UMLEmb:
UML for Embedded Systems
III. System Validation

Ludovic Apvrille,
ludovic.apvrille@telecom-paris.fr

LabSoC, Sophia-Antipolis, France

Goals

Learning objective

- Checking a SysML/AVATAR model against logical errors
- Checking a SysML/AVATAR model against temporal errors

Content

- Simulation
- Formal verification
  - Safety properties, observers
- Prototyping

Outline

Model Simulation

Introduction

Formal verification

Rapid prototyping and code generation

Simulation

Simulation enables model debugging and therefore the early detection of design errors in the life cycle of the system

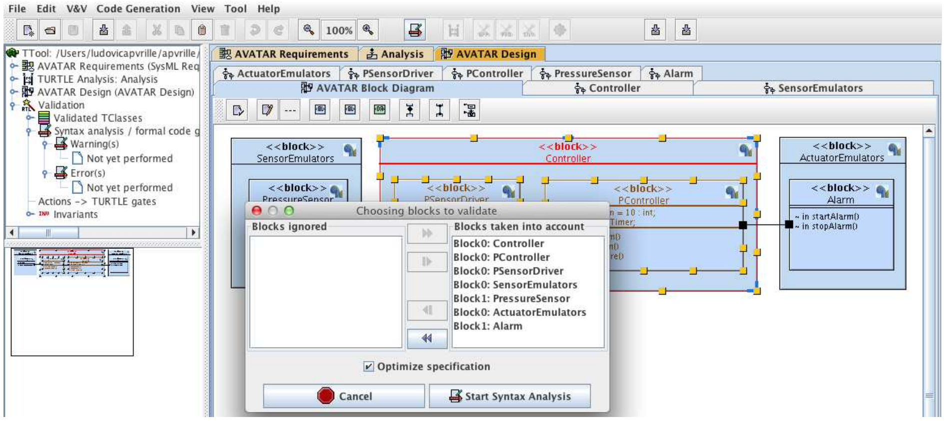
Driving the simulation

- Step by step simulation
- "Random" simulation
- Breakpoints

Tracing the simulation

- Simulation trace in the form of a sequence diagram
- Each already visited branch within each state machine is clearly identified
- Attribute values may be displayed

Checking Design Diagrams against Syntax Errors

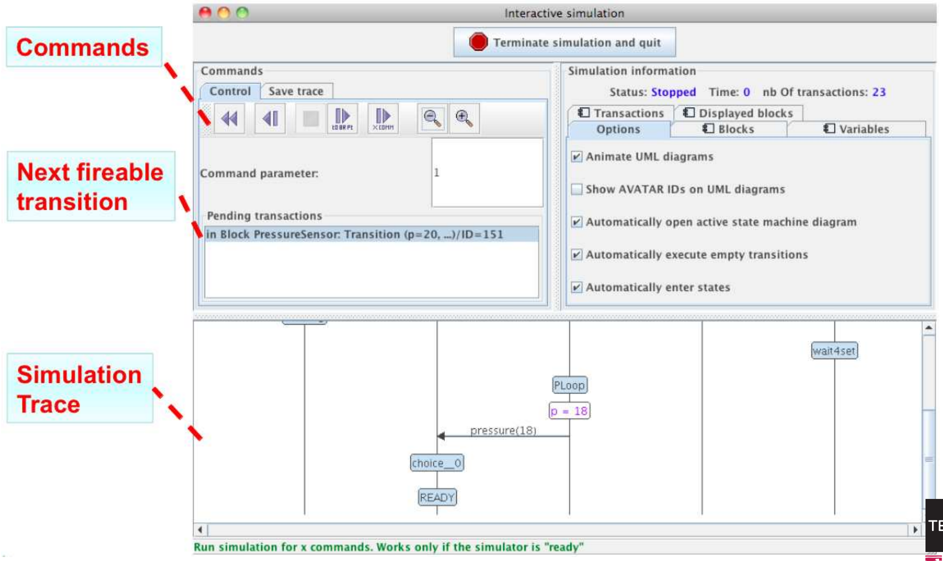


Choosing blocks to validate

Blocks ignored	Blocks taken into account
	Block0: Controller Block0: PController Block0: PSensorDriver Block0: SensorEmulators Block1: PressureSensor Block0: ActuatorEmulators Block1: Alarm

☑ Optimize specification

Simulator Interface


Commands

Next fireable transition

Simulation Trace

Status: Stopped Time: 0 nb Of transactions: 23

Command parameter: 1

Pending transactions

in Block PressureSensor: Transition (p=20, ...)/ID=151

☑ Animate UML diagrams

□ Show AVATAR IDs on UML diagrams

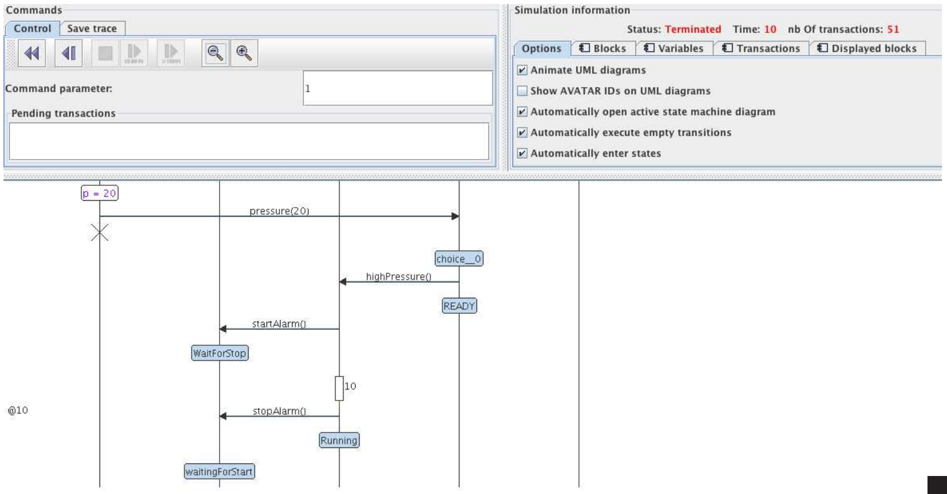
☑ Automatically open active state machine diagram

☑ Automatically execute empty transitions

☑ Automatically enter states

Run simulation for x commands. Works only if the simulator is "ready"

Simulator Trace (Sequence Diagram)



Commands

Control | Save trace

Command parameter: 1

Pending transactions

Simulation information

Status: Terminated Time: 10 nb Of transactions: 51

Options | Blocks | Variables | Transactions | Displayed blocks

☑ Animate UML diagrams

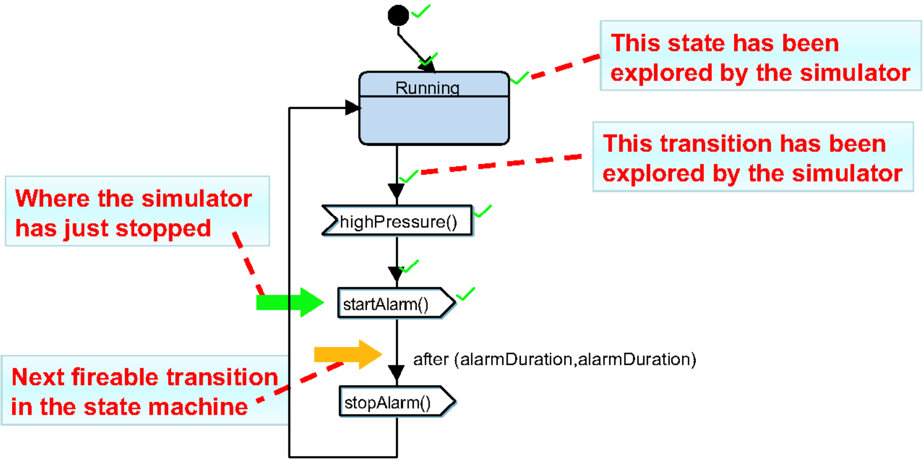
☐ Show AVATAR IDs on UML diagrams

☑ Automatically open active state machine diagram

☑ Automatically execute empty transitions

☑ Automatically enter states

Simulator Trace within a State Machine



This state has been explored by the simulator

This transition has been explored by the simulator

Where the simulator has just stopped

Next fireable transition in the state machine

Outline

Model Simulation

Formal verification

Introduction

Global view in TTool

Properties

Observers

Rapid prototyping and code generation

Introduction to Formal Verification

Formal verification intends to explore all possible system execution paths, and to verify properties along those execution paths

