

---

# **Hubs for VirtuosoNext™**

**Guillermina Cledou, José Proença**

**Jul 29, 2021**



# GETTING STARTED

<b>1</b>	<b>Local installation</b>	<b>3</b>
1.1	Requirements . . . . .	3
1.2	Installation steps . . . . .	3
1.3	Running the framework . . . . .	4
<b>2</b>	<b>Using the tools</b>	<b>5</b>
2.1	Server vs Lightweight mode . . . . .	5
2.2	Widgets . . . . .	5
<b>3</b>	<b>Publications</b>	<b>19</b>
<b>4</b>	<b>Support</b>	<b>21</b>
<b>5</b>	<b>VirtuosoNext™ RTOS</b>	<b>23</b>
<b>6</b>	<b>How to use the tools</b>	<b>25</b>



Hubs for VirtuosoNext™ is a web-based toolset to build, compose, depict, and analyse **Timed Hub Automata**. These automata give semantics to *Hubs*, interacting entities on the Real-Time operating system VirtuosoNext™ developed by [Altreonic](#).

The toolset is developed in Scala, and uses ScalaJS to generate JavaScript. The toolset is developed as a sub-module of [ReoLive](#).



## LOCAL INSTALLATION

VirtuosoNext toolset is developed in Scala, and uses ScalaJS to generate JavaScript. The toolset is developed as a sub-module of [ReoLive](#), and as such it requires ReoLive to run.

In addition it uses `verifyta` a command line tool from the [Uppaal](#) Real-Time Model Checker to verify properties of Timed Hub Automata.

Before installing ReoLive and VirtuosoNext toolset see the full list of requirements below.

### 1.1 Requirements

- Scala building tools ([SBT](#))
- [Uppaal](#) Real-Time Model Checker (optional)
- Java Runtime Environment ([JRE](#))

### 1.2 Installation steps

Clone the [ReoLive](#) repository

```
$ git clone git@github.com:ReoLanguage/ReoLive.git
$ cd ReoLive
```

Pull the git submodules (which will include VirtuosoNext):

```
$ git submodule update --init
```

Use your favourite editor to edit the path to Uppaal executables in the `global.properties` configuration file. For example, in Mac OS the typical path would be:

```
...
verifytaCmd = /Applications/uppaal/bin-Darwin/verifyta
...
```

Run the compilation script:

```
$ ./compile.sh
```

### 1.3 Running the framework

After running the script `compile.sh` you can already access to the *lightweight* version of VirtuosoNext by opening

```
$ open site/hubs.html
```

and accessing the *LW Hubs* tab.

This lightweight version is a single javascript file autogenerated during compilation, and gives access to most of the tools.

However, if you want to access the verification tool for Timed Hubs Automata, it requires access to your Uppaal installation to run the models in Uppaal.

For this you need to run the server and access the *VirtuosoNext* tab, as follows.

Start the server using `sbt`

```
$ sbt server/run
```

Open the VirtuosoNext toolset in a browser

```
$ open http://localhost:9000/hubs
```



## USING THE TOOLS

### 2.1 Server vs Lightweight mode



Fig. 1: Server (Full Hubs) vs Lightweight mode (LW Hubs)

While the toolset is developed in Scala, the code is compiled both into *JVM* binaries that are executed on a **server** (*Full Hubs*), and into **JavaScript** using *ScalaJS* to produce an interactive web page (*LW Hubs*).

Both versions provide *almost* the same functionality, with the server additionally supporting the live verification of properties through the Uppaal model checker. Everything else apart from the live verification is computed by the browser using the generated JavaScript libraries.

### 2.2 Widgets

The toolset is organized in a set of widgets, each of them providing some functionality: edition, visualization, or analysis.

By clicking in a widgets' name it is possible to *open* or *close* such a widget. By default only some widgets are open.

All widgets, except the *Examples* widget use the hub specified in the *Hub composer* to carried on with their functionality.

Below we explain each widget in detail.

#### 2.2.1 Hub Composer

It is **the editor** where users can specify *hubs* and *tasks* with different interaction semantics.

