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

Simulation

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

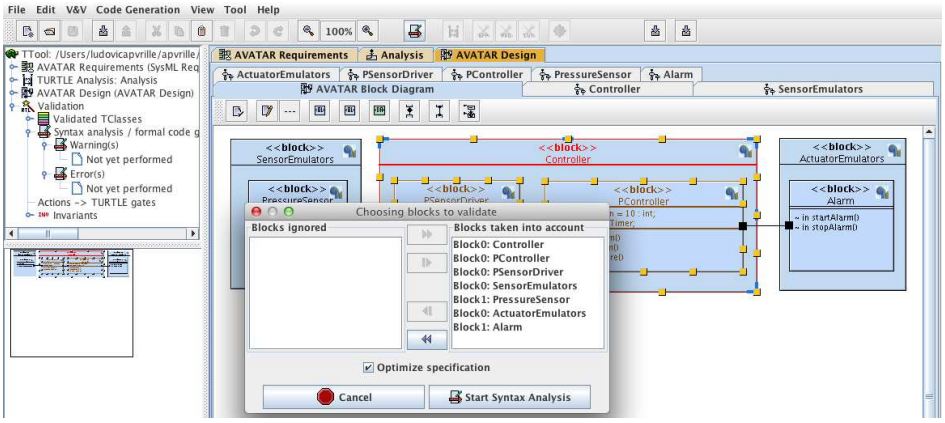
Driving the simulation

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

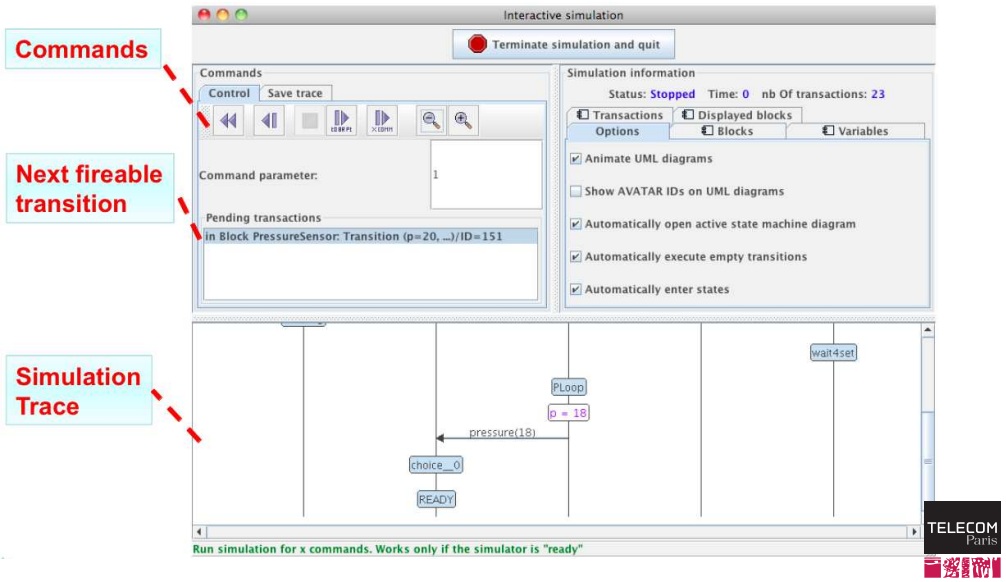
Tracing the simulation

- 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

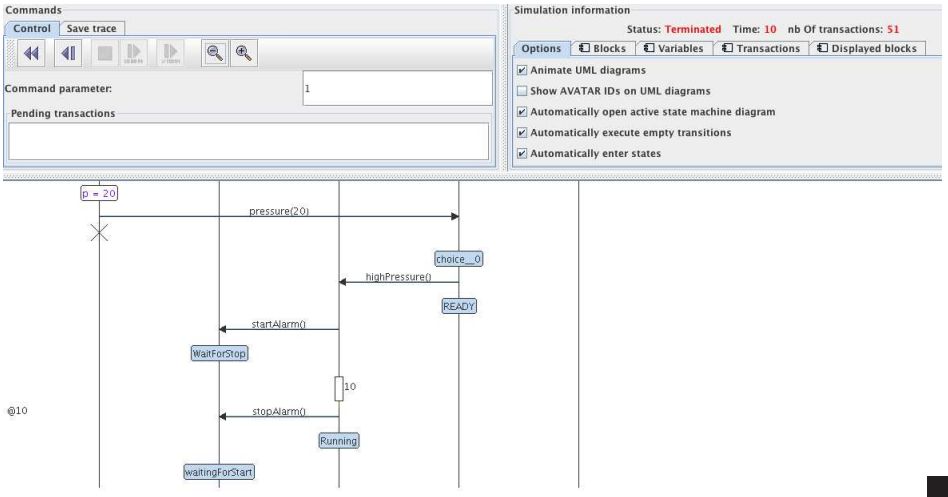
Checking Design Diagrams against Syntax Errors



