Contents

- Introduction
  - Mechatronic systems design challenges
  - Embedded Control Systems software
- Model-driven Design Methodology
- Case study
- Results & Conclusions
Introduction – Mechatronics challenges

- Developing **Reliable** and **Robust** Embedded Control Software for mechatronic applications is **too** costly and **too** time consuming.

- Reasons:
  - Complexity, Heterogeneity, Lack of Predictability, Late Integration

- Approaches to tackle the problem
  - Concurrent Engineering, Model Driven Design, Early Integration

**Trade-off concurrent design flow ⇔ integration efficiency**
Mechatronics: Embedded Control Systems

- Essential Properties Embedded Control Software
  - Dynamic behavior of the physical system essential for SW
  - Real-time constraints with low-latency requirement
  - Dependability: Safety, Reliability
- Layered Software structure

- Model-driven Design
  - Heterogeneous modeling
  - Multiple Models of Computation
  - Multiple Modeling formalisms
ECS Design Methodology

- **Aim**
  - Efficient Concurrent Design
  - Fast Integration
  - Reliable Result

- **Approach:**
  - Model-Driven Design
  - Concurrent Design
  - Code Synthesis
ECS Design Methodology

- **Way of Working**
  - **Abstraction**
    - **Hierarchy**
      - Split into subsystems
      - Cope with complexity
  - **Model-driven design**
    - **Design Space Exploration**
      - Aspect models
      - Make choices
      - Limit solution space
    - **Step-wise refinement**
      - Add detail
      - Lower abstraction
  - **Implementation**
  - **Realization**

- **Concurrent design trajectory**
  - Mechanics, Electronics, **SW: Discrete Event, Continuous Time**
  - **Model-level Early Integration** where needed
Design Methodology Discrete Event

- **Approach**
  - Stepwise & local refinement
  - Verification by simulation & model checking

- **Way of working**
  - System partitioning into concurrent actors
  - C-model : Abstract interactions between concurrent actors
  - M-model : Interaction between different MoCs
  - R-model : Timing low-level behavior
  - Property preserving code synthesis

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Models</th>
<th>Abstraction level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>C-model</td>
<td>High</td>
</tr>
<tr>
<td>Multi MoC</td>
<td>M-model</td>
<td></td>
</tr>
<tr>
<td>Real-time</td>
<td>R-model</td>
<td>Low</td>
</tr>
</tbody>
</table>
Design Methodology Continuous Time

- **Approach**
  - Stepwise & local refinement
    - From model towards controller code
  - Verification by simulation

- **Way of Working**
  - Model & Understand
    - Physical system dynamics
  - Simplify model, derive the control laws
  - Interfaces & target
    - Add non-ideal components (AD, DA, PC)
  - Dependability: Safety, Reliability, …
  - Integrate control laws into ES
    - Scaling/conversion factors
    - Via local refinement:
      - {Software/Processor/Hardware}-In the Loop
Case study Overview

- Goals
  - Apply our methodology
    - Real-world setup with industrial complexity
    - Concurrent model-driven design
    - Trade-off integrated design flow ⇔ partial separated design flow
  - Integration efficiency analysis
    - Comparison with other test cases on the same setup
Case Study Production cell

Production cell demonstrator

- Based on:
  - Stork Plastics Molding machine

- Architecture
  - CPU (ECS) + FPGA (digital I/O)
  - Distributed Control possible

- 6 Production Cell units
  - Action in the production process
    - Molding, Extraction, Transportation, Storage
  - Synchronize with neighbors
  - Deadlock possible on > 7 blocks
Concurrent Design of Embedded Control Software

Case Study Production cell

- Embedded Control System Software Design
  - Jointly
    - Specs, partitioning, interfaces
  - Concurrently
    - SW partitions
  - Jointly
    - SW integration & testing

Diagram: Model-driven concurrent design process

- Continuous Time Control
- Discrete Event Control
- Integration

Diagram: Embedded software

- User interface
  - Non real-time
- Supervisory control & interaction
- Soft real-time
  - Sequence control
  - Loop control
  - Safety layer

Diagram: I/O hardware

- D/A
  - Power amplifier
- A/D
  - Filtering/Scaling

Diagram: Physical system

- Actuators
  - Physical process
- Sensors
Case Study Partitioning & Hierarchy

- Embedded Software
  - Discrete Event partition
  - Continuous/Discrete Time partition

- Based on
  - Top level system model
  - Production Cell Units (PCUs)

- Layered Software structure

- Interface

![Diagram of embedded hardware and software components]

UNIVERSITY OF TWENTE.
Discrete Event Software Design

- Modeling tools: SHESim/Rotalumis
  - POOSL: Parallel Object-Oriented Specification Language
  - SHESim: Graphical tool for model construction and simulation
  - Rotalumis: Fast execution engine built in C++

