Teaching Cyber-Physical Systems: A Programming Approach

Kerstin Bauer and Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern

WESE 2012
Outline

1. Modeling Cyber-physical Systems
   - Cyber-physical vs. Hybrid Systems
   - Languages for Hybrid Systems

2. Extending Synchronous Programs to Hybrid Programs
   - Synchronous Languages
   - From Synchronous to Hybrid Systems

3. Our Course on Cyber-physical Systems
   - Part 1: Modeling Hybrid Systems
   - Part 2: Simulation of Hybrid Systems
   - Part 3: Verification of Hybrid Systems
   - Part 4: Real-Time Requirements
Cyber-physical vs. Hybrid Systems

- **embedded systems** are application-specific computer systems that are integrated in physical systems.
- **cyber-physical systems** are systems whose physical parts are tightly integrated with its embedded systems and/or information systems.

\[ \iff \] cyber-physical system behaviors often include both continuous and discrete behaviors \[ \iff \] hybrid systems.

\[ \iff \] design of cyber-physical systems requires modeling, simulation, and verification of hybrid system behaviors.
**Cyber-physical vs. Hybrid Systems**

- **embedded systems**
  are application-specific computer systems that are integrated in physical systems

- **cyber-physical systems**
  are systems whose physical parts are tightly integrated with its embedded systems and/or information systems

  \[ \equiv \]  
  cyber-physical system behaviors often include both continuous and discrete behaviors \[ \equiv \]  
  hybrid systems

  \[ \equiv \]  
  design of cyber-physical systems requires **modeling, simulation, and verification** of hybrid system behaviors
Hybrid Automata [Alur et al. 1995]

- can be represented as finite state transition systems
- **continuous behaviors**
  - each state is labeled with a set of differential equations
  - these define the change of variable’s values over time
- **discrete behaviors**
  - each transition is labeled by assignments to variables
  - without taking time, variables change on transitions
  - transitions can be taken if trigger conditions are valid
- HA were developed as formal models for verification
- HA do not scale with complex discrete behaviors
  \[ \text{\color{red}{state space explosion!}} \]
Hybrid Systems as Equation Systems?

- many tools describe hybrid systems as equations systems
  - each equation describes the behavior of one variable
  - equation systems can be grouped into modules
  - and modules can be connected by wires

- note: discrete programs can also be represented as equations:

```plaintext
while(σ) {
    x₁ = case (φ₁,₁) do E₁,₁ ... (φ₁,n₁) do E₁,n₁;
    ...
    xₘ = case (φₘ,₁) do Eₘ,₁ ... (φₘ,nₘ) do Eₘ,nₘ;
}
```

- but: software engineers advocate the use of more sophisticated statements
many tools describe hybrid systems as equations systems
- each equation describes the behavior of one variable
- equation systems can be grouped into modules
- and modules can be connected by wires

note: discrete programs can also be represented as equations:

\[
\text{while}(\sigma) \{
\begin{align*}
  x_1 &= \text{case } (\varphi_{1,1}) \text{ do } E_{1,1} \ldots (\varphi_{1,n_1}) \text{ do } E_{1,n_1}; \\
  &\vdots \\
  x_m &= \text{case } (\varphi_{m,1}) \text{ do } E_{m,1} \ldots (\varphi_{m,n_m}) \text{ do } E_{m,n_m};
\end{align*}
\}
\]

but: software engineers advocate the use of more sophisticated statements
Languages for Hybrid Systems

- software engineers advocate the use of more sophisticated statements
  - to increase the readability of the programs
  - to allow the reuse of the programs
- many decades of research in programming languages developed
  - structured programming by statements like loops, ...  
  - encapsulation and reuse by modules
  - generic/polymorphic data types
  - ...
- so why should we be satisfied with equation systems for hybrid systems?
only a few languages and tools exist [Carloni et al. 2006] and these specialize on design phases:

- **modeling**: Modelica, ...
- **simulation**: MATLAB/Simulink, SystemC-AMS, ...
- **verification**: PHAVer, HyTech, ...

however, we need all of these together

using many languages and tools in a single course?
our course on cyber-physical systems is based on our own language Quartz

Quartz was derived from the synchronous language Esterel

in the following part of the talk,

- we consider the synchronous part of Quartz
- its compilation to guarded actions,
- its operational meaning
- and then the extension to hybrid systems
Synchronous Languages