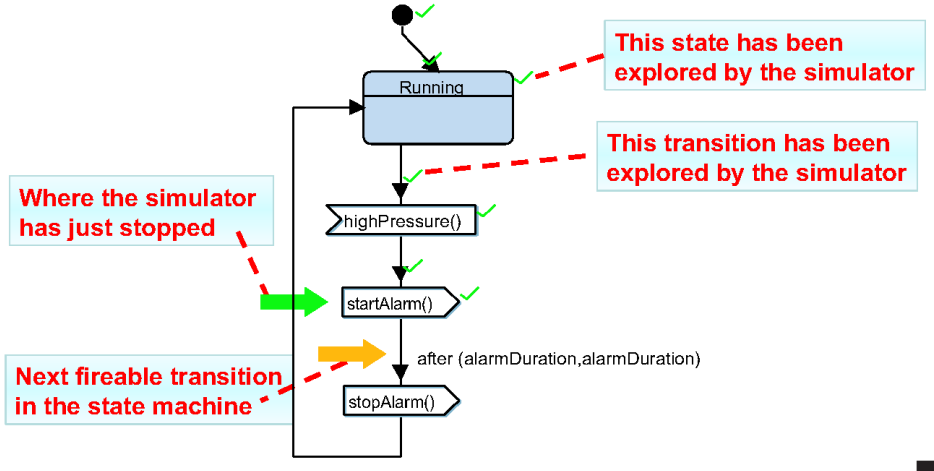
Simulator Interface



Simulator Trace (Sequence Diagram)



Simulator Trace within a State Machine



Outline

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

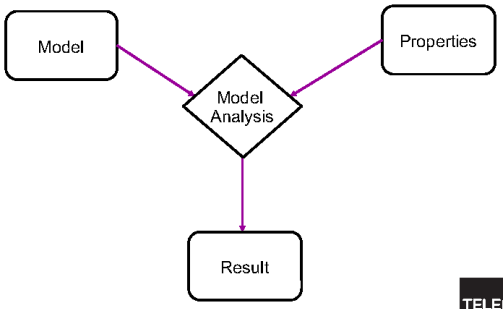
Simulation vs. Formal Verification

Simulation explores execution paths in the model relying on

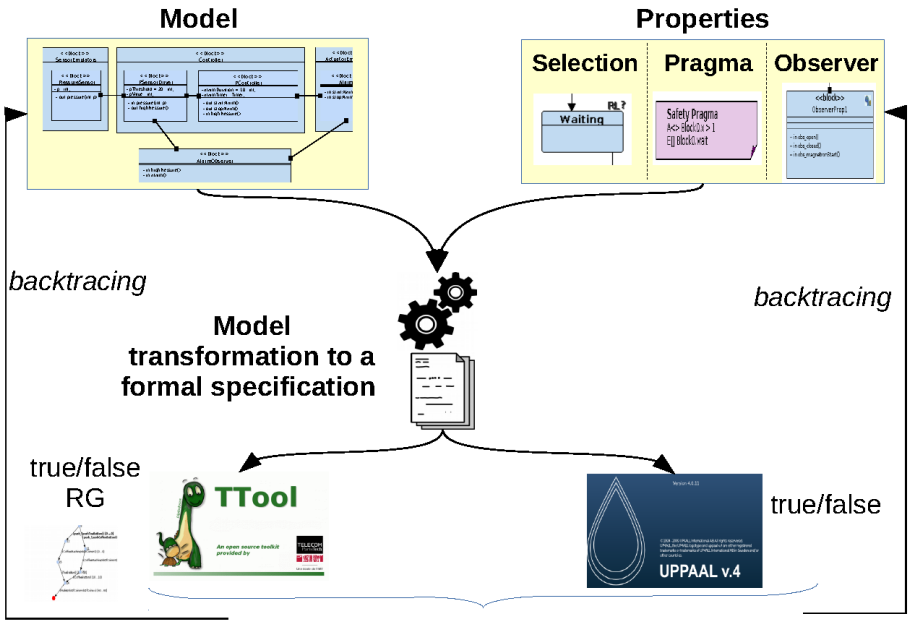
- The experience of the Human who guides the simulation
- Random selection in case of non deterministic choice (several transitions fireable at the same time)

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

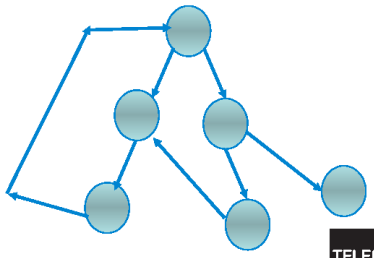
1. From the initial state
2. Search for fireable transitions and create new states
3. Compare new states with existing ones
4. GOTO 2, and take newly created states as initial states

Risk: state explosion problem

- Missing resources (e.g. memory)

