Une école de l'IMT

TTool

`ttool.telecom-paristech.fr`

Code generation from Avatar Design Diagrams in TTool

	Document Manager	Contributors	Checked by
Name	Ludovic APVRILLE	Ludovic APVRILLE	Ludovic APVRILLE
Contact	ludovic.apvrille@telecom-paristech.fr		
Date	November 3, 2017		

Contents

1 Preface

1.1 Table of Versions

Version	Date	Description & Rationale of Modifications	Sections Modified
1.0	13/06/2017	First draft	

1.2 Table of References and Applicable Documents

Reference	Title & Edition	Author or Editor	Year

1.3 Acronyms and glossary

Term	Description

1.4 Summary

This document describes the code generation principle for AVATAR design diagrams implemented in TTool. It describes how to configure TTool for generating code, how to generate the code, how to compile it, how to execute it. Finally, the document explains how to have the generated code to connect with an external graphical interface.

2 Configuration

2.1 TTool configuration

At first, if not already configured[1], you must open the configuration file of TTool. The default file is located in:

`TTool/bin/config.xml`

Open your configuration file, and set the following lines accordingly with your TTool installation:

- Main directory in which the generated code and the avatar runtime library are located:

  ```
  <AVATARExecutableCodeDirectory data="../executablecode/" />
  ```

- Host that is intended to perform the code compilation and execution. Default value is "localhost".

  ```
  <AVATARExecutableCodeHost data="localhost"/>
  ```

- Compilation command to compile the generated code:

  ```
  <AVATARExecutableCodeCompileCommand data="make -C ../executablecode/" />
  ```

- Execution command. This will start the application generated from your model:

  ```
  <AVATARExecutableCodeExecuteCommand data="../executablecode/run.x" />
  ```

2.2 External tools

The previous configuration assumes that a **C compiler**, referenced by the provided Makefile (default = "gcc"[2]) is installed on your machine, as well as the **POSIX-1 librairies**. Also, a Mafile utility must be installed (e.g., "GNU make"[3]).

[1] TTool comes already configured
[2] `https://gcc.gnu.org/`
[3] `https://www.gnu.org/software/make/`

3 A first example

This very first example explains how to generate the code from an AVATAR design model, and how to introduce your own basic C functions in the code generation process.

3.1 Getting the example

Be sure to get the latest version of TTool including the remote loading of models (June 2017 and after). Do: File, Open from TTool repository, and select "HelloWorldCodeGeneration.xml".

3.2 Understanding the model

This models contains a design diagram composed of one MainBlock. This later regularly executes the "printHelloWorld" method (see Figure 1).

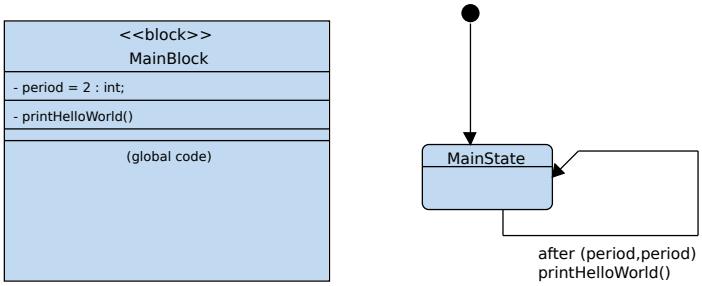

Figure 1: Hello world model

You may then check the syntax of the diagram, and select the "interactive simulation icon". From the window that opens, make a step-by-step simulation, and observe the behaviour of the system. This behaviour is simulated, that is, there is no executable code that is generated to simulate the model.

3.3 Generating executable code

To generate executable code, click on the "check syntax" icon, and then click on the "code generation" icon representing a gear. The following window should open (see Figure 3).

- You can add debugging information to the generating code ("Put debug information ...") if you wish the generated code to print information in the default output when executing. Typical debugging information is: state entering/exiting and send / receive of signals.
- Tracing capabilities enable to draw a sequence diagram representing the execution of the code.
- User defined code can be included (or not). **Uncheck this option first**.
- The time unit manipulated by TTool can be set to seconds, milliseconds or microseconds. For example, if "sec" is selected, it means that "after(2)" will be transformed as a waiting of two seconds. Default is "sec", keep it like this for the incoming tutorial.