Hubs are specified by **composing** predefined hubs. We provide the list of primitive hubs below, followed by an explanation on how to composed them.

Tasks are defined as a sequence of input or output ports, each of which can connect to the environment following the sequence order, and using a specified interaction semantics. This is explained below in further detail.

---

#### Load the hub

```

1 mainW // try different scenarios
2 {
3     dupl3(a?,b!,c!,d!) =
4     dupl(a,b,ab) dupl(ab,c,d)
5     ,
6     seq(s1?,p1?,s2?,p2?,get!) =
7     dupl3(p1,d11,d12,d13)
8     dupl3(p2,d21,d22,d23)
9     drain(s1,d21) drain(s2,d11)
10    drain(d12,d42) drain(d22,d32)
11    dupl(e1,d41,d42) dupl(e2,d32,d31)
12    eventFull(d31,e1) event(d41,e2)
13    merger(d13,d23,get)
14    ,
15    // Scenarios
16    mainW() = // waits (possibly for ever)
17    task<t1>(W p1!,W s1!) // 'put1' goes first
18    task<t2>(W s2!,W p2!) // 'start2' goes first
19    task<act>(W get?)
20    seq(s1,p1,s2,p2,get)
21 }

```

Fig. 2: Hub composer - Example code

Whenever a hub is specified, it is required to **load the hub** so that other widgets can analysed or visualised such a hub. The hub can be loaded either by pressing shift + enter in the *Hub Composer*, or by clicking in the update icon on the top right of the widget.

## Primitive Hubs

Usage of predefined hubs is illustrated in the following tables. They are separated into original hubs in VirtuosoNext™ and newly proposed hubs.

Original Hubs from VirtuosoNext™

Hub	Usage
Port	port (in, out)
Event	event (in, out)
DataEvent	dataEvent (in, out)
Semaphore	semaphore (in, out)
Resource	resource (in1, in2)
FIFO	fifo (in, out)
BlackBoard	blackboard (in, out)

Newly proposed hubs

Hub	Usage
Drain	drain (in1, in2)
Merger	merger (in1, in2, out)
Exclusive Router	xor (in, out1, out2)
Duplicator	dupl (in, out1, out2)
EventFull	eventFull (in, out)
DataEventFull	dataEventFull (in, out)
FIFOFull	fifoFull (in, out)
BlackBoardFull	blackboardFull (in, out)
Timer	timer (in, out) or timer<n> (in, out) ( <i>n</i> a positive integer, 0 when omitted)

## Tasks

We can model tasks by using a predefined construct defined by the following grammar:

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

A task tries to communicate with the environment through its IO ports in the order established by the declaration and following the specified interaction semantics.

These interaction semantics determine how a task waits on a request to succeed. These can be:

- waiting (W) – a task waits indefinitely until the request can be served
- non-waiting (NW) – either the requests is served without delay or the request fails
- waiting with time-out (WT) – waits either until the request is served or the specified time-out has expired.

## Examples

The following code specifies a task named T1, with an input port a and an output port b.

```
task<T1>(W a?, 4 b!)
```

T1 first tries to read from the environment on port a waiting until it succeeds (W). When it succeeds, it tries to send data through b, but it waits only 4 units of time, after this time whether it succeeds, it starts again, trying to read in a. This semantics is given by the following THA.

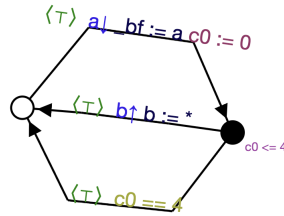


Fig. 3: T1 semantics

Similarly, the following code specifies a task named T2, with an output port c. The task **periodically** tries to send data through c every 5 units of time.

```
task<T2>(NW c!) every 5
```

Informally, the tasks tries to send data through c without waiting (NW). Whether it succeeds, it will wait 5 units of time before starting again and trying to send data again. Formally, this semantics is given by the following THA.