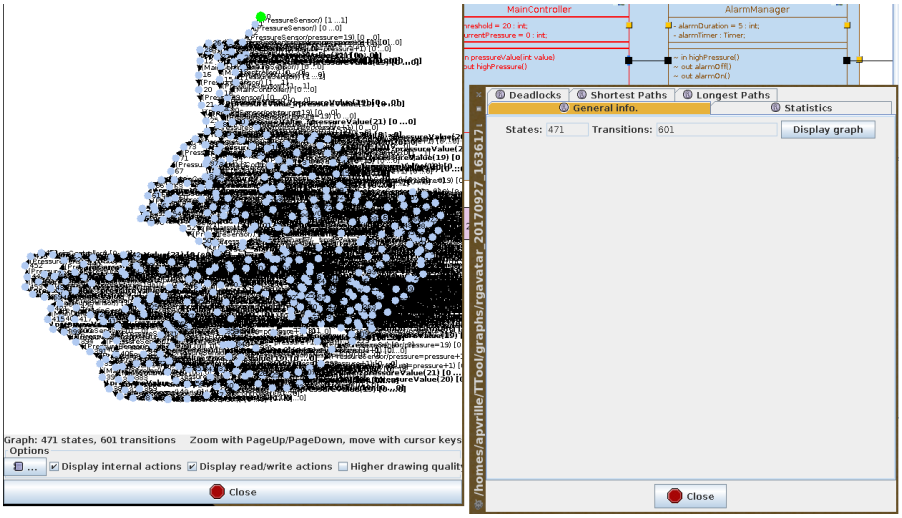
(Some) Solutions

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

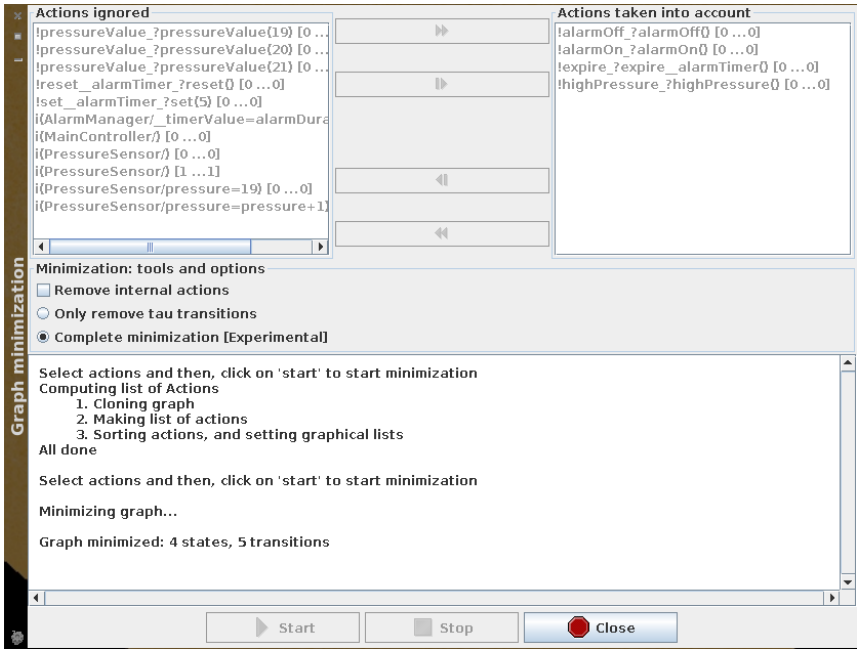


Reachability Graph Generation in TTool

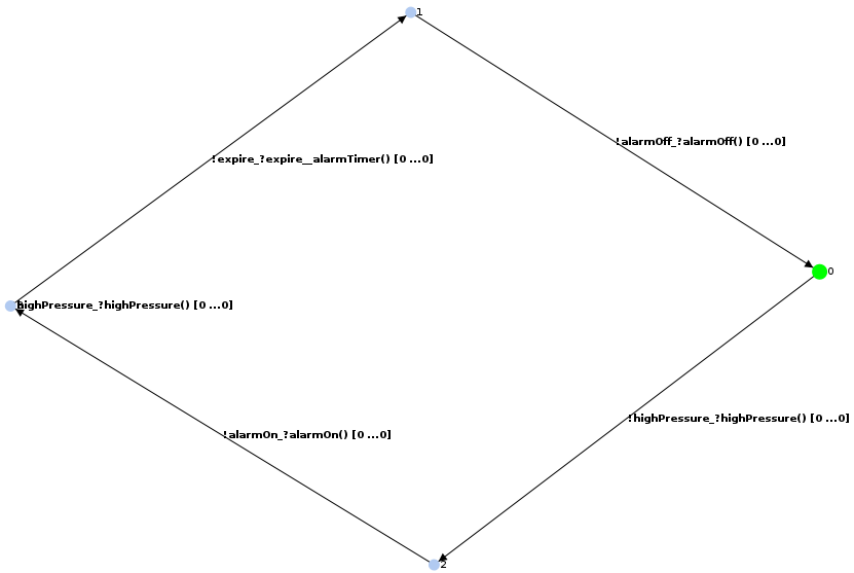
- Internal feature
  - "Syntax checking", then "Avatar Model Checker"



