Coordination of tasks on a Real-Time OS

Guillermina Cledou[1], José Proença[1,2], Bernhard H.C. Sputh[3], and Eric Verhulst[3]

[1] HASLab/INESC TEC, Universidade do Minho, Portugal
[2] CISTER, ISEP, Portugal
[3] Altreonic NV, Belgium
mgc@inesctec.pt, pro@isep.ipp.pt,
{bernhard.sputh,eric.verhulst}@altreonic.com

Abstract. VirtuosoNext^{TM} is a distributed real-time operating system (RTOS) featuring a generic programming model dubbed *Interacting Entities*. This paper focuses on these interactions, implemented as so-called *Hubs*. Hubs act as synchronisation and communication mechanisms between the application tasks and implement the services provided by the kernel as a kind of Guarded Protected Action with a well defined semantics. While the kernel provides the most basic services, each carefully designed, tested and optimised, tasks are limited to this handful of basic hubs, leaving the development of more complex synchronization and communication mechanisms up to application specific implementations. In this work we investigate how to support a programming paradigm to compositionally build new services, using notions borrowed from the Reo coordination language, and relieving tasks from coordination aspects while delegating them to the hubs. We formalise the semantics of hubs using an automata model, identify the behaviour of existing hubs, and propose an approach to build new hubs by composing simpler ones. We also provide tools and methods to analyse and simplify hubs under our automata interpretation. In a first experiment several hub interactions are combined into a single more complex hub, which raises the level of abstraction and contributes to a higher productivity for the programmer. Finally, we investigate the impact on the performance by comparing different implementations on an embedded board.

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 [16] 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 from a visual description and provides tools to analyse the timing behaviour in detail.

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

This paper starts by formalising hubs using an automata model, which we call Hub Automata, inspired in Reo's parametrised constraint automata semantics [1]. This formalism brings several advantages. On the one hand, it brings a generic language to specify hubs, which can be interpreted by VirtuosoNext's kernel task. New hubs can be built by specifying new Hub Automata, or by composing the Hub Automata from existing hubs. On the other hand, it allows existing (and new) hubs to be formally analysed, estimating performance and memory consumption, and verifying desired properties upfront. Furthermore, we show that by using more specific hubs one can shift some of the coordination burden from the tasks to the hubs, leading to easier and less error prone programming of complex protocols, as well as leaving room for optimizations. In some cases it can also reduce the amount of context switches between application tasks and the kernel task of VirtuosoNext, improving performance.

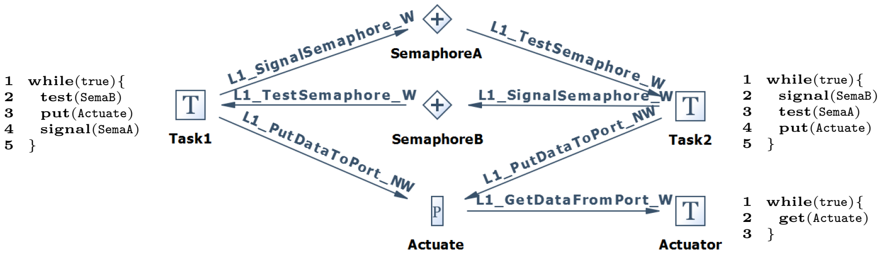
We implemented a prototype implementation, available online,[4] to compose hubs based on our Hub Automata semantics, and to analyse and simplify them. We also compared the execution times on an embedded system between different orchestration scenarios of tasks, one using existing hubs and another using a more refined hub built out of the composition of hubs, evidencing the performance gains and overheads of using composed hubs.

Summarising, our key contributions are the formalisation of coordinating hubs in VirtuosoNext (Section 3), a compositional semantics of hubs (Section 4), and a set of tools to compose and analyse hubs, together with a short evaluation of the execution times of a given scenario using composed hubs (Section 5).

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

[4] http://github.com/arcalab/hubAutomata



```
while(true){
  test(SemaB)
  put(Actuate)
  signal(SemaA)
}
```

```
while(true){
  signal(SemaB)
  test(SemaA)
  put(Actuate)
}
```

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

Fig. 1: Example architecture in VirtuosoNext, where two tasks communicate with an actuator in a round robin sequence through two semaphores and a port.

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 scheduler of tasks, the hub services, and the internode communication and routing.

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

2.1 Example of an architecture

A program in VirtuosoNext is a fixed set of tasks, each running on a given computational node, and interacting with each other via dedicated interaction entities, called hubs. Consider the example architecture in Fig. 1, where tasks Task1 and Task2 send instructions to an Actuator task in a round robin sequence. SemaphoreA tracks the end of Task1 and the beginning of Task2, while SemaphoreB does the reverse, and port Actuate forwards the instructions from each task to the Actuator. In this case two Semaphore hubs were used, depicted by the diamond shape with a '+', and a Port hub, depicted by a box with a 'P'. Tasks and hubs can be deployed on different processing nodes, but this paper will consider only programs deployed in the same node, and hence omit references to nodes. This and similar examples can be found in the VirtuosoNext's manual [13].