- embedded and cyber-physical systems require programs with a notion of time
- physical time is often too concrete for many reasons
- motivated by the success in HW design, synchronous languages introduced clocks in programs
- synchronous programming languages were developed around twenty years ago in France and Israel
- well-known synchronous languages are Esterel, Lustre, Signal, and also some variants of StateCharts
- our synchronous language: Quartz (it’s a cousin of Esterel)
Micro/Macro Steps

- **reactive systems**
  - executions are divided into discrete reaction steps
  - within each reaction, new inputs are read
  - and current outputs as well as next internal state are computed

- **synchronous languages distinguish between micro- and macro steps**
  - micro steps (= atomic actions) are executed in zero time
  - macro steps require one unit of logical time
  - `pause` statement declares start/end of reaction step
  - macro step = code between two `pause` statements
  - parallel macro steps require the same (logical) time
Micro/Macro Steps

- **reactive systems**
  - executions are divided into discrete reaction steps
  - within each reaction, new inputs are read
  - and current outputs as well as next internal state are computed

- **synchronous languages distinguish between micro- and macro steps**
  - micro steps (= atomic actions) are executed in zero time
  - macro steps require one unit of logical time
  - `pause` statement declares start/end of reaction step
  - macro step = code between two `pause` statements

- parallel macro steps require the same (logical) time
The Synchronous Model of Computation

{ 
    b = true;
    p : pause;
    if(a)
        b = true;
    r : pause;
}

||

{ 
    q : pause;
    if(!b)
        c = true;
    a = true;
    s : pause;
}

- first step: execute \( b = \text{true} \) and reach locations \( p \) and \( q \)
- second step: execute code between \( p \) and \( r \) together with code between \( q \) and \( s \)
- thus, \( a = \text{true} \); \( b = \text{true} \), but not \( c = \text{true} \)
- note the back-and forth communication of the two threads
- a dynamic schedule must be found that respects the data dependencies
Some Statements of Quartz

- **nothing** (empty statement)
- **ℓ : pause** (macro step)
- **x = τ, next(x) = τ** (assignments)
- **if(σ) S₁ else S₂** (conditional)
- **S₁ ; S₂** (sequence)
- **S₁ || S₂** (parallel statement)
- **do S while(σ)** (loop)
- **[weak] [immediate] abort S when(σ)** (abortion)
- **[weak] [immediate] suspend S when(σ)** (suspension)
- **{α x; S}** (local variables)
The Averest Tool

- compile program to set of guarded actions \((\gamma, \alpha)\)
- \((\gamma, \alpha)\) means: whenever \(\gamma\) holds, execute action \(\alpha\)
- actions are
  - immediate assignments \(x = \tau\)
  - delayed assignments \(\text{next}(x) = \tau\)
- causal execution order
  - immediate assignments must be execute in data order
  - i.e., values must be written before read in the same step
  - simulators use value unknown \((\perp)\) and compute reaction as a fixpoint
  - code generators prefer static schedules
Example: ABRO (Program Code)

```plaintext
module ABRO(event ?a, ?b, ?r, !o) {
    loop
        abort {
            {wa: await(a); || wb: await(b);} 
            emit(o);
            wr: await(r);
        } when(r);
    }
```
**Example: ABRO (Guarded Actions)**

```plaintext
system ABRO:
interface:
  a, b, r: input event bool
  o: output event bool
locals:
  w0, wa, wb, wr: label bool
guarded actions:
  !r&w0&!a|r&(wr|wa|wb)|w0 => next(wa)=true
  !r&wb&!b|r&(wr|wa|wb)|w0 => next(wb)=true
  !r&(wr|a&wa&b&wb|!wa&b&wb|!wb&a&wa) => next(wr)=true
  !r&(a&wa&b&wb|!wa&b&wb|!wb&a&wa) => o=true
```
The Averest Tool

