
UML for Embedded Systems

# Laboratory Session Introduction

Ludovic Apvrille, Daniel Knorreck
{ludovic.apvrille, daniel.knorreck}@telecom-paristech.fr

# 1 SysML versus UML

UML 2.0 is a flexible and powerful modeling language suited for capturing large and complex systems. Its emphasis is placed on software systems and consequently on discrete systems. Additionally, UML can easily be customized for a specific purpose - e.g., an application domain - by means of profiles. A so called meta-model of a profile relates all introduced modeling elements to building blocks of UML, thus extending existing mechanisms. *SysML* is such a profile proposing few but significant extensions to UML. The OMG defines SysML as follows [4]:
*SysML supports the specification, analysis, design, and verification and validation of a broad range of complex systems. These systems may include hardware, software, information, processes, personnel, and facilities. [. . . ]*
*Currently it is common practice for systems engineers to use a wide range of modeling languages, tools and techniques on large systems projects. In a manner similar to how UML unified the modeling languages used in the software industry, SysML is intended to unify the diverse modeling languages currently used by systems engineers. [. . . ]*
*Since SysML uses UML2 as its foundation, systems engineers modeling with SysML and software engineers modeling with UML2 will be able to collaborate on models of software-intensive systems. This will improve communication among the various stakeholders who participate in the systems development process and promote interoperability among modeling tools. It is anticipated that SysML will be customized to model domain-specific applications, such as automotive, aerospace, communications, and information systems.*

The major difference of SysML with what we've studied together is the class diagram of UML, which is named *Block Diagram* in SysML. A SysML Block Diagram describes Systems in terms of structured blocks which are composed of sub-blocks. In the scope of this lab, a SysML block is equivalent to a UML2 class, apart from the following difference: we will interconnect ports of blocks directly on the Block Diagram, and not on a Composite Structure Diagram. Thus, the design is made upon only one diagram (a SysML Block Diagram). Other methodological stages are unmodified, i.e., the analysis stage relies on Use Case Diagrams and Sequence Diagrams, and the detailed design stage relies on State Machine Diagrams.

However, don't panic, everything will be progressively introduced along this tutorial, and along other labs.

# 2 The AVATAR SysML profile

The methodology usually adopted with AVATAR comprises a requirement capture phase, that we will ignore in our lab sessions. Otherwise, the methodology used with AVATAR will be exactly the one we've learnt together i.e.,:

1. **System analysis**. A system may be analyzed using usual UML diagrams, such as Use Case Diagrams and Sequence Diagrams.
2. **System design**. The system is designed in terms of communicating SysML blocks described in an AVATAR Block Diagram.
3. **Detailed system design**. The system is designed in terms of behaviors described with AVATAR State Machines.
4. **Simulation and formal verification** can finally be conducted over the system design and detailed design.

Simulation consists in exploring one possible execution of the system, i.e., one possible sequence of actions the system may perform. Such a sequence of actions is usually called a *trace*. Formal verification consists in exploring all possible execution of the systems, i.e. identifying all possible traces.

## 2.1 Block and State Machine Diagrams

An AVATAR block defines a list of *attributes*, *methods* and *signals*. Signals can be sent over synchronous or asynchronous channels. Channels are defined using connectors between ports. Those connectors contain a list of signal associations.

AVATAR State Machine Diagrams are built upon SysML State Machines, including hierarchical states. In accordance with SysML, transitions are annotated with the following elements:

- A **Guard** is a condition (for example $a == 5$ in Figure 1) which disables a transition in case it evaluates to false.
- **Actions** (like $a = a + 1$ or *doSomething*() in Figure 1) are performed when the transition is taken.