Content

- Brief introduction on formal verification
- How to model and prove safety properties
  - Example: the pressure controller

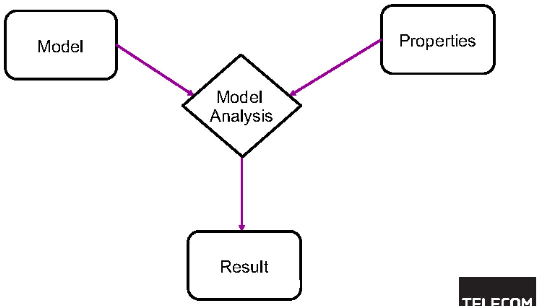
Simulation vs. Formal Verification

Simulation explores execution paths in the model relying on

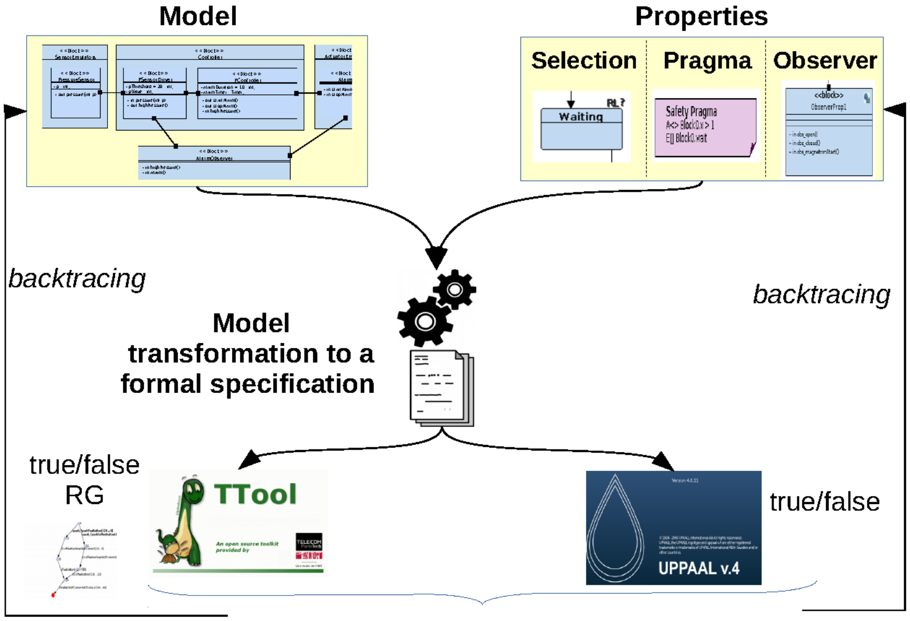
- The experience of the Human who guides the simulation
- Random selection in case of non deterministic choice (several transitions fireable at the same time)

Formal verification

- Formally checks a model of the system against (a subset of) its expected properties
- **Formal verification does not rely on chance but on mathematics!**



Safety Verification in TTool



Properties

Example of general properties

- The system shall always reach a given final state
- From any state the system may return to its initial state
- Deadlock freeness
- No unspecified reception (signals are sent but never received)
- No livelock (systems cannot exit given routines)
- Never used modeling elements (transitions/states are not reachable)

Properties (Cont.)

Specific properties

E.g. "At any time, one station of the LAN holds the token."

Safety: Nothing bad will happen

E.g. "The microwave oven will not start heating as long as the door remains open."

Liveness: "Something good will eventually happen"

E.g. "All connections requests from a pilot will be acknowledged by an air traffic controller."

Reachability Analysis

Principle of reachability graph generation

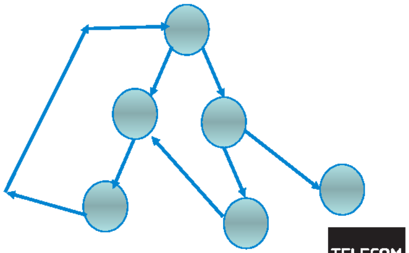
1. From the initial state
2. Search for fireable transitions and create new states
3. Compare new states with existing ones
4. GOTO 2, and take newly created states as initial states

Risk: state explosion problem

- Missing resources (e.g. memory)

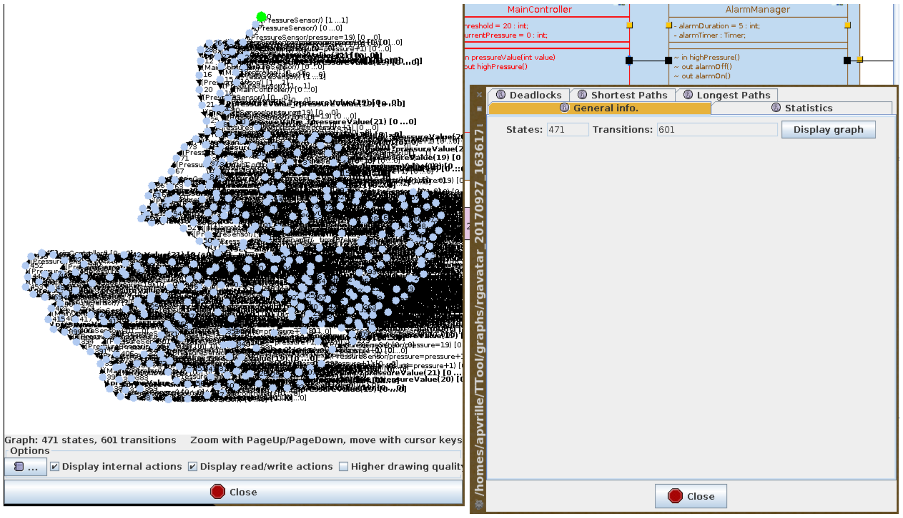
(Some) Solutions

- State coding (hash functions)
- Partial exploration of the graph



Reachability Graph Generation in TTool

- Internal feature
  - "Syntax checking", then "Avatar Model Checker"



