



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

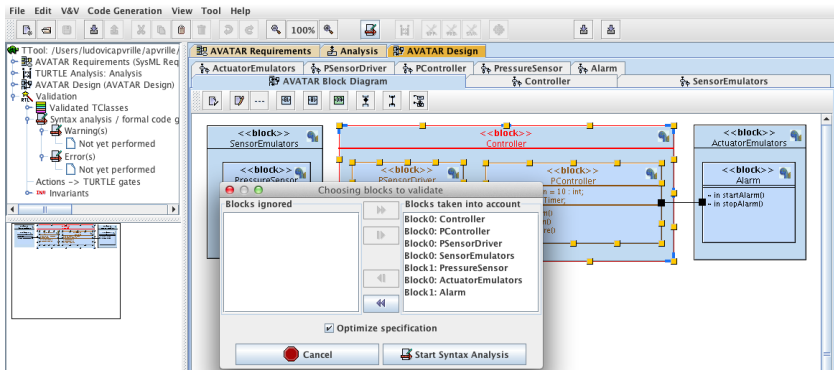
Driving the simulation

- 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



Simulator Interface

Commands

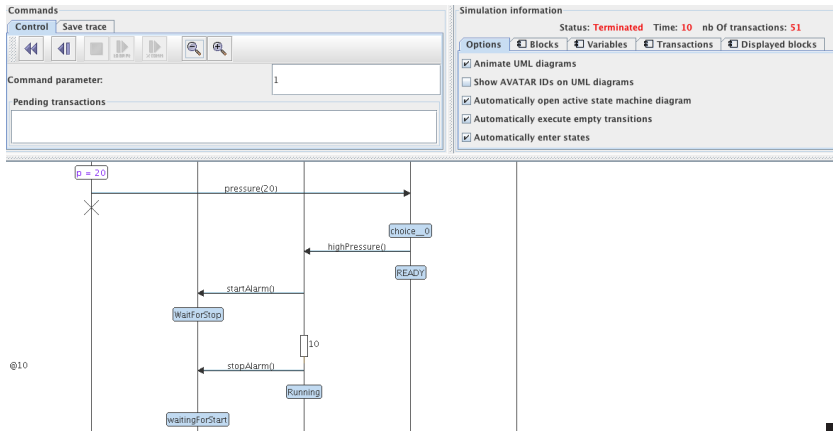
Next fireable transition

Simulation Trace

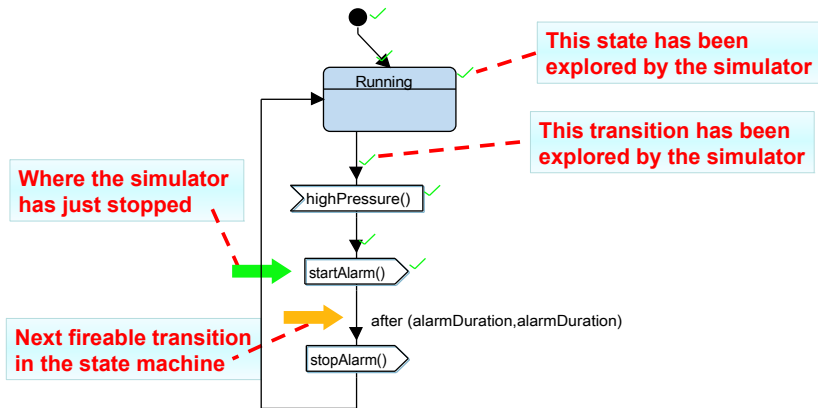
The screenshot shows the 'Interactive simulation' window. At the top, there is a 'Terminate simulation and quit' button. Below it, the 'Commands' panel includes a 'Control' sub-panel with navigation icons (back, forward, stop, play, refresh) and a 'Command parameter' input field containing the value '1'. A 'Pending transactions' list shows 'in Block PressureSensor: Transition (p=20, ...)/ID=151'. The 'Simulation information' panel on the right shows 'Status: Stopped', 'Time: 0', and 'nb Of transactions: 23'. It also has tabs for 'Options', 'Blocks', and 'Variables'. The 'Options' tab is active, showing several checked options: 'Animate UML diagrams', 'Automatically open active state machine diagram', 'Automatically execute empty transitions', and 'Automatically enter states'. The 'Displayed blocks' tab shows a state machine trace with lifelines for 'choice_0' and 'PLoop'. The trace shows a 'READY' state on 'choice_0', a 'pressure(18)' message to 'PLoop', and a 'p = 18' message back to 'choice_0'. A 'wait4set' block is also visible on the right lifeline.