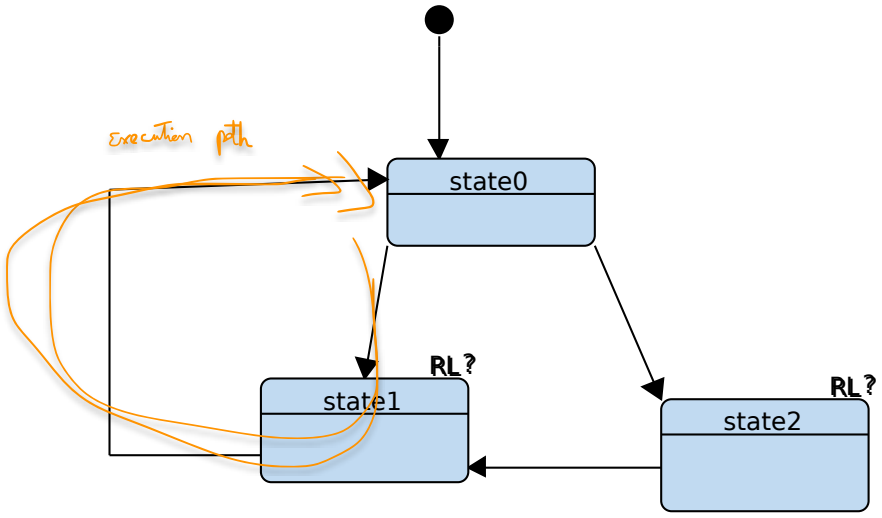
Minimization of Reachability Graph



Minimized Reachability Graphs

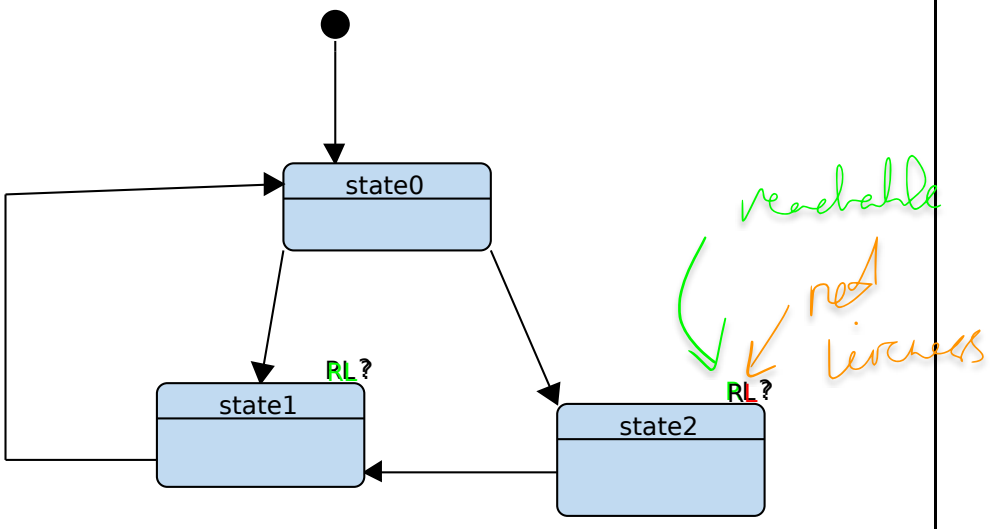


Selecting States for Verification



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

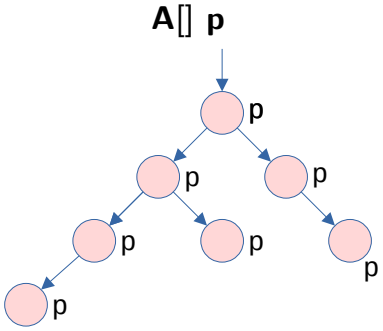
Verification Backtracing

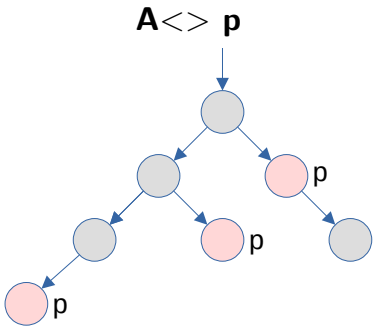


How to obtain this result in TTool? "Syntax checking" then "Safety verification" then check "selected states" in reachability and liveness sections

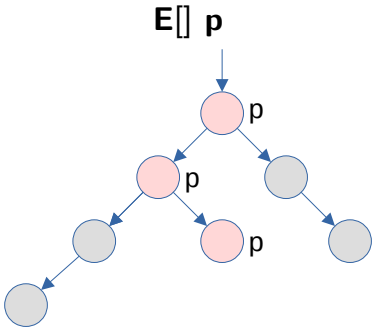
Safety Pragmas

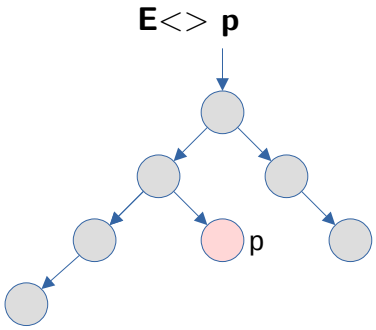
- TCTL = Timed Computation Tree Logic
- Two main operators: A (All paths), E (One path)
- Two modifiers: [] (All states), <> (one state)
- A (boolean) property p





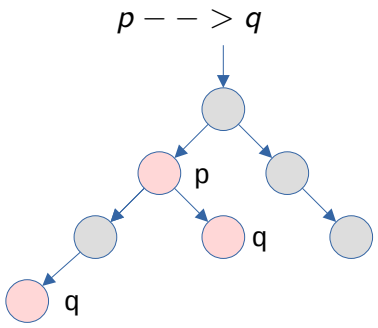
Safety Pragmas (Cont.)