In addition to that, AVATAR State Machines enrich SysML transitions with temporal parameters:

- **Delay:** $after(t_{min}, t_{max})$. It models a variable delay during which the activity of the block is suspended, waiting for a delay between $t_{min}$ and $t_{max}$ to elapse. For instance in Figure 1 a delay of 1 to 3 time units may be encountered.
- **Complexity:** $computeFor(t_{min}, t_{max})$. It models a time during which the activity of the block actively executes instructions, before transiting to the next state: that computation may last from $t_{min}$ to $t_{max}$ units of time (4 to 7 time units in the example in Figure 1)

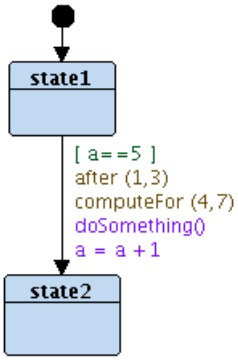



Figure 1: Transition parameters

## 2.2 Simulation of AVATAR models

AVATAR models can be simulated i.e. it is possible to obtain one possible execution of an AVATAR model. That execution can be presented in the form of a sequence diagram.

## 2.3 Model transformation to UPPAAL

UPPAAL is a formal language based on communicating automata. A formal language is a language defined in a mathematical way, relying on words defined with an alphabet. Constructs of the language have an operational semantics i.e. a precise mathematical meaning is given to each construct of the language.
On the contrary, **UML is not formally defined**, since it is defined using a document in plain text (English), and definitely not with a mathematical approach. That is, two persons reading the same UML diagram may have a different understanding of that diagram. Conversely, two persons reading the same UPPAAL description have exactly the same understanding of the system.

A design in AVATAR can be automatically translated to UPPAAL, using an algorithm implemented in TTool (see Figure 2). That algorithm being formally defined, and since a UPPAAL description is also formally defined, we can say that an AVATAR design is also a formal language. And so, two persons reading the same AVATAR model shall have exactly the same understanding of the system.

UPPAAL is not only the name of a language, but it is also the name of the toolkit that can handle specifications given in the UPPAAL language. UPPAAL - the toolkit - has simulation

capabilities i.e., it can simulate step-by-step a UPPAAL specification, and it can display the simulation trace in the form of a Sequence Diagram. However, TTool integrates its own simulator that can directly animate your SysML models, and so, this is not very likely that you will have to use the UPPAAL simulator.
UPPAAL also has formal verification capabilities, that is, you can specify in a formal way properties you want to prove on your system (for example, proving that a Coffee Machine never delivers a beverage for free): for each property, UPPAAL can answer "true", false, or "couldn't prove it". The "couldn't prove it" applies when UPPAAL was not able to prove the property: in that latter case, it is most likely that the system has so many traces that your computer can't compute all traces (lack of memory, or it would take far too much time). That situation is called *combinatory explosion*.

Simulation and formal verification are further explained in this tutorial.