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

Simulator Trace (Sequence Diagram)



Simulator Trace within a 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

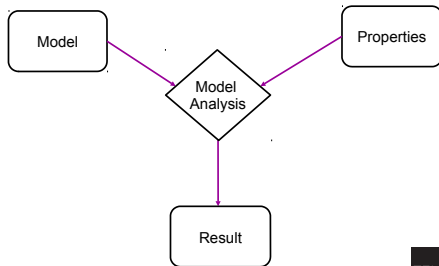
Simulation vs. Formal Verification

Simulation explores execution paths in the model relying on

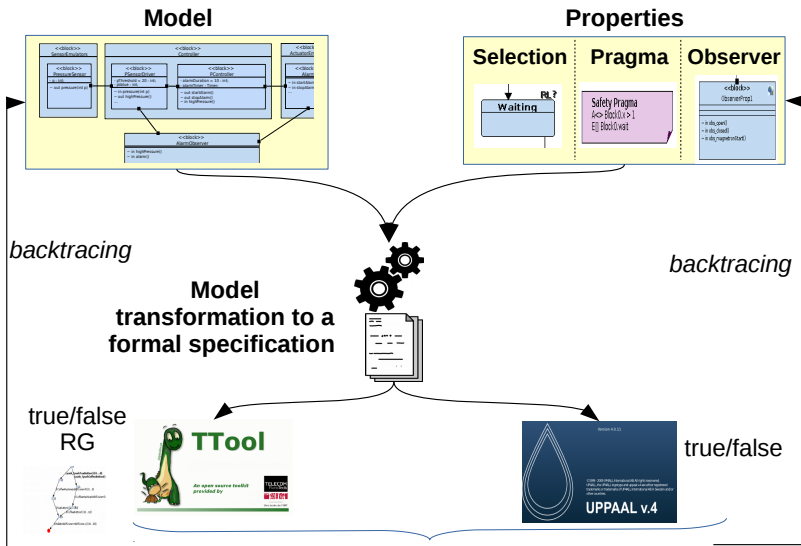
- The experience of the Human who guides the simulation
- Random selection in case of non deterministic choice (several transitions fireable at the same time)

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

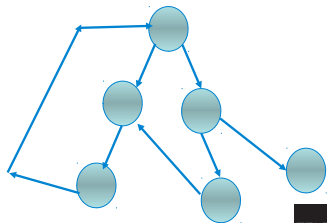
1. From the initial state
2. Search for fireable transitions and create new states
3. Compare new states with existing ones
4. GOTO 2, and take newly created states as initial states

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"

The screenshot displays the TTool interface. On the left, a large, dense reachability graph is shown, consisting of numerous blue circular nodes connected by black lines. The graph represents the state space of the system. At the top of the graph, several nodes are labeled with their state identifiers, such as 'PressureSensor() [1...1]', 'PressureSensor() [0...0]', and 'PressureSensor() [0...0]'. Below the graph, a control panel shows 'Graph: 471 states, 601 transitions' and 'Zoom with PageUp/PageDown, move with cursor keys'. There are also checkboxes for 'Display internal actions', 'Display read/write actions', and 'Higher drawing quality', along with a 'Close' button.

On the right side of the interface, a state transition diagram is visible. It shows two main components: 'MainController' and 'AlarmManager'. The 'MainController' has variables 'pressure' and 'alarm' and methods 'inPressure()' and 'outPressure()'. The 'AlarmManager' has variables 'alarmDuration' and 'alarmTimer' and methods 'inHighPressure()' and 'outAlarmOn()'. The diagram illustrates the interactions between these components, with arrows indicating the flow of control and data.