2.2 Task coordination via Hubs

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

Table 1: Examples of existing Hubs in VirtuosoNext

Hub	**Waiting Lists for Service Requests**
P 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. Succeeds if not set yet; and `test` – checks if an event happened, in which case succeeds, and clears the event.
D DataEvent	`update` – sets an event and buffers some data, overriding any previous data. Always succeeds; `read` – reads the data. Succeeds if the event is set; and `clear` – clears the buffer and the event.
⊕ Semaphore	`signal` – signals the semaphore, incrementing an internal counter `c`. Succeeds if $\texttt{c} < \texttt{MAX}$;[5] and `test` – checks if $\texttt{c} > \texttt{0}$, in which case succeeds, and decrements `c`.
🔒 Resource	`lock` – locks a logical resource and buffers the `id` of the requesting task. Succeeds only if the resource is free; and `unlock` – unlocks the resource. Succeeds only if locked by the same task.
▥ FIFO	`enqueue` – buffers some data in the queue. Succeeds if the queue is not full; and `dequeue` – gets data from the queue. Succeeds if the queue is not empty.
BB Blackboard	`update` – buffers some data, overriding any previews data, incrementing a sequence number. Always succeeds; `read` – reads the data and the sequence number. Succeeds if not empty. Reader tasks can use the sequence number to attest the freshness of the data; and `wipe` – clears the buffer.

Any number of tasks can make put or get requests to a given hub. Such requests will be queued in waiting lists (at each corresponding hub) until they can be served. Waiting lists are ordered by task priority – requests get served by following such an order. In addition, requests can use different interaction semantics. As such, the interaction can be *blocking*, *non-blocking* or *blocking with a time-out*, which will determine how much time, if any, a task will wait on a request to succeed – indefinitely, none, or a certain amount of time, respectively.

There are various hubs available, each with its predefined semantics [13]. Table 1 describes some of them and their put and get service request methods.

3 Deconstructing Hubs

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

[5] Here, `MAX` represents `L1_UINT32_MAX` in VirtuosoNext™, which is $2^{32} - 1$.

3.1 Syntax

We formalise the behavioural semantics of a hub using an automata model, which we call *Hub Automata*. We start by introducing some preliminary concepts.

Definition 1 (Guard). *A guard $\phi \in \Phi$ is a logical formula given by the grammar below, where $x \in \mathcal{X}$ is a variable, $\overline{x}$ denotes a sequence of variables, and $pred \in \mathcal{P}red$ is a predicate.*

$$\phi \ := \ \top \mid \bot \mid pred(\overline{x}) \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi$$

We say $\Phi(\mathcal{X})$ is the set of all possible guards over variables in $\mathcal{X}$.

Definition 2 (Update). *An update $u \in \mathcal{U}$ is an assignment of variables $x \in \mathcal{X}$ to expressions $e \in \mathcal{E}$, a sequence of updates, or updates in parallel, given by the grammar below, where $d \in \mathcal{D}$ is a data value, and $f \in \mathcal{F}$ is a deterministic function without side-effects.*

$$\begin{aligned} u &:= x \leftarrow e \mid u;u \mid u|u && \text{(update)} \\ e &:= d \mid x \mid f(\overline{x}) && \text{(expression)} \end{aligned}$$

We write $\mathcal{U}(\mathcal{X})$ to denote the set of all updates over variables in $\mathcal{X}$.

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

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

Definition 3 (Hub Automata). *A* Hub Automaton *is a tuple $H = (L, \ell_0, P, \mathcal{X}, v_0, \rightarrow)$ where L is a finite set of locations, ℓ_0 is the initial location, $P = P_I \uplus P_O$, is a finite set of ports, with P_I and P_O representing the disjoint sets of input and output ports, respectively, $\mathcal{X}$ is a finite set of internal variables, $v_0 : \mathcal{X} \rightarrow \mathcal{D}$ is the initial valuation that maps variables in $\mathcal{X}$ to a value in $\mathcal{D}$, and $\rightarrow \subseteq L \times \Phi(\mathcal{X} \cup \hat{\mathcal{P}}) \times 2^P \times \mathcal{U}(\mathcal{X} \cup \hat{\mathcal{P}}) \times L$ is the transition relation.*

For a given transition $(l, g, \omega, u, l') \in \rightarrow$, also written $l \xrightarrow{g,\omega,u} l'$, l is the source location, g is the guard defining the enabling condition, ω is the set of ports triggering the transition, u is the update triggered, and l' is the target location.

Informally, a Hub Automaton is a finite automaton enriched with *variables* and an *initial valuation* of such variables; and where transitions are enriched with *multi-action* transitions, and *logic guards* and *updates* over variables. A transition $l \xrightarrow{g,\omega,u} l'$ is enabled only if (1) all of its ports ω are ready to be executed simultaneously, and (2) the current valuation satisfies the associated guard g. Performing this transition means applying the update u to the current valuation, and moving to location l'. This is formalised in the following section.

Fig. 2 depicts the Hub Automata for each of the hubs described in Section 2.2, except the Resource hub (for space restrictions). Consider, for example, the Hub Automaton for the FIFO hub, implemented using an internal circular queue, with size N and with elements of type T. Initially, the FIFO is at location *idle* and its internal variables are assigned as follows: $c \mapsto 0$, $f \mapsto 0$, $p \mapsto 0$, and $bf_i \mapsto null$ for all $i \in \{0 \dots N-1\}$. Here c is the current number of elements in the queue, f and p are the pointers to the front and last empty place of the queue, respectively, and each bf_i holds the value of the i-th position in the queue. The FIFO can *enqueue* an element —if the queue is not full ($c < N$)— storing the incoming data value in bf_p, and increasing the c and p counters; or it can *dequeue* an element—if the queue is not empty ($c \geq 1$), updating the corresponding variables.