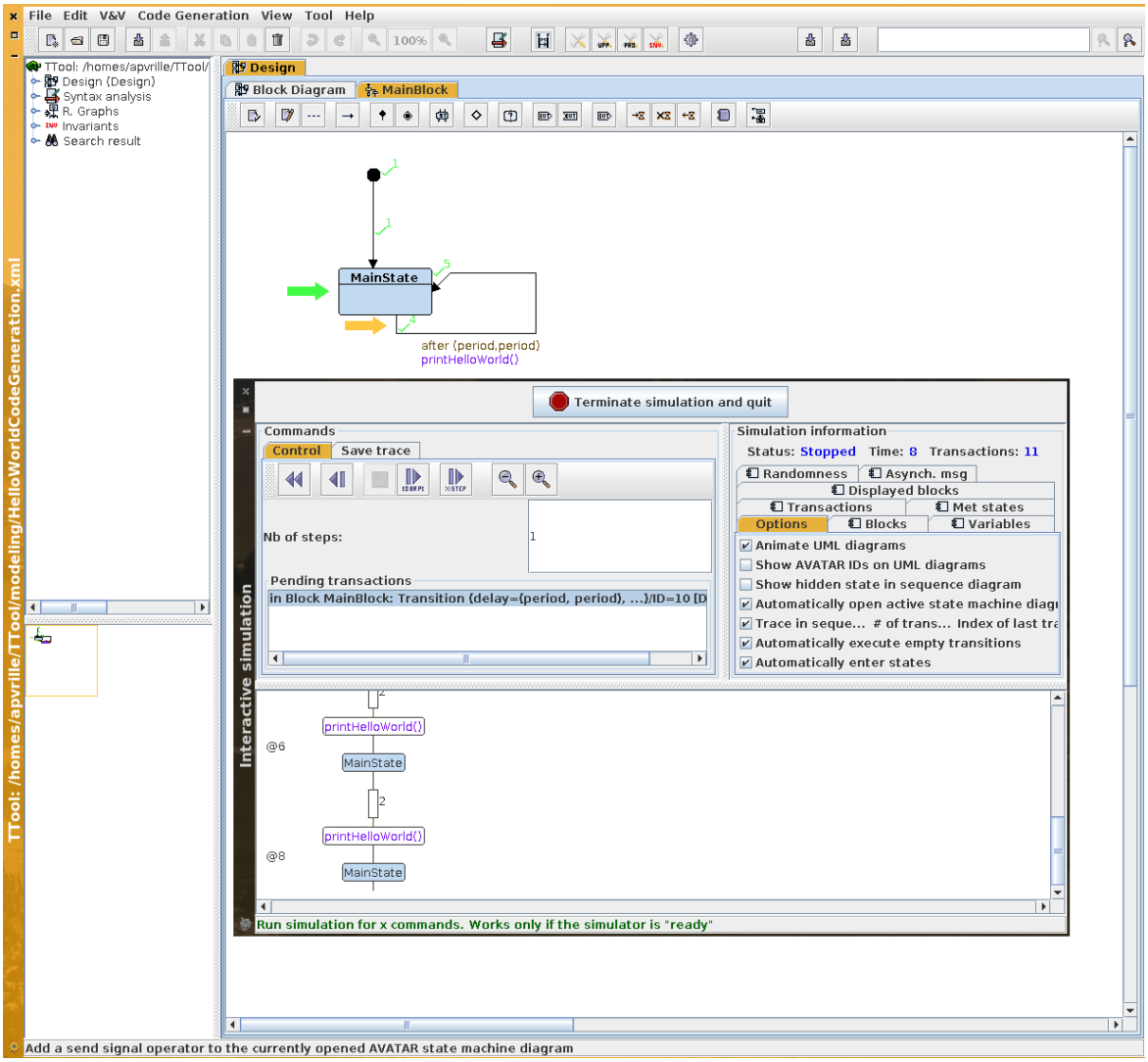

Figure 2: Functional simulation of the Hello world model

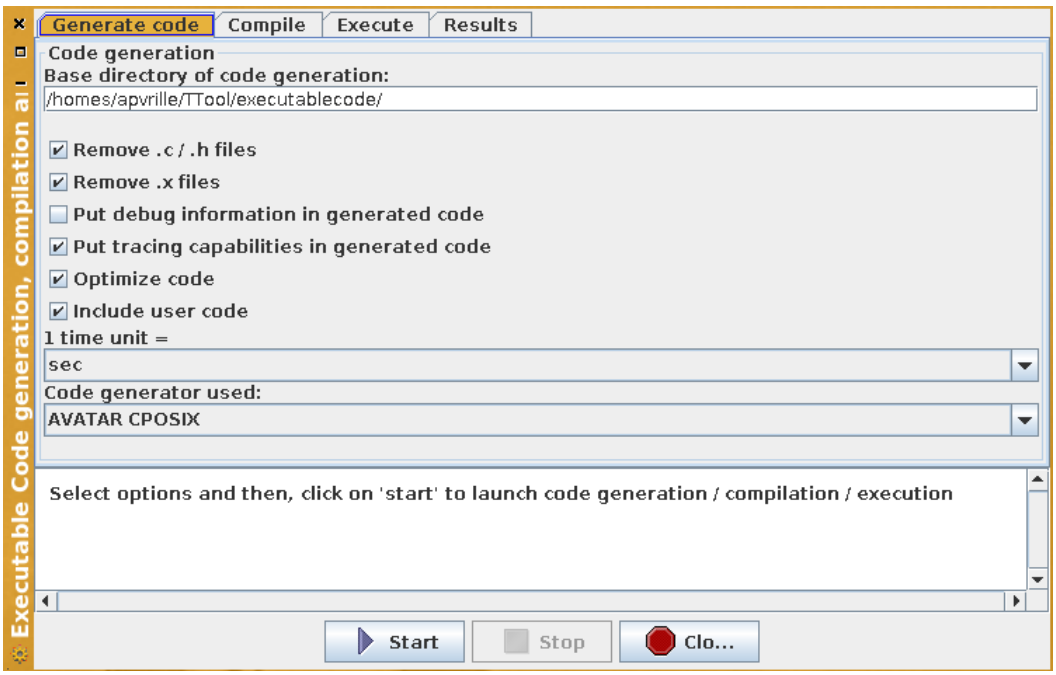

Figure 3: Generating C/POSIX code for the Hello world model

3.4 Compiling the generated code

Once the code has been generated, the dialog window should automatically switch to the "Compile" tab (see Figure 4). There, you should notice two alternative possibilities:

- Compile the code for your localhost (**You should select this option**).
- Compile the code for the SoCLib platform. This option is not addressed in this document.

If the compilation fails, it is probably due to a bad installation of a C compiler. You could also edit the Makefile you have selected (see section 2) to adapt it to your localhost particularities. Note that the compilation process also compiles the Avatar runtime C sources[4], and links all resulting object files together.

You may obvisouly try to compile the code from a terminal. e.g.:

```
$ cd TTool/executablecode
$ make
echo Making directories
Making directories
mkdir -p ./lib
mkdir -p ./lib/generated_src/
mkdir -p ./lib/src/
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/generated_src/main
    ↪ .o -c generated_src/main.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/generated_src/
    ↪ MainBlock.o -c generated_src/MainBlock.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/request.o -c
    ↪ src/request.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/message.o -c
    ↪ src/message.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/myerrors.o -c
    ↪ src/myerrors.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/debug.o -c src
    ↪ /debug.c
```