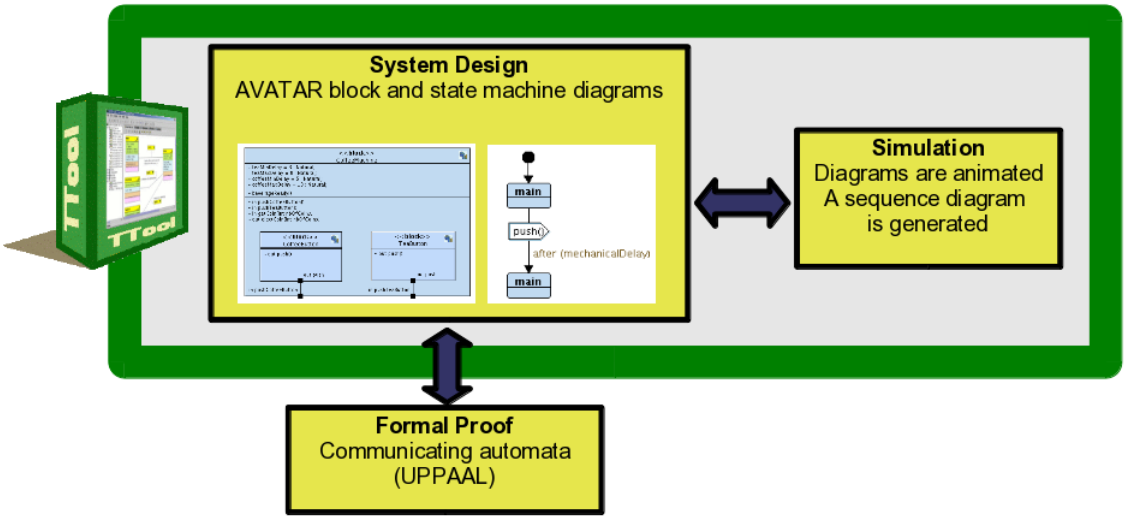



Figure 2: Simulation and verification capabilities

# 3 Toolkit: TTool

## 3.1 Overview

The free software toolkit *TTool* [3] supports several UML / SysML profiles, in particular AVATAR, TURTLE [2] and DIPLODOCUS [1]. TTool offers UML modeling edition facilities, and well as press-button approaches for simulation and formal verification. TTool and its profiles are supported by several academic and industrial partnerships.

TTool is interfaced to verification tools (e.g., to UPPAAL) that implement *reachability* and *liveness* analysis. Thus, the developer may easily find out whether a system state is reached in every possible execution (*liveness*) or whether there exists at least one execution reaching a state (*reachability*). To perform a formal analysis, it suffices to right click on the corresponding symbol: The UPPAAL verifier is invoked with the corresponding verification formula (expressed in a formal language called CTL), and the result is directly fed back to TTool.

## 3.2 Getting started

Let's start as follows:

1. **Start TTool** by invoking the start up script *"ttool.exe"*, probably located in your *TTool_local/bin* directory.

2. To **create a new model** i.e. click on the *new* icon, and then right click on the document panel (the main large panel)and choose *New Avatar Design* in the context menu (see Figure 3). As usual, an existing model may be opened using the *Open*  icon in the main toolbar.

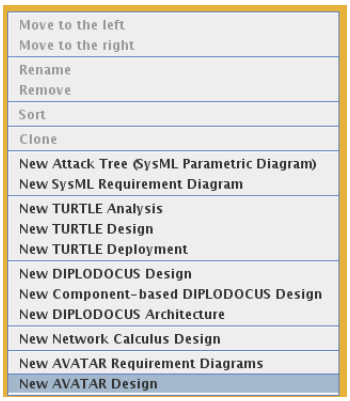



Figure 3: Create a new AVATAR diagram

3. Design the **Block Diagram** which represents the static view of your model. You may want to **add Blocks** by selecting the corresponding icon  in the toolbar and placing them by left clicking on the document panel. The *Port connector* tool  allows you to **interconnect blocks** with communication primitives. After having selected the connector, yellow connection points show up on the blocks. Left click first on the

source point and then on the destination point. To **change the properties of a block**, double click on its body and a dialog window will pop up (see Figure 4). There is a separate tab for the modification of Attributes, Methods and Signals. To **change the name of a block**, double click on the «*block*» descriptor. Do not forget to explicitly **link input to output ports** by double clicking on the respective port connector in your model. A dialog appears (depicted in Figure 5) which also provides options for the channel type and the FIFO size.

