# MOSIG PDES / Embedded Systems

Florence Maraninchi and Pascal Raymond

Verimag / Grenoble INP

2012-2013



## Teachers and Organization

- Florence.Maraninchi@imag.fr 
- Pascal.Raymond@imag.fr 



## Organization, Online Resources

A single place for all documents and slides:
`http://ensiwiki.ensimag.fr/index.php/MOSIG_2_-_Option_DEMIPS_-_UE_Embedded_Systems`

Or, go to `http://ensiwiki.ensimag.fr/`, and search for DEMIPS. The page of the course should appear in the list of documents found.

**Or...**

google "`maraninchi mosig`".



## Books?

There's no book that could cover the full range of subjects we will be looking at during the course.
On Pascal's part (model-checking), there are books.
On Florence's part (general introduction and notions, definition of models and modeling embedded systems), there will be papers.



## VERIMAG Lab (20 years!)

`www-verimag.imag.fr`

Embedded Systems
(Domain-Specific) Languages, validation methods (automatic test, formal verification), development methods, model-driven design, modeling, components, security, formal models, etc.

Application Domains: embedded control systems, systems-on-a-chip, sensor networks, distributed algorithms and systems, middleware, component-based systems, robotics, electronic voting, ...



## Contents of the course (2012-2013 Edition)

- Formal models that can be used to represent the concurrent and timed aspects of modern computer systems
- Synchronous programming of embedded systems
- Principles and applications of model-checking
- Modeling principles; synchronous models, asynchronous models

## Prerequisites

- Programming with an imperative language (C, Java, Ada, ...)
- Parallel programming, at least one style (threads in Java, tasks in Ada, ... )
- Basic synchronous circuits,
- Operating systems,
- Low-level software,
- Automata and formal languages



## Calendar

See webpage (ensiwiki)



## Evaluation

- Pascal's part of the course contains practical exercises (P)
- There is a 3h written exam for the first session (E), about both Florence and Pascal's parts
- The mark for the first session is E, *modified by P*
- There is a 2h written exam for the second session (E'), on both parts
- The mark for the second session is E' (we forget about P)



# Part I

# General Introduction to Embedded Systems



## Outline

# Embedded Systems: Computer Systems in Everyday-Life Objects

- Smart buildings and Energy
- Trains, subways, cars ...
- Consumer electronics (phones, digital cameras, ...)
- Telecom equipments
- Smart cards
- Computer Assisted Surgery
- Avionics and space












# Ex 1: Embedded Control

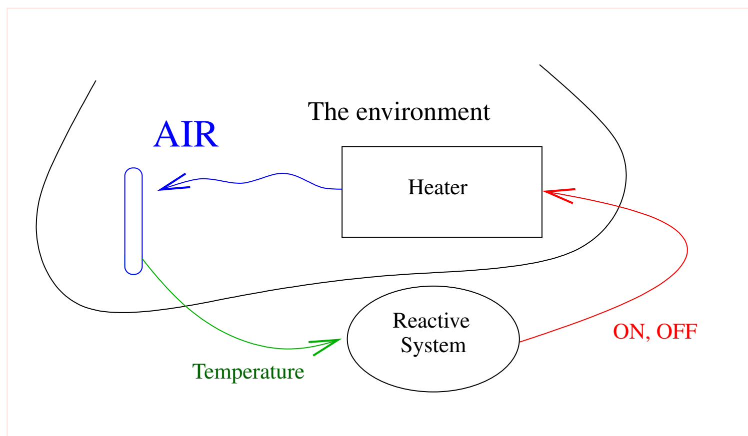
In trains, cars, aircraft, space objects, nuclear power plants, ...
Systems: ABS, fly-by-wire, automatic flights, security control, ...

- The environment is a physical system, not a human being
- There are quite strong real-time constraints
- They are safety-critical systems
- The computer system is the implementation of a control engineering solution
- The computer system is reactive





# Embedded Control - Reactive Systems








# Embedded Control - Real-Time Systems

A typical real-time program:

```
initializations
while (true) {
      --- point (1)
      get inputs
           from the sensors
      compute outputs
           and update memory
      write outputs
           on the actuators
      --- point (2)
}
```

The time it takes to execute the code between points (1) and (2) defines the sampling period of the program. This is real time.

The outputs to the environment may have some influence on future inputs. This is reactivity.