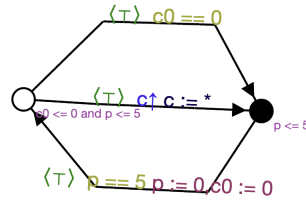


Fig. 4: T2 semantics

## Composition

### Preo syntax

Composition using the **Preo** syntax is defined in a pointfree style, i.e., without naming the ports. For example, the `dupl(in,out1,out2)` hub, is specified simply as `dupl`, and the `timer<n>(in,out)` is specified as `timer(n)` or just `timer` for default values.

Composition of hubs and tasks can be sequential `;` (outputs to inputs) or parallel `*` (appending inputs and outputs). A type system guarantees that composition is correct.

The sequential composition requires that the number of outputs match the number of inputs in the sequence.

```
dupl ; fifo * event
```

This code specifies a duplicator hub where the first output connects to the input of a `fifo` hub, and the second output connects to the input of an `event` hub.

More complex examples are available in the [Examples](#) widget [online](#).

Preo syntax is extended as well with integers and booleans expression that can simplify the definition of complex hubs.

- `hub ^n` :  $n$  hubs of type `hub`,  $n$  a positive integer
- `hub !` : as many `hub` such that their inputs and outputs connect correctly with another hubs that may connect in sequence with `hub`

```
// for fifo hubs in parallel, composed in sequence with as many merger hubs needed (2,
↪in this case).
fifo^4 ; merger!
```

**Note:** Checkout [Typed Connector Families and Their Semantics](#) to read the theory behind Preo.

### Treo syntax

In the **Treo** syntax hubs are specified by explicitly naming their port.

Furthermore, a new hub needs to be declare in a function like manner, by specifying their ports as parameters and declaring whether each parameter is an input port `?` or an output port `!`.

Composition is specified by declaring two hubs separated with spaces. Composed hubs with shared port names will synchronize over such ports.

```
// Main block (Preo Syntax)
// uses the hub myDupl specified in the code block
myDupl

{
    // a hub using Treo syntax
    myDupl(in?,out1!,out2!) =
        dupl(in,o1,o2)
        fifo(o1,out1)
        event(o2,out2)
    ,

    // an equivalent hub to myDupl, declared using Preo syntax
    otherDupl = dupl ; fifo * event
}
```

More complex examples are available in the [Examples](#) widget [online](#).

## Specifying Hubs

The main Hub is specified following the *Preo syntax*.

It is possible to declare various hubs, using the *Preo syntax* or *Treo syntax*, by declaring them in a function like manner inside a block `{ / }` and referencing their names. Various hubs specified inside the block are separated by `, .`

```
// Main block (Preo Syntax)
// uses the hub myDupl specified in the code block
timer(5) ; myDupl

{
    // a hub using Treo syntax
    myDupl(in?,out1!,out2!) =
        dupl(in,o1,o2)
        fifo(o1,out1)
        event(o2,out2)
    ,

    // an equivalent hub to myDupl, declared using Preo syntax
    otherDupl = dupl ; fifo * event
    ,

    // yet another hub equivalent to myDupl
    yetAnotherDupl(i?, o1!, o2!) = otherDupl(i,o1,o2)
}
```

## 2.2.2 Circuit of the instance

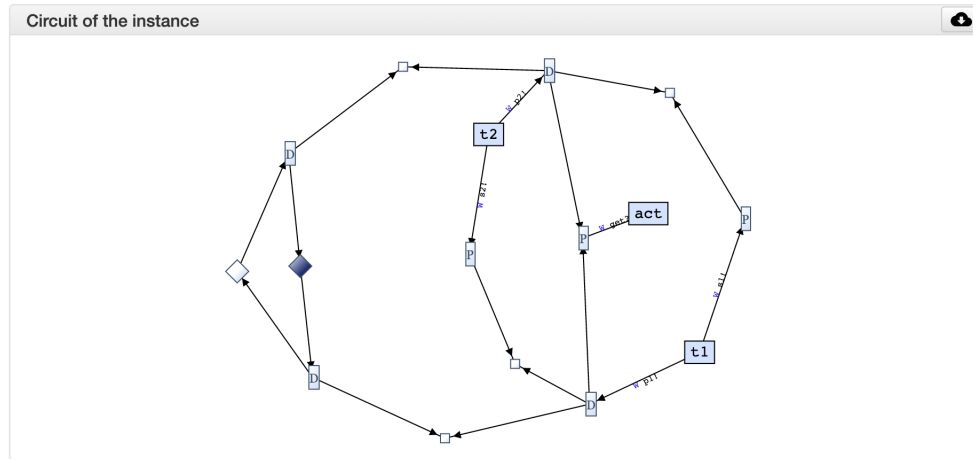


Fig. 5: Hub circuit - Two task,  $t1$  and  $t2$ , write in sequence to another task  $act$