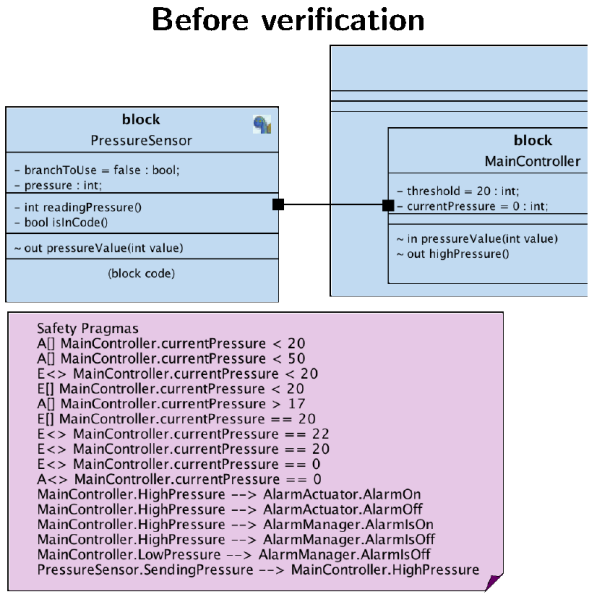


Safety Pragmas (Cont.)

- Leads to
- $p - - > q$

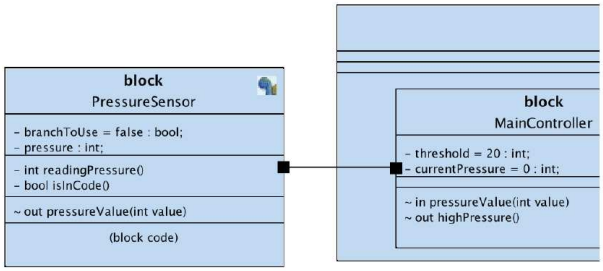


Safety pragmas in TTool



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

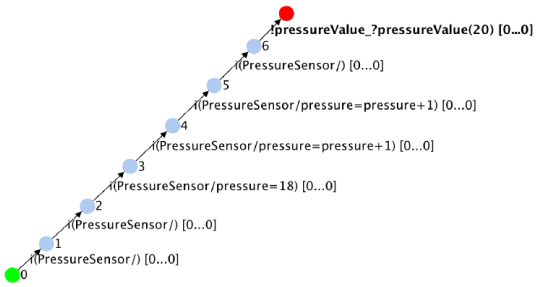
Safety Pragmas
- F A[] MainController.currentPressure < 20
- T A[] MainController.currentPressure < 50
- T E<> MainController.currentPressure < 20
- T E[] MainController.currentPressure < 20
- F A[] MainController.currentPressure > 17
- F E[] MainController.currentPressure == 20
- F E<> MainController.currentPressure == 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



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

Observer-Guided Verification

Observers

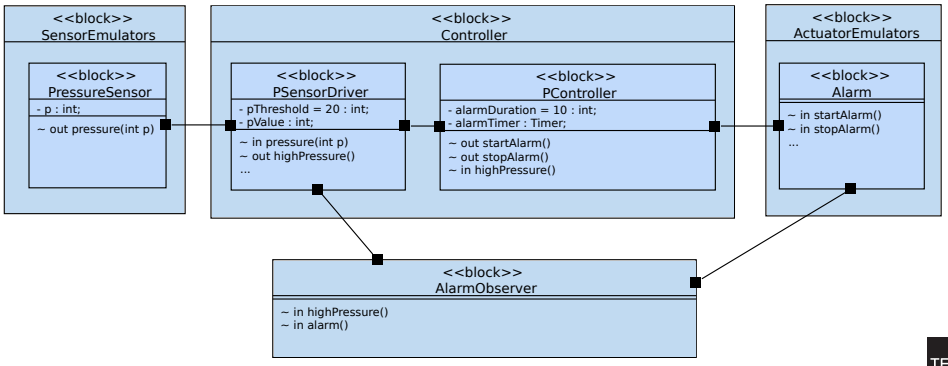
- Expression of (complex) properties within the design
- Observer should have an *error* state whose reachability can be searched for in TTool/UPPAAL
- The observer should remain non-intrusive
  - At least, as long as the observed property is satisfied