# Real-Time Programming Problems

- Write code that is sufficiently fast (you're not always allowed to answer: "*try a bigger machine*")
- Be able to tell how fast your program is, *in advance* (Worst-Case-Execution-Time static evaluation)
- It's not always possible to write single-loop code, because of the *intrinsic parallelism* of a reactive system.
  e.g., between two independent sensor-computing-actuator lines

# Embedded Control - Safety

Criticity:
A fault is very “expensive”
(human lives, damage to the environment, ...).

HW Fault-tolerance:
Examples: several sensors of the same kind, plus a voter, several copies of the same code, running on several processors
SW Fault-tolerance:
Example: Several distinct versions developed independently from the same specification.





# Embedded Control - Centralized or Distributed Systems?

Fact: centralized systems are far simpler than distributed ones.

But:

- Fault-tolerance forces distribution
- Physical position of sensors and computers also forces distribution
- Efficiency sometimes requires distribution

Another fact: distributed real-time programming is very hard.





# Ex 2: Consumer Electronics

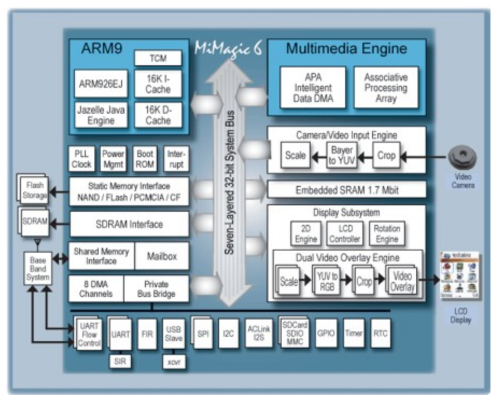
Digital cameras, set-top boxes, mobile phones, internet tablets, all kinds of portable devices...

- The environment is: a physical system (radio link) + a human being
- There are real-time constraints on the radio part
- They are business-critical systems
- The memory capacity and the processor speed are limited, the size is important, energy consumption is a very important constraint
- The hardware architecture is complex, and dedicated to the device (several processors, a DMA, a MPEG decoder, buses, radio components, ...) and the software is very hard to build.





# Consumer Electronics - Systems-on-a-Chip




A complete computer system... on a single chip.
Several HW elements, embedded SW on the computing units, (RT)OS on some of the computing units, ...





# Systems-on-a-Chip: SW Development Virtual Prototyping

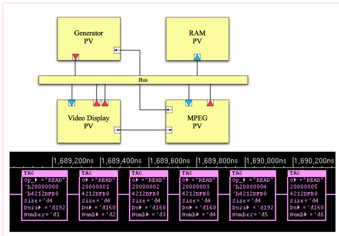
The SW developers should start developing SW for a particular HW platform, long before the circuit corresponding to this HW platform is available.

Solution: virtual prototyping = write a program (SW) able to simulate the behaviours of the HW platform, w.r.t. the SW.



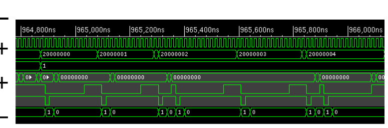


# Systems-on-a-Chip: Virtual Prototypes




TLM
+ Early Available
+ Fast Simulations
– Not synthesizable

RTL
Synthesizable+
Cycle and Data Accurate+
Slow Simulations–



# Systems-on-a-Chip: Summary

Main difficulties:

- Choose the right abstraction of the HW behavior
- Ensure that the SW developed on the virtual prototype will work, unchanged, on the real HW
- Define several abstractions, depending on the use (functional validation, time performances, energy consumption, ...)





# Example: HW and Drivers USB 3.0 Verification Techniques: Testing USB at the System Level

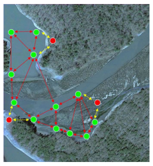
Recent announce for a seminar:
“As designer engineers work to integrate USB 3.0 into their devices, they will have to be cognizant of a number of complex issues. SuperSpeed links introduce 21 new state machines to USB operation - many of which rely on timers for entrance and exit. In addition, consumer devices often require seamless translation into other technologies such as, PCI Express, SATA, Fibre Channel and DDR3. And in mobile applications, vBUS power draw can limit the design's performance.”

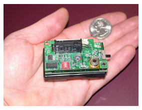




# Ex 3: Sensor Networks

Environment monitoring, logistics, ...





- The environment is: a physical system (radio link + physical inputs on the sensors)
- The memory capacity and the processor speed are very limited, energy consumption is THE key point
- The hardware architecture of a node is quite simple
- The software (MAC and routing protocols, application code) is crucial for energy consumption

The main problem is cross-layer design.