Below the state transition diagram, a 'General info.' panel displays 'States: 471' and 'Transitions: 601'. There is a 'Display graph' button and a 'Close' button at the bottom right of the interface.

Minimization of Reachability Graph

Actions ignored

```
!pressureValue_?pressureValue(19) [0 ...0]
!pressureValue_?pressureValue(20) [0 ...0]
!pressureValue_?pressureValue(21) [0 ...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...

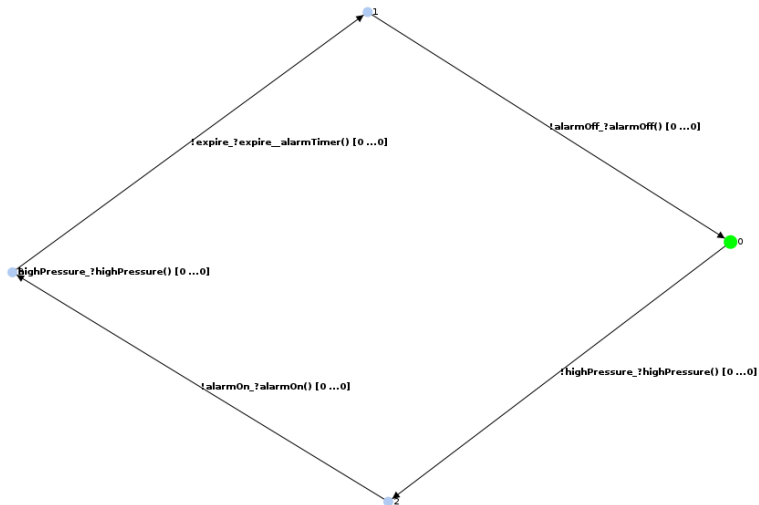
Graph minimized: 4 states, 5 transitions

▶ Start

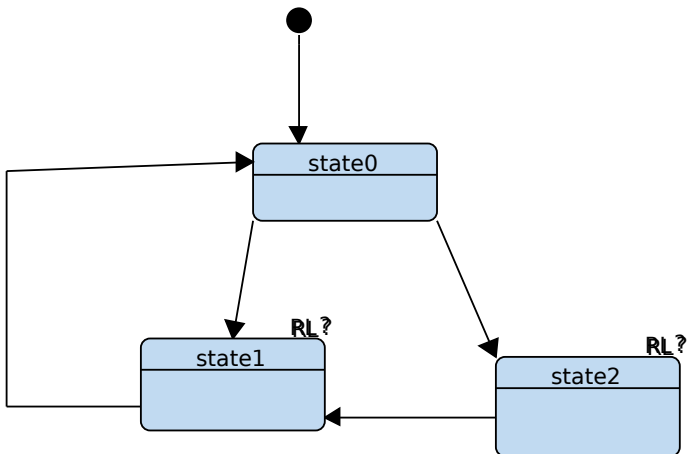
■ Stop

● Close

Minimized Reachability Graphs

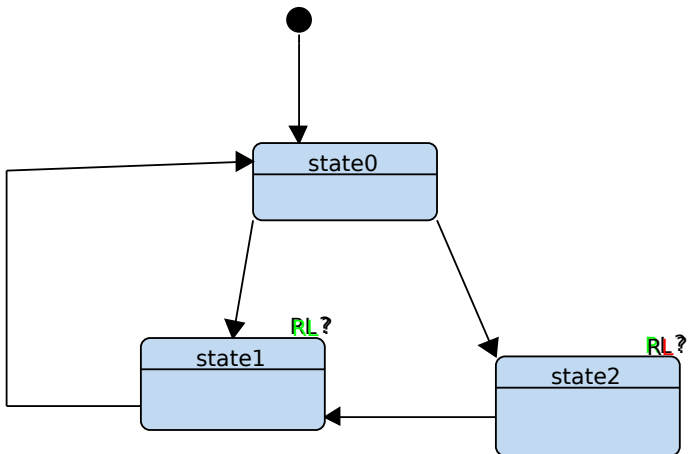


Selecting States for Verification



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

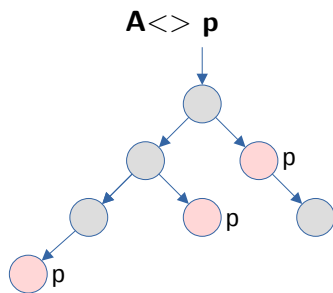