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

Hub Automata can be used to describe new hubs to restrict synchronous interactions between tasks. Fig. 2 includes two hubs that do not exist in VirtuosoNext (hubs with *): a Duplicator broadcasts a given input to two outputs atomically, and a Drain receives two inputs synchronously without storing any value.

3.2 Semantics

We start by defining guard satisfaction, used by the semantics of Hub Automata.

Definition 4 (Guard Satisfaction). *The satisfaction of a guard g by a variable valuation v, written $v \models g$, is defined as*

$$\begin{array}{llll}
v \models \top & \textit{always} & v \models \phi_1 \wedge \phi_2 & \textit{if } v \models \phi_1 \textit{ and } v \models \phi_2 \\
v \models \bot & \textit{never} & v \models \phi_1 \vee \phi_2 & \textit{if } v \models \phi_1 \textit{ or } v \models \phi_2 \\
v \models \neg\phi & \textit{if } v \not\models \phi & v \models \textit{pred}(\overline{x}) & \textit{if } \textit{pred}(v(\overline{x})) \textit{ evaluates to } \texttt{true}
\end{array}$$

Definition 5 (Update application). *Given a serialisation function σ that converts general updates into sequences of assignments, the application of an update u to a valuation v is given by $v[\sigma(u)]$, where $v[-]$ is defined below.*

$$\begin{array}{ll}
v[x \leftarrow e](x) = e & \\
v[x \leftarrow e](y) = v(y) \quad \textit{if } x \neq y
\end{array}
\qquad v[u_1; u_2](x) = (v[u_1])[u_2](x)$$

The serialisation function σ is formalised in Section 4.3, after describing how to compose Hub Automata. We will omit σ when not relevant. The execution