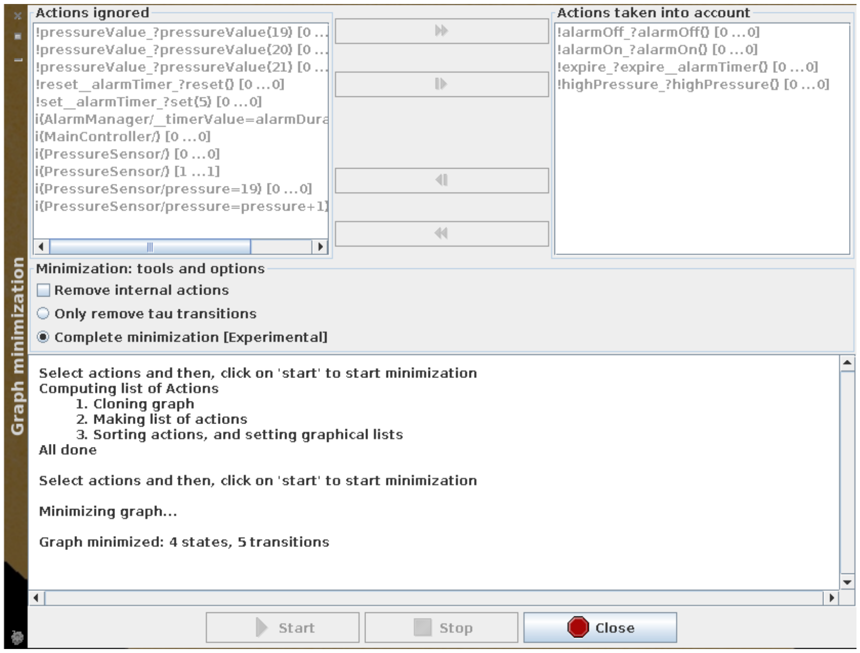
States: 471 Transitions: 601

Graph: 471 states, 601 transitions Zoom with PageUp/PageDown, move with cursor keys

Options

☑ Display internal actions ☑ Display read/write actions □ Higher drawing qualit

Minimization of Reachability Graph



Actions ignored

- !pressureValue_?pressureValue(19) [0 ...
- !pressureValue_?pressureValue(20) [0 ...
- !pressureValue_?pressureValue(21) [0 ...
- !reset__alarmTimer_?reset() [0 ...0]
- !set__alarmTimer_?set(5) [0 ...0]
- i(AlarmManager/__timerValue=alarmDura
- i(MainController/) [0 ...0]
- i(PressureSensor/) [0 ...0]
- i(PressureSensor/) [1 ...1]
- i(PressureSensor/pressure=19) [0 ...0]
- i(PressureSensor/pressure=pressure+1)

Actions taken into account

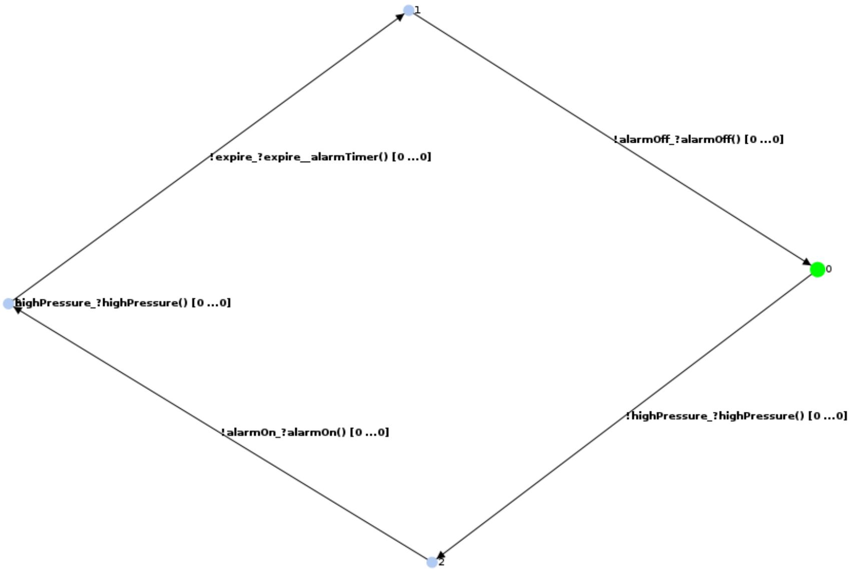
- !alarmOff_?alarmOff() [0 ...0]
- !alarmOn_?alarmOn() [0 ...0]
- !expire_?expire__alarmTimer() [0 ...0]
- !highPressure_?highPressure() [0 ...0]

Minimization: tools and options

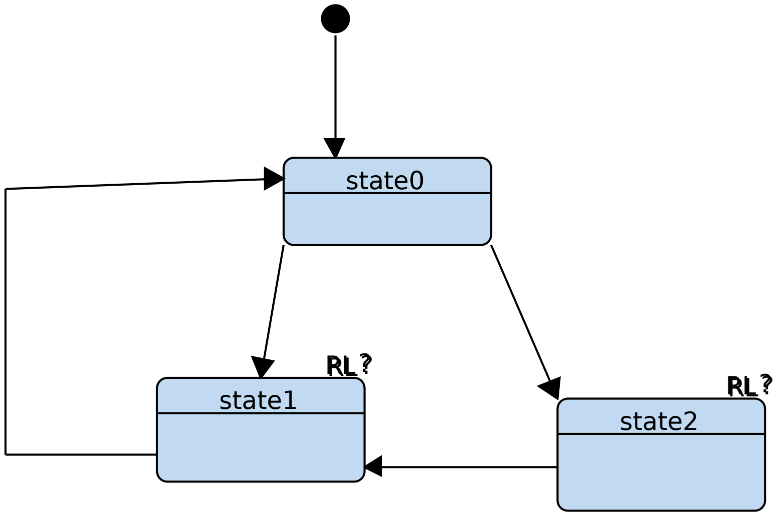
☐ Remove internal actions
○ Only remove tau transitions
● Complete minimization [Experimental]

```
Select actions and then, click on 'start' to start minimization
Computing list of Actions
        1. Cloning graph
        2. Making list of actions
        3. Sorting actions, and setting graphical lists
All done

Select actions and then, click on 'start' to start minimization

Minimizing graph...

Graph minimized: 4 states, 5 transitions
```

Minimized Reachability Graphs

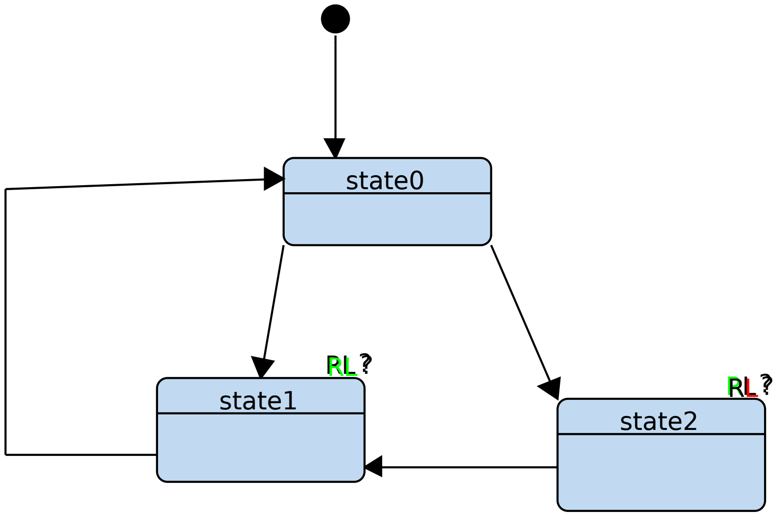

1
!expire_?expire__alarmTimer() [0 ...0]
!alarmOff_?alarmOff() [0 ...0]
0
!highPressure_?highPressure() [0 ...0]
!alarmOn_?alarmOn() [0 ...0]
!highPressure_?highPressure() [0 ...0]
2

Selecting States for Verification



How to activate "RL" in TTool? Simply right-click on a state and select "Check for Reachability / Liveness"

Verification Backtracing

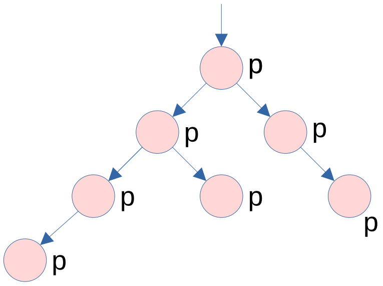


How to obtain this result in TTool? "Syntax checking" then "Safety verification" then check "selected states" in reachability and liveness sections