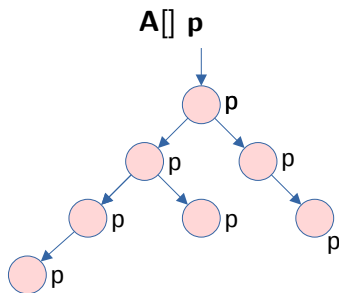
Verification Backtracing



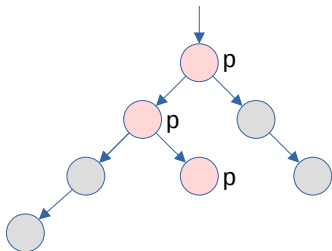
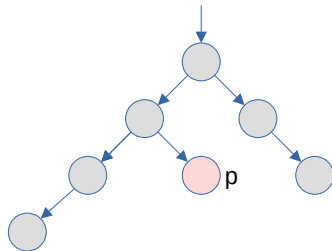
How to obtain this result in TTool? "Syntax checking" then "Safety verification" then check "selected states" in reachability and liveness sections

Safety Pragmas

- TCTL = Timed Computation Tree Logic
- Two main operators: A (All paths), E (One path)
- Two modifiers: \square (All states), $\langle \rangle$ (one state)
- A (boolean) property p

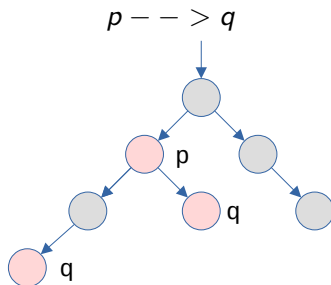


Safety Pragmas (Cont.)

 $E[] p$

 $E<> p$


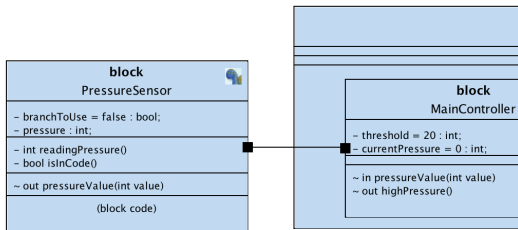
Safety Pragmas (Cont.)

- Leads to
- $p \text{ -- } > q$



Safety pragmas in TTool

Before verification



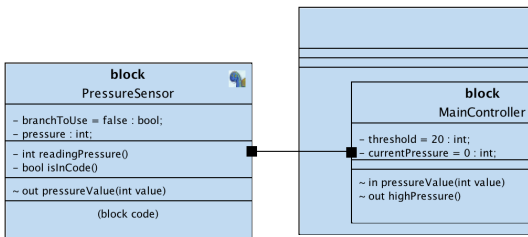
Safety Pragmas