[4]These sources are located in TTool/executablecode/src

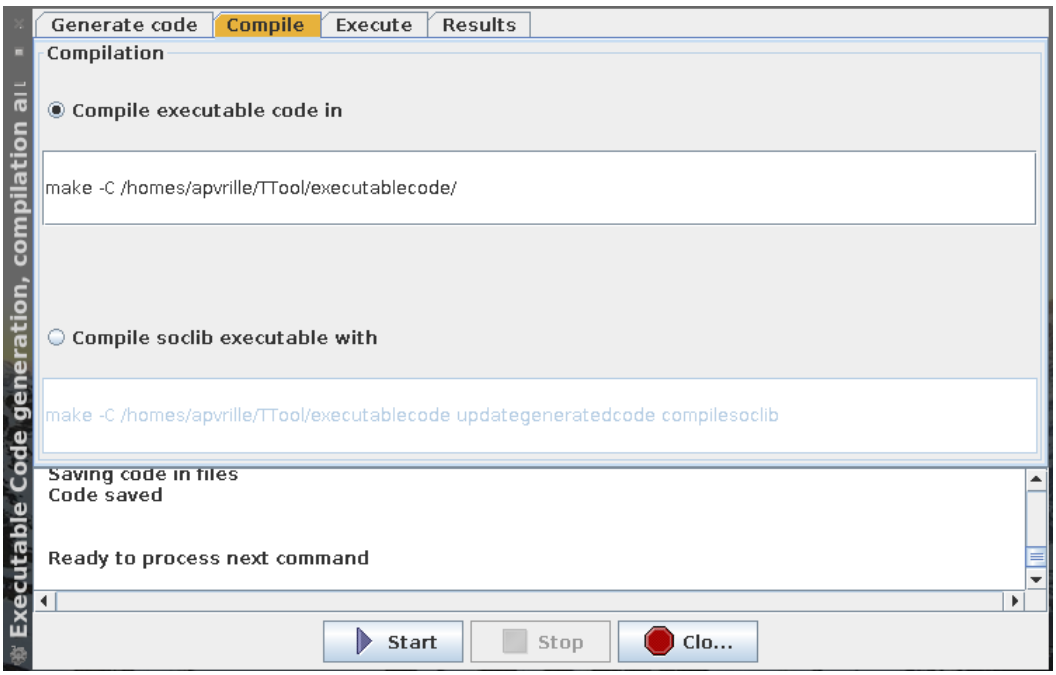

Figure 4: Compiling the Hello world model generated C code

```
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/syncchannel.o
    ↪ -c src/syncchannel.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/asyncchannel.o
    ↪  -c src/asyncchannel.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/
    ↪ request_manager.o -c src/request_manager.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/random.o -c
    ↪ src/random.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/mytimelib.o -c
    ↪  src/mytimelib.c
/usr/bin/gcc -O1 -pthread -Wall  -I. -I. -Isrc/  -Igenerated_src/ -o lib/src/tracemanager.o
    ↪  -c src/tracemanager.c
/usr/bin/gcc -O1 -pthread -ldl -lrt -Wall  -I. -I. -Isrc/  -Igenerated_src/ -L. -L..  -o
    ↪ run.x lib/generated_src/main.o lib/generated_src/MainBlock.o lib/src/request.o lib/
    ↪ src/message.o lib/src/myerrors.o lib/src/debug.o lib/src/syncchannel.o lib/src/
    ↪ asyncchannel.o lib/src/request_manager.o lib/src/random.o lib/src/mytimelib.o lib/
    ↪ src/tracemanager.o -lm  2>&1 | c++filt
```

3.5 Executing the generated code

Once the generated code has been successfully compiled and linked, the execution tab is selected (see Figure 5). There are three possible options to execute the compiled program:

- Running the program ("Run code")
- Running the program and activating the backtracing
- Running the program in the SoCLib environment (no covered in this document).

Select the second option. An execution trace should be displayed in the console of the dialog window.

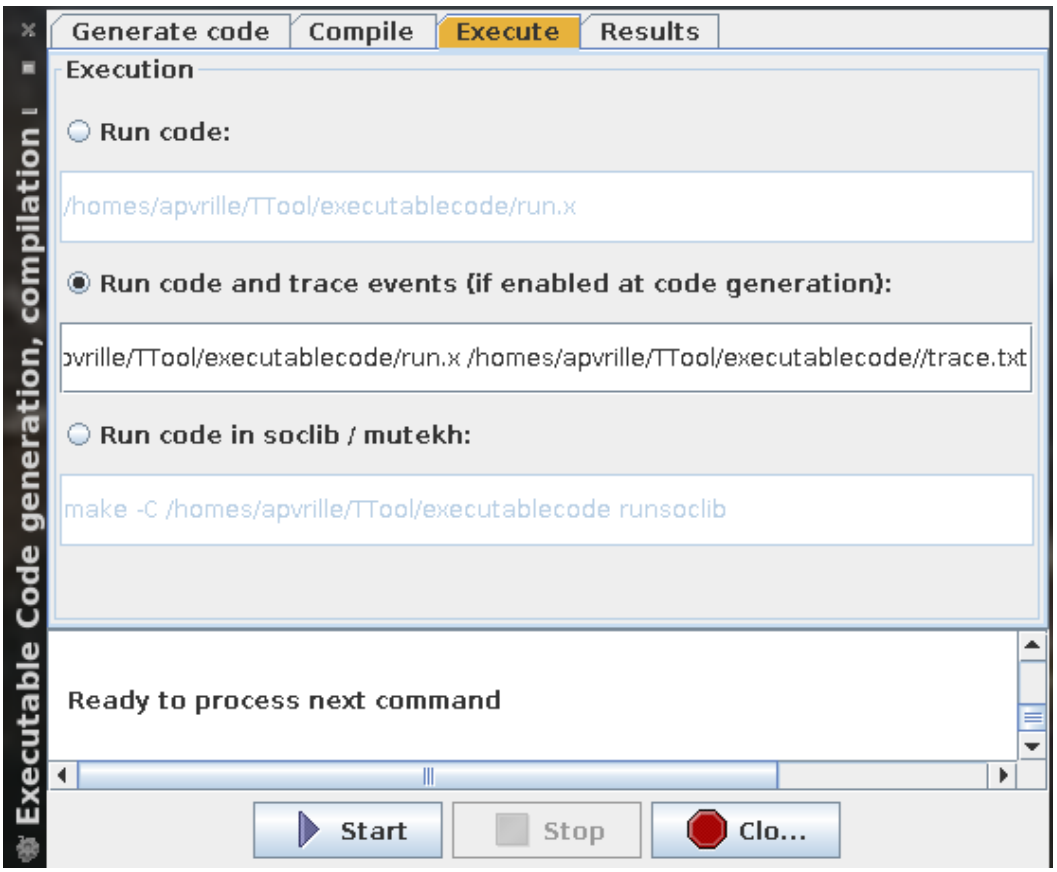

Figure 5: Executing the Hello world program

3.6 Backtracing

After you have started the program, switch to the "Results" tab. You should see a window similar to the one display in Figure 6. There are two options:

- Displaying the execution trace of the localhost program
- Displaying the execution trace of the SoCLib program (option is not covered in this manual)

Select the first option, and click on the "show simulation trace" button. A new window should open, displaying the execution of the model under the form of a UML Sequence Diagram (see Figure 7)