This widget shows the architectural view of the hub specified in the *Hub Composer*, i.e. how primitive hubs and tasks are connected to form a more complex hub.

**Blue boxes** with names represent *tasks*; **white circles**, if any, represent free *input/output ports*, i.e. ports that haven't been connected yet; and **the rest of the nodes** represent *primitive hubs*.

**Arrows** represent *connections* from output to input ports. Incoming and outgoing arrows from tasks are labeled with the corresponding interaction semantics (W, NW, n - n a positive integer), the port's name (only when using the **Treo** syntax), and the type of port (input or output).

## 2.2.3 Hub Automaton of the instance

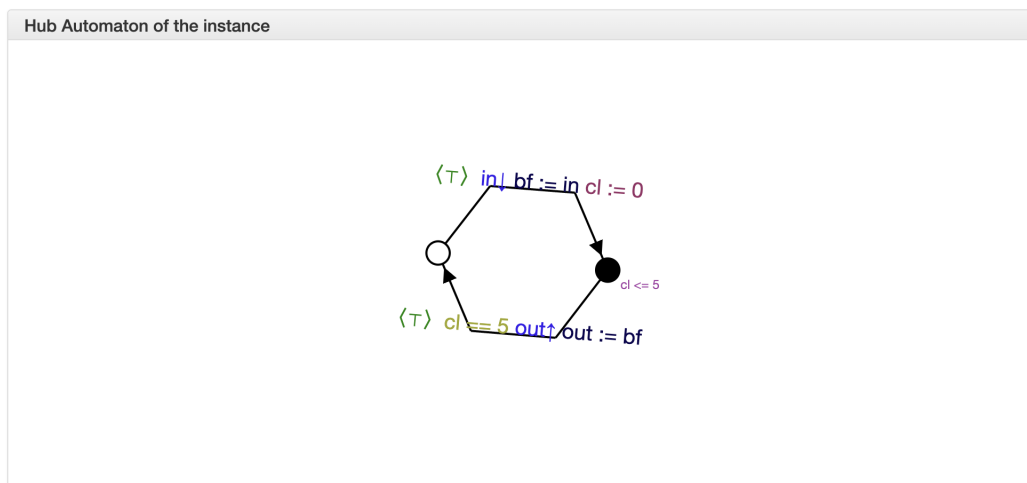


Fig. 6: (Timed) Hub Automaton - Example automaton for a hub timer (5)

This widget shows the simplified and serialized automaton of the hub specified in the *Hub Composer*.

A **white circled location** represents the initial state. All locations have a **clock invariant**, represented by a purple label next to the location node, e.g.  $cl \leq 5$  (right location). Locations that do not show any clock invariant are locations

with trivially satisfied invariants, namely  $\top$ .

Transitions are labeled as followed:

- **guard constraint**, represented by a green label within angle brackets, e.g.  $\langle \top \rangle$
- **clock constraint**, if any, represented by a yellow label following the guard constraint, e.g.  $cl == 5$  (bottom transition)
- **synchronizing ports**, represented by blue labels, e.g.  $in \downarrow$ , where  $\downarrow$  represents an input port, and  $\uparrow$  an output port
- **updates**, if any, represented by a dark blue label, e.g.  $bf := in$  (top transition)
- **clock updates**, if any, represented by a purple label, e.g.  $cl := 0$  (top transition)

**Warning:** There is a known issue where labels of the automaton are not visualised in the current version of Firefox.

## 2.2.4 Examples



Fig. 7: Examples - A set of example hubs written in *Preo* and *Treo* syntax

This widget provides a set of example hubs, from primitive (e.g. *Port* and *Port - 2 sources*) to more complex ones (e.g. *Alternator* and *Sequencer*).