```

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.AlarmsOn
MainController.HighPressure --> AlarmManager.AlarmsOff
MainController.LowPressure --> AlarmManager.AlarmsOff
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.AlarmsOn
- ✗ MainController.HighPressure --> AlarmManager.AlarmsOff
- ✗ MainController.LowPressure --> AlarmManager.AlarmsOff
- ✗ 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.AlarmsOn
F MainController.HighPressure --> AlarmManager.AlarmsOff
F MainController.LowPressure --> AlarmManager.AlarmsOff
F PressureSensor.SendingPressure --> MainController.HighPressure
  
```



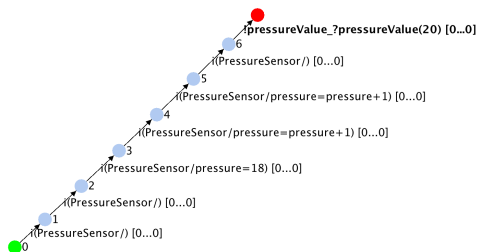
Safety Pragmas

```

✓ 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.AlarmsOn
✓ F MainController.HighPressure --> AlarmManager.AlarmsOff
✓ F MainController.LowPressure --> AlarmManager.AlarmsOff
✓ 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



Trace proving that \forall MainController.currentPressure < 20 is false

Observer-Guided Verification

Observers

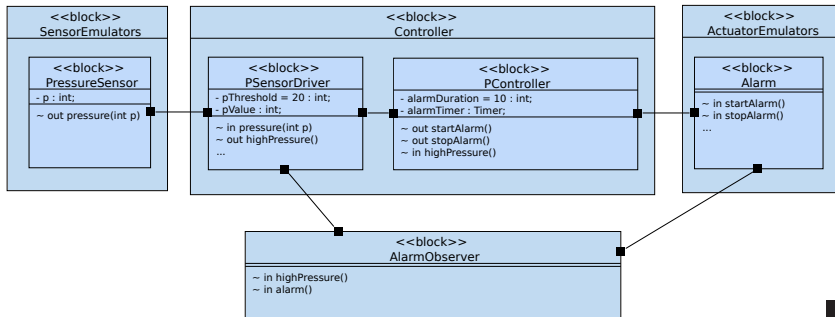
- Expression of (complex) properties within the design
- Observer should have an *error* state whose reachability can be searched for in TTool/UPPAAL
- The observer should remain non-intrusive
 - At least, as long as the observed property is satisfied

Example: Pressure Controller

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

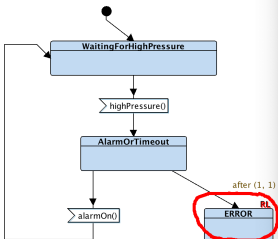
Pressure Controller: Design of an Alarm Observer

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



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 `trace5.txt`

Reachability Graph Generation
 /Users/ludovicaprille/TTool/graphs/rgavatar\$.aut
 Save RG in dotty:
 /Users/ludovicaprille/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

