

Hubs for VirtuosoNext: Online Verification of Real-Time Coordinators

Guillermina Cledou^a, José Proença^b, Bernhard H.C. Spath^c, Eric Verhulst^c

^a*HASLab/INESC TEC, Universidade do Minho, Portugal*

^b*CISTER, ISEP, Portugal*

^c*Altreonic NV, Belgium*

Abstract

VirtuosoNextTM 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.spath@altreonic.com (Bernhard H.C. Spath), bernhard.spath@altreonic.com (Eric Verhulst)

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 main-line 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,*

38 τ will eventually fire in less than 3 time units”. The verification uses UP-
 39 PAAL, resorting to an intermediate DSL for the logic that hides locations and
 40 auxiliary variables and clocks.

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

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

- 52 • Specification of hubs interpreted as **timed** (hub) automata,
 53 – capturing hubs currently present in VirtuosoNext (without real time),
 54 – including hubs not present in VirtuosoNext (some **with real time**).
- 55 • Online tools (<http://arcatools.org/hubs>) to analyse hubs,
 56 – using a DSL to specify hubs built by composing simpler hubs,
 57 – **using a DSL to specify timed contracts of tasks’ interactions**,
 58 – interpreting composed hubs as the composition of their **timed** au-
 59 tomata (c.f. [7]),
 60 – generating graphs and composed automata with dynamic layouts,
 61 – **introducing a temporal logic focused on interactions**,
 62 – **generating UPPAAL specifications and logic formulas**,
 63 – **running UPPAAL to verify properties**, and
 64 – including other analysis of hubs.

65 The rest of this paper is organized as follows. [Section 2](#) provides some con-
 66 text on how hubs coordinate tasks in VirtuosoNext, and how we can formally
 67 model existing and new hubs. [Section 3](#) presents the software architecture
 68 and functionality. [Section 4](#) introduces the verification tools, including timed
 69 contracts of tasks, the dynamic temporal logic, and the usage of UPPAAL to
 70 verify properties. [Section 5](#) exemplifies how to verify the behaviour of a com-
 71 plex hub under different scenarios. Finally, [Sections 6](#) and [7](#) discuss some
 72 related work and conclude with highlights and future directions, respectively.

73 2. Distributed tasks in VirtuosoNext

74 A VirtuosoNext *system* is executed on a target system, composed of pro-
75 cessing *nodes* and communication *links*. Orthogonally, an *application* consists
76 of a number of *tasks* coordinated by *hubs*. Unlike links, hubs are indepen-
77 dent of the hardware topology. When building application images, the code
78 generators of VirtuosoNext map tasks and hubs onto specific nodes, taking
79 into account the target platforms. A special *kernel task*, running on each
80 node, controls the scheduling of tasks, the hub services, and the internode
81 communication and routing.

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

89 2.1. Example of an architecture

90 A program in VirtuosoNext is a fixed set of tasks, each running on a
91 given computational node, and interacting with each other via dedicated
92 interaction entities, called hubs. Consider the example architecture in Fig. 1,
93 where tasks Task1 and Task2 send instructions to an Actuator task in a round
94 robin sequence. SemaphoreA tracks the end of Task1 and the beginning of
95 Task2, while SemaphoreB does the reverse, and port Actuate forwards the
96 instructions from each task to the Actuator. In this case two Semaphore
97 hubs were used, depicted by the diamond shape with a '+', and a Port hub,
98 depicted by a box with a 'P'. Tasks and hubs can be deployed on different
99 processing nodes, but this paper will consider only programs deployed in the
100 same node, and hence omit references to nodes. This and similar examples
101 can be found in the VirtuosoNext's manual [8].

102 2.2. Task coordination via Hubs

103 Hubs are coordination mechanisms between tasks that coordinate via *put*
104 and *get* service requests to transfer information from one task to another.
105 This can be a data element, the notification of an event occurrence, or some
106 logical entity that needs to be protected for atomic access. A call to a hub

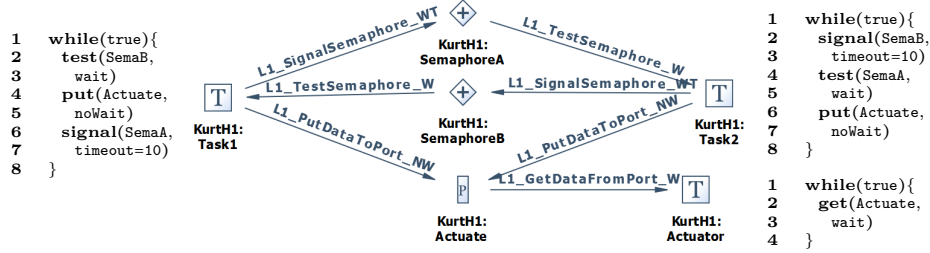


Figure 1: Example application in VirtuosoNext, whereby two tasks communicate with an actuator in a round robin sequence through two semaphores and a port.