- C-model: handshake diagram formalized in POOSL model
  - Partitions design into a set of concurrent actors
  - Actors synchronize action by a handshake sequence
  - Models untimed abstract interactions between actors
**Discrete Event Software Design**

- **M-model:**
  - Refinement of C-model
  - Adds interfaces to low-level behavior
  - Focuses on interactions between high-level DE-control and DT/CT loop control (MoC interaction)
  - Externally observable behavior is kept the same

```plaintext
To_Extraction_buffer()()
[last] out ! request;
  sel
    out ? grant{ blocked:=false }
  or
    out ? postpone{ blocked:=true }
    motor ! stop;
    out ? grant{ blocked:=false }
    motor ! start;
les;
sensorL! off{ last:=false; load:=load-1 }
out ! end;
To_Extraction_buffer()()

Discrete_Sensor()()
  sel
    sensor ! on
  or
    sensor ! off
les;
Discrete_Sensor()()

Discrete_Motor()()
  motor ! start;
  motor ! stop;
Discrete_Motor()()
```
Discrete Event Software Design

- **R-model:**
  - Refinement of M-model
  - Adds low-level behavior
  - Both DT and DE behavior
  - Adds timing
  - Again, externally observable functional behavior is kept the same

- **Automatic code synthesis:**
  - Automatic mapping to target platform
  - Property-preserving code generation
  - Building blocks with common interface
  - Mathematically proven timing relation between model and implementation

---

Discrete()()

\[
\begin{align*}
\text{sel} & \quad [ \text{curstate} \land \neg \text{prestate} ] \\
& \quad \text{sensor} \! \text{on} \{ \text{prestate} := \text{curstate} \} \\
\text{or} & \quad [ \neg \text{prestate} \land \neg \text{curstate} ] \\
& \quad \text{sensor} \! \text{off} \{ \text{prestate} := \text{curstate} \} \\
\text{les;} & \\
\text{Discrete}().
\end{align*}
\]

Continuous()()

\[
\begin{align*}
\text{[ curstate} = \text{prestate} ] \\
\text{curstate} := \text{sensor} \text{ Read}; \\
\text{delay} 0.01; \\
\text{Continuous}().
\end{align*}
\]
Continuous Time Software Design

- **Goal**
  - Loop Controller Algorithm in C++ POOSL dataclass
  - Low Level Safety & Sanity Check
  - Event Interface (start/stop/error …)

- **Modeling Tools & Languages: 20-sim**
  - Physical System Model: ODE, bond graphs, data flow
  - Code Synthesis: template based C/C++
Continuous Time Software Design

Controlled Motion Rotation Robot

- Ref Position (m)
- Motor current (A)
- Rotation velocity (rad/s)
- PWM Output (x100%)
- Real Pos (rad)
- Error (m)
- Forward Finished
- Backward Finished

class Controller_Rotation: public PoosiDataClass
{
    /* the model functions */
    void Initialize (double *u, double *y, double time);
    void Read (double *u, double *y);
    void Calculate (double *u, double *y, double time);
    void Write (double *u, double *y);
    void Terminate (double *u, double *y);
};
Integration

- Discrete Event
  - Last iteration:
    - Timing

- Continuous Time / Discrete Time
  - Last iteration:
    - Event interface
    - Target unit test

- Code synthesis
  - Stepwise
  - Partial code generation
  - Template based
  - Simulation feedback

- Target tests
Results & Discussion

- Short integration & testing phase
  - < 2 days, previous case: > week
- Almost running first time right
  - Minor timing issue with magnet on/off traction delay
- Concurrent, but separated design
  - Minimal information exchange
    - Refinements on interfaces, data types, timing
- Required
  - Good partitioning
  - Building blocks approach
- Working setup
Results Movie
Results & Discussion

- Trade-off Concurrent Design ↔ Integration
  - Minimal design interaction
  - Minimal information exchange
    - Refinements on interfaces, data types, timing
  - Designers attitude
    - Focus on own partition, but think across discipline boundaries

- Possible Improvements
  - Model-based integration tests
    - Physical system model could be used to test the final software ↔ Virtual Prototyping
  - Tool support:
    - Automated consistency checks
    - Tool ↔ Tool integration
    - Model ↔ Model interaction
Conclusions

- Mechatronics / Cyber Physical Systems
  - Synergistic design approach
  - Close cooperation between disciplines ⇔ integrated design
- Integrated design ⇔ concurrent design efficiency
  - Trade-off between early integration and late integration time
  - Choice is project specific
- Good partitioning of the mechatronic system
  - Allows concurrent, but partly separated design
- Case: fully integrated design is not always needed
  - More efficient work flow with still predictable integration
- Methodology not limited to SW implementation ECS
  - ECS in FPGA realization available
Ongoing work

- Embedded Control System software for our Humanoid Soccer Robot
  - Vision processing
  - Supervisory control
  - Sequence control
  - Path planning
  - Soccer strategy
  - Low level loop control
    - 24 degrees of freedom