Some examples are written in *Preo* syntax, such as *Alternator (no variables)*, and others in *Treo* syntax, such as *Alternator*.

By clicking on one of the examples, the corresponding code will be loaded in the *Hub Composer* and it will trigger the update of other widgets that are opened.

## 2.2.5 Context Switch Analysis

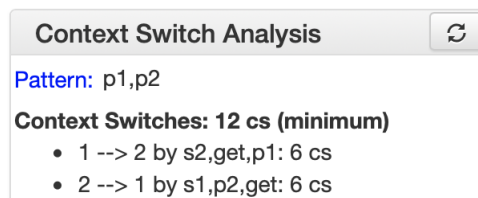


Fig. 8: Context Switch Analysis - Minimum number of context switches for the trace  $p_1, p_2$  from the hub example from *Circuit of the instance*

This widget is an interactive panel to estimate the minimum number of context switches that a given trace in the current hub will have if implemented in VirtuosoNext™.

A trace is a sequence of ports executions. In the example, the trace  $p_1, p_2$  captures any trace in which  $p_1$  executes, followed by the execution of port  $p_2$ . In both cases,  $p_1$  and  $p_2$  could execute synchronously with other ports.

It is possible to express  $n$ -sequential executions of the same port  $p$  as  $p^n$ . For example  $p^3$ , instead of  $p, p, p$ .

The trace can be specified in the text box next to the *Pattern*:. After which, it is required to load the trace by either pressing `shift + enter` or clicking on the load icon on the top right of the box.

The widget will present the analysis below by stating the minimum number of context switches required, showing the transitions that follow such a trace and the number of context switches per transition.

In the example, the trace  $p_1, p_2$  requires in the best case *12 CS*. Starting from the initial state *1* it transitions to state *2* by executing synchronously ports  $s_2$ ,  $get$ , and  $p_1$ . Context switches occur when the execution changes from the **Kernel** to some user **task** and vice-versa. Hubs execute in the Kernel task.

The following table summarises the possible sequence of CS between the Kernel task (executing the hub) and the user tasks responsible for the synchronisation requests on ports  $s_2$ ,  $s_1$ ,  $get$ ,  $p_1$ , and  $p_2$ . **Each line represents 1 CS.**

Notice that this is just an example. In reality, the order in which the kernel selects which task to execute next depends on many factors, including the priority of the tasks, and other tasks that might be executing.

#	Control From	Synchronisation Request	Control To
1	Kernel		Task with $s_2$
2	Task with $s_2$	$s_2$	Kernel
3	Kernel		Task with $get$
4	Task with $get$	$get$	Kernel
5	Kernel		Task with $p_1$
6	Task with $p_1$	$p_1$	Kernel
7	Kernel		Task with $s_1$
8	Task with $s_1$	$s_1$	Kernel
9	Kernel		Task with $get$
10	Task with $get$	$get$	Kernel
11	Kernel		Task with $p_2$
12	Task with $p_2$	$p_2$	Kernel

For example, assuming the execution starts in the Kernel and there are not other tasks executing apart from the ones mentioned. The Kernel selects the next task to execute (based on priority, etc.), in this case, the task responsible for  $s_2$ , and it takes 1 CS to change control to the such a task. This task then request to synchronise on port  $s_2$  and the control goes back to the kernel (+1 CS).

Please notice that this widget is experimental.

## 2.2.6 Hub Automaton Analysis

Hub Automaton Analysis
<b>Memory: 33 bits</b> <ul style="list-style-type: none"> <li>• 2 state(s): 1 bit(s)</li> <li>• 1 variable(s) of type int: 1 * 32 bit(s)</li> <li>• 0 clocks: 0 bits</li> </ul>
<b>Code size estimation: 5 loc</b> <ul style="list-style-type: none"> <li>• 2 transition(s), 2 state(s), 1 variable(s) (1 loc each)</li> <li>• 0 non-empty guards (1 loc each)</li> <li>• 0 assignment instructions (1 loc each)</li> </ul>