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 $c < \text{MAX}$; 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.

134 2.3. Beyond VirtuosoNext: Custom Complex Hubs

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

143 More complex hubs can be built by plugging existing hubs together, also
 144 not currently supported by VirtuosoNext. For example, the composition
 145 $\rightarrow \textcircled{T} \rightarrow \boxed{D} \swarrow \searrow$ denotes a new hub that waits for a given time after receiving
 146 a value from its left port, and then synchronously sends it to both of the
 147 right ports. Fig. 2 describes a more complex architecture of a sequencer
 148 protocol than the one in Fig. 1, which we will use as a running example. Un-
 149 like in the sequencer in Fig. 1, the sequencing behaviour is captured by the
 150 hub (exogenous coordination), and it is not scattered among the compo-
 151 nents (endogenous coordination), making it easier to analyse and adapt or
 152 maintain. I.e., tasks in the original architecture are responsible to use the
 153 semaphores and the actuator in the right order to have an alternating be-
 154 haviour; 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 [*]	$\text{put}_1, \text{put}_2$ – signals some data entering the ports. Both put_1 and put_2 must synchronize to succeed.
 Duplicator	$\text{put}_1, \dots, \text{put}_n$ – signals some data entering the port; and $\text{get}_1, \dots, \text{get}_m$ – 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.

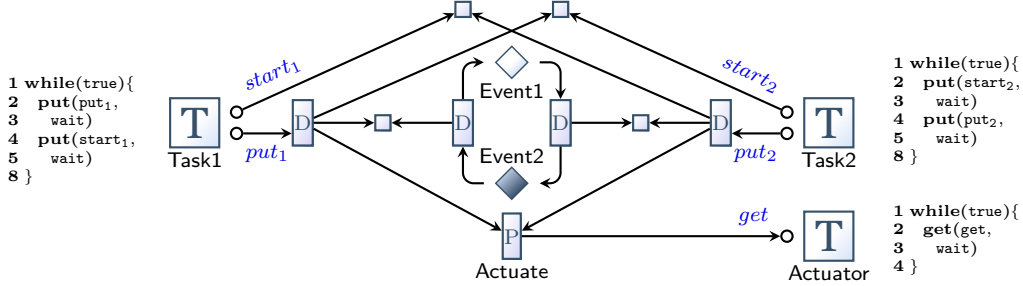


Figure 2: Alternative architecture for the sequencer protocol in Fig. 1.

155 a value, put_i , unaware of the coordination protocol.

156 2.4. Formal semantics in a nutshell

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

164 As in timed automata, there is a notion of clock variables that capture
 165 the dense time that passes since they were last reset. Initially, all clocks are
 166 set to zero, and are incremented simultaneously. THA additionally supports
 167 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).

168 **Example: Custom Alternator.** Fig. 3 (left) shows the THA that cap-
 169 tures the behaviour specified by the architecture in Fig. 2. Initially the
 170 automaton is in location $L1$, and it can move to a new location $L2$ by
 171 atomically performing actions from the three involved tasks (top transition),
 172 namely, put_1 , get , and $start_2$. While doing so, a special variable associated
 173 to port get , is assigned with the value sent through port put_1 in $\widehat{get} \leftarrow \widehat{put}_1$.
 174 The remaining transition behaves similarly.

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

188 3. Software Framework

189 3.1. Software Architecture

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

193 It provides support for generating an interactive website to use the tools, ei-
194 ther in a standalone lightweight JavaScript version, or in a Client-Server ver-
195 sion that enables the support of off-the-shelf applications from the browser.
196 The off-the-shelf tools include the UPPAAL real-time model checker used by
197 the Hubs module to verify temporal properties of the hubs.

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

203 The modules and the framework are developed in Scala, an object-oriented
204 programming language with functional features [12]. The Client-Server ver-
205 sion is compiled into JavaScript using Scala.js¹ to run on the client side, and
206 JVM binaries to run on the server side. The server is based on the Play
207 Framework² for Scala. The lightweight and the client side version use the
208 D3.js³ library to build interactive graphics in JavaScript. Note that cur-
209 rently the server is only used to model-check properties using UPPAAL, and
210 everything else is computed by the browser using the generated JavaScript
211 libraries.