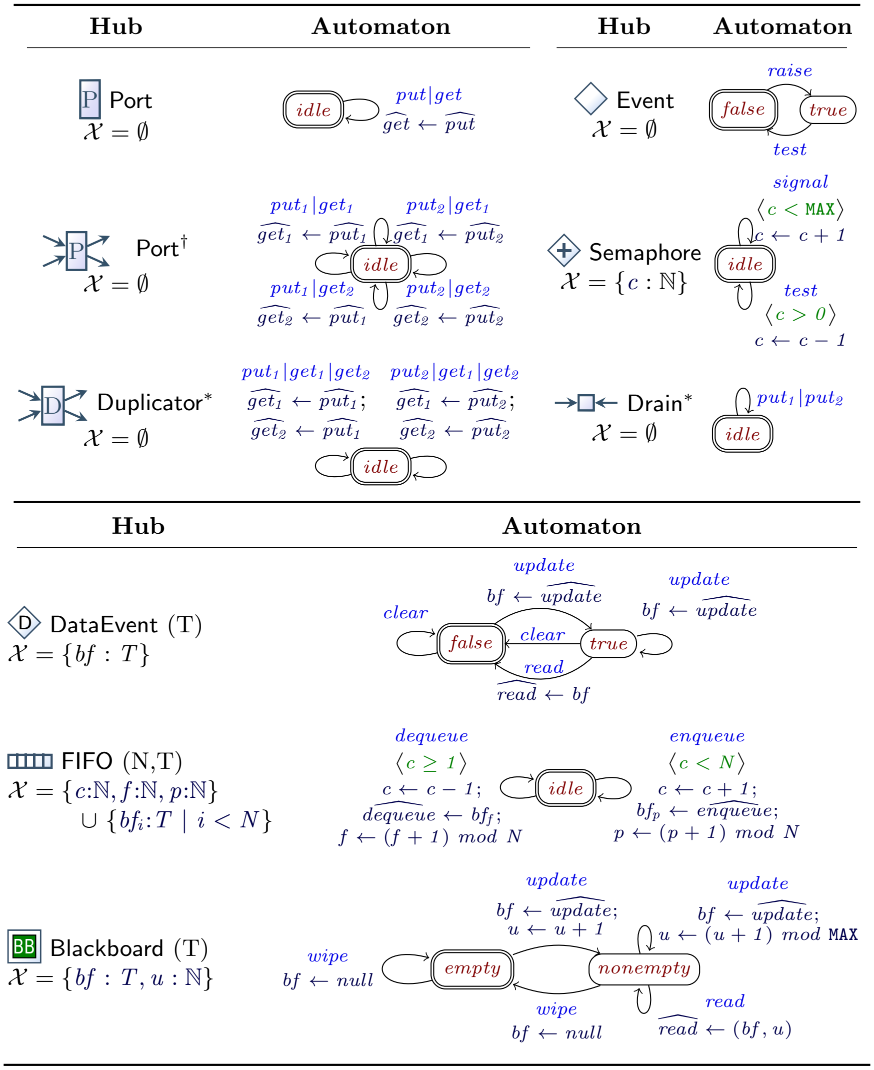

Hub
Automaton
Hub
Automaton
P Port
$\mathcal{X} = \emptyset$
idle
put|get
$\widehat{get} \leftarrow \widehat{put}$
Event
$\mathcal{X} = \emptyset$
false
true
raise
test
P Port†
$\mathcal{X} = \emptyset$
$put_1|get_1$
$\widehat{get_1} \leftarrow \widehat{put_1}$
$put_2|get_1$
$\widehat{get_1} \leftarrow \widehat{put_2}$
idle
$put_1|get_2$
$\widehat{get_2} \leftarrow \widehat{put_1}$
$put_2|get_2$
$\widehat{get_2} \leftarrow \widehat{put_2}$
Semaphore
$\mathcal{X} = \{c : \mathbb{N}\}$
signal
$\langle c < \mathtt{MAX} \rangle$
$c \leftarrow c + 1$
idle
test
$\langle c > 0 \rangle$
$c \leftarrow c - 1$
D Duplicator*
$\mathcal{X} = \emptyset$
$put_1|get_1|get_2$
$\widehat{get_1} \leftarrow \widehat{put_1}$;
$\widehat{get_2} \leftarrow \widehat{put_1}$
$put_2|get_1|get_2$
$\widehat{get_1} \leftarrow \widehat{put_2}$;
$\widehat{get_2} \leftarrow \widehat{put_2}$
idle
Drain*
$\mathcal{X} = \emptyset$
$put_1|put_2$
idle
Hub
Automaton
D DataEvent (T)
$\mathcal{X} = \{bf : T\}$
update
$bf \leftarrow \widehat{update}$
update
$bf \leftarrow \widehat{update}$
clear
false
clear
true
read
$\widehat{read} \leftarrow bf$
FIFO (N,T)
$\mathcal{X} = \{c{:}\mathbb{N}, f{:}\mathbb{N}, p{:}\mathbb{N}\}$
$\cup \{bf_i{:}T \mid i < N\}$
dequeue
$\langle c \geq 1 \rangle$
$c \leftarrow c - 1$;
$\widehat{dequeue} \leftarrow bf_f$;
$f \leftarrow (f + 1) \bmod N$
idle
enqueue
$\langle c < N \rangle$
$c \leftarrow c + 1$;
$bf_p \leftarrow \widehat{enqueue}$;
$p \leftarrow (p + 1) \bmod N$
BB Blackboard (T)
$\mathcal{X} = \{bf : T, u : \mathbb{N}\}$
update
$bf \leftarrow \widehat{update}$;
$u \leftarrow u + 1$
update
$bf \leftarrow \widehat{update}$;
$u \leftarrow (u + 1) \bmod \mathtt{MAX}$
wipe
$bf \leftarrow null$
empty
nonempty
wipe
$bf \leftarrow null$
read
$\widehat{read} \leftarrow (bf, u)$

Fig. 2: Automata semantics of hubs – from VirtuosoNext except those with *. Port† captures how VirtuosoNext interprets multiple calls to the same port.

of an automaton is defined as sequences of steps that do not violate the guards, and such that each step updates the current variable valuation according to the corresponding update.

Definition 6 (Semantics of Hubs). *The semantics of a Hub Automaton $H = (L, \ell_0, P, \mathcal{X}, v_0, \rightarrow)$ is given by the rule below, starting on configuration (ℓ_0, v_0).*

$$\frac{\ell \xrightarrow{g,p,u} \ell' \quad v \models g \quad v' = v[u]}{(\ell, v) \xrightarrow{p} (\ell', v')} \qquad (\mathsf{seq})$$

For example, the following is a valid trace of a Fifo hub with size 3 (Fig. 2).

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

4 Reconstructing Hubs

Two hubs can be composed to form a more complex one, following the same ideas as in Reo [1]. The composition is done on top of two simpler operations: *product* and *synchronisation*. This section starts by defining these two operations, followed by an example and by a suitable definition of serialisation of updates.

4.1 Hub composition

The *product* takes two hubs with disjoint ports and variables, and produces a new hub where they behave in an interleaving or synchronous fashion, i.e. fully concurrent. The *synchronisation* operation is conducted over a Hub Automaton H and it links two ports a and b in P such that they can only operate in a synchronous manner.

Definition 7 (Product of Hub Automata). *Let H_1 and H_2 be two Hub Automata with disjoint sets of ports and variables. The product of H_1 and H_2, written $H_1 \times H_2$, is a new Hub Automaton defined as*

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

where $\rightarrow$ is defined as follows:

$$\frac{l_1 \xrightarrow{g_1,\omega_1,u_1} l'_1}{(l_1, l_2) \xrightarrow{g_1,\omega_1,u_1} (l'_1, l_2)} \qquad \frac{l_2 \xrightarrow{g_2,\omega_2,u_2} l'_2}{(l_1, l_2) \xrightarrow{g_2,\omega_2,u_2} (l_1, l'_2)} \qquad \frac{l_1 \xrightarrow{g_1,\omega_1,u_1} l'_1 \qquad l_2 \xrightarrow{g_2,\omega_2,u_2} l'_2}{(l_1, l_2) \xrightarrow{g_1 \wedge g_2\ ,\ \omega_1 \cup \omega_2\ ,\ u_1 | u_2} (l'_1, l'_2)}$$

Definition 8 (Synchronisation of Hub Automata). *Let H be a Hub Automaton, a and b two ports in P, and x_{ab} a fresh variable. The synchronisation of a and b is given by $\Delta_{a,b}(H)$, defined below.*

$$\begin{array}{rcl}
\Delta_{a,b}(H) & = & (L, l_0, (P \backslash \{a, b\}), \mathcal{X} \cup \{x_{ab}\}, v_0, \rightarrow') \\
\rightarrow' & = & \{l \xrightarrow{g,\omega,u} l' \mid a \notin \omega \text{ and } b \notin \omega\} \cup \\
 & & \{l \xrightarrow{g',\omega',u'} l' \mid l \xrightarrow{g,\omega,u} l', a \in \omega, b \in \omega, \omega' = \omega \backslash \{a, b\}, \\
 & & \qquad\qquad g' = g[x_{ab}/\hat{a}][x_{ab}/\hat{b}],\ u' = u[x_{ab}/\hat{a}][x_{ab}/\hat{b}]\}
\end{array}$$

