Trends in Embedded Software Engineering

Prof. Dr. Wolfgang Pree
Department of Computer Science
Universität Salzburg

cs.uni-salzburg.at
MoDECS.cc
PREEtec.com
Contents

- Why focus on embedded software?
- Better abstractions
  - timing & concurrency
  - transparent distribution
- Modularization (component models)
- Further challenges
What are we talking about?

Information processing that is tightly integrated with physical processes is called *embedded computing*
Many innovations in embedded systems will result from software features
A special breed: real-time embedded computing
## Hard versus soft real-time systems

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Hard</th>
<th>Soft</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response time</td>
<td>Hard-required</td>
<td>Soft-desired</td>
</tr>
<tr>
<td>Peak-load performance</td>
<td>Predictable</td>
<td>Degraded</td>
</tr>
<tr>
<td>Control of pace</td>
<td>Environment</td>
<td>Computer</td>
</tr>
<tr>
<td>Redundancy</td>
<td>Active</td>
<td>Checkpoint</td>
</tr>
<tr>
<td>Error detection</td>
<td>Autonomous</td>
<td>User assisted</td>
</tr>
</tbody>
</table>

Kopetz97
Sample hard RT system: engine controller

- 10 ms for 360° at 6000 rpm
- Temporal accuracy of 3 µsec required
- Up to 100 concurrent software tasks

=> A hard real-time system is only correct if the concurrent tasks meet the strict real-time deadlines
Priority scheduling anomalies

- reducing task execution times
- adding a processor to a distributed computing system
- relaxing precedence constraints

All these can increase processing time.
Priority inversion problem

- low-priority process keeps high-priority process from running
- might cause deadlocks
Most existing programming models are not sufficient

In order to overcome the inherent problems of concurrent, real-time software development, new abstractions, methods and tools are required.
Automotive sector: more software ...

- More than 80% of innovations in the automotive industry are expected to result from embedded software over the next decade.
- Key goal: ultra-reliable embedded software

The challenge:
- Up to some millions (!) LOC
- Complex distributed computing platform
  - 20 – 80 computing nodes
  - Several bus systems
- Massively concurrent software tasks
Dr. Hans-Georg Frischkorn (Senior Vice President Electric/Electronics, BMW Group):

“Whereas today, software is developed for and tailored to the delivered electronic control unit [processor], a main paradigm shift is expected: **Software in cars will become much more independent of specific hardware**—a similar development to that started in the computer industry 30 years ago.”

Thus, **radically new abstractions, methods and tools are required.**
Abstractions
Lifting the level of abstraction for general-purpose programming

HW + OS

assembly language

C

C++

Java

C#

Oberon

LISP

Prolog

compilation
Abstractions for embedded systems—a missing link

- mathematical modeling (e.g., Simulink)
- embedded software model
- general-purpose programming languages

HW + OS
Abstractions for embedded systems—a missing link

- timing & concurrency
- efficient to implement
- reusable components

mathematical modeling (eg, Simulink)

software model

general purpose programming languages

HW + OS
Key question: What are appropriate abstractions?

- Do they cover the relevant aspects of a design?
- Which embedded software properties do they deliver?
- Do they scale?
- Are the resulting designs understandable?
- Can compilers generate efficient code?
Caveat: visual representations are often associated with “good abstractions“

Real-Time UML: priorities (widely misued in practice), etc.

Simulink: offers only functions for modularization
Required embedded software properties

focus: ultra-reliable embedded control
Determinism

- If a software component is called twice with the same input values at the same time instants, it should both times produce the same output values at the same time instants.

Example: If a control component receives the same sensor values and the same driver input, it should always react exactly in the same way.

- Consequences:
  - Testability: each behavior can be reproduced.
  - Minimal jitter.
Portability (= software standardization)

- The behavior of a software component should be specified independent of its implementation.

Example: The hardware, operating system, or bus architecture can be changed without changing the behavior of the application components.

- Consequences:
  - Upgradability of hardware.
  - Portability of software.
better quality, reduced costs

- deterministic timing behavior
- portability
- automatic code generation

embedded control software
better quality, reduced costs

- deterministic timing behavior
- portability
- automatic code generation
better quality, reduced costs

- deterministic timing behavior
- portability
- automatic code generation
better quality, reduced costs

- deterministic timing behavior
- portability
- automatic code generation
better quality, reduced costs

- deterministic timing behavior
- portability
- automatic code generation
Sample abstraction:
Logical Execution Time (LET)
State-of-the-art programming model leads to non-deterministic, non-portable software

- only constraint: calculation finishes before a deadline

- PROBLEM: results are available as soon as calculation is finished => NO determinism, NO portability
Logical Execution Time (LET), an abstraction invented at UC Berkeley

- results are available EXACTLY after the LET of a task

\[ \text{LET} \geq \text{WCET} \]
LET delivers the required software properties for ultra-reliable embedded systems

- **determinism** => testability, minimal jitter
- **compositionality** => extensibility, reuse
- **software standardization** => portability, upgradability of HW
platform first ...?
LET allows Transparent Distribution

- message sent according to bus schedule (TDMA)
LET might be appropriate for the time-triggered and event-triggered programming models

- xGiotto: has generalized fixed LET
- requires event scoping
Towards embedded software components
Hierarchical structuring is only a first step

A „subsystem“ (= function) in Simulink
current modularization

one embedded control system per computing node
required: domain-adequate language constructs for modularization

```
module M2 {
  import M1;
  ...
  start mode main [period= M1.refPeriod] {
    task [freq=1] sum(M1.inc.o, M1.dec.o);
    ...
  }
}
```
Compositionality

- The behavior of a software component should be independent of the overall system load and configuration.

Example: A new component can be added to a system without changing the behavior of the original components.

- Consequences:
  - Extensibility of systems.
  - Reuse of components.
Next-generation tools for embedded software development
We have developed a modeling language and a set of tools based on LET

- modularization
  - separation of concerns:
    - timing | functionality
    - platform | functional and temporal behavior
  - language construct for modules
- embedded software properties
  - deterministic timing behavior
  - portability (automatic platform code generation)
  - compositionality
- transparent distribution
Tools for developing ultra-reliable embedded software require solid foundations

- tool support
- development methods
- abstractions

visual/interactive editor, compiler, etc.
Giotto/Timing
Definition Language
LET, module
Timeline of beating the concepts into their shapes ...

UC Berkeley
Giotto

MoDECS
Timing Definition Language (TDL)
LET-based tool chain
MoDECS.cc  PREEtec.com

starting point: LET/Giotto (UC Berkeley)
VisualTDL in Simulink

TDL module → component model

platform spec.

TDL compiler + schedule generator

E-/S-Machines + TDLComm

INtime, OSEK, TTA, RT-Linux
Further embedded software challenges
Orthogonal challenges

- appropriate software abstractions as basis of model-based development
- support of multiple models of computation
- reengineering of existing systems
- requirements engineering
- process management
- risk analysis and error management
- usability
- security
- . . .
The end

Thank you for your attention!