# Transformational Systems

Typical example : a compiler

```
      else
         if Y then return False ;
         else return False ;
ubaye(7) gnatmake chap2.adb
gcc-4.1 -c chap2.adb
gnatbind -x chap2.ali
gnatlink chap2.ali
ubaye(8)
```

Inputs at the beginning, then some finite time computation, outputs at the end.

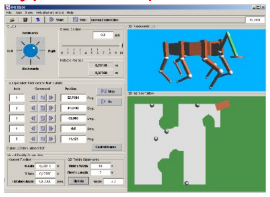
A transformational system has to terminate.





# Interactive Systems

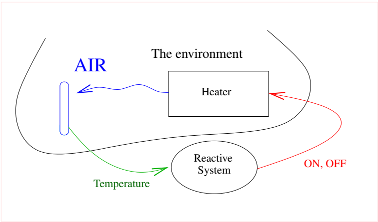
Typical example: a man-machine interface



loop-based behavior (does not necessarily terminate), where inputs come all the time (human actions on buttons, mouse, keyboard) and outputs are produced all the time also (changes of the interface, effects on the underlying computer system).

## Reactive Systems

Typical example: a heater controller.




The same as interactive systems, but the speed of the interaction is driven by the (physical) environment. The computer system should be sufficiently fast in order not to miss relevant evolutions of the environment.



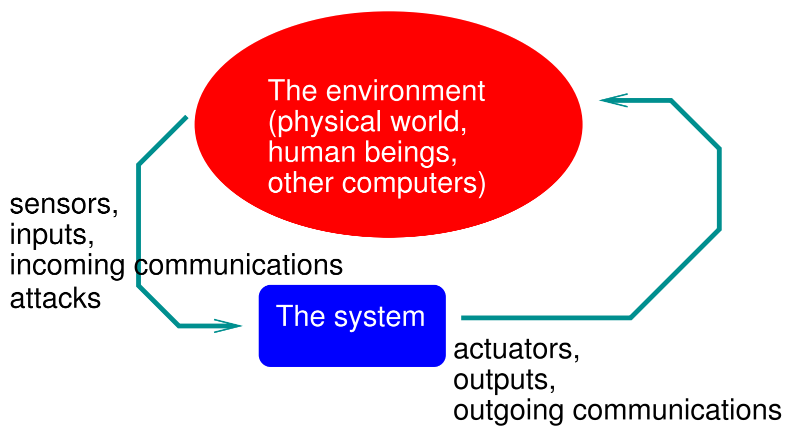








## External View (1)








## External View (2)

A Communicating Embedded Application is essentially reactive.

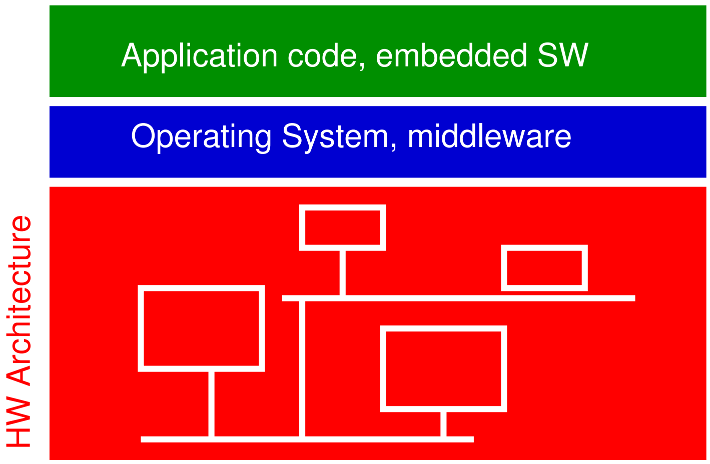
Externally observable properties:

- Correctness (functional property), or...
- ... Failure rate (functional property)
- Power consumption (non-functional property)
- Time (functional or non functional ?)
- Resistence to attacks (functional or non functional ?)





## Internal View








## Constraints

- (very) Scarce resources (memory, CPU, energy, ...)
- Real-time constraints and reactivity
- critical contexts of use (human lives, environment, business, ...) that imply strong and "in advance" validation methods for functional properties
- Importance of power Consumption and other extra-functional properties
- Fault-tolerance

## Main Difficulties for the Design of Embedded Systems

- Real-time parallel and distributed programming (choice of a programming language?)
- Relation with control engineering
- Intricate dependency between HW, application SW, and OS or middleware
- Certification authorities
- Several degrees of dynamicity (from simple reconfigurations to mobile code...)