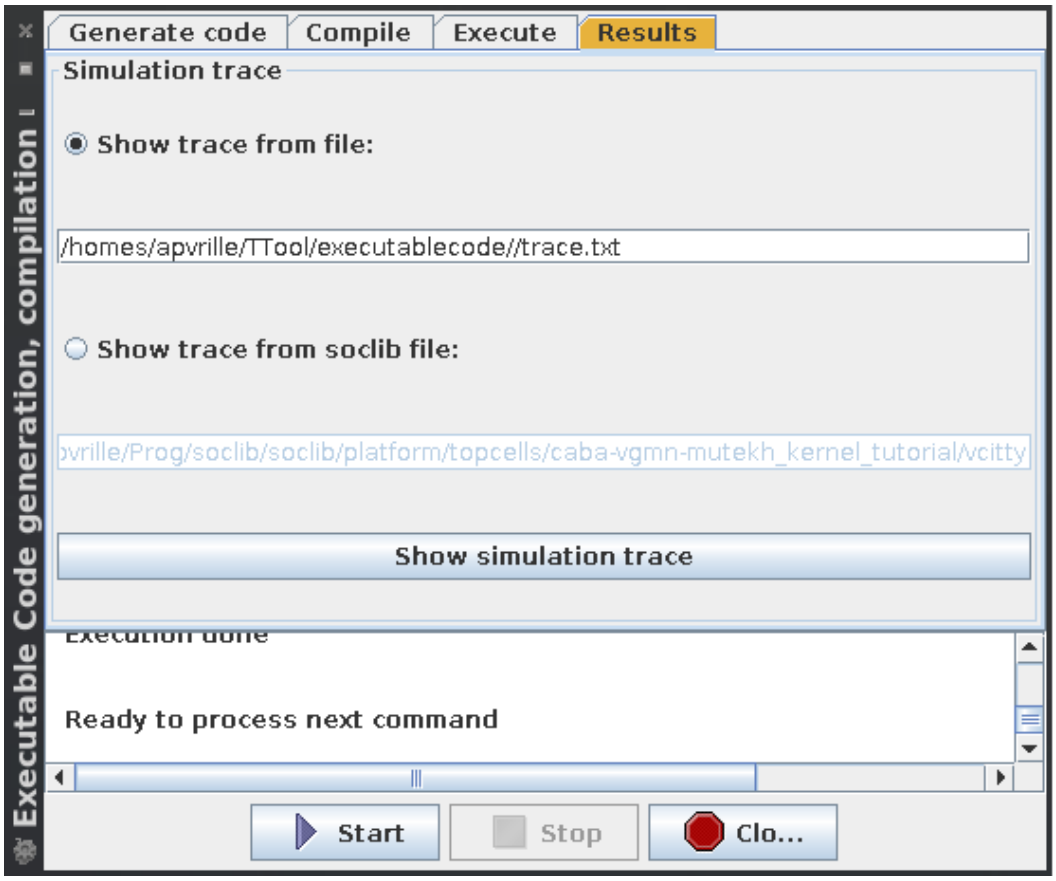

Figure 6: Backtracing dialog window

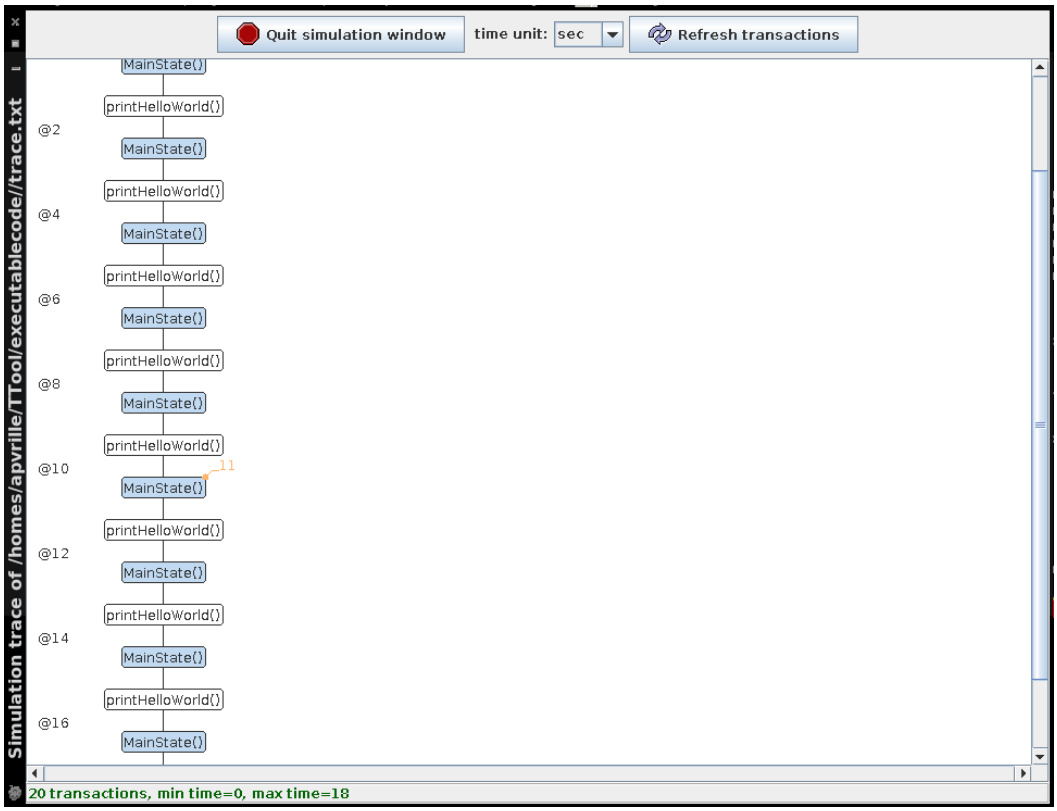

Figure 7: Executing the Hello world program

4 Enhancing model with user code

4.1 Principle

A user of TTool can provide its own C code within an AVATAR design diagram. When a model is enhanced with custom C code, the custom C code may prevent the compilation process to succeed: if the compilation fails, you need to reconsider the code you have inserted according to the errors provided by the compiler.
Basically, there are two types of custom code as show in the "prototyping" window (see Figure 8). To open this window, simply double-click in the attributes/methods/signal/code part of a given block.

- **The global code of the model**
- **The local code of a given block**. This code is itself split into two sub parts:
  - The global code of the block.
  - The code implementing methods of the block.