Example: Pressure Controller

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

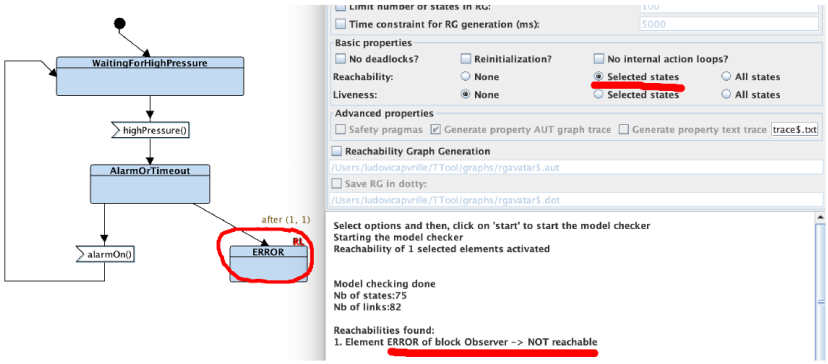
Pressure Controller: Design of an Alarm Observer

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



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



Outline

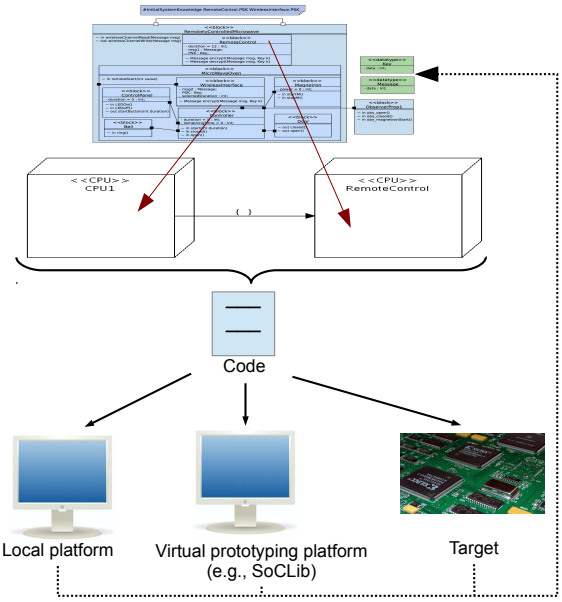
Introduction to Rapid Prototyping

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

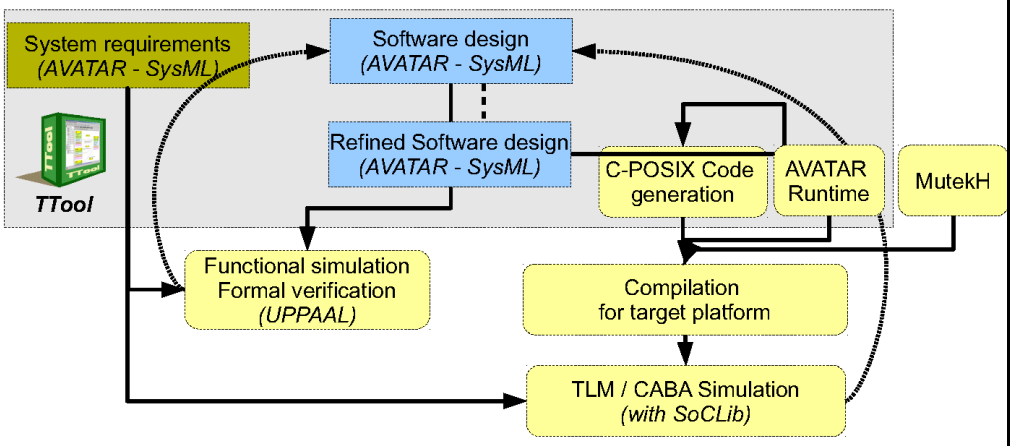
Code Generation: Overview



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 Prototying: 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

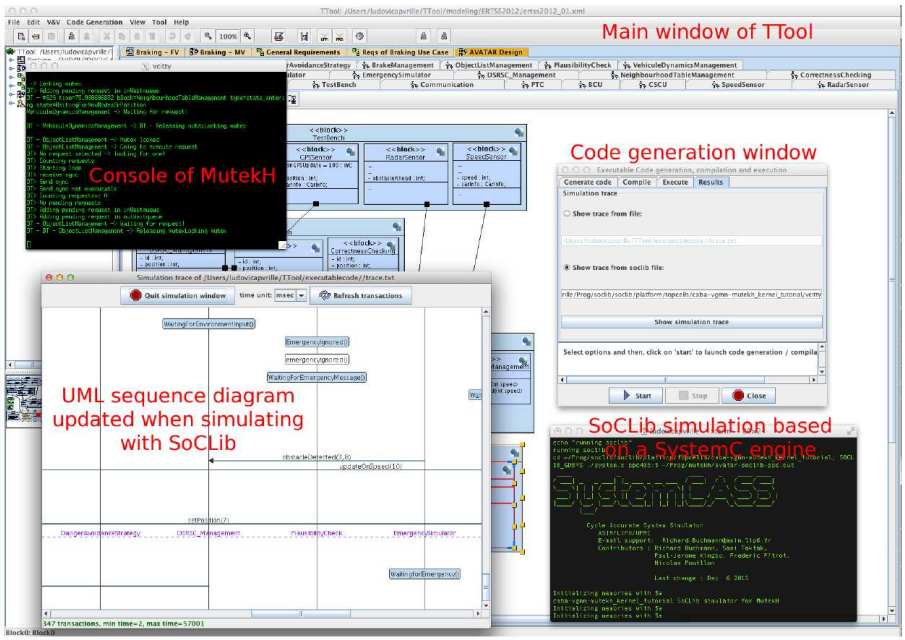
Hardware platform simulator: SoCLib (www.soclib.fr)