## General-Purpose or Domain-Specific Languages?

Ada, Java for real-time programming?

A DSL (Domain-Specific Languages) may have specific operations dedicated to reading sensors, writing actuators, synchronizing on time, etc., but it also has less than general-purpose languages.

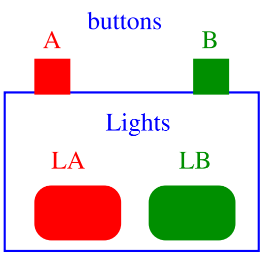
**Example**

A programming language for embedded systems should not allow to write programs for which the memory is not statically bounded (implies: no recursion, no dynamic allocation)





## A Special Note on Parallel Systems (1)




Behavior to be programmed:
— Each time A is pressed, toggle light A
— Each time B is pressed, toggle light B

How to program such a system?





## A Special Note on Parallel Systems (2)

```
StateA := OFF ;
while (true) {
    read button A
    if (buttonA) {
        StateA := not StateA ;
    }
    if (StateA) {
        LightA.ON
    }
    else {
        LightA.OFF
    }}
```

```
StateB := OFF ;
while (true) {
    read button B
    if (buttonB) {
        StateB := not StateB ;
    }
    if (StateB) {
        LightB.ON
    }
    else {
        LightB.OFF
    }}
```

This is a solution for (one button, one light). How to describe two of them in parallel?





## A Special Note on Parallel Systems (3)

```
StateB := OFF ; StateA := OFF ;
while (true) {
    read button B ; read button A ;
    if (buttonB) { StateB := not StateB ; }
    if (buttonA) { StateA := not StateA ; }
    if (StateB)  { LightB.ON } else { LightB.OFF }
    if (StateA)  { LightA.ON } else { LightA.OFF }
}
```

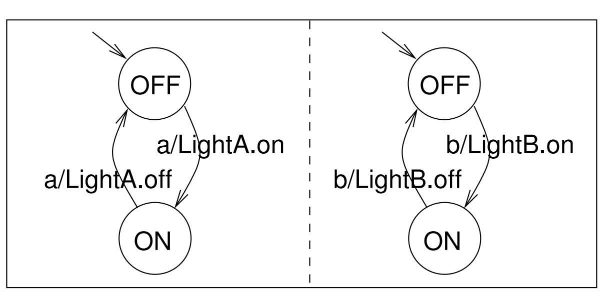
A solution with static scheduling: the code produced is sequential, but the high-level language may be parallel.





## A Special Note on Parallel Systems (4)

A Solution in an Automaton-Based Language
(Statecharts, Argos, SCADE, EsterelStudio/SyncCharts, ...)



## Validation and Certification

Validation:
Simulation, automatic testing, formal verification, ... are methods that help in analysing (functional) properties of a computer system before it is deployed.

Certification:
Examples: the DO178B norm for civil avionics, common criteria for smart cards, ...









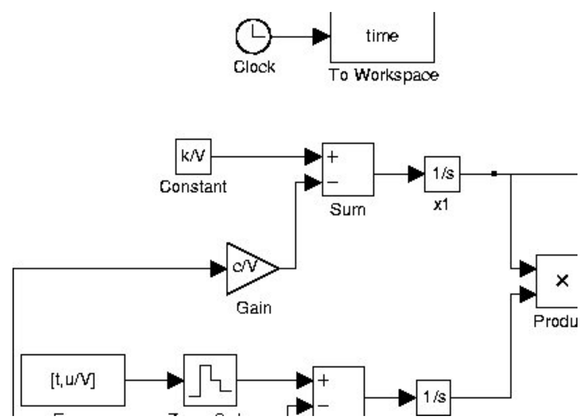








## A Simulink Diagram








## Development from Simulink

- A continuous control problem and solution, including a model of the environment
- A discrete solution for the controller part
- An implementation. *Automatic code generation from Simulink?* or manual encoding, considering the diagrams as a detailed specification?

A complete chain from Simulink to embedded code is an instance of the general model-driven approaches.