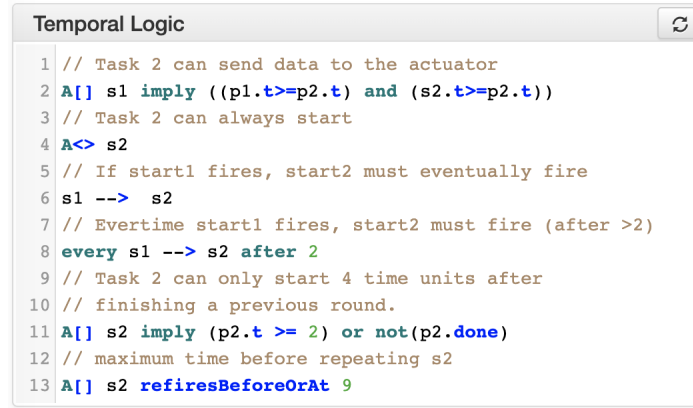
Fig. 9: THA Analysis - example of structural properties for the automaton of the hub specified in the *Hub Composer*



This widget provides a summary of some structural properties of the timed hub automaton. Currently:

- **Memory estimation** - minimum memory size (bits) required in terms of data (assumes Integer variables) and clock variables (Float variables), and in terms space needed to encode all states. Typically  $\lceil \log_2(n) \rceil$  bits are required to encode  $n$  states.
- **Code size estimation** - lines of code needed to encode the hub. Typically one line per: transition, state, variable, guard, and assignment instruction. We consider assignment instruction to clock resets and assignments on internal variables. Assignments from input to output ports are not consider as such.
- **Always available ports** - information about which ports of the hub are always ready to synchronise (up to some restrictions). This is, ports that are ready to execute in any state of the hub, possibly up to some restrictions imposed by guards, or synchronizations with other ports. For example in a data `dataEvent` hub, the input port is always ready to synchronize without delay, and without restrictions imposed by the hub - transitions with this port are single-action transitions and have a trivially satisfied guard.

## 2.2.7 Temporal Logic



```

Temporal Logic
1 // Task 2 can send data to the actuator
2 A[] s1 imply ((p1.t>=p2.t) and (s2.t>=p2.t))
3 // Task 2 can always start
4 A<> s2
5 // If start1 fires, start2 must eventually fire
6 s1 --> s2
7 // Evertime start1 fires, start2 must fire (after >2)
8 every s1 --> s2 after 2
9 // Task 2 can only start 4 time units after
10 // finishing a previous round.
11 A[] s2 imply (p2.t >= 2) or not(p2.done)
12 // maximum time before repeating s2
13 A[] s2 refiresBeforeOrAt 9

```

Fig. 10: Temporal Logic - example of temporal properties for the automaton of the hub specified in the *Hub Composer*

This widget is the editor where the user can specify a list of *timed behavioral properties*, and (if using the server version) *verify* them by relying on an instance of the Uppaal model checker running in our server (if using *ArcaTools*) or the user's computer (if using a local installation).

### The grammar

Properties are given using a **dynamic temporal logic** proposed for Timed Hub Automata, which can be seen as a subset of Uppaal Timed Computation Tree Logic (TCTL). This logic provides new operators to reason about the behaviour of the systems focusing on **actions**, i.e., on ports that are fired rather than on locations as Uppaal TCTL.

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

A valid property consists of a *path formula*  $pf$  given by the following grammar

```

// path formula
pf ::= A[] sf | E[] sf | A<> sf | E<> sf | sf --> sf | every a --> b [after n]

// state formula
sf ::= a | a.doing | a.done

```

(continues on next page)

(continued from previous page)

```

    | a refiresAfter n | a refiresAfterOrAt n | a refiresBefore n | a_
↪refiresBeforeOrAt n |
    | not sf
    | sf and sf | sf imply sf | sf or sf
    | ecc
    | deadlock | nothing

// extended clock constraints
ecc ::= c # n | c - # n | ecc and ecc | a.t # n

// clock constraints operators
# ::= = < | <= | == | >= | >

```

where  $a$  and  $b$  are port names,  $c$  is a clock, and  $n$  is an Integer.  $A$  and  $E$  are the universal and existential quantifiers over paths, while  $[\ ]$  and  $\langle \rangle$  are the universal and existential quantifiers over states.  $a.t$  is a special clock assigned to port  $a$  that is set to 0 every time  $a$  fires – i.e., after  $a$  fired, this clock tracks the time since  $a$  last fired.