212 3.2. Software Functionalities

213 We implemented a tool that *composes, simplifies, analyses, and verifies*
214 THA, available to use online or download on <http://arcatools.org/hubs>.
215 We organise the functionality by widgets, as depicted in Fig. 5. Our current
216 implementation allows specifications of composed hubs and tasks using a
217 textual representation based on Preo [13, 14] and Treo [15], by means of the
218 following widgets: ① the editor to specify the hub; ② the architectural view
219 of the hub; ③ the simplified automaton of the hub; ④ the timed automaton
220 to be imported by UPPAAL model checker; ⑤ a summary of some structural
221 properties of the automaton, such as required memory, size estimation of the
222 code, information about which hubs' ports are always ready to synchronise;
223 ⑥ an interactive panel to produce the minimum number of context switches
224 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/>

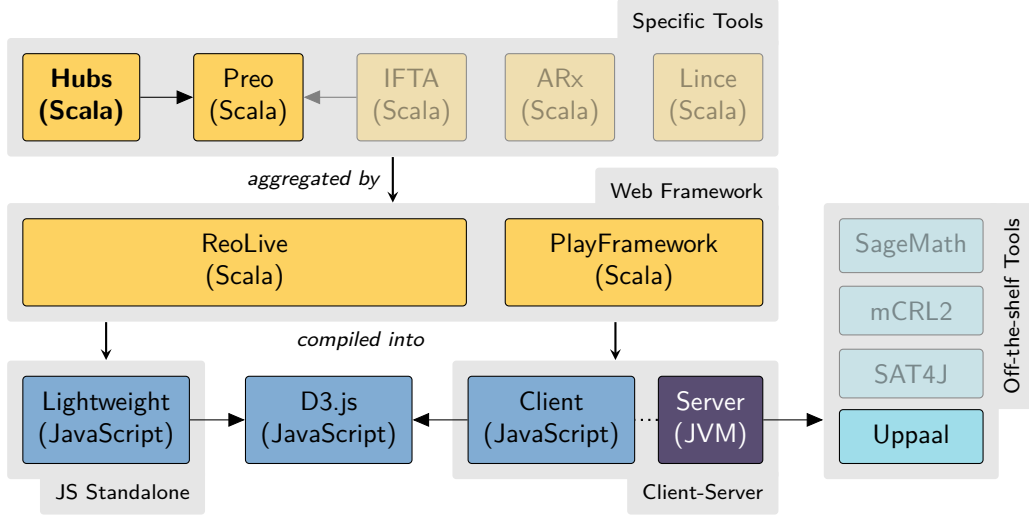


Figure 4: Software Architecture.

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.

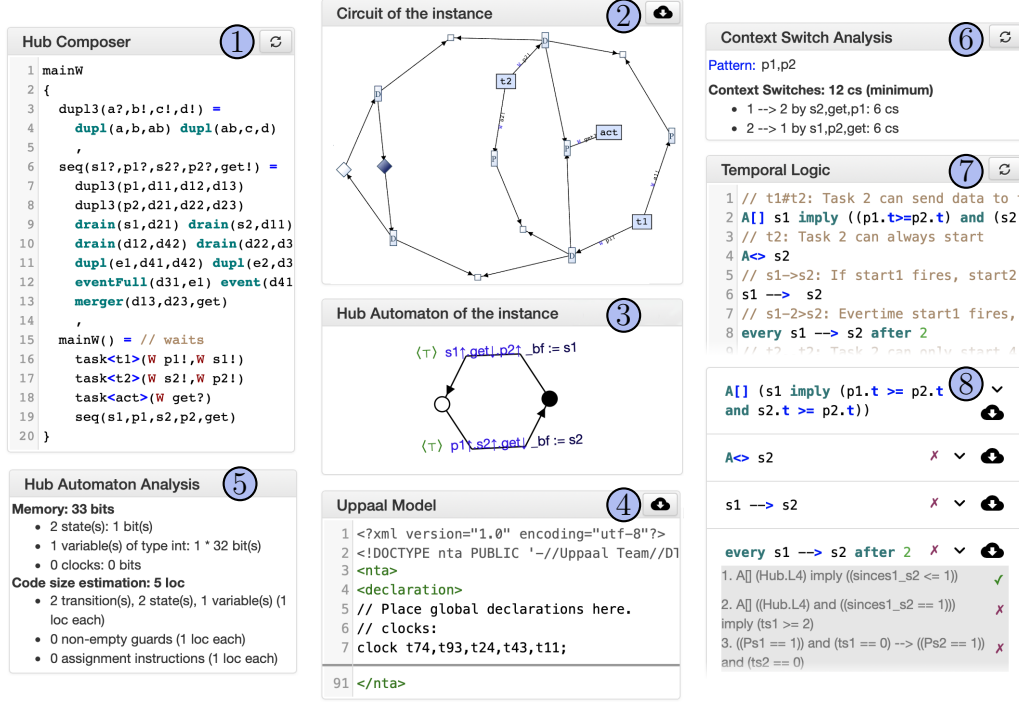


Figure 5: Screenshot of the widgets in the online analyser for VirtuosoNext's hubs.