- 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

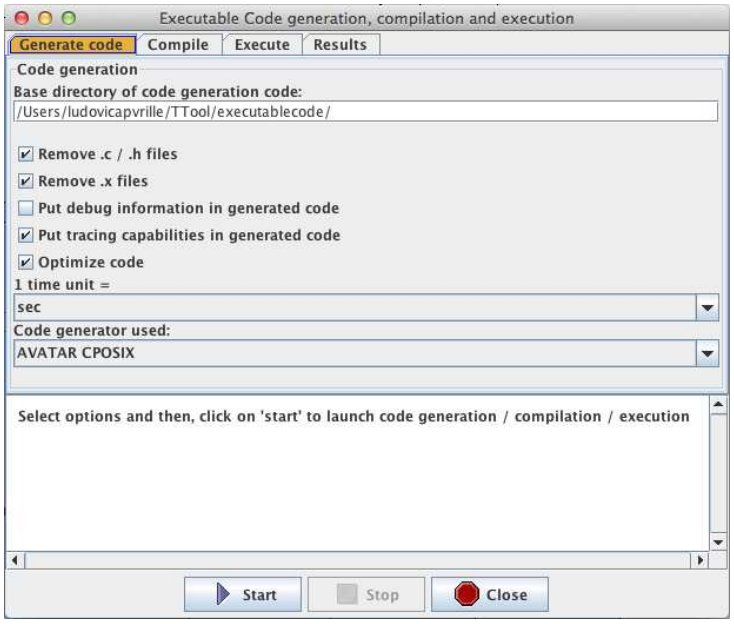
Embedded Operating System: MutekH (www.mutekh.org)

- 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

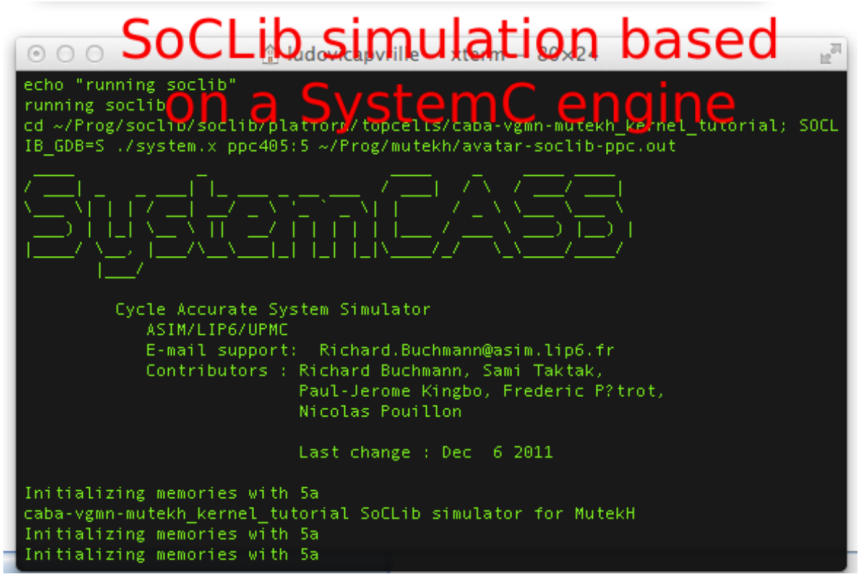
Virtual Prototyping: Graphical Environment