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

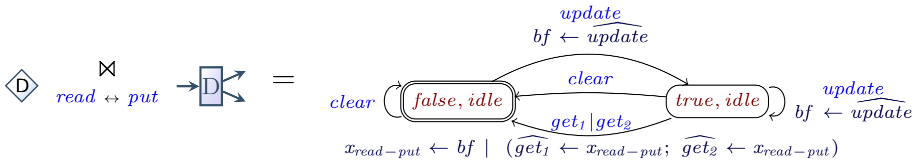

D
⋈
read ↔ *put*
D
=
$\widehat{update}$
$bf \leftarrow \widehat{update}$
clear
false, idle
clear
true, idle
get₁|get₂
$\widehat{update}$
$bf \leftarrow \widehat{update}$
$x_{read-put} \leftarrow bf \mid (\widehat{get_1} \leftarrow x_{read-put};\ \widehat{get_2} \leftarrow x_{read-put})$

Fig. 3: Example of composition between two Hub Automata, where a DataEvent automaton is composed with a Duplicator automaton by synchronising on actions *read* and *put* (left), resulting in the composed automaton on the right.

The composition of two Hub Automata consists of their product followed by the synchronisation of their shared ports.

Definition 9 (Composition of Hub Automata). *Let H_1 and H_2 be two Hub Automata with disjoint sets of ports and variables, and let $\{(a_0, b_0), \ldots, (a_n, b_n)\}$ be a finite (possibly empty) set of ports bindings, such that for each pair (a_i, b_i) for $0 \leq i \leq n$ we have that $(a_i, b_i) \in P_{I_{H_1}} \times P_{O_{H_2}}$ or $(a_i, b_i) \in P_{O_{H_1}} \times P_{I_{H_2}}$. The composition of H_1 and H_2 over such a set is defined as follows.*

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

Intuitively, composing two automata means putting them in parallel ($\times$), and then restrict their behaviour by forcing shared ports to go together (Δ). The first step joins concurrent transition into new transitions, placing updates in parallel. This emphasises the need for a serialisation process that guarantees a correct evaluation order of values to data in ports, which is the focus of Section 4.3.

Fig. 3 shows the composition of two Hub Automaton: a DataEvent, and a Duplicator with two output points. The composed automaton (right) illustrates the behaviour of the two hubs when synchronised over the actions *read* and *put*: whenever a data event is raised and the buffer updated, the hub can be tested simultaneously by two tasks through *get₁* and *get₂*. Both tasks will receive the stored data in the DataEvent Hub, before setting the event to false. Synchronised ports are removed from the composed model, and variables associated to such ports are renamed accordingly, i.e. $\widehat{read}$ and $\widehat{put}$, are both renamed to $x_{read\text{-}put}$.

4.2 Example: Round Robin tasks

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

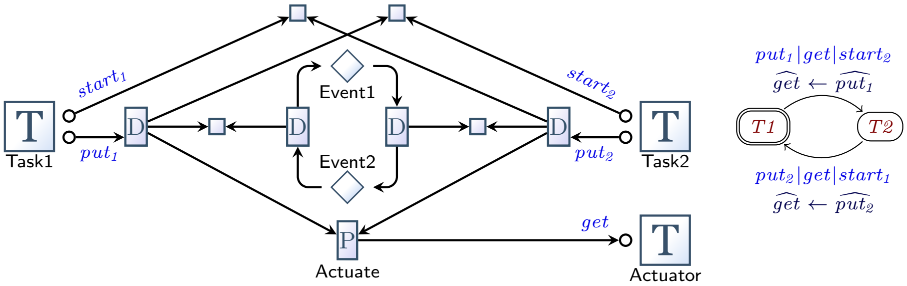


Fig. 4: Alternative architecture for the example in Fig. 1 – Reo connector (left) and its Hub Automaton (right) after updates have been serialised and simplified.

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

We propose an alternative hub that exploits shared ports (Fig. 4), built by composing a set of primitives from Fig. 2, which further restricts the behaviour of the coordinator. More specifically, when a task sends a data value to the actuator, the coordinator interprets it as the end of its round. Furthermore, it requires each task to send only when the other is ready to start – a different behaviour could be implemented to buffer the end of a task round (as in Fig. 1).

4.3 Serialisation of Updates

Recall that the application of an update u (Definition 5) requires a serialisation function σ that converts an update with parallel constructs into a sequence of assignments. This subsection proposes a serialisation algorithm that preserves dependencies between variables, and rejects updates that have variables with circular dependencies. It uses an intermediate dependency graph that is traversed based on Kahn's algorithm [9], and later discards intermediate assignments.