# Recommended readings - Lustre and SCADE

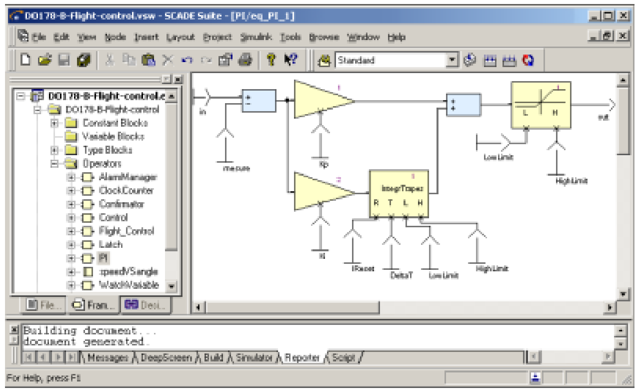
```
@article{ halbwachs91synchronous,
    author = "N. Halbwachs and P. Caspi and
              P. Raymond and D. Pilaud",
    title = "The synchronous data-flow
             programming language {LUSTRE}",
    journal = "Proceedings of the IEEE",
    volume = "79",
    number = "9",
    month = "September",
    year = "1991"}
```

www.esterel-technologies.com/products/scade-suite/overview.html





# A Scade diagram







# Main features

- A formal language
- Powerful bi-directional interface to requirements management tools like DOORS.
- KCG Code Generator qualification eliminates the need for low level testing.
- Verification tools
- *Import and reuse of Simulink block diagrams and Stateflow diagrams into SCADE.*



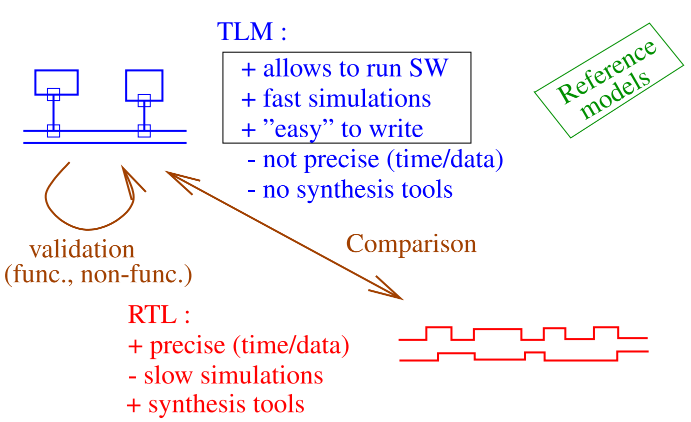








# A new abstraction level: TLM, and SystemC








# SystemC example

```
void module1::T1(){
  int a = 0;
  while(true){
   wait(e1);
   a++;e2.notify();
   a++;e3.notify();
}}
void module1::R1(){
  int b = 0;
  while(true){
    b++;wait(e2);
    b++;p2.f2(b);
}}
void module1::
          f1(int x){
   cout << x ;
   e1.notify();
   wait(e3);
}
```

```
void module2::T2(){
  int c;
  while(true){
    c++;  p1.f1(c);
    c++;  wait(e4);
  }
}

void module2::
          f2(int x){
    cout<< x ;
    e4.notify();
}
```

2 Some Industrial Practices
- Simulink in the automotive industry
- SCADE in the avionics industry
- SystemC for Systems-on-a-Chip
- Summary





## Summary: Programming or Modeling Languages

Software:
C, C++, SystemC, Java or RT Java, Ada, ...
Domain-Specific Languages (DSLs): Lustre/Scade, Simulink, ...

Hardware:
VHDL, Verilog, C, SystemC, ...





## Summary: Criticity

Safety-critical systems (e.g., nuclear plants):
Design norms, certification authorities, ...

Business-critical systems (e.g., mobile phones):
Methods to shorten time-to-market (virtual prototyping)





## Summary: Intrinsic Difficulties and Methods

Reactivity, distribution, real-time, fault-tolerance, ...

Use of "*models*"





1 What is an Embedded System?

2 Some Industrial Practices

3 Case-Study: HW and low-level SW
- The Hardware and the Protocol
- Consumption Automata
- Modeling The HW and the SW... and Their Interactions

4 Why Models? Model-Driven Approaches, Virtual Protoyping, Formal Verification

5 This Course



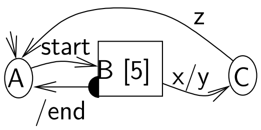


## The Example, and What it Illustrates

A simple Medium-Access-Control (MAC) protocol for sensor networks.
Typical case for:

- Modeling the HW with formal models (automata)
- Extracting an automaton from the C code of the low-level SW
- Composing the two automata for:
  - Detecting illegal uses of the HW API by the low-level SW
  - Computing the power consumption of a given scenario
  - ...