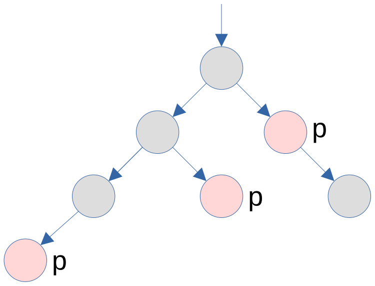
Safety Pragmas

- TCTL = Timed Computation Tree Logic
- Two main operators: A (All paths), E (One path)
- Two modifiers: [] (All states), <> (one state)
- A (boolean) property p

A[] p

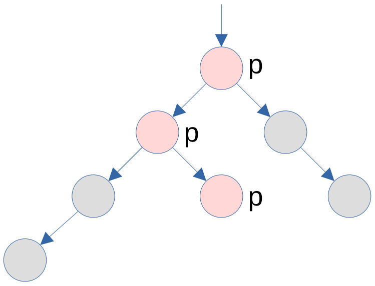


A<> p

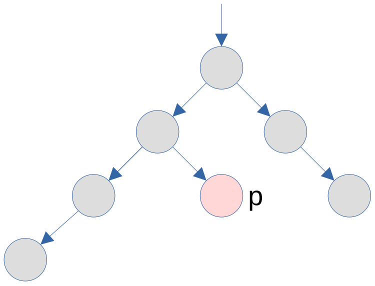


Safety Pragmas (Cont.)

E[] p

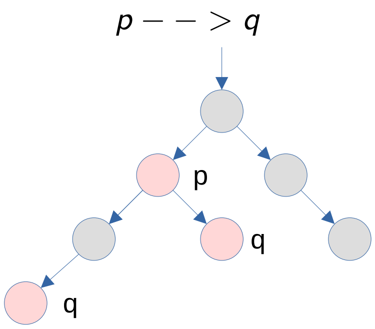


E<> p



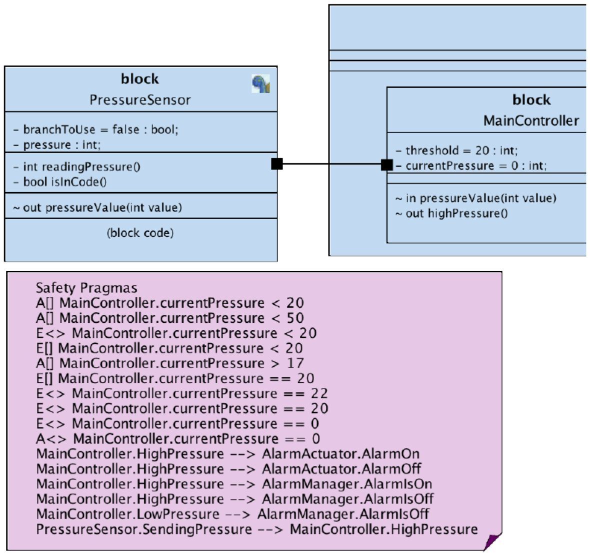
Safety Pragmas (Cont.)

- Leads to
- $p --> q$



Safety pragmas in TTool

Before verification



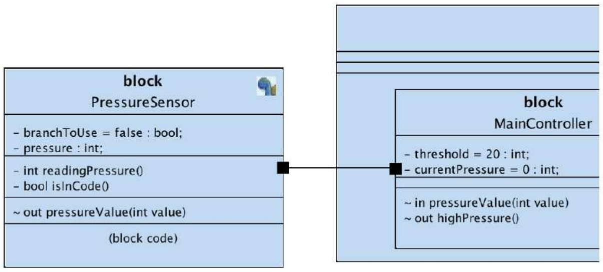
block
PressureSensor
- branchToUse = false : bool;
- pressure : int;
- int readingPressure()
- bool isInCode()
~ out pressureValue(int value)
(block code)

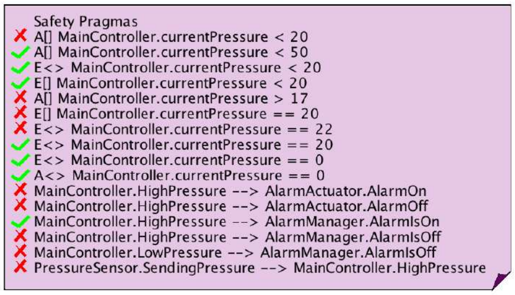
block
MainController
- threshold = 20 : int;
- currentPressure = 0 : int;
~ in pressureValue(int value)
~ out highPressure()

Safety Pragmas
A[] MainController.currentPressure < 20
A[] MainController.currentPressure < 50
E<> MainController.currentPressure < 20
E[] MainController.currentPressure < 20
A[] MainController.currentPressure > 17
E[] MainController.currentPressure == 20
E<> MainController.currentPressure == 22
E<> MainController.currentPressure == 20
E<> MainController.currentPressure == 0
A<> MainController.currentPressure == 0
MainController.HighPressure --> AlarmActuator.AlarmOn
MainController.HighPressure --> AlarmActuator.AlarmOff
MainController.HighPressure --> AlarmManager.AlarmIsOn
MainController.HighPressure --> AlarmManager.AlarmIsOff
MainController.LowPressure --> AlarmManager.AlarmIsOff
PressureSensor.SendingPressure --> MainController.HighPressure

Safety pragmas in TTool (Cont.)

After verification

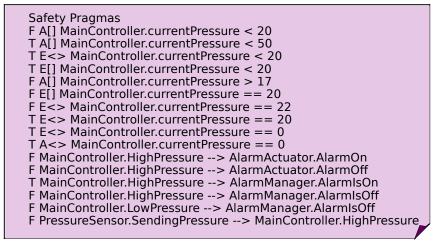




Safety Pragmas
✗ A[] MainController.currentPressure < 20
✓ A[] MainController.currentPressure < 50
✓ E<> MainController.currentPressure < 20
✓ E[] MainController.currentPressure < 20
✗ A[] MainController.currentPressure > 17
✗ E[] MainController.currentPressure == 20
✗ E<> MainController.currentPressure == 22
✓ E<> MainController.currentPressure == 20
✓ E<> MainController.currentPressure == 0
✓ A<> MainController.currentPressure == 0
✗ MainController.HighPressure --> AlarmActuator.AlarmOn
✗ MainController.HighPressure --> AlarmActuator.AlarmOff
✓ MainController.HighPressure --> AlarmManager.AlarmIsOn
✗ MainController.HighPressure --> AlarmManager.AlarmIsOff
✗ MainController.LowPressure --> AlarmManager.AlarmIsOff
✗ PressureSensor.SendingPressure --> MainController.HighPressure

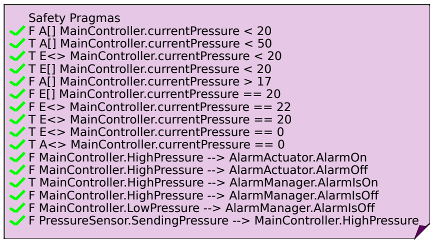
Safety pragmas in TTool (Cont.)