Contracts for tasks can be specified by the following grammar.

$$\begin{aligned}
 tk &:= \text{task}\langle \text{name} \rangle(\text{port}^*) [\text{every } n] & \text{mode} &:= W \mid NW \mid n \\
 \text{port} &:= \text{mode name io} & \text{io} &:= ! \mid ?
 \end{aligned}$$

241 This syntax has been briefly mentioned in [Section 2.2](#). For example, `task<T1>`
 242 `(W a?, 4 b!)` specifies a task that tries to read a value on its port `a`, waiting
 243 indefinitely (`W`), followed by a call to write a value to port `b` with a timeout of
 244 4 time units, after which it loops again following the same behaviour forever.
 245 This example, when extended with `every 5`, will periodically run every 5 time
 246 units. In our interpretation of a periodic run, every round of the execution
 247 of this task takes exactly 5 time units, and repeats forever. In each round `a`
 248 fires once and `b` either fires or times-out; hence `a` can wait at most 10 time
 249 units between 2 fires (when it fires at the beginning and end of consecutive
 250 rounds). If after 5 time units after the start of a round `a` fires and `b` cannot
 251 fire, then `b` will timeout and not fire for that round. As another example,
 252 `task<T2>(NW c!) every 5` will periodically try to send a value to port `c` every

where $a, b \in \mathcal{P}$ are ports, $n \in \mathbb{N}$, $\boxtimes \in \{\square, \diamond\}$, $\text{pred}(\bar{x})$ 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.

$$\begin{aligned} cc &::= c \boxdot n \mid c - c \boxdot n && \text{(clock constraint)} \\ \rho &::= a.\text{done} \mid a.\text{doing} \mid a \text{ refiresAfter } n \mid a \text{ refiresAfterOrAt } n && \text{(a-formula)} \end{aligned}$$

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, $a.\text{done}$ holds if a has fired at least once, $a.\text{doing}$ holds if a was the last port to be fired, and $a \text{ refiresAfterOrAt } 5$ holds in states where, if a fired before, then it cannot re-fire unless 5 units of time have passed. Regarding path properties, $A \boxtimes \psi$ 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 , $\square \psi$ holds if ψ holds for all states along p , $\diamond \psi$ holds if a state along p satisfies ψ , and $\psi_1 \text{ --> } \psi_2$ is a shorthand for $A \square (\psi_1 \text{ imply } (A \diamond \psi_2))$.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, $\text{every } a \text{ --> } b \text{ after } 5$ 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 $H, s \models \pi$, can be found in the companion report [7]. This grammar is enriched with a special clock $a.t$ 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 \wedge to indicate the generalised **and** for multiple state formulas, and P to denote the set of all ports used by a given THA.

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

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

⁴As in UPPAAL, nested path formulas are not supported explicitly. However, some are introduced through specific constructs like $\psi \text{ --> } \psi$.

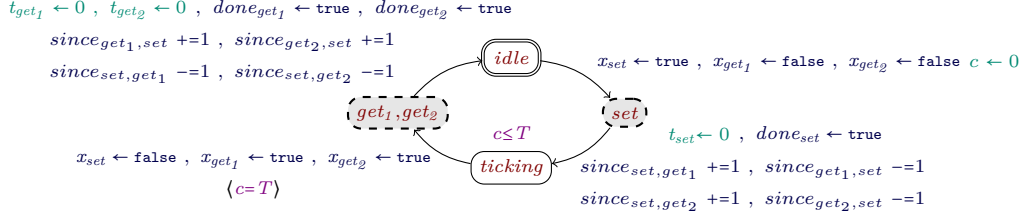


Figure 7: Encoded UPPAAL's automaton of the THA from Fig. 3; dashed locations are committed.

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 \bowtie 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 *get₁* was fired (or from the beginning), and it is 2 if it was fired more than

326 once since the last time *get₁* was fired. These variables are used when
 327 verifying formulas like **every** $a \dashrightarrow b$, where for each a fired, b should
 328 fire without a firing in between.

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

336 **Priority:** Recall that tasks are modelled as timed automata with a notion
 337 of priority (Section 4.1). This priority is meant only to prevent ports from
 338 discarding data and timing out when the hub is ready to communicate. This
 339 is encoded in UPPAAL using its notion of *channel priority*. *Channels* in UP-
 340 PAAL are labels of transitions in automata used to synchronise with channels
 341 of neighbour automata. Our encoding does not rely on channels since it pro-
 342 duces a single automaton, but we introduce a set of dummy channels $prio_p$
 343 that can always be fired⁵, where $p \in \mathbb{Z}$ denotes the priority of the channel
 344 (higher numbers mean higher priority). Transitions in an automaton are
 345 marked with priority 0 if it synchronizes with other automata, and with pri-
 346 ority -1 if it denotes a timeout. During composition, priorities of transitions
 347 that go together are added up, reducing the priority of transitions with more
 348 timeouts.