## Elements of Syntax for Automata With Timed States (Synchronous Semantics)




- Circles: normal states
- Squares: timed states [delay]
- x/y: input/output

Synchronous semantics: on a discrete time line...



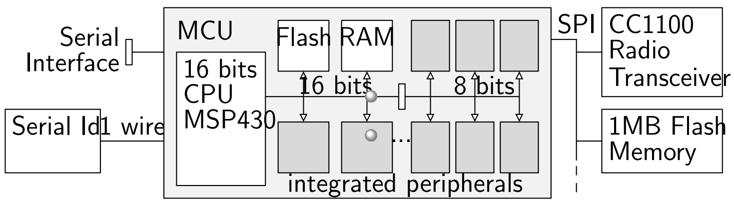








## The Hardware Of A Node (WSN430)

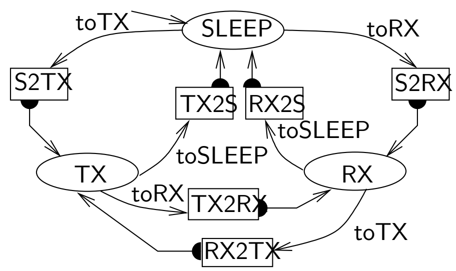



See: perso.ens-lyon.fr/eric.fleury/Upload/wsn430-docbook/index.html





## The Radio (Simplified)




Delays:
RX2S=TX2S=0.1 $\mu s$
S2RX=S2TX=88.4 $\mu s$
TX2RX=21.5 $\mu s$
RX2TX=9.6 $\mu s$

RX = receive; TX=transmit;
Square states: waiting for some delay;
toRX, toTX, ...: commands from the SW.





## The Protocol - C code

www.senslab.info/wp-content/uploads/2012/03/wsn430-lib-v2012-03-13.t

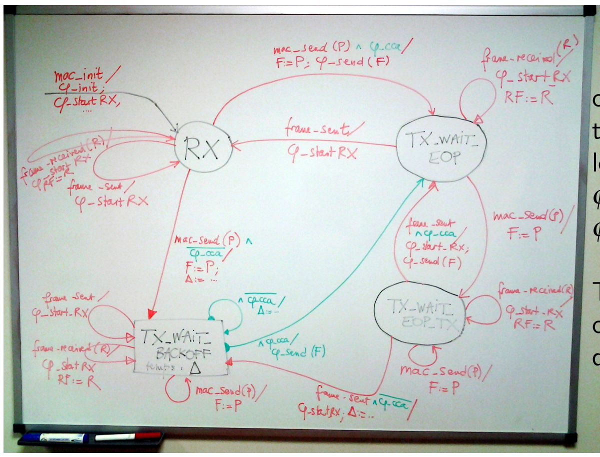
mac/simplemac.c

```
...
#define STATE_RX               0x0
#define STATE_TX_WAIT_EOP      0x1
#define STATE_TX_WAIT_EOP_TX   0x2
#define STATE_RX_WAIT_BACKOFF  0x3
...
uint16_t mac_send
(uint8_t packet[], uint16_t length, uint8_t dst_addr)
{   ...
  switch (mac_state) {
   case STATE_RX:              try_sending(); break;
   case STATE_RX_WAIT_BACKOFF: break;
   case STATE_TX_WAIT_EOP:     mac_state = STATE_TX_WAIT_EOP_TX; br
   case STATE_TX_WAIT_EOP_TX:  break;
}
```





## The Protocol - Automaton version




commands issued to the lower-level: $\phi$start_RX, $\phi$_send, ...

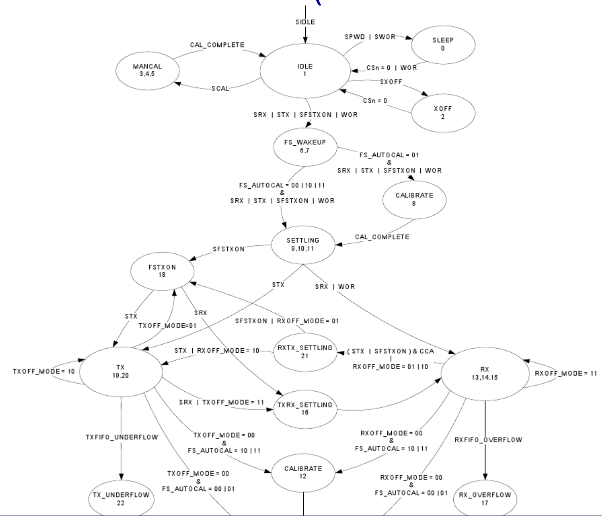
Timed states for counting back-off delays.