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

Building Dependency Graphs A dependency graph is a directed graph $D = (N, L)$, where N is a set of nodes, each representing an update of the form $x \leftarrow e$, and $L \subseteq N \times N$ is a set of links between nodes, where a link (n, m) indicates that n must appear before m in a sequence of assignments. Given D_1 and D_2, their composition, $D_1 \bowtie D_2 = (N_1 \cup N_2, L_1 \cup L_2)$ is a new dependency graph.

```
Algorithm 1: Dependency Graph for an update $u$
    input  : An update $u$ with parallel options $u_i$ for $i = 1..n$
    output: A Dependency Graph for $u$
graphs $\leftarrow \bigcup_{i=1}^{n} \texttt{struct}(u_i)$;
toVisit $\leftarrow$ graphs;
foreach $g \in$ graphs do
    toVisit $\leftarrow$ toVisit $\setminus \{g\}$;
    newLinks $\leftarrow$ newLinks $\cup \{\texttt{links}(n, m) \mid n \in N_g, m \in \bigcup_{v \in \text{toVisit}} N_v\}$;
newNodes $\leftarrow$ the set of all nodes from all $g \in$ graphs;
newLinks $\leftarrow$ newLinks $\cup$ the set of all links from all $g \in$ graphs;
return A dependency graph with newNodes and newLinks;
```

Given a dependency graph $D = (N, L)$, we say a node n is a *leaf* ($\texttt{leaf}_L(n)$) if $\nexists_{(m,o)\in L} \cdot o = n$, or a a *root* ($\texttt{root}_L(n)$) if $\nexists_{(o,m)\in L} \cdot o = m$. We first define $\texttt{struct}(u)$ recursively to consider dependencies between assignments imposed by the structure of the update (i.e., imposed by ; and |), defined as follows.

$$\frac{\texttt{struct}(x \leftarrow e)}{(\{x \leftarrow e\}, \{\})} \qquad \frac{\texttt{struct}(u_1 \mid u_2)}{\texttt{struct}(u_1) \bowtie \texttt{struct}(u_2)}$$

$$\frac{\texttt{struct}(u_1 \ ; \ u_2),\ \texttt{struct}(u_1) = (N_1, L_1),\ \texttt{struct}(u_2) = (N_2, L_2)}{(N_1 \cup N_2, L_1 \cup L_2 \cup \{(n, m) \mid n \in N_1, \texttt{leaf}_{L_1}(n), m \in N_2, \texttt{root}_{L_2}(m)\})}$$

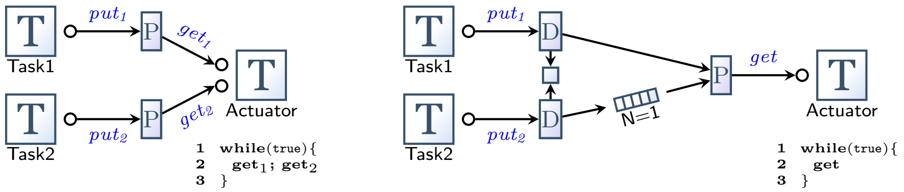
Secondly, we create dependency links between nodes of different subgraphs of u (generated by |) based on their dependency on variables. These links between two nodes n and m, noted as $\texttt{links}(n, m)$, are created as follows: from n to m if m depends on a variable produced by n; from m to n if n depends on a variable produced by m; and both ways if both conditions apply.

The complete algorithm to build a dependency graph is given in Algorithm 1. If the graph is not acyclic, Kahn's algorithm will return a topological order.

Simplification of Updates This step considers all transitions of the automaton to find and remove unnecessary and intermediate assignments. We consider *unnecessary assignments*: assignments to internal variables that are never used on the right-hand side (RHS) of an assignment nor on a guard; and assignments that depend only of internal variables that are never assigned (undefined variables). We consider *intermediate assignments*, assignments to internal variables that are followed by (eventually in the same sequence) an assignment where the variable is used on the RHS, and such that the variable is never used on guards.

5 Evaluation

We compare the two architectures from Section 4.2, using a variation of these, and provide both an analytical comparison, using different metrics, and a performance comparison, executing them in an embedded board.



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

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

Fig. 5: Architectural view of scenarios S$_{\textit{2-ports}}$ (left) and S$_{altern}$ (right).

Scenarios We compare four different scenarios in our evaluation, using the architectures from Section 4.2, and compile and execute them on a TI Launchpad EK-TM4C1294XL[6] board with a 120MHz 32-bit ARM Cortex-M4 CPU.
- **S$_{\boldsymbol{orig}}$** the initial architecture as in Fig. 1;
- **S$_{\boldsymbol{custom}}$** using a custom-made hub that follows the automaton in Fig. 4 without any data transfer;
- **S$_{\boldsymbol{altern}}$** using a custom-made hub that acts as S$_{custom}$, but discarding the `start` queues, and assuming that tasks start as soon as possible (Fig. 5 right); and
- **S$_{\boldsymbol{2\text{-}ports}}$** simple architecture with two ports, each connecting a task to the actuator, also discarding the `start` queue, whereas the actuator is responsible to impose the alternating behaviour (Fig. 5 left).

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

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

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

Note that conceptually the original architecture further requires the tasks to be well behaved, in the sense that a task should not signal/test a semaphore more times than the other task tests/signals it. In the refined architecture functionality is better encapsulated: tasks abstract from implementing coordination behaviour and focus only on sending data to the actuator, while the coordinator handles the order in which tasks are enabled to send the data. This contributes to a

[6] http://www.ti.com/tool/EK-TM4C1294XL#

Table 2: Possible sequence of context switches between the Kernel task (executing the hubs) and the user tasks for each scenario.

