Towards a Component Architecture for Hard Real Time Control Applications

Wolfgang Pree  
Josef Templ  
CS, University of Salzburg, Austria  
http://www.modecs.cc/

joint work with  
Thomas Henzinger, EECS, UC Berkeley  
Christoph Kirsch, EECS, UC Berkeley

Giotto project goals (1999/2000)

- **platform-independent, deterministic software model** for embedded control applications
  - platform does not have to be considered from the beginning
  - reusable software components that can be moved between ECUs
- **composition**
- basis for verifiable embedded software
Key ingredients of an embedded control software model (regarding the timing behavior)

- deterministic (FLET)
- appl.-centric
- (not platform-centric)

\[ \text{Giotto} \rightarrow \text{TDL} \]

**AUTOSAR-RTE**

**Basic Software**

OS, Communication Network, FT, ...

**HW**

**E-Machine**

[S-Machine]

**SW-Bus**

OS: OSEK[Time], VxWorks, QNX, ...

Comm.: [TT]CAN, FlexRay, TTA, ...

Programming models for real-time software
(C. Kirsch: Principles of RT Programming, EmSoft 02)

```
soft time <= real time
```

```
soft time = 0
```

```
soft time = real time
```

not deterministic

not composable

(implementation)
Platform-Independent Specification of Computation and Communication Activities

The Fixed Logical Execution Time (FLET) assumption: a precondition for deterministic RT composition
Sample TDL module (1)

```plaintext
module SampleModule {
    type ValPair;

    sensor
        int s1 uses gets1;
        ValPair s2 uses gets2;

    actuator
        int a1 uses seta1;
        ValPair a2 uses seta2;
}
```

Sample TDL module (2)

```plaintext
task t1 {
    input
        int i;
        ValPair i2;
    output
        int o := 1;    // initialization
    state
        int s uses myInit;
        ValPair s2 uses gets2;
        uses t1Impl(i, i2, o, s);  // external impl.
}

task t2 { . . . }
```
Sample TDL module (3)

start mode main [5000 ms] {
    task
        [2] t1(s1, s2)
        [1] if t2Guard(s1)
            then t2 {j := s1; k := t1.o;}
    actuator
        [1] a1 := t1.o;
        [10] if actGuard(s1, t1.o)
            then a2 := t1.o;
    mode
        [1] if failure() then stop()
}

mode stop [1000 ms] {
    mode [1] if restarted() then main()
}

Why we added the module construct to TDL

- unit of deployment (sensors, actuators, tasks, modes)
- unit of distribution
  - several modules on one ECUs (execute multiple control applications in parallel)
  - modules are movable between ECUs (at design time and even at run-time)
Introducing modules in TDL

module EngineControl {
  // Giotto/TDL code consisting of sensor, actuator,
  //    task and mode declarations
}

- named Giotto/TDL program
- provides name space
- loaded into E-machine
- may have a 'start' mode
- module = component

CPU partitioning

- start mode is executed after loading of a module
- module needs CPU time
- executing module = CPU partition
- dynamic loading of (independent) modules = dynamic
  partitioning of an ECU
- in principle unloading is also possible upon request
Module import

module AdvancedCar{
    import EngineControl;
    import BrakeByWire;
    import ...;
    // Giotto/TDL code consisting of sensor, ... declarations;
    //     may access public elements of imported modules
}

- import specifies static dependencies between modules
- allows the decomposition of large applications
- = static partitioning of an ECU

Information hiding

module EngineController {  
    public const maxRpm = 6500;
    //... more code
}

- sensors may be read by multiple modules
- actuator updates by multiple modules must be prevented
- TDL-rule: actuator update only in declaring module

=> modules partition the set of actuators
Module extension (I)

module ExtendedEngineControl {
    import EngineControl;
    actuator int newActuator uses setNewActuator;
    task newTask ...; //provides output variable res
    mode normal extends EngineControl.normal {
        task [1] newTask(...);
    }
}

- experimental feature
- allows *hot deployment* of new functionality
Scheduling issues

- **global hyper period** ‘hp’ = GCD of all activity periods of all modes of all partitions
  => activity periods should not be relative primes
- preemptive EDF scheduling per mode
- every partition gets a slot in hp
- **slot size allocated for the most CPU intensive mode**
- if all partitions execute most CPU intensive mode, CPU may be utilized up to 100%
- **dynamic loading** => dynamic scheduling + rescheduling if hp changes (background task)
- we experiment also with RM scheduling (OSEK)

Implementation status

- **TDL Compiler** implemented in Java using Coco/R
- **Java-based E-machine** with loading and execution of multiple modules is running
  - uses Java threads with suspend()/resume()
  - not strictly real time
  - alternative considered is ‘realtime’ Java
- in parallel we have implemented a **C-based E-machine** on top of OSEK, OSEK/Time etc.
Module distribution

modeling the timing + functionality of a module

functionality model, i.e., control laws

eg, Simulink-Editor

Timing model (TDL)

eg, Simulink Simulation Environment
fully automated generation of executable(s)

functionality model, i.e., control laws

eg, Simulink-Editor

Task1Impl
Task2Impl

... .

M1

TDL-Editor

timing model (TDL)

C-Code Generator

TDL Compiler

C Compiler

Linker

→ M1

independently developed TDL modules

eg, TDL+
Simulink

requirements

functional model

generated

application code

test

verification

validation

eg, TDL+
Simulink

generated

M1

M2

M3

eg, TDL+
Simulink

requirements

functional model

generated

application code

test

verification

validation
The modules can be assigned to any ECU as long as the time-safety property is not violated.

<table>
<thead>
<tr>
<th>module</th>
<th>@</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>ECU1</td>
</tr>
<tr>
<td>M2</td>
<td>ECU2</td>
</tr>
<tr>
<td>M3</td>
<td>ECU1</td>
</tr>
</tbody>
</table>

sample movement of modules (functionality) at run-time with time-safety checking
sample movement of modules (functionality) at run-time with time-safety checking
sample movement of modules (functionality) at run-time with time-safety checking

Sample movement of modules (functionality) at run-time with time-safety checking
sample movement of modules (functionality) at run-time with time-safety checking

Abstraction levels for control system development

- TDL language and component architecture
- state-of-the-art methods and tools for distributed development (e.g., DaVinci, SysDesign)
- operating/network system
- microcontroller abstraction
- ECU hardware

application-centric and deterministic (FLET)
platform-centric and/or non-deterministic (priorities, etc.)
platform-specific
The end

Thank you for your attention!