3 Case-Study: HW and low-level SW
- The Hardware and the Protocol
- Consumption Automata
- Modeling The HW and the SW... and Their Interactions





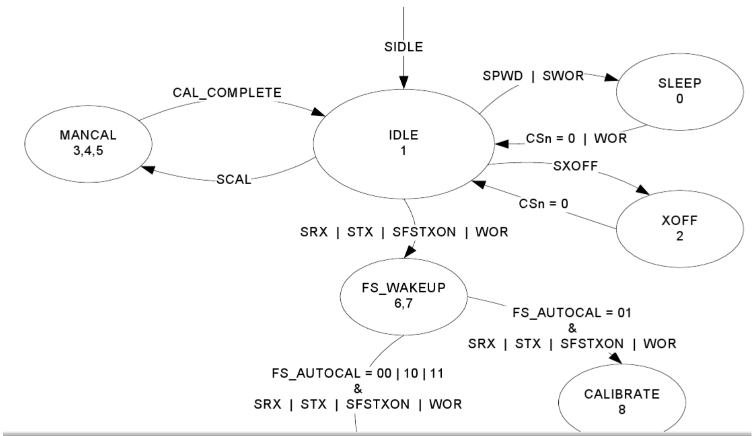
# Power-State Models (Texas Instruments CC1100)







# Power-State Models (Texas Instruments CC1100)








# Power-State Models

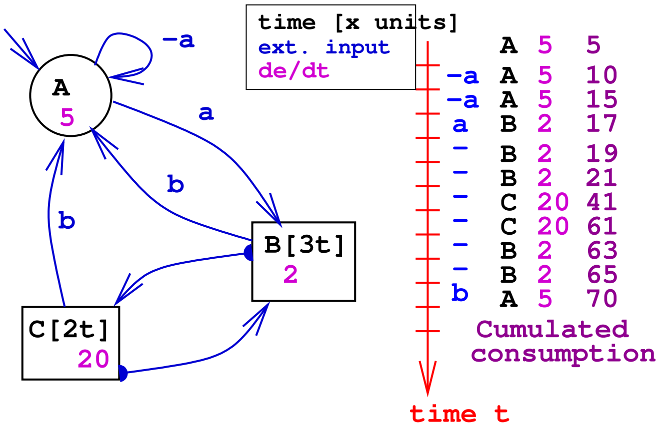
In a datasheet one can find:
— The automaton (power modes and transitions between them)
— Consumption per unit of time, in each mode
— Time and Consumption penalty for each transition

All this can be expressed by a consumption automaton.
= a discrete view of a linear-priced timed-automaton (LPTA),
or a simple case of hybrid automaton.





# Power-State Models: Formal View








3 Case-Study: HW and low-level SW
- The Hardware and the Protocol
- Consumption Automata
- Modeling The HW and the SW... and Their Interactions

## Remarks, Questions, and Discussion...

- The HW (radio device) is modeled by an automaton
- The MAC program is given as an automaton (nothing's missing, the automaton syntax is a programming language, you can get the actual running code from the automaton, or the automaton from the C code).
- What to do with these two objects? How to combine them?
- What does it mean to "run" the program together with the model of the radio? What can we observe?
- How do we know that the model is faithful to reality?





## Products of Models

A (synchronous) product of the HW and SW automata is an automaton in which a state represents the state of the HW together with the state of the SW.





## Example Illegal Use of the HW

Assume the HW (the radio) should not receive command C when in state S.

How to express this property on the product automata?
We'll see we can automatically prove that it's respected (or not).





## Total Consumption of a Scenario

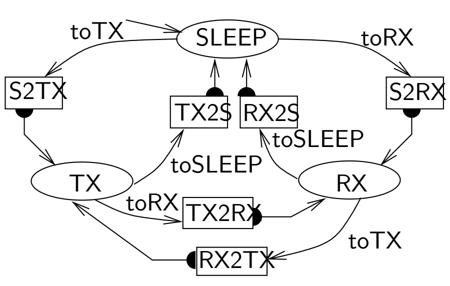
Given an input scenario for the SW, we can play it on the product automata, and get a sequence of states with the time spent in each of them.