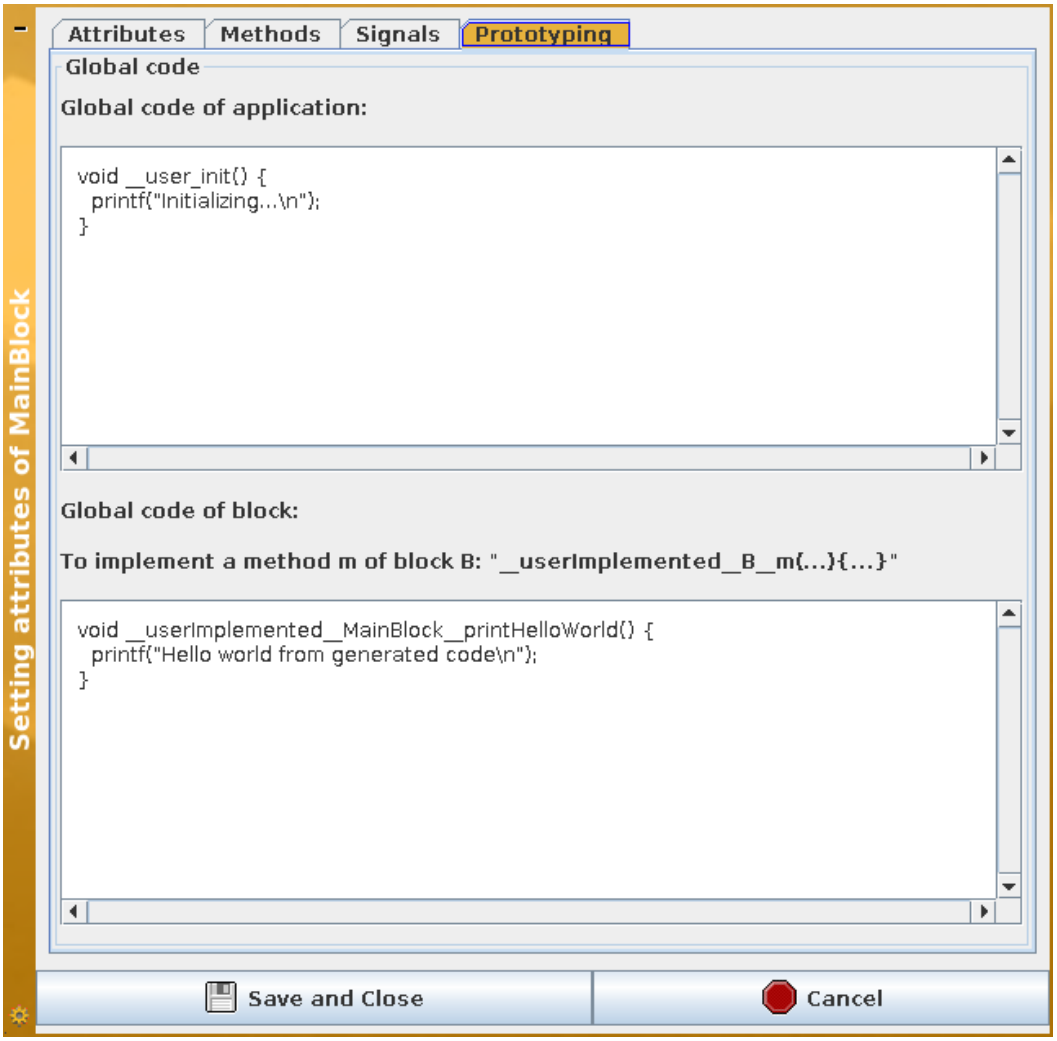

Figure 8: Global code of an application and global code of a block

4.2 Global code

The global code typically contains the declaration of global variables. Also, one specific method can be used to execute code when the application starts. The function prototype is:

`void __user_init()`

For instance, the global code of the HelloWorld example is as follows:

```
void __user_init() {
  printf("Initialializing\n");
}
```

4.3 Block code

The block code typically contains variables that are not declared as block attributes. Block attributes can indeed be directly used in the custom code. The block code can also provide an implementation for the block methods. For a method called "method" of the block "block", the function must be declared as "__userImplemented__block__method" and **you must ensure to check the "Implementation provided by the user" option** in the method definition window (see Figure 9). For instance, the following code correesponds to the block code of "MainBlock". It implements the method *printHelloWorld()* referenced in the model (see Figure 1).

```
void __userImplemented__MainBlock__printHelloWorld() {
  printf("Hello world from generated code\n");
}
```

4.4 Code generation and execution with custom code

Be sure to **check the "Include user code" option** on the code generation tab of the code generation window. You may also uncheck the "Put tracing capabilities in generated code" since we won't use this in this example. Then, follow the usual stages: compile, and then execute the program. You should now see the effect of the printf command in the console. It should like this:

```
Initializing...
Hello world from generated code
Hello world from generated code
Hello world from generated code
```

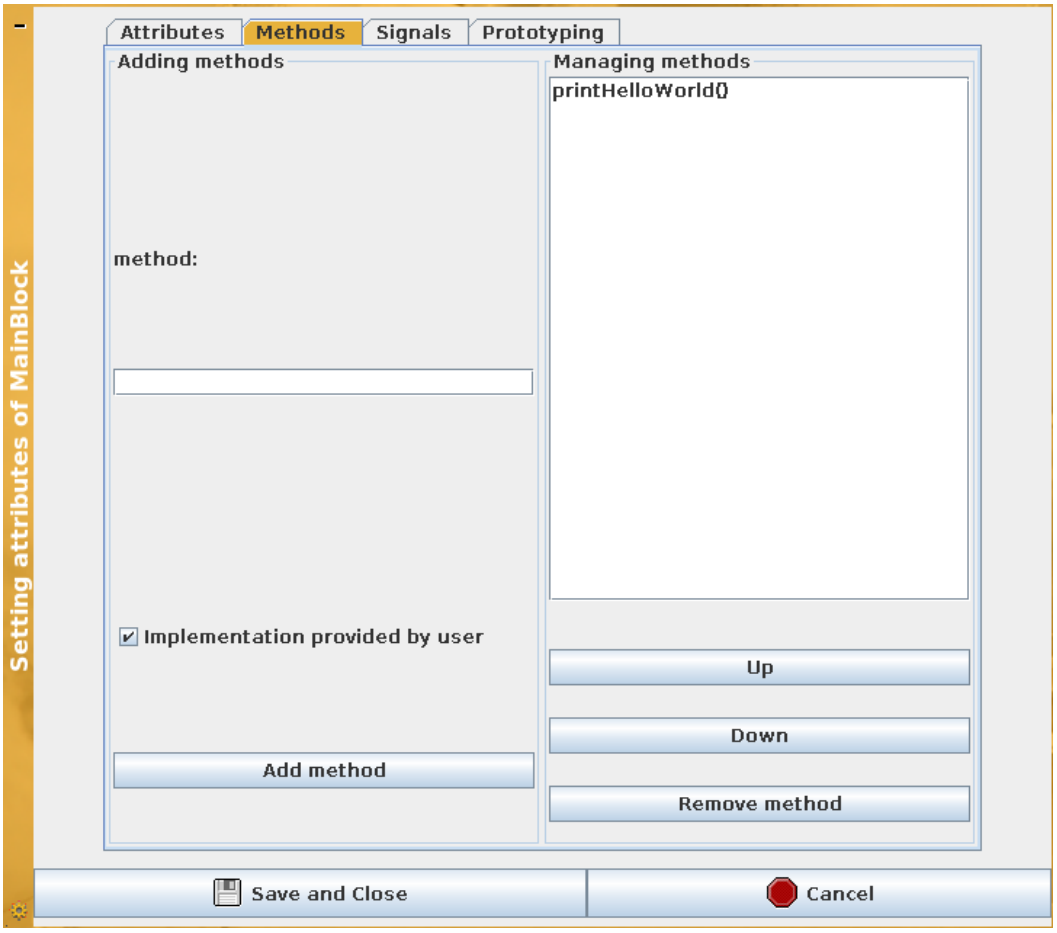

Figure 9: Selecting a method for which the user provides an implementation

5 Advanced model enhancement with user code

To follow this section, you have to use another TTool model called "MicroWaveOven_SafetySecurity_fullMethodo.xml", available on the network repository of TTool (File, Open from TTool repository).
This section explains how a code generated from TTool can be linked with an external software, e.g. to animate a graphical user interface. The provided example relies on datagrams and sockets to echange information between the Microwave Software (fully generated from TTool) and its graphical user interface (programmed "by hand"). We will now call these two software MS and GUI, respectively

5.1 GUI

The TTool distribution includes an external software which represents a graphical user interface of a microwave system. The source code (in Java) of this software is located in "TTool/executablecode/example":

```
$cd TTool/executablecode/example
$ls
DatagramServer.java  MainMicrowave.java           MicrowavePanel.java
Feeder.java          MainPressureController.java  README
```

This directory contains the java sources as well as a README file. First compile the java source code, and then execute the GUI, as follows:

```
$javac *.java
$java MainMicrowave
```

A window similar to the one of Figure 10 should open. This window is not yet animated. To do so, we need to build the MS.