The following table describes intuitively when each formula is satisfied.

Construct	Description
$A[] \text{ sf}$	Holds if in <b>all</b> possible paths, $\text{sf}$ holds in <b>all</b> states
$A<> \text{ sf}$	Holds if in <b>all</b> possible paths, $\text{sf}$ holds in <b>at least one</b> state
$E[] \text{ sf}$	Holds if in <b>at least one</b> path, $\text{sf}$ holds in <b>all</b> states
$E<> \text{ sf}$	Holds if in <b>at least one</b> path, $\text{sf}$ holds in <b>at least one</b> state
$\text{sf1} \rightarrow \text{sf2}$	Holds if whenever in every path where $\text{sf1}$ in some state $s$ , $\text{sf2}$ is eventually satisfied along the path from $s$ . It is a shorthand for $A[] (\text{sf1} \text{ imply } (A <> \text{sf2}))$ . Notice that neither Uppaal nor our logic allows nested path formulas.
$\text{every } a \rightarrow b \text{ after } n$	Holds if, whenever $a$ fires, $b$ will fire before $a$ fires again, but after 5 or more units of time since $a$ fired.
$a$	Holds at the time instance when port $a$ fires.
$a.\text{doing}$	Holds if $a$ was the last port to be fired.
$a.\text{done}$	Holds if $a$ has fired at least once.
$a \text{ refiresAfter } n$	Holds in states where, if $a$ fired, then it cannot refire until more than $n$ units of time passed.
$a \text{ refiresAfterOrAt } n$	Holds in states where, if $a$ fired, then it cannot refire until $n$ or more units of time passed.
$a \text{ refiresBefore } n$	Holds in states where $a$ fires before less than $n$ units of time passed since the beginning or since it last fired.
$a \text{ refiresBeforeOrAt } n$	Holds in states where $a$ fires before $n$ or less units of time passed since the beginning or since it last fired.
$\text{not sf}$	Holds in states where $\text{sf}$ is not satisfied
$\text{sf1 and sf2}$	Holds in states where both $\text{sf1}$ and $\text{sf2}$ are satisfied
$\text{sf1 or sf2}$	Holds in states where $\text{sf1}$ or $\text{sf2}$ are satisfied
$\text{sf1 imply sf2}$	Holds in states where if $\text{sf1}$ is satisfied, $\text{sf2}$ is satisfied as well. In states where $\text{sf1}$ is not satisfied the property is trivially satisfied.
$\text{nothing}$	Holds in states where no action has fired previously.
$\text{deadlock}$	Holds in states where there are no outgoing action transitions neither from the state itself or any of its delay successors.
$c \# n$	Holds in states where the current value of clock $c$ , $\eta(c)$ , satisfies the condition $\eta(c) \# n$ .
$c1 - c2 \# n$	Holds in states where the current value of clock $c1$ and $c2$ , satisfy the condition $\eta(c1) - \eta(c2) \# n$ .
$\text{ecc1 and ecc2}$	Holds in states where both clock constraints $\text{ecc1}$ and $\text{ecc2}$ are satisfied.
$a.t \# n$	Holds in states where the current value of clock $a.t$ satisfies the condition $\eta(a.t) \# n$ .

## The widget

To analyse the properties the user needs to load the properties by either pressing `shift + enter` or by clicking on the load icon on the top right of the widget.

Even when using the lightweight version, the widget provides the necessary information to verify each property using Uppaal manually.