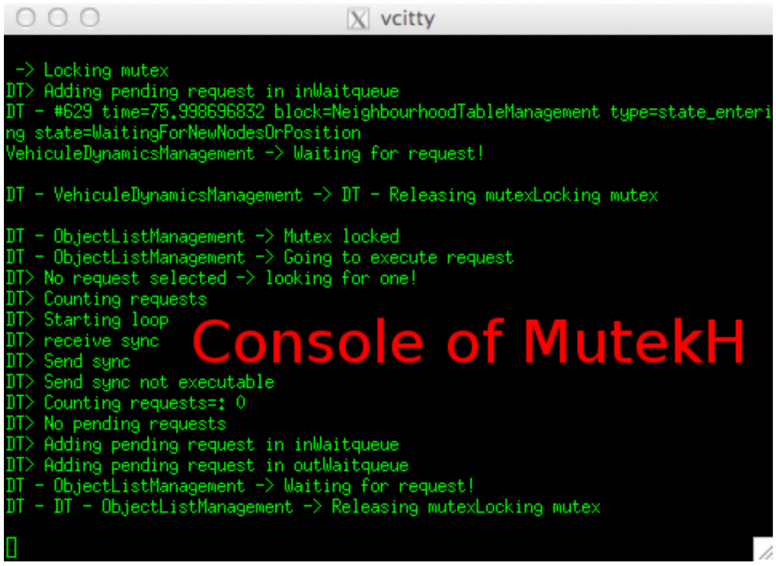
(Virtual) Prototyping: Code Generation



Virtual Prototyping: SocLib Simulation

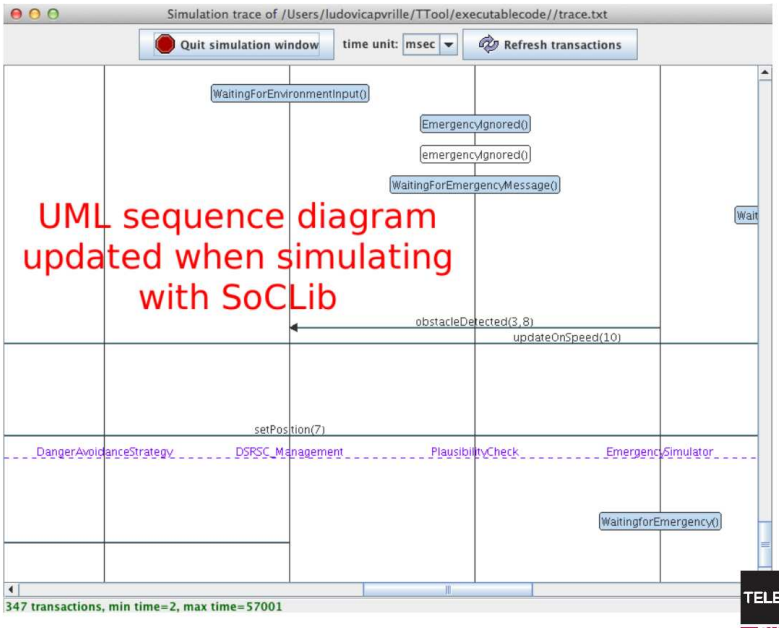


Virtual Prototyping: Console



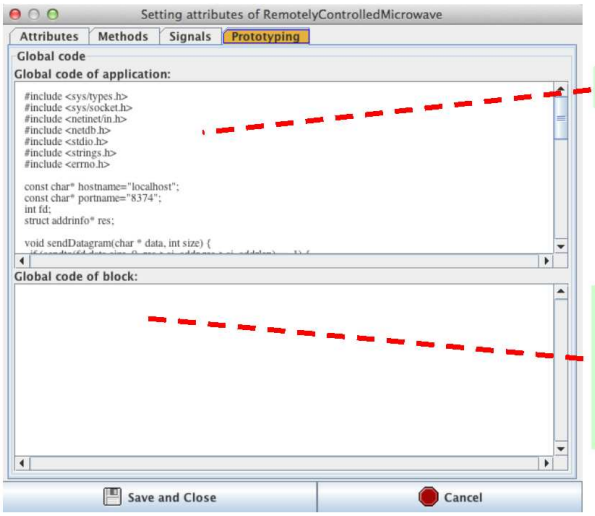
(Virtual) Prototyping: Trace

TTool displays execution traces in a sequence diagram



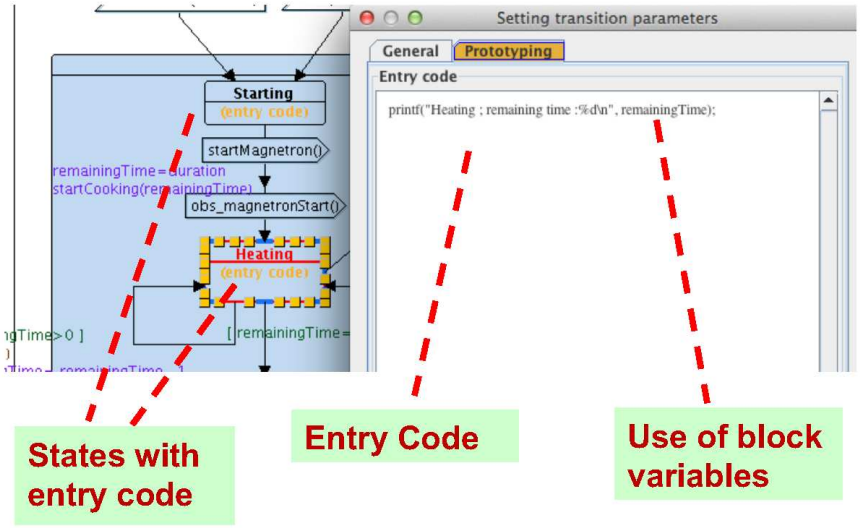
Customizing Generated Code with Your Own Code: Application and Block Code

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



Customizing Generated Code with Your Own Code: State Entry Code

- Code executed whenever a state is reached



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