5.2 Microwave Software (MS)

Open the model in TTool, then double-click on a block method, and finally select the "Prototyping" tab. You can now review the global and local code.

5.2.1 Global code

The goal of the global code is to setup the communication infrastructure to the GUI. To do so, the global code is as follows.

- It declares headers.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <strings.h>
#include <errno.h>
```

- It defines the global constants and variables, including the hostname to use, and the port.

```
const char* hostname="localhost";
const char* portname="8374";
int fd;
struct addrinfo* res;
#define MAX_DGRAM_SIZE  549
```

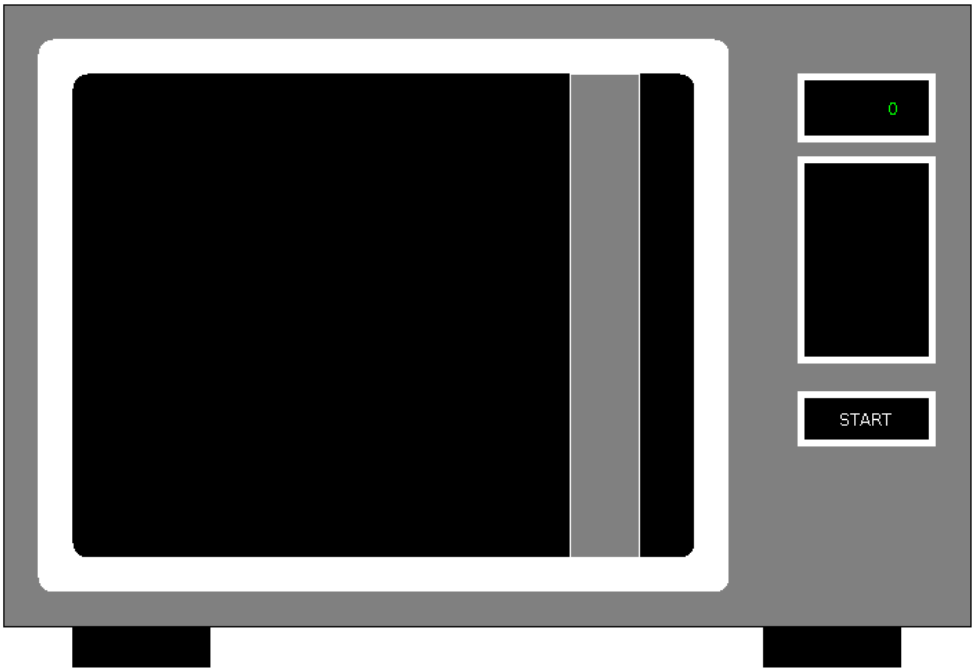

Figure 10: Window of the Graphical User Interface

- It defines the global constants and variables used to be able to interact from the GUI to MS, and the functions to send datagrams to the MS. For example, when you click on the "start" button, a datagram is sent from GUI to MS. This part is explained in more details in section 5.4

```
pthread_t thread__Datagram;

// Handling start datagrams
int start = 0;
pthread_mutex_t startMutex ;
pthread_cond_t  noStart;



void startDatagram() {
  pthread_mutex_lock(&startMutex);
  start = 1;
  pthread_cond_signal(&noStart);
  pthread_mutex_unlock(&startMutex);
}

// Assumes fd is valid
void* receiveDatagram(void *arg) {
  printf("Thread receive datagram started\n");

  char buffer[MAX_DGRAM_SIZE];
  struct sockaddr_storage src_addr;
  socklen_t src_addr_len=sizeof(src_addr);

  while(1) {
    printf("Waiting for datagram packet\n");
    ssize_t count=recvfrom(fd,buffer,sizeof(buffer),0,(struct sockaddr*)&src_addr,&
        ↪ src_addr_len);
    if (count==-1) {
      perror("recv failed");
    } else if (count==sizeof(buffer)) {
```

```
      perror("datagram too large for buffer: truncated");
    } else {
      //printf("Datagram size: %d.\n", (int)(count));
      if (strncmp(buffer, "START", 5) == 0) {
        //printf("++++++++++++++++++++++++ START\n");
        startDatagram();
      }
    }
  }
}
```

- It defines a function to send a datagram.

```
void sendDatagram(char * data, int size) {
  if (sendto(fd,data,size, 0, res->ai_addr,res->ai_addrlen)==-1) {
        printf("Error when sending datagram");
        exit(-1);
    }
}
```

- It defines the user initialization function. This function puts in an "addinfo struct" information for sending the datagram. Then, it gets the target IP address. Following this, it creates a reference to the right socket. Finally, it tries to send a test packet ("salut"). Whenever a failure is encountered, the application exits. When the C code is generated from the model, all the global code is integrated into a file called "main.c".

```
void __user_init() {
  const char* content = "salut";
  struct addrinfo hints;

  memset(&hints,0,sizeof(hints));
  hints.ai_family=AF_UNSPEC;
  hints.ai_socktype=SOCK_DGRAM;
  hints.ai_protocol=0;
  hints.ai_flags=AI_ADDRCONFIG;

  int err=getaddrinfo(hostname,portname,&hints,&res);
  if (err!=0) {
    printf("failed to resolve remote socket address (err=%d)",err);
    exit(-1);
  }
  fd=socket(res->ai_family,res->ai_socktype,res->ai_protocol);
  if (fd==-1) {
    printf("%s",strerror(errno));
    exit(-1);
  }
  if (sendto(fd,content,sizeof(content),0,
      res->ai_addr,res->ai_addrlen)==-1) {
    printf("%s",strerror(errno));
    exit(-1);
  }

  // Start a thread to receive datagrams
  pthread_create(&thread__Datagram, NULL, receiveDatagram, NULL);
}
```

Note: the network code of the GUI application is located in *DatagramServer.java*.

5.2.2 Local code

The local code associates the call to a method of a block to the sending (or receiving) of a datagram packet. Let's take the example of the "Door" block. The GUI should be informed about door opening or closing operations. In the model, each time the door is closed, the

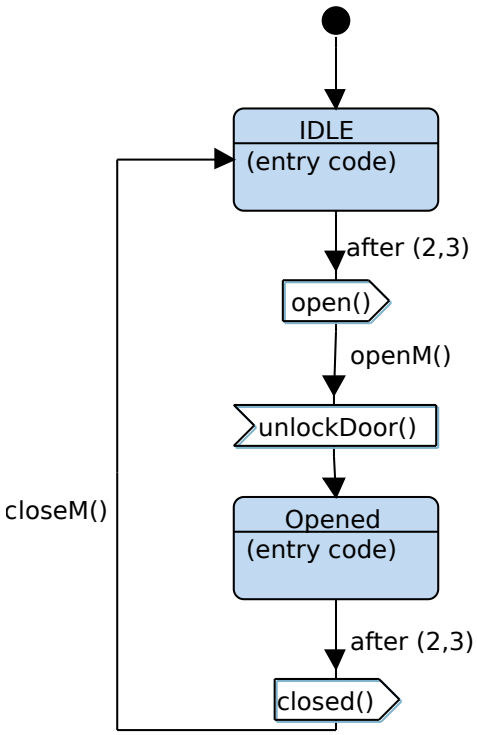

Figure 11: State Machine Diagram of the Door block

openM() method is closed. Also, each time the door is closed, the *closeM()* method is called (see Figure 11).

In the local code, we first need to state that a *sendDatagram()* function is externally defined.