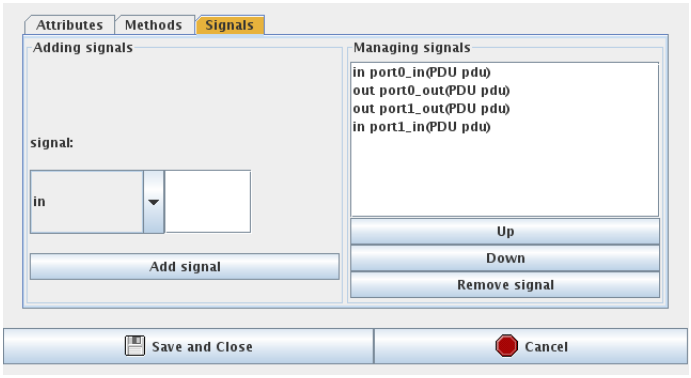



Figure 4: Block attributes dialog

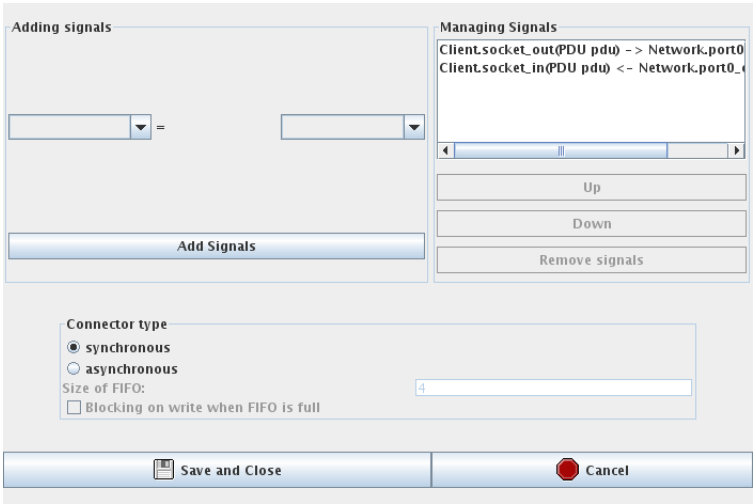
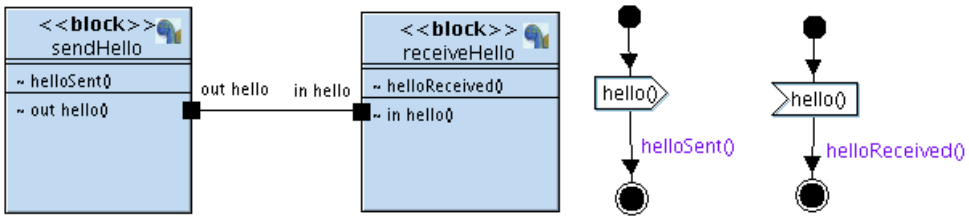



Figure 5: Signal attributes dialog

**Try it out:** Build the Hello World Block Diagram depicted in Figure 6a. The block *sendHello* has one outgoing signal called *hello* and one method *helloSent()*. The block *receiveHello*

inversely receives a signal called *hello* and comprises a method *helloReceived()*. Incoming and outgoing ports should be connected to each other using a **synchronous channel**. Note that the ports of the two blocks are filled in black, which means that you have used a synchronous channel. Ports filled in white correspond to asynchronous channels.




(a) Block diagram (b) SendHello (c) ReceiveHello

Figure 6: The Hello World AVATAR model

4. **Describe the behavior of your blocks** with **State Machine Diagrams**. A dedicated tab for each block is provided above the toolbar. The latter contains buttons for placing states  , signal send operators  , signal receive operators  , set timer operators  , cancel timer operators  , wait for timer expiration operators  and non-deterministic choice operators  amongst others.

   A statechart always starts with a start state  , and you may use the stop state  to terminate a sub-activity. A typical statechart comprises sequences of the following operators in the given order: a state, a signal reception and actions (sending signals, setting timers, etc.). The expiration of timers are pretty much treated in the same way as signal receptions.
   Operators can be connected together using **Transitions**. The parameter dialog of transitions (c.f. Figure 7) shows up when you double click on the respective arrow. As stated in 2.1, transitions are characterized by a guard (a condition to be satisfied for the transition to be enabled), minimal and maximal *after* values (delay for the transition to be taken), *compute for* values (time penalty due to computations) and actions (variable operations, method invocation, etc.) to be executed upon firing of the transition. As a rule of thumb, you can think of *after* delays as being imposed on the system by external components (mechanical delays,...), whereas *compute for* delays account for operations performed by the block itself.

**Try it out:** Describe the behavior of the two blocks you created previously. The *sendHello* block merely sends the *hello* signal which is subsequently received by the *receiveHello* block. After that, the blocks execute their respective method *helloSent/helloReceived*. Figure 6b and Figure 6c may guide you in this task.

5. **Check the syntax** of your model by clicking on the *Syntax analysis* button  in the Toolbar of TTool. Identify the syntax error in your model with the aid of the tree view panel in the upper left corner of the window (double click on the nodes *Validation*, *Syntax Analysis / formal code generation*, Error(s), as described in Figure 8). Double clicking on the error directly navigates you to the badly formatted command. Please note: If the syntax has not been validated, you cannot proceed with simulation nor formal verification.