349 4.3.2. Encoding Formulas by Example

350 The UPPAAL⁶ model checker supports a subset of TCTL formulas for
 351 timed automata [16], which we took into account when proposing the logic
 352 for THA. The key differences with our logic are: the use of locations ($ta.\ell$)
 353 in state formulas, the absence of references to actions (or ports) and their
 354 associated clocks, and the absence of the **every**-path formula. Hence, when
 355 encoding our logic into UPPAAL's TCTL, each of the missing constructs are
 356 mimicked using the extra variables and clocks, and using references to known
 357 locations in the automata encoding.

⁵This 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
$A \Diamond \text{put}_2 \text{ and } \text{get}$	$A \Diamond x_{\text{put}_2} \text{ and } t_{\text{put}_2} = 0 \text{ and } x_{\text{get}} \text{ and } t_{\text{get}} = 0$
$A \Box \text{act.doing or nothing}$	$A \Box x_{\text{act}} \text{ or } (\text{not } x_{\text{get}} \text{ and not } x_{\text{put}_1} \text{ and not } x_{\text{put}_2})$
every $\text{put}_1 \dashrightarrow \text{put}_2$ after 2	$\begin{cases} x_{\text{put}_1} \dashrightarrow x_{\text{put}_2} \\ A \Box \text{cmt}(\text{put}_2) \text{ imply } \text{since}_{\text{put}_1, \text{put}_2} \leq 1 \\ A \Box \left(\begin{array}{l} \text{cmt}(\text{put}_2) \text{ and} \\ \text{since}_{\text{put}_1, \text{put}_2} = 1 \end{array} \right) \text{ imply } t_{\text{put}_1} \geq 2 \end{cases}$
$A \Diamond \text{put}_1 \text{ refiresAfterOrAt } 2$	$A \Diamond (\text{done}_{\text{put}_1} \text{ and } \text{cmt}(\text{put}_1)) \text{ imply } t_{\text{put}_1} \geq 2$

To refer to the committed locations introduced in [Section 4.3.1](#) we will use the following shorthand, where a is a port:

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

358 The encoding of examples of key formulas is presented in [Table 3](#) – the
359 general encoding of formulas follows the same structure as in these examples,
360 and is omitted in this paper. This proof relies on the fact that the observable
361 behaviour is not modified by adding new intermediate committed states to
362 an automaton that has no committed states, and by adding new variable
363 assignments that are never read.

364 5. Example: Verifying the sequencer protocol

365 Recall our running example illustrated in [Fig. 2](#) of a sequencer proto-
366 col. We illustrate the proposed specification constructs for tasks and time-
367 sensitive behavioural properties by verifying different properties under dif-
368 ferent scenarios, i.e., connecting tasks with different models of interaction
369 to the hub. The goal is to provide some insight on how to use our tools to
370 understand the different expected behaviours of a hub in different scenarios.

371 We create 5 different scenarios with 2 producer tasks and an actuator task,
372 varying on how the producer tasks interact with the hub. More specifically,
373 using wait, non-wait, and timeout calls to the hubs, at different periodicities.
374 These scenarios are presented in [Table 4](#) (left column). Notice that the first
375 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}	$\psi_{s1 \rightarrow s2}$	$\psi_{s1 \xrightarrow{2} s2}$	$\psi_{\leq 9}$
task<T1>(W put ₁ !, W st ₁ !) task<T2>(W st ₂ ! , W put ₂ !) task<Ac>(W get?)	✓	✗	✗	✗	✗
task<T1>(W put ₁ !, W st ₁ !) every 3 task<T2>(W st ₂ ! , W put ₂ !) every 3 task<Ac>(W get?)	✓	✓	✓	✗	✓
task<T1>(NW put ₁ !, NW st ₁ !) every 3 task<T2>(NW st ₂ ! , NW put ₂ !) every 3 task<Ac>(W get?)	✓	✓	✓	✓	✓
task<T1>(3 put ₁ !, 3 st ₁ !) every 6 task<T2>(3 st ₂ ! , 3 put ₂ !) every 6 task<Ac>(W get?)	✓	✓	✓	✗	✓
task<T1>(NW put ₁ !, 3 st ₁ !) every 2 task<T2>(W st ₂ ! , 3 put ₂ !) 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.