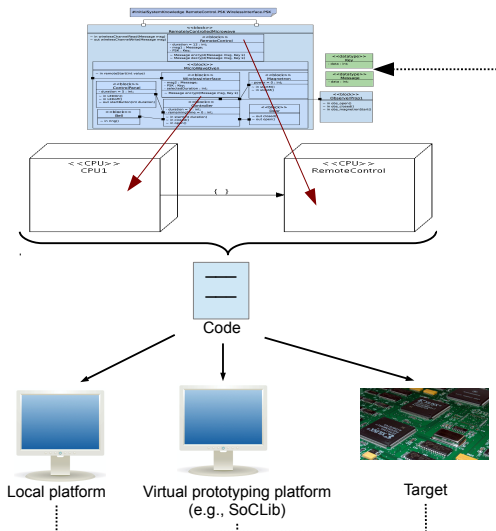
Introduction to Rapid Prototyping

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

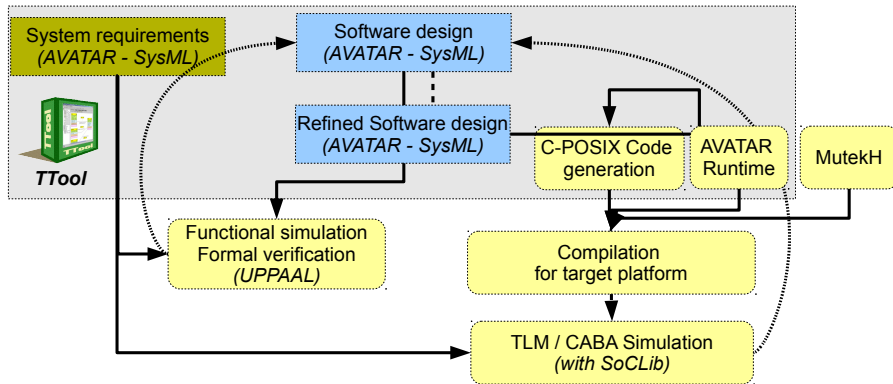
Code Generation: Overview



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

TTool /Users/ladovicapville/TTool/modeling/ERTSS2012/ertss2012_01.xml

File Edit V&V Code Generation View Tool Help

100%

Braking - FV Braking - MV General Requirements Reqs of Braking Use Case AVATAR Design

AvoidanceStrategy BrakeManagement ObjectListManagement PlausibilityCheck VehicleDynamicsManagement
 Simulator EmergencySimulator DSRSK_Management NeighbourhoodTableManagement CorrectnessChecking
 TestBench Communication PTC BCU CSCU SpeedSensor RadarSensor

Console of MutekH

Main window of TTool

Code generation window

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

UML sequence diagram updated when simulating with SoCLib

SoCLib simulation based on a SystemC engine

SystemC ASS

TELECOM Paris

38/46

Une école de l'IMT

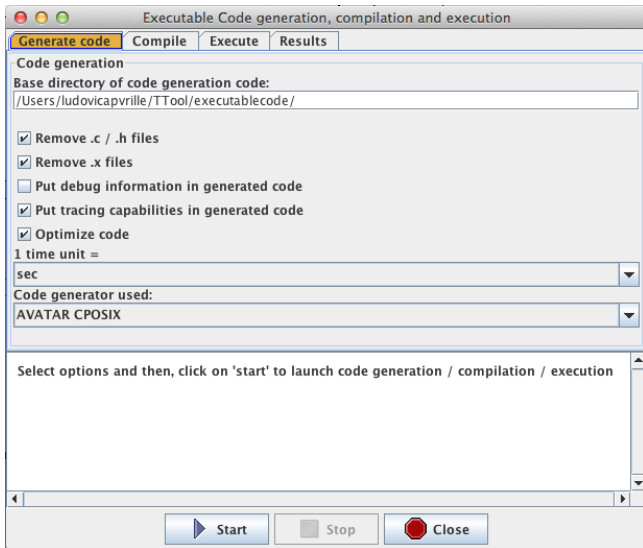
UMLEmb - System Validation

IP PARIS

The screenshot displays the TTool interface with several key components:

- Main Window:** Shows a project tree on the left and a central workspace containing a UML sequence diagram. The diagram includes messages like 'WaitingForEmergencyRequest', 'EmergencyIgnored()', 'WaitingForEmergencyMessage()', 'EmergencyDetected()', 'setPosition()', and 'WaitingForEmergency()'. A 'Block: Block0' is visible at the bottom.
- Console of MutekH:** A black window on the left showing a log of simulation events, including 'Waiting for request!', 'Releasing autoLocking mutex', and 'Requesting request in initialization'.
- Code Generation Window:** A dialog box on the right titled 'Code generation window' with tabs for 'Generate code', 'Compile', 'Execute', and 'Results'. It contains options for 'Simulation trace' and 'Show trace from file', along with a 'Start' button.
- SoCLib Simulation Window:** A window at the bottom right showing 'SoCLib simulation based on a SystemC engine' with a 'SystemC ASS' logo and system initialization logs.
- Toolbar and Menus:** The top of the window features a toolbar with icons for simulation and code generation, and a menu bar with options like 'File', 'Edit', 'V&V', and 'Code Generation'.

(Virtual) Prototyping: Code Generation



Virtual Prototyping: SocLib Simulation

The screenshot shows a terminal window with the following content:

```

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

```

Below the terminal output is the SystemC logo, which consists of the text "SystemC ASS" in a stylized, green, outlined font.