```
extern void sendDatagram(char *data, int size);
```

Then, we need to define the user defined C code for both *openM()* and *closeM()*. We also define constant strings in order to uniquely identifiydatagram packets sent to the GUI. In the code, "10" corresponds to the length of the datagram.

```
const char* openD = "Open Door";
const char* closeD = "Close Door";

void __userImplemented__Door__openM() {
  sendDatagram(openD, 10);
}

void __userImplemented__Door__closeM() {
   sendDatagram(closeD, 10);
}
```

The corresponding packets are also defined in the GUI application e.g. see *MainMicrowave.java*.

5.3 GUI animation

Generate the C code from TTool (be sure to check the "Include user code" option). Start the GUI from a terminal, and then, start the MS application from TTool (you can also start MS from a terminal). You should see the animations of the GUI while the generated application executes. For example, Figure 12 shows the microwave when the door opened, and Figure 13 shows the microwave in heating mode.

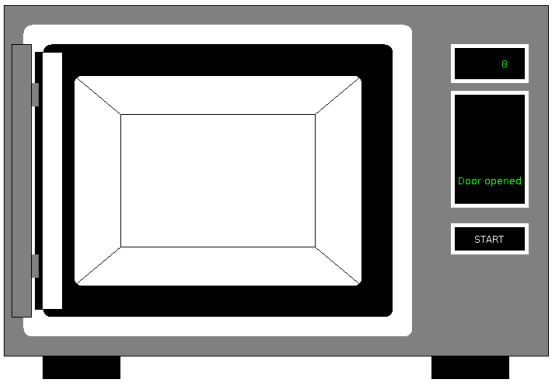

Figure 12: GUI when the door is opened

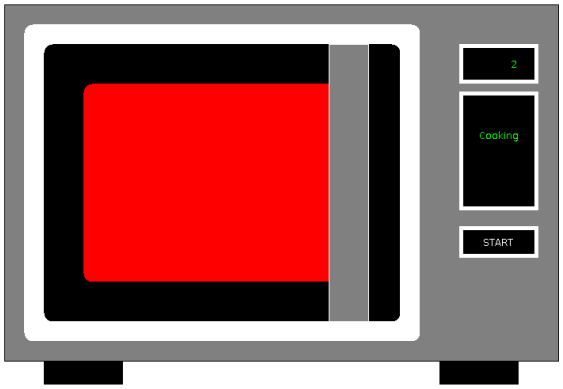

Figure 13: GUI when the microwave is cooking

5.4 GUI actions

5.4.1 GUI side

The GUI contains a "start" button. When the user clicks on this button, the GUI sends a "START" datagram packet to the MS. The GUI is indeed programmed as follows:

- In MainMicrowave.java:

```
public void mouseClicked(MouseEvent e){
        int x = e.getX();
        int y = e.getY();

        System.out.println("Mouse clicked!!!");

        // START?
        if ((x>630)&&(x<720)&&(y>335)&&(y<365)) {
            System.out.println("Mouse clicked on start");
            if (ds != null) {
                ds.sendDatagramTo("START");
            }
            System.out.println("Action on start sent");
        }
}
```

ds.sendDatagram(..) calls a DatagramServer object that sends a datagram to the destination from which it got its first packet.

5.4.2 MS side

On MS side, the global code starts in *user_init()* a thread that handles datagram receiving :

```
pthread_create(&thread__Datagram, NULL, receiveDatagram, NULL);
```

receiveDatagram() waits for datagram packets. When it gets a packet, it checks if it contains the "START" string. If so, it calls *startDatagram()*. This function works as follows:

1. A lock is put on a mutex ("startMutex")
2. The "start" variable is set to 1
3. A call is made on the condition variable "noStart"
4. The mutex is unlock

The *ControlPanel* block defines a *start()* method called before it sends the "startButton" signal, see Figure 14.

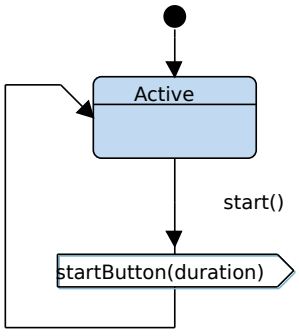

Figure 14: Window of the Graphical User Interface

Thus, when calling start(), the MS wants to wait for the "START" datagram. To do so, the *ControlPanel* block implements "start()" as follows. First, it refers to externally defined elements: the start variable ("start"), the mutex (startMutex) and the condition variable ("startMutex").

```
extern int start;
extern pthread_mutex_t startMutex ;
extern pthread_cond_t  noStart;
```

The method itself works as follows:

1. It puts a lock on the mutex.
2. It waits untils "start" is equal to at least 1. Meanwhile, it waits on the "noStart" condition variable.
3. When "start" is finally equal to 1 or more, it sets "start" to 0.
4. It unlocks the mutex

Then, the execution of the *ControlPanel* block can continue with the sending of the "startButton" signal. Note that this is thanks to the mutex facility that the datagram receinving facility and the *ControlPanel* block cannot modify "start" at the same time.

```
void __userImplemented__ControlPanel__start() {
  pthread_mutex_lock(&startMutex);
  printf("Waiting for next start");
  while(start < 1) {
    pthread_cond_wait(&noStart, &startMutex);
  }
  start = 0;
  pthread_mutex_unlock(&startMutex);
  printf("****** MW can start cooking\n");
}
```

6 Another exampe of advanced model enhancement with user code

We now consider a PressureController (PC) system. This xml TTool model is available via the TTool model repository.

6.1 GUI

The TTool distribution includes an external software which represents a graphical user interface of the pressure controller environment: pressure sensor and alarm. The source code (in Java) of this software is located in "TTool/executablecode/example": *MainPressureController.java*. First compile the java source code, and then execute the GUI, as follows:

```
$javac MainPressureController.java
$java MainPressureController
```

A window similar to the one of Figure 15 should open. This window is not yet animated. To do so, we need to build the PC. The GUI and Pc exchange UDP packets to inform each other about modifications:

- Each time the slider is moved, a datagram packet with the pressure value is sent by GUI to PC.
- Each time the alarm actuator is triggered, a datagram packet is sent by PC to GUI. This packet contains "+" when the alarm must be activated, and "-" when it must be deactivated.

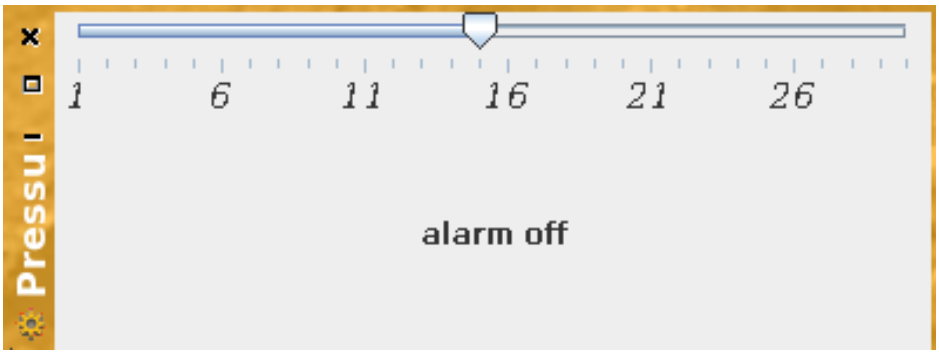

Figure 15: Graphical User Interface of the Pressure Controller System

6.2 Pressure Controller (PC)

The Pressure Controller is build upon a set of blocks representing the system itself, and two blocks representing the environment (the pressure sensor, and the alarm actuator), see Figure 16

Since we would like the pressure controller to interact with its environment, we have customized the methods of *PressureSensor* and *AlarmActuator*.

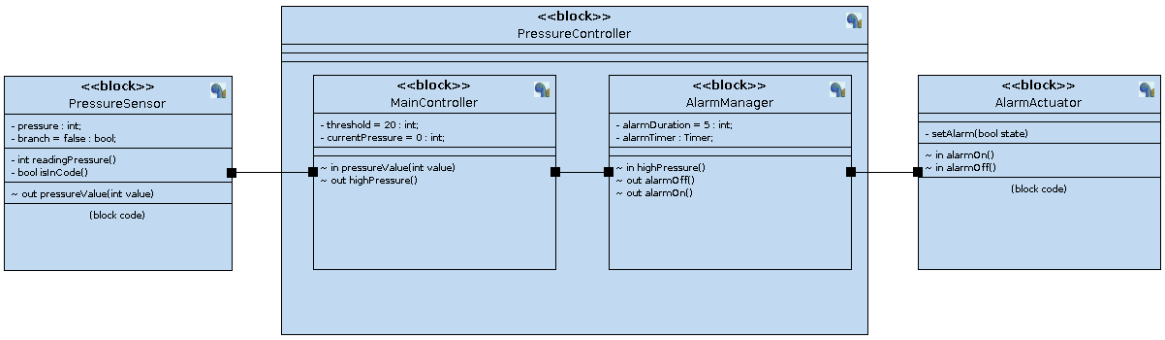

Figure 16: Pressure Controller System: Avatar Design

6.2.1 PressureSensor

The pressure sensor (see Figure 17) monitors the pressure with a period of 1 unit of time. The model does not act the same when simulating the model or executing its code. To do this, we have a "IsInCode" method that is **not** executed when the model is considered for functional simulation or formal verification. Indeed, the "branch" boolean is set to false by default, so the random command is executed (and not the *readingPressure()* method). On the contrary, when executing the C code of this model, *IsIncode()* returns true, so *branch* is equal to true, so the left branch is executed, and the pressure is read (*readingPressure()*).

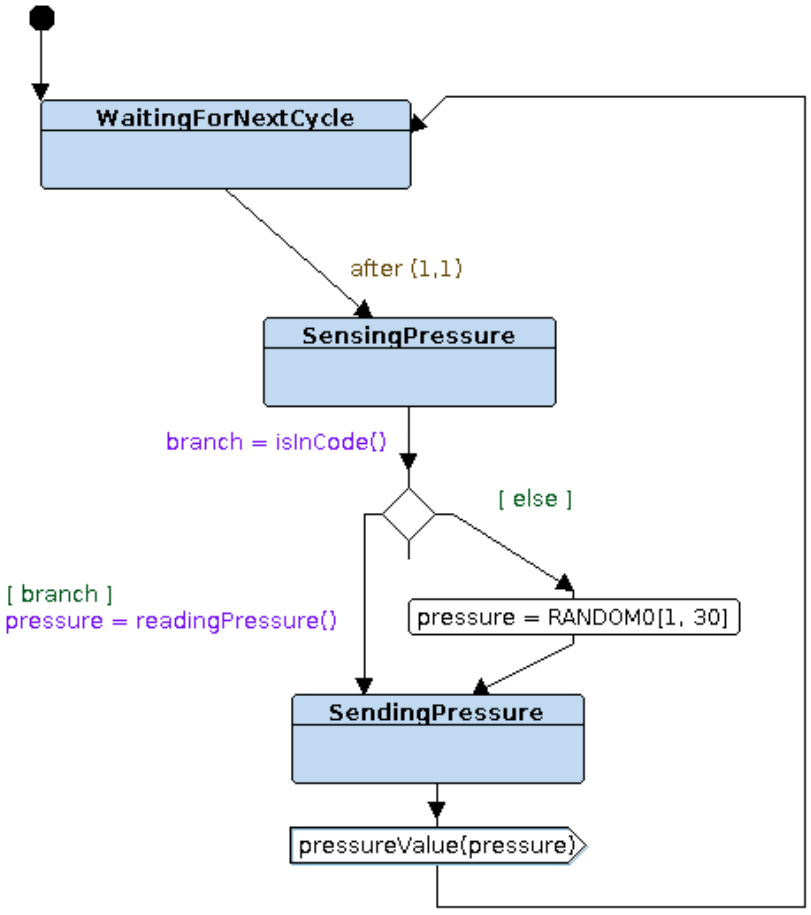

Figure 17: State Machine Diagram of Pressure Sensor

6.2.2 AlarmActuator

The alarm actuator (see Figure 18) waits for an oder to activate or deactivate the alarm. In both case, it calls a user implement method *setAlarm()* with a different parameter ("true", "false").

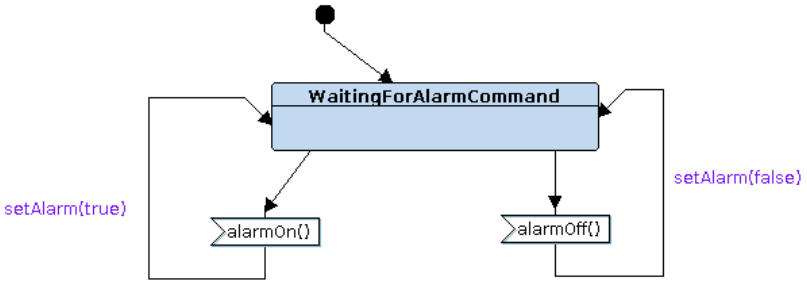

Figure 18: State Machine Diagram of Pressure Sensor

7 Customizing the code generator

We are currently working on a plug-in facility in order to be able to customize the AVATAR-to-C code generator. Send us an email to be informed about updates, or stay connected to `https://twitter.com/TTool_UML_SysML`