After loading the properties, a new box appears showing the results. In particular, for each property, the result box shows:

- whether it is satisfied (✓ or ✗). This is shown only when using the server version (Full Hubs)
- its encoding using Uppaal’s temporal logic syntax. This is accessed by clicking on the expand button . Notice that a property using our logic might be translated into several Uppaal properties. In this case, we show for each Uppaal property whether it is satisfied - all should be satisfied in order to satisfy the original property.
- the Uppaal model needed to verify such a property and the property itself encoded using Uppaals’ syntax. This can be downloaded by clicking on .

<code>A[] (s1 imply (p1.t &gt;= p2.t and s2.t &gt;= p2.t))</code>	✓	⌵	⬇
1. <code>A[] (((Ps1 == 1)) and (ts1 == 0)) imply ((tp1 &gt;= tp2) and (ts2 &gt;= tp2))</code>	✓		
<code>A&lt;&gt; s2</code>	✗	⌵	⬇
<code>s1 --&gt; s2</code>	✗	⌵	⬇
<code>every s1 --&gt; s2 after 2</code>	✗	⌵	⬇
1. <code>A[] (Hub.L4) imply ((sinces1_s2 &lt;= 1))</code>	✓		
2. <code>A[] ((Hub.L4) and ((sinces1_s2 == 1))) imply (ts1 &gt;= 2)</code>	✗		
3. <code>((Ps1 == 1)) and (ts1 == 0) --&gt; ((Ps2 == 1)) and (ts2 == 0)</code>	✗		
<code>A[] (s2 imply (p2.t &gt;= 2 or not (p2.done)))</code>	✗	⌵	⬇
<code>A[] s2 refiresBeforeOrAt 9</code>	✗	⌵	⬇

Fig. 11: Verification Information - Output result from loading the properties in the Temporal Logic box.

## One Uppaal model per property

Depending on the kind of property, the model may need to incorporate more or less auxiliary variables in order to support such a query. For example, *a.done* query requires to add a Boolean variable *a\_done* to the model, initialized as false and set to true whenever port *a* fires (never set to false again). Thus, each property has its own Uppaal model.

## Manual verification using Uppaal

Although the user can automatically verify properties from the temporal logic widget, as explained above, it is possible to download the model and import the model from the Uppaal model checker.

After running Uppaal, go to **File -> Open System** and select the .xml model downloaded either from the *Uppaal Model* or *Temporal Logic* widget.

- **Editor:** shows the automaton of the hub and the structure of the Uppaal Project. Global declarations of variables, clocks, and channels, can be found under *Declarations*, while local declarations can be found under *Hub -> Declarations*. The initialization of the system is found under *System declarations*.
- **Simulator:** provides tools to simulate executions by selecting an enabled transition, while highlighting the current location in the automaton, among other functionality.
- **Verifier:** provides functionality to write temporal properties and verify them. If the model imported was downloaded from the *Temporal Logic* widget, it will show the corresponding property for which the model was created.

### 2.2.8 Uppaal Model



```

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

```

Fig. 12: Uppaal Model - Uppaal timed automaton model of the hub specified in the *Hub Composer*.

This widget provides the base Uppaal timed automaton model of the hub specified in the *Hub Composer*. By base we mean that the model does not have any auxiliary variable or committed states in between states of the original model, as is the case with Uppaal models generated in the *Temporal Logic* widget.

The model can be downloaded and imported into the Uppaal model checker for further analysis.



## PUBLICATIONS

If you are interested in knowing more about the theory behind Timed Hub Automata, here are some key publications that fuel the toolset.

### 2019

- [Coordination of tasks on a Real-Time OS](#)  
Guillermina Cledou, José Proença, Bernhard H.C. Spath, and Eric Verhulst,  
Coordination





## SUPPORT

We are available for support in case you encounter any issue or have trouble using the tools.

- Guillermina Cledou: `mgc at inescotec dot pt`
- José Proença: `pro at isep dot ipp dot pt`



## **VIRTUOSONEXT™ RTOS**

**VirtuosoNext™** is a distributed real-time operating system (RTOS) featuring a generic programming model dubbed *Interacting Entities*, 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.



## HOW TO USE THE TOOLS

Hubs for VirtuosoNext™ toolset is available to use [online](#) or to download and install locally following the [installation guidelines](#).

Either case, read [Using the tools](#) to learn more about how to use the tools.