- A designer expects a pragma to be true or to be false
- → Expected result can be indicated with a "T" or "F" before the pragma



Safety Pragmas
F A[] MainController.currentPressure < 20
T A[] MainController.currentPressure < 50
T E<> MainController.currentPressure < 20
T E[] MainController.currentPressure < 20
F A[] MainController.currentPressure > 17
F E[] MainController.currentPressure == 20
F E<> MainController.currentPressure == 22
T E<> MainController.currentPressure == 20
T E<> MainController.currentPressure == 0
T A<> MainController.currentPressure == 0
F MainController.HighPressure --> AlarmActuator.AlarmOn
F MainController.HighPressure --> AlarmActuator.AlarmOff
T MainController.HighPressure --> AlarmManager.AlarmIsOn
F MainController.HighPressure --> AlarmManager.AlarmIsOff
F MainController.LowPressure --> AlarmManager.AlarmIsOff
F PressureSensor.SendingPressure --> MainController.HighPressure

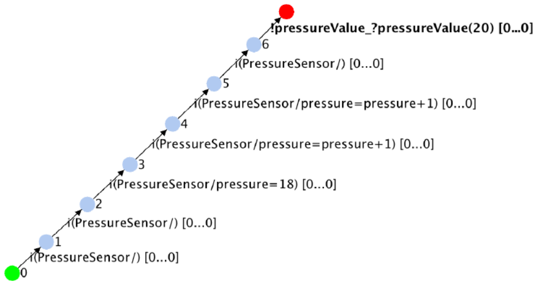
→



Safety Pragmas
✓ F A[] MainController.currentPressure < 20
✓ T A[] MainController.currentPressure < 50
✓ T E<> MainController.currentPressure < 20
✓ T E[] MainController.currentPressure < 20
✓ F A[] MainController.currentPressure > 17
✓ F E[] MainController.currentPressure == 20
✓ F E<> MainController.currentPressure == 22
✓ T E<> MainController.currentPressure == 20
✓ T E<> MainController.currentPressure == 0
✓ T A<> MainController.currentPressure == 0
✓ F MainController.HighPressure --> AlarmActuator.AlarmOn
✓ F MainController.HighPressure --> AlarmActuator.AlarmOff
✓ T MainController.HighPressure --> AlarmManager.AlarmIsOn
✓ F MainController.HighPressure --> AlarmManager.AlarmIsOff
✓ F MainController.LowPressure --> AlarmManager.AlarmIsOff
✓ F PressureSensor.SendingPressure --> MainController.HighPressure

Verification Traces

- Traces intend to explain why a pragma is satisfied or not (e.g. proof or counterexample)
- A trace can be displayed as a graph


!pressureValue_?pressureValue(20) [0...0]
6
i(PressureSensor/) [0...0]
5
i(PressureSensor/pressure=pressure+1) [0...0]
4
i(PressureSensor/pressure=pressure+1) [0...0]
3
i(PressureSensor/pressure=18) [0...0]
2
i(PressureSensor/) [0...0]
1
i(PressureSensor/) [0...0]
0

Trace proving that $A[]MainController.currentPressure < 20$ *is false*

Observer-Guided Verification

Observers

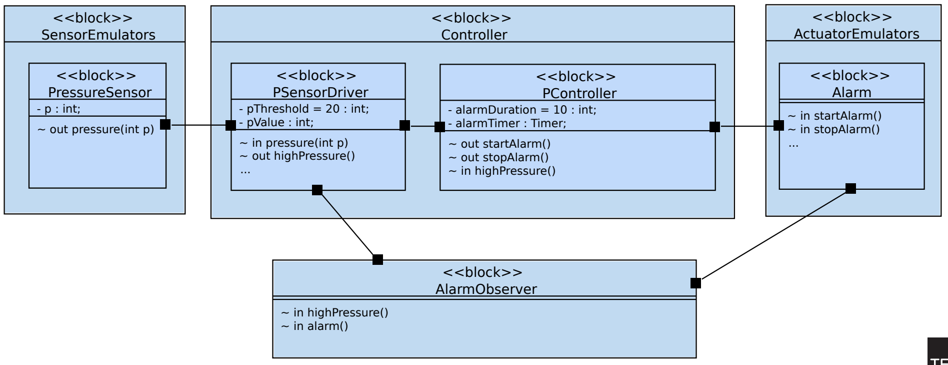
- Expression of (complex) properties within the design
- Observer should have an *error* state whose reachability can be searched for in TTool/UPPAAL
- The observer should remain non-intrusive
  - At least, as long as the observed property is satisfied

Example: Pressure Controller

- Observer that verifies the alarm rings in zero time when a high pressure is detected

Pressure Controller: Design of an Alarm Observer

- An "AlarmObserver" block is added to the design
- AlarmObserver fetches information from the pressure sensor and the alarm



<<block>>
SensorEmulators

<<block>>
PressureSensor
- p : int;
~ out pressure(int p)

<<block>>
Controller

<<block>>
PSensorDriver
- pThreshold = 20 : int;
- pValue : int;
~ in pressure(int p)
~ out highPressure()
...

<<block>>
PController
- alarmDuration = 10 : int;
- alarmTimer : Timer;
~ out startAlarm()
~ out stopAlarm()
~ in highPressure()

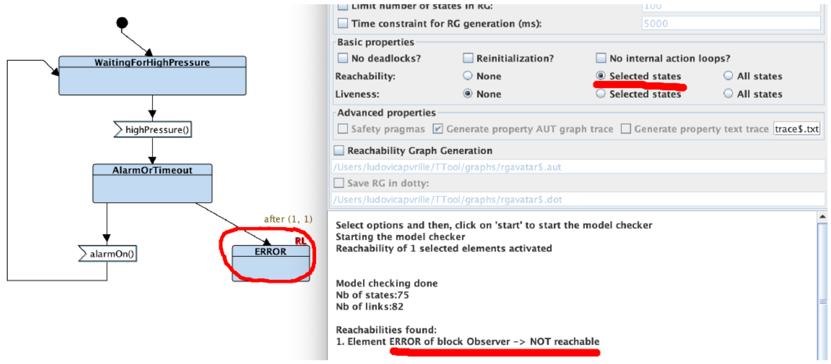
<<block>>
ActuatorEmulators

<<block>>
Alarm
~ in startAlarm()
~ in stopAlarm()
...

<<block>>
AlarmObserver
~ in highPressure()
~ in alarm()

Pressure Controller: Design of an Alarm Observer (Cont.)

- Whenever the observer gets a *highPressure* signal, it goes into the state ERROR after 1 unit of time if it hasn't received yet an *alarm* signal
- The reachability of ERROR is searched for



□ Limit number of states in RG: 100
□ Time constraint for RG generation (ms): 5000

Basic properties
□ No deadlocks? □ Reinitialization? □ No internal action loops?
Reachability: ○ None ● Selected states ○ All states
Liveness: ● None ○ Selected states ○ All states

Advanced properties
□ Safety pragmas ☑ Generate property AUT graph trace □ Generate property text trace trace$.txt

□ Reachability Graph Generation
/Users/ludovicapvrille/TTool/graphs/rgavatar$.aut
□ Save RG in dotty:
/Users/ludovicapvrille/TTool/graphs/rgavatar$.dot

```
Select options and then, click on 'start' to start the model checker
Starting the model checker
Reachability of 1 selected elements activated


Model checking done
Nb of states:75
Nb of links:82

Reachabilities found:
1. Element ERROR of block Observer -> NOT reachable
```