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

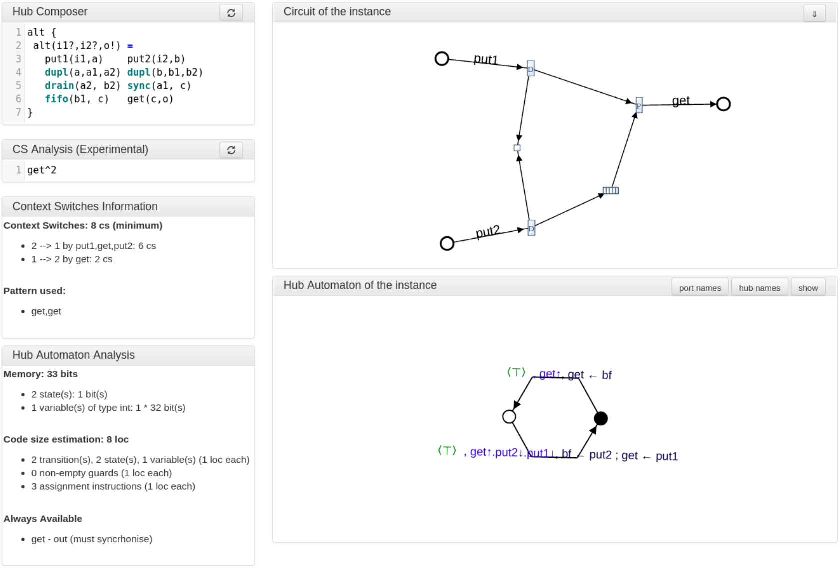
better understanding of the behaviour of both the tasks and the coordination mechanism. In addition, by knowing the semantics of each hub and by looking at the architecture in Fig. 1 is not enough to determine the behaviour of the composed architecture, but it requires to look at the implementation of the tasks to get a better understanding of what happens. However, in Fig. 4 these two premises are sufficient to understand the composed behaviour.

Measuring execution times on the target processor We compiled, executed, and measured the execution of 4 systems: (1) S_{orig}, (2) a variation of S_{custom} implemented as a dedicated task, which we call $\mathit{Task}[S_{custom}]$, (3) a variation of S_{custom} that abstracts away from the actual instructions (implemented as a native hub, which we call $\mathit{NoData}[S_{custom}]$), and (4) $\mathrm{S}_{\mathit{2\text{-}ports}}$. The results of executing 1000 rounds using our TI Launchpad board are presented below, whereas the end of each round consists of the actuator receiving an instruction from both tasks (i.e., 500 values from each task).

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

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

Online analysis tools We implemented a prototype that composes, simplifies, and analyses Hub Automata, available online,[7] and depicted in Fig. 6. The generated automata can be used to produce either new hubs or dedicated tasks that perform coordination. These generated automata can also be formally analysed to provide key insight information regarding the usefulness and drawbacks of such a hub. Our current implementation allows specifications of composed hubs using a textual representation based on ReoLive [3,14], and produces (1) the architectural view of the hub, (2) the simplified automaton of the hub, and (3) a summary of some properties of the automaton, such as required memory, size estimation of the code, information about which hubs' ports are always ready to synchronise, and minimum number of context switches for a given trace.


Hub Composer
CS Analysis (Experimental)
Context Switches Information
Hub Automaton Analysis
Circuit of the instance
put1
put2
get
Hub Automaton of the instance
port names
hub names
show
⟨⊤⟩ , get↑, get ← bf
⟨⊤⟩ , get↑.put2↓.put1↓, bf ← put2 ; get ← put1

```
alt {
 alt(i1?,i2?,o!) =
   put1(i1,a)    put2(i2,b)
   dupl(a,a1,a2) dupl(b,b1,b2)
   drain(a2, b2) sync(a1, c)
   fifo(b1, c)   get(c,o)
}
```

```
get^2
```

Context Switches: 8 cs (minimum)

- 2 --> 1 by put1,get,put2: 6 cs
- 1 --> 2 by get: 2 cs

Pattern used:

- get,get

Memory: 33 bits

- 2 state(s): 1 bit(s)
- 1 variable(s) of type int: 1 * 32 bit(s)

Code size estimation: 8 loc

- 2 transition(s), 2 state(s), 1 variable(s) (1 loc each)
- 0 non-empty guards (1 loc each)
- 3 assignment instructions (1 loc each)

Always Available

- get - out (must syncrhonise)

Fig. 6: Screenshot of the online analyser for VirtuosoNext's hubs.

[7] http://github.com/arcalab/hubAutomata

6 Related work

The global architecture of VirtuosoNext RTOS, including the interaction with hubs, has been formally analysed using TLA+ by Verhulst et al. [16]. More concretely, the authors specify a set of concrete hubs, their waiting lists, and the priority of requests, and use the TLC model checker to verify a set of safety properties over these. Our approach uses a formalism focused on the interactions, abstracting away waiting lists, and aims at the analysis and code generation of more complex hubs built compositionally.

The automata model proposed here is mainly inspired by Reo's paramiterised constraint automata [1] and constraint automata with memory cells [8], both used to reason about data-dependent coordination mechanism. In the former states can store variables which are updated or initialised when transiting, while the latter treats variables as first-class objects, as in here, allowing to efficiently deal with infinite data domains. Both approaches use data constraints as a way to assign values to ports, and define updates as a way to modify internal variables. Here, we treat both variables more uniformly, requiring a serialization method, and postponing it until obtaining the final composed automaton.

The composition and the restrictions imposed here on the input and output ports are similar to those introduced by Interface Automata [4] to deal with the composition of open systems. However, [4] imposes additional restrictions to ensure automata compatibility, i.e. whenever an automaton is ready to send an output, which is an input of the other, the latter should be able to receive it.