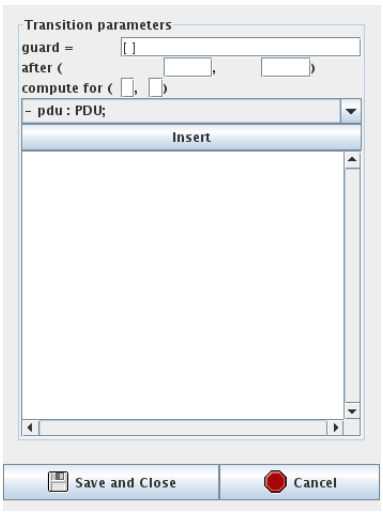



Figure 7: Transition attributes dialog

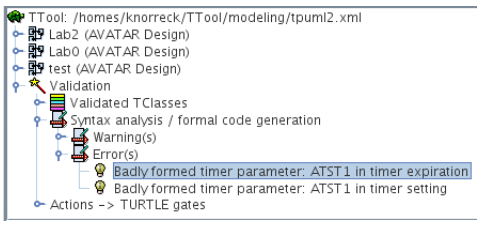



Figure 8: Tracing syntax errors

6. **Perform simulations, and accordingly animate your UML models** by clicking on the appropriate sequence diagram icon in the toolbar (*Interactive simulation*). The window depicted in Figure 9 should open. You can select the pending transitions directly on the SysML diagrams, go back one transition, reset the simulation, etc. Value of variables of blocks are displayed in a table. Also, each time a noticeable action occurs in the simulation, a corresponding action is drawn in the sequence diagram representing the execution of the system.

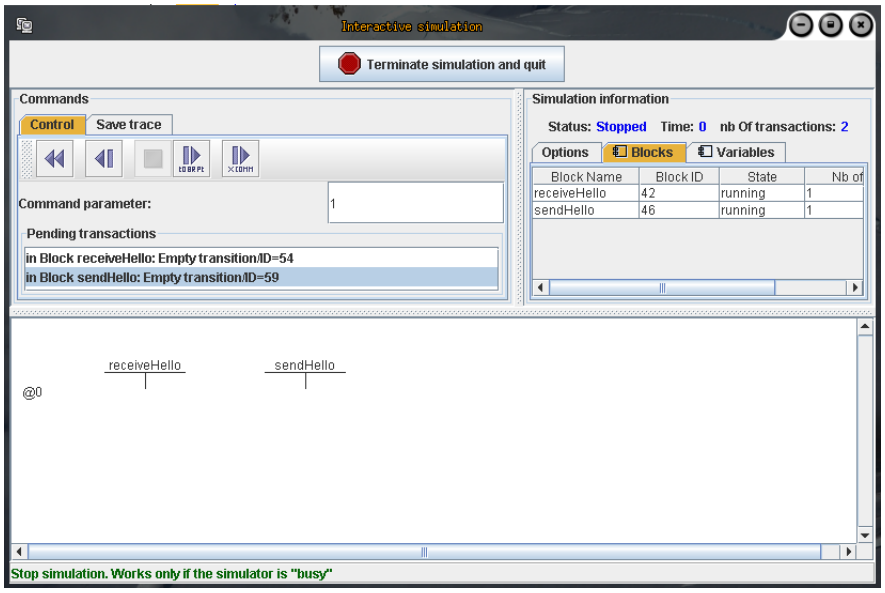



Figure 9: Interactive simulation window

**Try it out:** Simulate your model several times. You should be able to observe two different behaviors. Guess the reason why!