Outline

Model Simulation

Formal verification

Rapid prototyping and code generation

- Code generation
- Virtual prototyping
- Customizing code generation in TTool

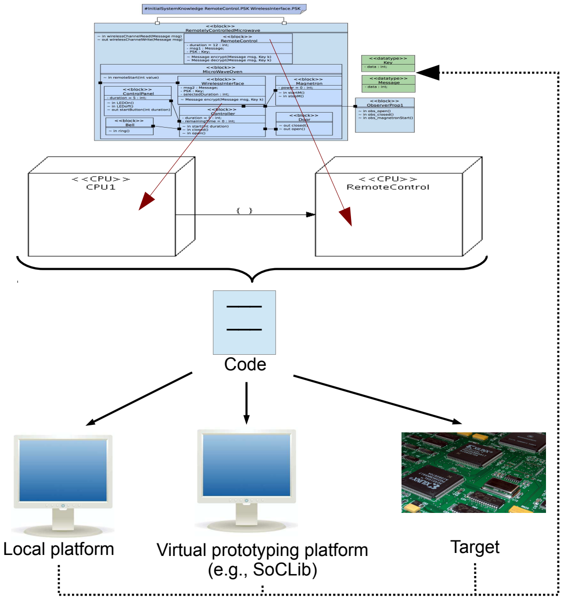
Introduction to Rapid Prototyping

Rapid prototyping intends to experiment with the execution of code produced from models

Content

- Overview of code generation in TTool
- Transformation of AVATAR design diagrams into executable code
- Application to a microwave oven

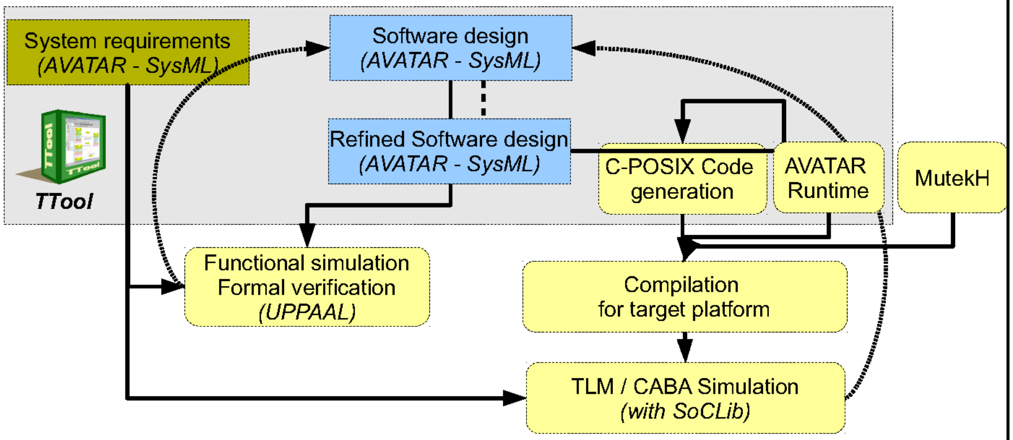
Code Generation: Overview



Principle of Code Generation

- Only AVATAR design diagrams are taken into account
- Generated code relies on POSIX threads
  - One thread per block
- Synchronous communications between blocks is implemented in the AVATAR runtime with POSIX mutex
  - Asynchronous communications relies on linked lists managed in the AVATAR runtime
  - Time is handled based on POSIX *clock_gettime*() with *CLOCK_REALTIME* option
  - . . .

Virtual Prototyping: Method



Virtual Prototyping Steps

1. Model refinement
2. Selection of an OS, setting of options of this OS (scheduling algorithm, . . .)
3. Selection of a hardware platform, and selection of a task allocation scheme
4. Code generation (press-button approach)
5. Manual code improvement - Code might also be manually added at model level
6. Code compilation and linkage with OS
7. Simulation platform boots the OS and executes the code
8. Execution analysis: directly in TTool (sequence diagram), with debuggers (e.g., *gdb*), or with custom graphical interfaces

Support: SoCLib and MutekH

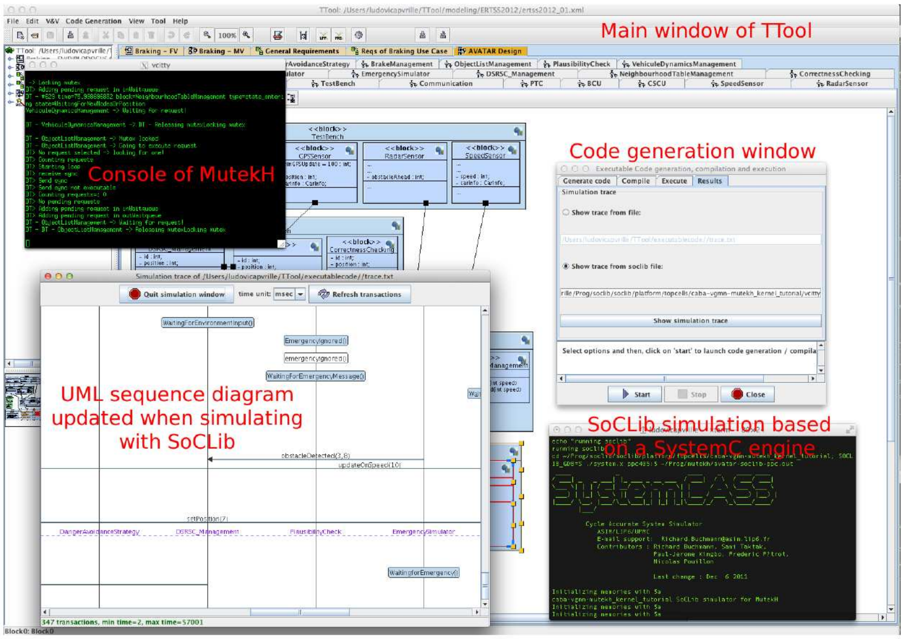
Hardware platform simulator: SoCLib (www.soclib.fr)

- Virtual prototyping of complex Systems-on-Chip
- Supports several models of processors, buses, memories
  - Example of CPUs: MIPS, ARM, SPARC, Nios2, PowerPC
- Two sets of simulation models:
  - TLM = Transaction Level Modeling
  - CABA = Cycle Accurate Bit Accurate

Embedded Operating System: MutekH (www.mutekh.org)

- Natively handles heterogeneous multiprocessor platforms
- POSIX threads support
- Note: any Operating System supporting POSIX threading and that can be compiled for SoCLib could be used

Virtual Prototyping: Graphical Environment



```
-> Locking mutex
DT> Adding pending request in inWaitqueue
DT - #629 time=75.998696832 block=NeighbourhoodTableManagement type=state_entering state=WaitingForNewNodesOrPosition
VehiculeDynamicsManagement -> Waiting for request!

DT - VehiculeDynamicsManagement -> DT - Releasing mutexLocking mutex

DT - ObjectListManagement -> Mutex locked
DT - ObjectListManagement -> Going to execute request
DT> No request selected -> looking for one!
DT> Counting requests
DT> Starting loop
DT> receive sync
DT> Send sync
DT> Send sync not executable
DT> Counting requests=; 0
DT> No pending requests
DT> Adding pending request in inWaitqueue
DT> Adding pending request in outWaitqueue
DT - ObjectListManagement -> Waiting for request!
DT - DT - ObjectListManagement -> Releasing mutexLocking mutex
```

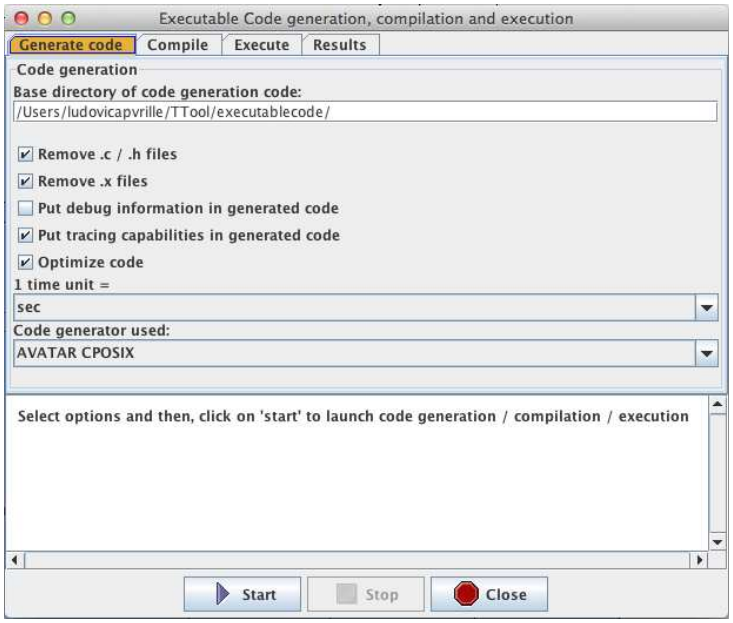
Simulation trace of /Users/ludovicapvrille/TTool/executablecode/trace.txt