Averest Design Flow

Quartz → Compilation → AIF Module

... → Linking → Transformation → AIF System

Transformation → Verification

Transformation → Simulation

AIF System → SW Synthese

AIF System → HW Synthese

http://www.averest.org
continuous transitions are implemented by flow statements

- a flow statement describes a part of a continuous transition
- it lists the differential equations
- and a condition $\sigma$ that defines the end of the continuous transition
- several flow statements may run in parallel
- simply take the union of the differential equations, and terminate as soon as one release condition holds

```latex
flow \{ \\
x_1 \leftarrow s_1; \\
\vdots \\
x_m \leftarrow s_m; \\
drv(y_1) \leftarrow t_1; \\
\vdots \\
drv(y_n) \leftarrow t_n; \\
\} \text{ until}(\sigma);
```
hybrid variables:
- discrete assignments \( x = E \) and \( \text{next}(x) = E \)
- continuous assignments \( x \leftarrow E \) and \( \text{drv}(x) \leftarrow E \)

discrete vs. continuous values
- reactions consist of a discrete and a continuous part
- \( \text{cont}(x) \) denotes the value during the continuous part, while \( x \) denotes the discrete value

release conditions
- new guarded actions \((\gamma, \text{release}(\sigma))\)
- continuous transition terminates if \( \sigma \) holds
A Hybrid Reaction Step

- **discrete reaction**
  - read new inputs (\( \Rightarrow \) partial variable environment)
  - evaluate immediate assignments in zero time
  - \( \Rightarrow \) complete variable environment for this reaction

- **continuous reaction**
  - evaluate continuous actions
  - takes physical time
  - terminates if one release condition becomes true
  - \( \Rightarrow \) updated variable environment

- **state transition**
  - evaluate delayed assignments \( \text{next} (x) = E \)
  - \( \Rightarrow \) new internal state
The Flow Statement

- **statement**
  
  $\ell_1$: `pause;
x = 0.0;

  $\ell_2, \ell'_2$: `flow{ drv(x) <- 1.0; } until(cont(x)>=1.0);

- **meaning:**

  - $\ell_1$: $x = 0.0$
  - $\ell_2$: $drv(x) \leftarrow 1.0$
  - $\ell'_2$: $cont(x) \geq 1.0$
  - $\ell_2$: $cont(x) \geq 1.0$
  - $\ell'_2$: $\neg cont(x) \geq 1.0$
Our Course on Cyber-physical Systems

- course covers modeling, simulation, and verification
- we don’t want to waste time by learning many languages
- instead, we focus on Quartz and Averest
- advantages for students: they
  - can use Quartz programs for modeling,
  - have access to internal system representations,
  - can work interactively with the system,
  - and can therefore write their own simulators and verification procedures

=⇒ goal: stimulate research interest of the students
course covers modeling, simulation, and verification
we don’t want to waste time by learning many languages
instead, we focus on Quartz and Averest
advantages for students: they
can use Quartz programs for modeling,
have access to internal system representations,
can work interactively with the system,
and can therefore write their own simulators and verification procedures

goal: stimulate research interest of the students
Part 1: Modeling Hybrid Systems (4 Weeks)

- syntax and semantics of hybrid automata
- syntax and semantics of Quartz programs
- different aspects like non-zenoness and urgent transitions
- students learn to model hybrid systems using Quartz
- short overview on other tools like MATLAB/Simulink and Modelica is given
- running examples are already introduced here
Part 2: Simulation of Hybrid Systems (2 Weeks)

- brief introduction to numerical solution of differential equations
- methods to improve numeric accuracy (second order integration, etc.)
- students write and evaluate their own simple simulators
- we demonstrate main problems of simulators, i.e. discrete event detection/zero crossing
Part 3: Verification of Hybrid Systems (4 Weeks)

- short introduction to bisimulation and temporal logics
- region graph construction to reduce hybrid systems to finite state systems
- this will again be implemented based on guarded actions by the students
- students learn to specify and formally verify hybrid systems
- students develop and use appropriate abstractions for region graph constructions
Part 4: Real-Time Requirements (2 Weeks)

- Many embedded systems have to react to input stimuli in time to fulfill their requirements.
- Students write a discrete controller and have to derive real-time bounds for it.
- They should also prove that with the derived real-time bounds, the system will work correctly.
- They shall also evaluate given faulty controllers with given worst-case reaction times.
- WCET analysis is seen outside the scope of this course, but would be a nice complement.
Conclusions

- course covers modeling, simulation, and verification
- we advocate: modeling systems by programs
- but we don’t want to waste time by learning many languages
- instead, we focus on Quartz and Averest
- students can interactively work with Averest and can test their prototypes for simulation and verification
References and Further Reading