Finite-memory automata [10] and nominal automata [12,15] are models that deal with infinite alphabets, focusing on the expressiveness of their variants and on the decidability of some of their properties, which is not the goal of this paper. Finite-memory automata uses substitution instead of equality tests over the input alphabet with the support of a finite set of registers (variables) associated to the automata, and nominal automata are based on nominal sets, which can be seen as infinite sets with a finite representation.

Formal analysis of RTOS are more typically focused on the scheduler, which is not the focus of this work. For example, theorem provers have been used to analyse schedulers for avionics software [7]. Carnevali et al. [2] use preemptive Time Petri Nets to support exact scheduling analysis and guide the development of tasks with non-deterministic execution times in an RTOS with hierarchical scheduling. Dietrich et al. [5] analyse and model check all possible execution paths of a real-time system to tailor the kernel to particular application scenarios, resulting in optimisations in execution speed and robustness. Dokter et al. [6] propose a framework to synthesise optimised schedulers that consider delays introduced by interaction between tasks. Scheduling is interpreted as a game that requires minimising the time between subsequent context switches.

7 Conclusions

This paper proposes an approach to build and analyse hubs in VirtuosoNext, which are services used to orchestrate interacting tasks in a Real Time OS that runs on embedded devices. In VirtuosoNext, complex coordination mechanisms are the responsibility of the programmer, who can use a set of fundamental hubs to coordinate tasks, but have to implement more complex interaction mechanisms as application specific code, deteriorating readability and maintainability.

Our proposed formal framework provides mechanisms to design and implement complex hubs that can be formally analysed to provide the same level of assurance that predefined hubs provide. Currently, the framework allows to build complex hubs out of simpler ones, and analyse some aspects of the hubs such as: memory used, estimated lines of codes, and always available ports.

Preliminary tests on a typical set of scenarios have confirmed our hypothesis that using dedicated hubs to perform custom coordination can result in performance improvements. In addition, we claim that moving coordination aspects away from tasks enables a better understanding of the tasks and hubs behaviour, and provides better visual feedback regarding the semantics of the system.

Ongoing work to extend our formal framework includes: **runtime behaviour analysis**, by taking into account the time-sensitive requests made to hubs and some contracts that tasks are expected to obey; **variability support** to analyse and improve the development of families of systems in VirtuosoNext, since VirtuosoNext provides a simple and error-prone mechanism to allow topologies to be applied to the same set of tasks; **code refactoring and generation** applied to existing (on-production) VirtuosoNext programs, probably adding new primitive hubs, by extracting the coordination logic from tasks and into new complex hubs; and **analysis extension** to support a wider range of analysis to Hub Automata, such as the number of context switches required to perform certain behaviour, or the model checking of liveness and safety properties using mCRL2 (c.f. [3,11]).

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, within the CISTER Research Unit (UID/CEC/04234); also 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).

References

1. Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
2. Laura Carnevali, Giuseppe Lipari, Alessandro Pinzuti, and Enrico Vicario. A formal approach to design and verification of two-level hierarchical scheduling sys-

tems. In Alexander Romanovsky and Tullio Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, pages 118–131, Berlin, Heidelberg, 2011. Springer.

3. Rúben Cruz and José Proença. Reolive: Analysing connectors in your browser. In Manuel Mazzara, Iulian Ober, and Gwen Salaün, editors, *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, volume 11176 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2018.
4. Luca de Alfaro and Thomas A. Henzinger. *Interface-Based Design*, pages 83–104. Springer Netherlands, Dordrecht, 2005.
5. Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by rtos-aware control-flow analysis. *ACM Trans. Embed. Comput. Syst.*, 16(2):35:1–35:25, January 2017.
6. Kasper Dokter, Sung-Shik Jongmans, and Farhad Arbab. Scheduling games for concurrent systems. In Alberto Lluch Lafuente and José Proença, editors, *Coordination Models and Languages*, pages 84–100, Cham, 2016. Springer.
7. Vu Ha, Murali Rangarajan, Darren Cofer, Harald Rues, and Bruno Dutertre. Feature-based decomposition of inductive proofs applied to real-time avionics software: An experience report. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 304–313, Washington, DC, USA, 2004. IEEE Computer Society.
8. S.-S.T.Q. Jongmans, T. Kappé, and F. Arbab. Constraint automata with memory cells and their composition. *Science of Computer Programming*, 146:50 – 86, 2017. Special issue with extended selected papers from FACS 2015.
9. A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.
10. Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, November 1994.
11. Natallia Kokash, Christian Krause, and Erik P. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *FAC*, 24(2):187–216, 2012.
12. Alexander Kurz, Tomoyuki Suzuki, and Emilio Tuosto. On Nominal Regular Languages with Binders. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures*, pages 255–269, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
13. Altreonic NV. *OpenComRTOS-Suite Manual and API Manual (1.4.3.3)*. `http://www.altreonic.com/sites/default/files/OpenComRTOS_API-Manual.pdf`.
14. José Proença and Alexandre Madeira. Taming hierarchical connectors. In *Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, 2019*, Lecture Notes in Computer Science (to appear), 2019.
15. Lutz Schröder, Dexter Kozen, Stefan Milius, and Thorsten Wißmann. Nominal Automata with Name Binding. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 124–142, 2017.
16. Eric Verhulst, Raymond T Boute, José Miguel Sampaio Faria, Bernhard HC Sputh, and Vitaliy Mezhuyev. *Formal Development of a Network-Centric RTOS: software engineering for reliable embedded systems.* Springer Science & Business Media, 2011.