347 transactions, min time=2, max time=57001

Select options and then, click on 'start' to launch code generation / compila

```
echo "running soclib"
running soclib
cd ~/Prog/soclib/soclib/platform/topcells/caba-vgmn-mutekh_kernel_tutorial; SOCLIB_GDB=S ./system.x ppc405:5 ~/Prog/mutekh/avatar-soclib-ppc.out

        Cycle Accurate System Simulator
           ASIM/LIP6/UPMC
           E-mail support:   Richard.Buchmann@asim.lip6.fr
           Contributors : Richard Buchmann, Sami Taktak,
                          Paul-Jerome Kingbo, Frederic P?trot,
                          Nicolas Pouillon

                          Last change : Dec  6 2011

Initializing memories with 5a
caba-vgmn-mutekh_kernel_tutorial SoCLib simulator for MutekH
Initializing memories with 5a
Initializing memories with 5a
```

(Virtual) Prototyping: Code Generation



Code generation

Base directory of code generation code: /Users/ludovicapvrille/TTool/executablecode/

☑ Remove .c / .h files

☑ Remove .x files

☐ Put debug information in generated code

☑ Put tracing capabilities in generated code

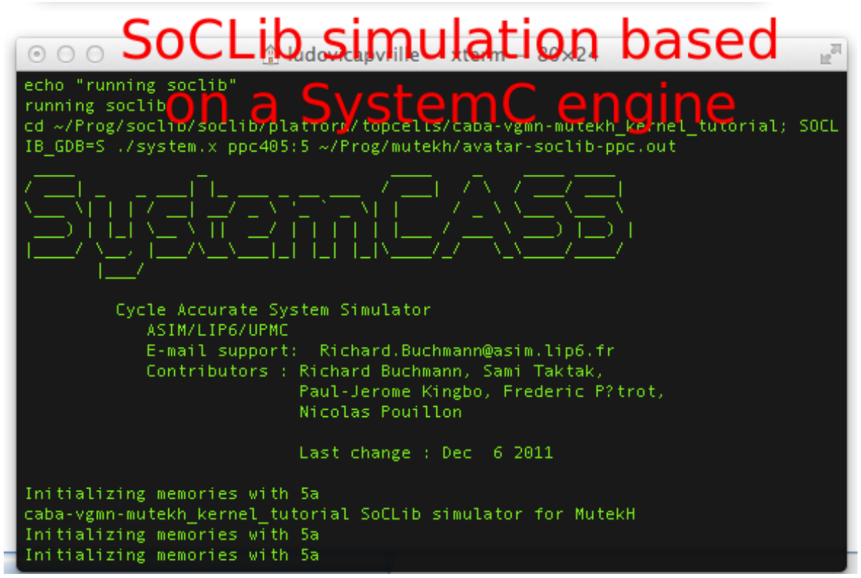
☑ Optimize code

1 time unit = sec

Code generator used: AVATAR CPOSIX

Select options and then, click on 'start' to launch code generation / compilation / execution

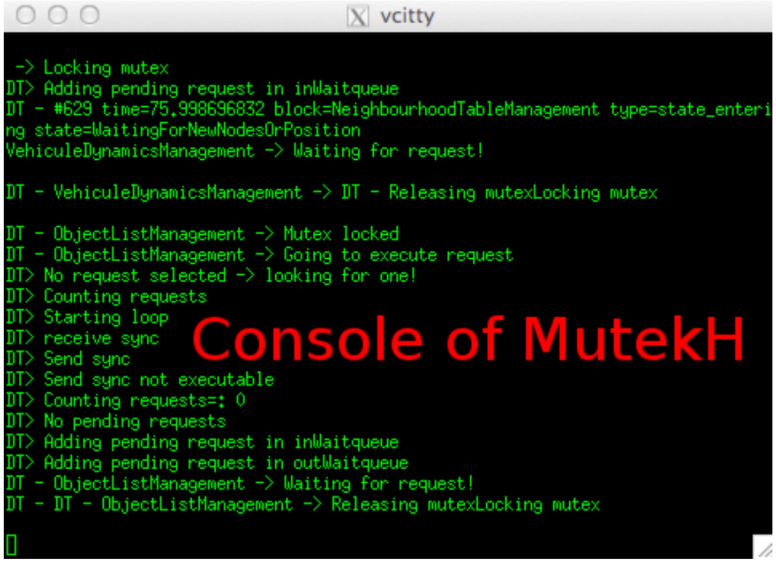
Virtual Prototyping: SocLib Simulation



SoCLib simulation based on a SystemC engine

```
echo "running soclib"
running soclib
cd ~/Prog/soclib/soclib/platform/topcells/caba-vgmn-mutekh_kernel_tutorial; SOCL
IB_GDB=S ./system.x ppc405:5 ~/Prog/mutekh/avatar-soclib-ppc.out
 ____            _                  ____    _    ____ ____
/ ___| _   _ ___| |_ ___ _ __ ___  / ___|  / \  / ___/ ___|
\___ \| | | / __| __/ _ \ '_ ` _ \| |     / _ \ \___ \___ \
 ___) | |_| \__ \ ||  __/ | | | | | |___ / ___ \ ___) |__) |
|____/ \__, |___/\__\___|_| |_| |_|\____/_/   \_\____/____/
       |___/

         Cycle Accurate System Simulator
            ASIM/LIP6/UPMC
            E-mail support:  Richard.Buchmann@asim.lip6.fr
            Contributors : Richard Buchmann, Sami Taktak,
                           Paul-Jerome Kingbo, Frederic P?trot,
                           Nicolas Pouillon

                           Last change : Dec  6 2011

Initializing memories with 5a
caba-vgmn-mutekh_kernel_tutorial SoCLib simulator for MutekH
Initializing memories with 5a
Initializing memories with 5a
```