$\psi_{t1 \# t2} = \{ \mathbf{A} \Box \text{start}_1 \text{ imply } ((\text{put}_1.t \geq \text{put}_2.t) \text{ and } (\text{start}_2.t \geq \text{put}_2.t)) \}$
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.

$\psi_{t2} = \{ \mathbf{A} \Diamond \text{start}_2 \}$
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.

$\psi_{s1 \rightarrow s2} = \{ \text{start}_1 \dashrightarrow \text{start}_2 \}$
If start₁ fires, start₂ 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

393 deadlock occurs after T1 finishes the 1st round firing both ports, and it
 394 fails to fire *put₁* in the 2nd round. Both T1 and T2 wait to fire *start* in
 395 their 2nd round, and time cannot pass at the end of the T1's round.

396 $\psi_{s1 \rightarrow s2} = \{\text{every } \textit{start}_1 \text{ --> } \textit{start}_2 \text{ after } 2\}$
 397 *Everytime \textit{start}_1 fires, \textit{start}_2 must eventually fire before \textit{start}_1 again,*
 398 *and wait at least 2 time units before firing \textit{start}_2 .* This is a variation of
 399 the previous property with a periodicity. All but the 3rd scenario fail
 400 to satisfy this property: the 1st scenario fails because rounds can be
 401 faster than 2; the 2nd and 4th fail because \textit{start}_1 can be executed at
 402 the end of a round, and \textit{put}_2 at the beginning of the following round;
 403 the last scenario fails for the same reason $\psi_{s1 \rightarrow s2}$ does.

404 $\psi_{\leq 9} = \{\mathbf{A} \square \textit{start}_2 \text{ refiresBeforeOrAt } 9\}$
 405 *Task 2 starts within 9 time units after finishing a previous round. Only*
 406 *the first scenario fails, since it can take an infinite amount of time*
 407 *between two fires of \textit{start}_2 . The 2nd scenario can take up to 6 time*
 408 *units between fires of \textit{start}_2 , the 4th can take up to 9 time units when*
 409 *\textit{start}_2 fires at the beginning of a round, and right before timing out in*
 410 *the follow up round (6+3 time units).*

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

419 6. Related work

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

427 Timed Hub Automata is inspired by existing automata-based models for
 428 Reo [2, 17, 3, 5], involving data, variables, and time. The semantics based on
 429 timed automata provide encodings of Reo connectors using the same notion
 430 of time used by UPPAAL, as we do, and further exploit the notion of automata
 431 composition embedded in UPPAAL. Unlike these approaches, we introduce
 432 a notion of sequential and parallel updates, and facilitate the verification
 433 process for the end user, by providing support for a fine-tuned language for
 434 specifying logical properties agnostic of locations and for describing timed
 435 scenarios. We avoid exposing the user to UPPAAL, using a similar automata
 436 model that is better suited for multiple actions.

437 Formal analysis of RTOS are more typically focused on the scheduler,
 438 which is not the focus of this work. The following are examples of relevant
 439 scheduling analysis. Ha et al. [18] used theorem provers to analyse schedulers
 440 for avionics software. Carnevali et al. [19] used preemptive Time Petri Nets
 441 to support exact scheduling analysis and guide the development of tasks with
 442 non-deterministic execution times in an RTOS with hierarchical scheduling.
 443 Dietrich et al. [20] analysed and model checked all possible execution paths of
 444 a real-time system to tailor the kernel to particular application scenarios, re-
 445 sulting in optimisations in execution speed and robustness. Dokter et al. [21]
 446 proposed a framework to synthesise optimised schedulers that consider delays
 447 introduced by interaction between tasks, interpreting scheduling as a game
 448 that requires minimising the time between subsequent context switches.

449 7. Conclusions

450 This article presents a toolset to construct and analyse hubs in Virtuoso-
 451 Next, which are services used to orchestrate interacting tasks in a Real Time
 452 OS that runs on embedded devices. When using VirtuosoNext, programmers
 453 can orchestrate individual tasks by using a set of core hubs, provided as
 454 services by the OS. More complex interaction mechanisms must be encoded
 455 within the tasks, which is hard to debug and maintain.

456 Our proposed formal framework provides mechanisms to design and im-
 457 plement complex hubs that can be formally analysed and verified to provide
 458 the same level of assurance that predefined hubs provide. Currently, the
 459 framework allows to (1) **construct** complex hubs out of simpler ones, (2)
 460 **verify** timed properties using a variation of TCTL used by UPPAAL tailored
 461 to reason about interactions with hubs, and (3) **analyse** some aspects of
 462 the hubs such as: memory used, estimated lines of codes, always available

463 ports, and minimum number of context switches required to perform certain
464 behaviour. This is publicly available both to run online using our web inter-
465 face, and to download and execute locally (<http://arcatools.org/hubs>).