7. **Convert your AVATAR model to a UPPAAL one, and perform formal proof with UPPAAL**. **Formal verification** is the act of proving whether a given property is satisfied by a system, or not, using formal techniques. Notions of *liveness* and *reachability* were introduced in section 3. Don't be scared by those definitions: performing proves with TTool becomes a piece of cake as the graphical interface makes the underlying theory transparent to the user! Proceed as follow:

   - First, tag an operator of a state machine for which you want to study the reachability or the liveness. To do so, right click on the operator, and select "Check for accesibility / liveness with UPPAAL", see Figure 10

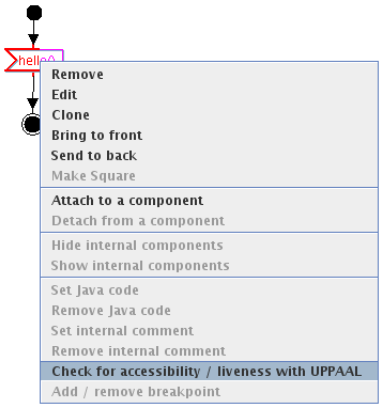



Figure 10: Tagging an operator for formal verification

- Second, check the syntax of your model
- At last, click on the toolbox icon in the toolbar (*Formal verification with UPPAAL*): the window depicted in Figure 11 should open. In this Figure, the *Start* button launches the verification procedure and upon its termination, verification results are displayed in the textbox of the window.

**Try it out:** Tag an operator of your choice in your design and run the verification with the parameters depicted in Figure 11. Interpret the results.

# 4 Synchronous versus asynchronous communication

A Reachability Graph (RG) captures all possible executions of a system. You can think of it as a way to represent together all traces you may obtain at simulation step. Those traces are represented in terms of nodes, and transitions between these nodes.
In case your AVATAR model does not stipulate a strict order among events, all possible interleavings between various system events are represented in the RG.
In our example, the RG depicts the two possible executions you observed during simulation (see Figure 12). Your task is first to figure out what the transitions named $t1$; $t2$ and $t3$ possibly stand for. After that, draw the Reachability Graph for the two blocks connected with an **asynchronous** channel. To do this, you may want to modify the channel type in your AVATAR model to be able to simulate and experiment in UPPAAL.

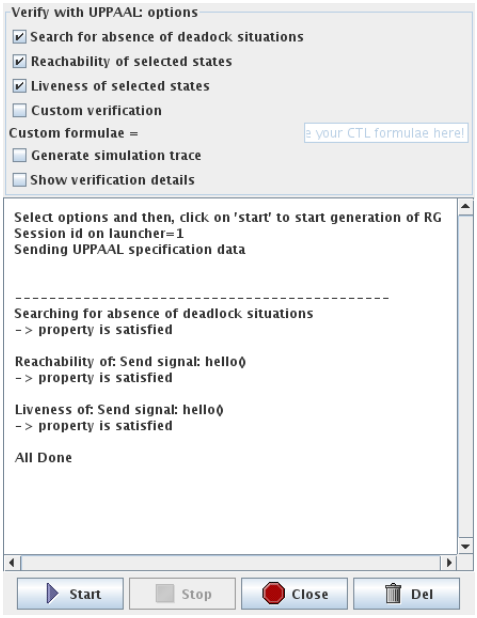



Figure 11: Formal Verification based on UPPAAL and driven from TTool

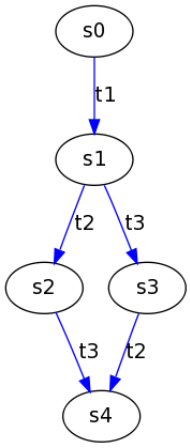



Figure 12: Reachability Graph of the Hello World AVATAR model

# References

[1] L. Apvrille. TTool for DIPLODOCUS: An Environment for Design Space Exploration. In *Proceedings of the 8th Annual International Conference on New Technologies of Distributed Systems (NOTERE'2008)*, Lyon, France, June 2008.

[2] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes. TURTLE: A real-time UML

profile supported by a formal validation toolkit. In *IEEE transactions on Software Engineering*, volume 30, pages 473–487, Jul 2004.

[3] Ludovic Apvrille and Pierre De Saqui-Sannes. Making formal verification amenable to real-time UML practitioners. In *Proceedings of the 12th European Workshop on Dependable Computing*, Toulouse, France, May 2009.

[4] OMG. Omg systems modeling language. In *http://www.sysmlforum.com/docs/specs/OMGSysML-v1.1-08-11-01.pdf*, 2008.