Virtual Prototyping: Console

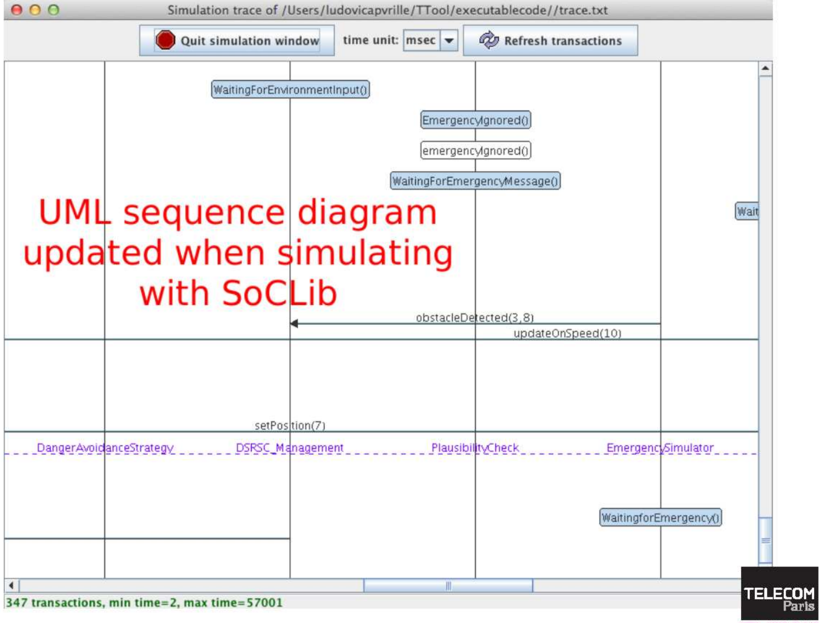


```
 -> Locking mutex
DT> Adding pending request in inWaitqueue
DT - #629 time=75.998696832 block=NeighbourhoodTableManagement type=state_enteri
ng state=WaitingForNewNodesOrPosition
VehiculeDynamicsManagement -> Waiting for request!

DT - VehiculeDynamicsManagement -> DT - Releasing mutexLocking mutex

DT - ObjectListManagement -> Mutex locked
DT - ObjectListManagement -> Going to execute request
DT> No request selected -> looking for one!
DT> Counting requests
DT> Starting loop
DT> receive sync
DT> Send sync
DT> Send sync not executable
DT> Counting requests=: 0
DT> No pending requests
DT> Adding pending request in inWaitqueue
DT> Adding pending request in outWaitqueue
DT - ObjectListManagement -> Waiting for request!
DT - DT - ObjectListManagement -> Releasing mutexLocking mutex
```

Console of MutekH

(Virtual) Prototyping: Trace

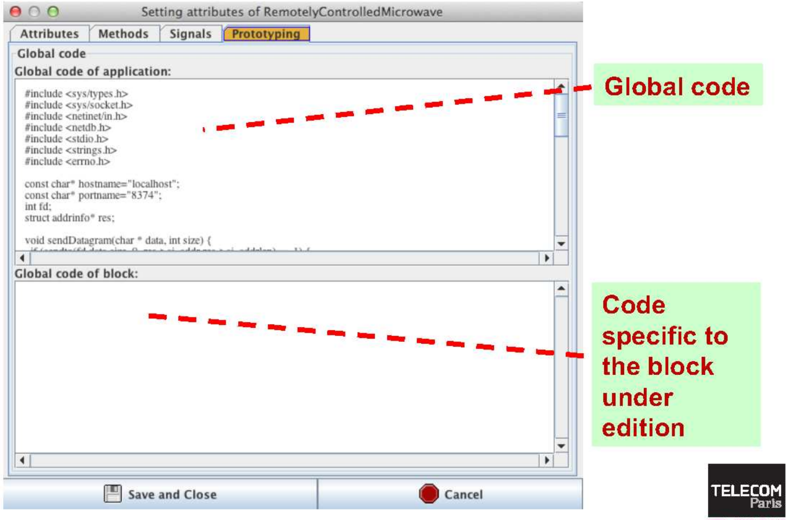
TTool displays execution traces in a sequence diagram



UML sequence diagram updated when simulating with SoCLib

Customizing Generated Code with Your Own Code: Application and Block Code

- Global code of the application
  - Inclusion of header files, global variables, . . .
- Code global to one given block



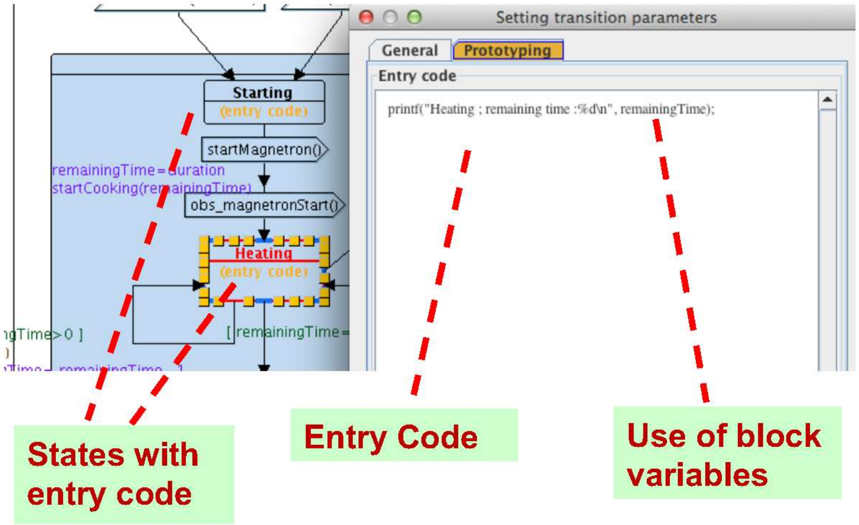
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <strings.h>
#include <errno.h>

const char* hostname="localhost";
const char* portname="8374";
int fd;
struct addrinfo* res;

void sendDatagram(char * data, int size) {
```

Global code

Code specific to the block under edition

Customizing Generated Code with Your Own Code: State Entry Code

- Code executed whenever a state is reached



```
printf("Heating ; remaining time :%d\n", remainingTime);
```

States with entry code

Entry Code

Use of block variables

Use of Customized Generated Code

Console debug

- Using e.g. *printf()* function

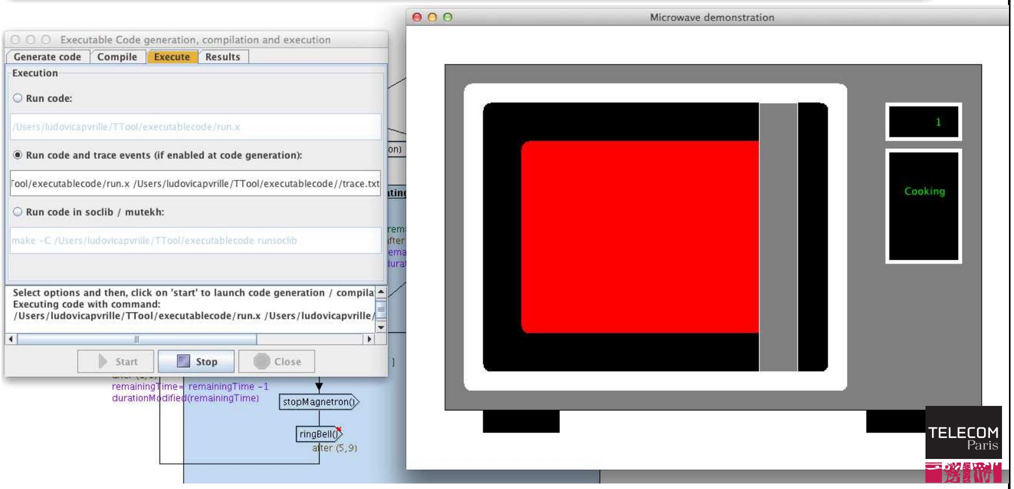
Connection to a graphical interface

- Piloting the code with a graphical interface
- Visualizing what's happening in the executed code
- Connection to graphical interface via, e.g., *sockets*

Use of Customized Generated Code (Cont)

Graphical interface for the microwave oven

- Socket connection to a graphical interface programmed in Java



Execution

○ Run code:

/Users/ludovicapvrille/TTool/executablecode/run.x

● Run code and trace events (if enabled at code generation):

ool/executablecode/run.x /Users/ludovicapvrille/TTool/executablecode//trace.txt

○ Run code in soclib / mutekh:

make -C /Users/ludovicapvrille/TTool/executablecode runsoclib

```
Select options and then, click on 'start' to launch code generation / compila
Executing code with command:
/Users/ludovicapvrille/TTool/executablecode/run.x /Users/ludovicapvrille/
```

remainingTime= remainingTime -1
durationModified(remainingTime)