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

470 Ongoing work to extend our formal framework includes:

- 471 • **variability support** to analyse and improve the development of fam-
472 ilies of systems in VirtuosoNext, since VirtuosoNext provides a simple
473 and error-prone mechanism to allow topologies to be applied to the
474 same set of tasks;
- 475 • **code refactoring and generation** applied to existing (on-production)
476 VirtuosoNext programs, probably adding new primitive hubs, by ex-
477 tracting the coordination logic from tasks into new complex hubs; and
- 478 • **analysis extension** to support a wider range of analysis to Hub Au-
479 tomata, such as the model checking of liveness and safety properties
480 using other tools, e.g. mCRL2 (c.f. [13, 22]).

481 *Acknowledgements.* This work is financed by the ERDF – European Regional De-
482 velopment Fund through the Operational Programme for Competitiveness and In-
483 ternationalisation – COMPETE 2020 Programme and by National Funds through
484 the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnolo-
485 gia, within project POCI-01-0145-FEDER-029946 (DaVinci). This work is also
486 partially supported by National Funds through FCT/MCTES, within the CIS-
487 TER Research Unit (UIDB/04234/2020); by the Norte Portugal Regional Oper-
488 ational Programme (NORTE 2020) under the Portugal 2020 Partnership Agree-
489 ment, through ERDF and also by national funds through the FCT, within project
490 NORTE-01-0145-FEDER-028550 (REASSURE); by the Operational Competitive-
491 ness Programme and Internationalization (COMPETE 2020) under the PT2020
492 Partnership Agreement, through ERDF, and by national funds through the FCT,
493 within project POCI-01-0145-FEDER-029119 (PReFECT); and by the FCT within
494 project ECSEL/0016/2019 and the ECSEL Joint Undertaking (JU) under grant
495 agreement No 876852. The JU receives support from the European Union’s Hori-
496 zon 2020 research and innovation programme and Austria, Czech Republic, Ger-
497 many, Ireland, Italy, Portugal, Spain, Sweden, Turkey.

References

- [1] E. Verhulst, R. T. Boute, J. M. S. Faria, B. H. Spath, V. Mezhyuev, 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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1007/978-3-319-68972-2_4).
- [6] G. Cledou, J. Proença, B. H. C. Spath, 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](https://doi.org/10.1007/978-3-030-22397-7_15).
- [7] G. Cledou, J. Proença, B. H. C. Spath, E. Verhulst, Verification of Real-Time Coordination in VirtuosoNext (extended version) (May 2020). [doi:10.5281/zenodo.3818020](https://doi.org/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.

- 529 [9] J. Bengtsson, W. Yi, Timed Automata: Semantics, Algorithms and
530 Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 87–124.
531 [doi:10.1007/978-3-540-27755-2_3](https://doi.org/10.1007/978-3-540-27755-2_3).
- 532 [10] S. Meng, F. Arbab, On resource-sensitive timed component connectors,
533 in: M. M. Bonsangue, E. B. Johnsen (Eds.), Formal Methods for Open
534 Object-Based Distributed Systems, 9th IFIP WG 6.1 International Con-
535 ference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings,
536 Vol. 4468 of Lecture Notes in Computer Science, Springer, 2007, pp.
537 301–316. [doi:10.1007/978-3-540-72952-5_19](https://doi.org/10.1007/978-3-540-72952-5_19).
- 538 [11] F. Arbab, Reo: a channel-based coordination model for component
539 composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366. [doi:
540 10.1017/S0960129504004153](https://doi.org/10.1017/S0960129504004153).
- 541 [12] M. Odersky, L. Spoon, B. Venners, Programming in scala, Artima Inc,
542 2008.
- 543 [13] R. Cruz, J. Proença, ReoLive: Analysing connectors in your browser,
544 in: M. Mazzara, I. Ober, G. Salaün (Eds.), Software Technologies:
545 Applications and Foundations - STAF 2018 Collocated Workshops,
546 Toulouse, France, June 25-29, 2018, Revised Selected Papers, Vol. 11176
547 of Lecture Notes in Computer Science, Springer, 2018, pp. 336–350.
548 [doi:10.1007/978-3-030-04771-9_25](https://doi.org/10.1007/978-3-030-04771-9_25).
- 549 [14] J. Proença, A. Madeira, Taming hierarchical connectors, in: H. Hoj-
550 jat, M. Massink (Eds.), Fundamentals of Software Engineering - 8th
551 International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Re-
552 visited Selected Papers, Vol. 11761 of Lecture Notes in Computer Science,
553 Springer, 2019, pp. 186–193. [doi:10.1007/978-3-030-31517-7_13](https://doi.org/10.1007/978-3-030-31517-7_13).
- 554 [15] K. Dokter, F. Arbab, Treo: Textual syntax for reo connectors, in: S. Bli-
555 udze, S. Bensalem (Eds.), Proceedings of the 1st International Workshop
556 on Methods and Tools for Rigorous System Design, MeTRiD@ETAPS
557 2018, Thessaloniki, Greece, 15th April 2018, Vol. 272 of EPTCS, 2018,
558 pp. 121–135. [doi:10.4204/EPTCS.272.10](https://doi.org/10.4204/EPTCS.272.10).
- 559 [16] G. Behrmann, A. David, K. G. Larsen, A Tutorial on Uppaal, Springer
560 Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 200–236. [doi:10.1007/
561 978-3-540-30080-9_7](https://doi.org/10.1007/978-3-540-30080-9_7).