This gives the total consumption of the scenario.





## The Radio (with Consumptions)




Consumptions:
$\gamma$(SLEEP)=2
$\gamma$(RX)=15
$\gamma$(TX)=16
$\gamma$(S2TX)=$\gamma$(S2RX)=
$\gamma$(RX2S)=$\gamma$(RX2TX)=
$\gamma$(TX2RX)= $\gamma$(TX2S)=8

Delays:
RX2S=TX2S=0.1 $\mu$s
S2RX=S2TX=88.4 $\mu$s
TX2RX=21.5 $\mu$s
RX2TX=9.6 $\mu$s

4. Why Models? Model-Driven Approaches, Virtual Protoyping, Formal Verification
   - Understand Complexity
   - Model-Driven Development Methods
   - Formal Models for Formal Verification





## Simple Models Help Understand the Complexity of Computer Systems

- Industrial practice for embedded applications : an ever growing wide variety of languages and models
  C/C++, SystemC, Ada, Simulink, Scade/Lustre, VHDL or Verilog, UML, SysML, AADL, IPXact, Java, ...
- How to understand the relationship between the two ?
- Computer science : a relatively stable and small set of well understood models of computation (for embedded systems: models of time and concurrency)





## Modeling Concurrency

- Synchronous models are good at: modeling parallelism between processes that share a common clock; describing synchronous circuits; describing a "logical" parallelism that will be compiled into sequential code;
- Asynchronous models are good at: modeling parallelism when no common clock exists (or when does not want to rely on the clocks and their synchronization when developing the SW);
- GALS (Globally Asynchronous Locally Synchronous) or LAGS models can mix the two





## Modeling Time

- Synchronous models are intrinsically timed, because of the existence of a global clock
- Asynchronous models are intrinsically untimed; adding time to an asynchronous model requires ad-hoc modifications.





## We will study...

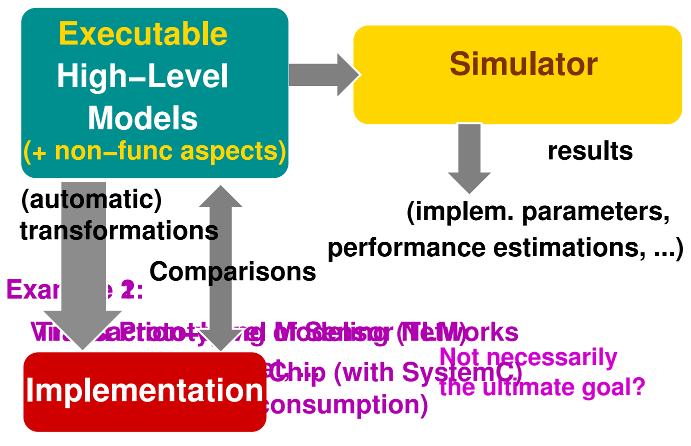
- Synchronous models, and how they can be turned into synchronous programming languages for real-time reactive systems
- Untimed asynchronous models
- How to add time to an asynchronous model
  (*you already know asynchronous parallel programming*)
- Systems to models to formal verification tools





4. Why Models? Model-Driven Approaches, Virtual Protoyping, Formal Verification
   - Understand Complexity
   - Model-Driven Development Methods
   - Formal Models for Formal Verification

## Model-Based Impl. plus Virtual Prototyping





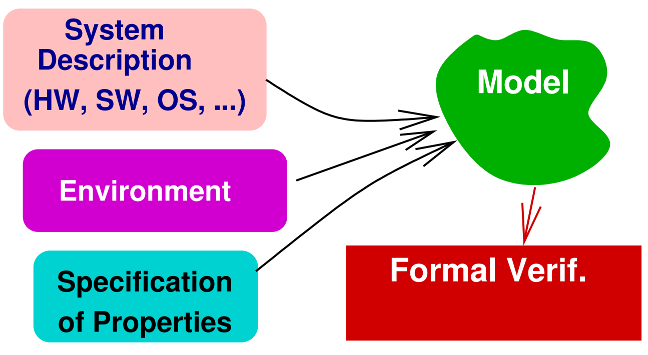









## From Systems to Models to Formal Verification














## Contents

- This general introduction
- Synchronous programming of embedded systems (Lustre/SCADE), compilation of synchronous languages
- Formal verification: model-checking
- Modeling embedded systems for formal validation (synchronous and asynchronous models, expressive power, modeling biases, ...)