Below the logo is the text:

```

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

```

vcitty
-> 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

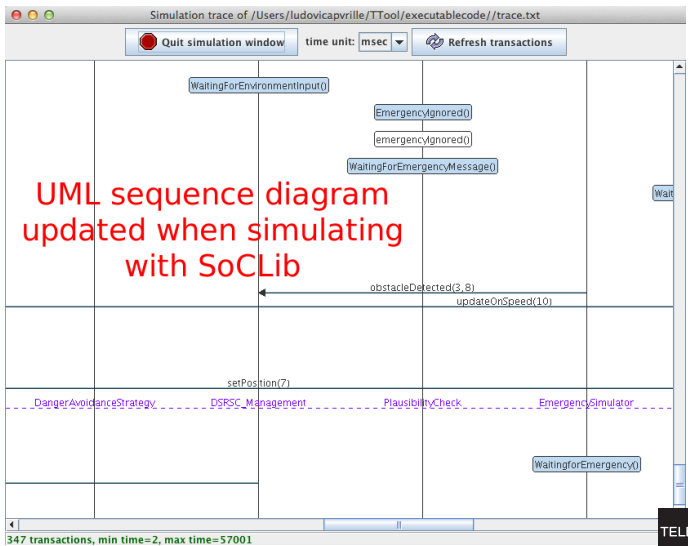
DT - ObjectListManagement -> Mutex locked
DT - ObjectListManagement -> Going to execute request
DT> No request selected -> looking for one!
DT> Counting requests
DT> Starting loop
DT> receive sync
DT> Send sync
DT> Send sync not executable
DT> Counting requests=: 0
DT> No pending requests
DT> Adding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

[]
  
```

Console of MutekH

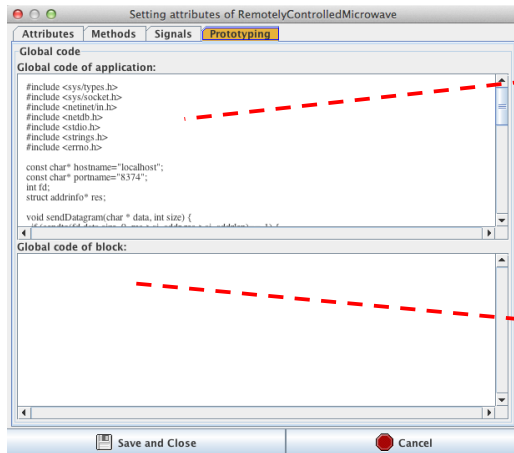
(Virtual) Prototyping: Trace

TTool displays execution traces in a sequence diagram



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



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

The image shows a UML state machine diagram on the left and a 'Setting transition parameters' window on the right. The state machine has two states: 'Starting (entry code)' and 'Heating (entry code)'. The 'Starting' state has an entry code box and is followed by actions: 'startMagnetron()', 'startCooking(remainingTime)', and 'obs_magnetronStart()'. The 'Heating' state is a state with a body and an entry code box. A transition from 'Starting' to 'Heating' is guarded by the condition '[remainingTime = 0]'. A transition from 'Heating' back to 'Starting' is guarded by the condition '[remainingTime > 0]'. The 'Setting transition parameters' window has a 'Prototyping' tab and an 'Entry code' field containing the code: `printf("Heating ; remaining time :%d\n", remainingTime);`. Red dashed lines connect the 'Starting' state to the 'States with entry code' box, the 'Heating' state to the 'States with entry code' box, the entry code boxes to the 'Entry Code' box, and the code in the Prototyping window to the 'Use of block variables' box.

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

The image displays two windows from a software application. The left window, titled "Executable Code generation, compilation and execution", has three tabs: "Generate code", "Compile", and "Execute". The "Execute" tab is selected, showing options for running code. Below the code execution area are "Start", "Stop", and "Close" buttons. The right window, titled "Microwave demonstration", shows a graphical representation of a microwave oven with a red interior and a digital display showing "1" and "Cooking". Below the microwave window is a UML state machine diagram with states "stopMagnetron()" and "ringBell()", and transitions labeled with expressions like "remainingTime + durationModified()", "remainingTime - 1", and "after(5,9)".