- 562 [17] S.-S. Jongmans, T. Kappé, F. Arbab, Constraint automata with mem-
 563 ory cells and their composition, *Science of Computer Programming* 146
 564 (2017) 50 – 86, special issue with extended selected papers from FACS
 565 2015. doi:<https://doi.org/10.1016/j.scico.2017.03.006>.
- 566 [18] V. Ha, M. Rangarajan, D. D. Cofer, H. Rueß, B. Dutertre, Feature-based
 567 decomposition of inductive proofs applied to real-time avionics software:
 568 An experience report, in: A. Finkelstein, J. Estublier, D. S. Rosenblum
 569 (Eds.), 26th International Conference on Software Engineering (ICSE
 570 2004), 23-28 May 2004, Edinburgh, United Kingdom, IEEE Computer
 571 Society, 2004, pp. 304–313. doi:[10.1109/ICSE.2004.1317453](https://doi.org/10.1109/ICSE.2004.1317453).
- 572 [19] L. Carnevali, G. Lipari, A. Pinzuti, E. Vicario, A formal approach to
 573 design and verification of two-level hierarchical scheduling systems, in:
 574 A. B. Romanovsky, T. Vardanega (Eds.), *Reliable Software Technolo-*
 575 *gies - Ada-Europe 2011 - 16th Ada-Europe International Conference on*
 576 *Reliable Software Technologies*, Edinburgh, UK, June 20-24, 2011. Pro-
 577 ceedings, Vol. 6652 of *Lecture Notes in Computer Science*, Springer,
 578 2011, pp. 118–131. doi:[10.1007/978-3-642-21338-0_9](https://doi.org/10.1007/978-3-642-21338-0_9).
- 579 [20] C. Dietrich, M. Hoffmann, D. Lohmann, Global Optimization of Fixed-
 580 Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis,
 581 *ACM Trans. Embed. Comput. Syst.* 16 (2) (2017) 35:1–35:25. doi:
 582 [10.1145/2950053](https://doi.org/10.1145/2950053).
- 583 [21] K. Dokter, S. Jongmans, F. Arbab, Scheduling games for concurrent
 584 systems, in: A. Lluch-Lafuente, J. Proença (Eds.), *Coordination Models*
 585 *and Languages - 18th IFIP WG 6.1 International Conference, COORDI-*
 586 *NATION 2016, Held as Part of the 11th International Federated Confer-*
 587 *ence on Distributed Computing Techniques, DisCoTec 2016, Heraklion,*
 588 *Crete, Greece, June 6-9, 2016, Proceedings*, Vol. 9686 of *Lecture Notes*
 589 *in Computer Science*, Springer, 2016, pp. 84–100. doi:[10.1007/978-](https://doi.org/10.1007/978-3-319-39519-7_6)
 590 [3-319-39519-7_6](https://doi.org/10.1007/978-3-319-39519-7_6).
- 591 [22] N. Kokash, C. Krause, E. P. de Vink, Reo + mcrl2: A framework for
 592 model-checking dataflow in service compositions, *Formal Asp. Comput.*
 593 24 (2) (2012) 187–216. doi:[10.1007/s00165-011-0191-6](https://doi.org/10.1007/s00165-011-0191-6).

594 **Required Metadata**

595 **Current code version**

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1.0
C2	Permanent link to code/repository used of this code version	https://github.com/arcalab/hubAutomata/releases/tag/v1.0
C3	Legal Code License	MIT
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Scala, ScalaJS, Scala Play Framework, JavaScript
C6	Compilation requirements, operating environments & dependencies	Requirements: Java Runtime Environment – https://www.java.com/en/download/ , Uppaal Model Checker (optional) – http://www.uppaal.org/ , Scala Building Tools – https://www.scala-sbt.org/ .
C7	If available Link to developer documentation/manual	https://hubs.readthedocs.io
C8	Support email for questions	mgc@inesctec.pt, pro@isep.ipp.pt

Table 5: Code metadata (mandatory)