Distributed, Time-Safe TDL Execution—Concepts, Tools and Run-time Infrastructure

Emilia Farcas
Prof. Dr. Wolfgang Pree
Dr. (ETH) Josef Templ
Department of Computer Science
Universität Salzburg

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

- Motivation
- Transparent Distribution
- Bus Schedule Generation Tool
- TDL Run-time Environment
- Tool Chain
- Development Process and Module Distribution
Motivation

Some benefits of distribution:
- Fault tolerance
- Scalability
- Less wiring

MOST-Bus

CAN-Bus
Introduction to Distributed TDL

<table>
<thead>
<tr>
<th>Unit of distribution:</th>
<th>TDL module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior:</td>
<td>as if executed locally</td>
</tr>
<tr>
<td>Communication:</td>
<td>via broadcast (bus)</td>
</tr>
<tr>
<td>Medium access control:</td>
<td>TDMA (time-slotting)</td>
</tr>
<tr>
<td>Cooperation model:</td>
<td>Producer-Consumer (Push)</td>
</tr>
</tbody>
</table>
The advantages of transparent distribution

- The **functional and temporal behavior** of a system is the same no matter where a component is executed.
- Developer’s perspective: **NO difference** between local and distributed execution of components.
- OEM-supplier perspective: the components can be developed independently.
transparent distribution in a nutshell

software component
(hard real-time control system)
transparent distribution in a nutshell
transparent distribution in a nutshell
transparent distribution in a nutshell

component M2 added later, if required even at run-time

M2
M1

exactly defined communication semantics (TDL programming model)
transparent distribution in a nutshell

- deterministic timing and communication behavior
- independent of the computing and communication platform

=> portability through automatic code generation and run-time environment
Example of Distributed TDL
message sent according to bus schedule (TDMA)
if the consumer runs slower e.g. by a factor of 2
- redundant message are avoided
- saves bandwidth
if the consumer needs a variable later than the producer’s FLET
the release of the consumer can be delayed until the message with the input variable is received
Bus Schedule Generation Tool
What Does the Tool Do?

It generates a global bus schedule file, which contains the following information:

- Which node has to send a packet and when.
- Which nodes have to receive a packet and when.
- The content for bus packets (a corresponding datagram, which has one or more items).
What Does the Tool Need as Input?

- TDL modules
- Platform description file
  - module to node assignment
  - physical bus properties (e.g., bus frequency, protocol overhead, inter frame gaps, min/max payload)

The tool automatically detects:
- Who has to communicate with whom.
- Which messages are needed in a communication cycle (bus period).
Who Has to Communicate with Whom

Result: a set of messages.

- A message has:
  - a Producer
  - one or more Consumers
  - size.

- Producers: sensors, task output ports.
- Consumers: actuators, task input ports, guard arguments.
Messages Needed in a Bus Period

Result: a set of message instances, with individual timing constraints:
- Release Offset
- Deadline

- Basic Producer-Consumer:
  - Send messages with the frequency of the Producer.
  - Message deadline = Producer LET.
  - \( \text{BusPeriod} = \text{LCM(Producer.period)} \)

- Optimized Producer-Consumer:
  - Send messages only when they are needed by the Consumers.
  - Message deadline depends on the optimization (e.g., = consumer LET).
  - \( \text{BusPeriod} = \text{LCM(Producer.period, Consumer.period)} \)
Message Deadline in Optimization II

- Producer Deadline
- Message Deadline

Node 1:
- Comm 1
- LET1
- Message M1
- Node 2:
- Comm 2
- LET2
- Message M2

Time
Message Scheduling

Current approach:
- Scheduling in 2 steps:
  - Schedule first the messages.
  - Schedule then the tasks with deadlines constraints from messages.
- Optimizations:
  - Build bus schedulers which allow more flexibility for the task scheduler.
  - Try several bus schedulers and get feedback from the Time-Safety-Check (TSC) for tasks.
  - Schedule individual messages or merge messages sent from the same node.
Scheduling Algorithms

- Heuristic algorithm - “Latest Deadline Last” - LDL
  - Adapted from Reversed EDF (Latest Release Time - LRT) - treats deadlines as release times and vice versa
  - Schedule messages as late as possible

  **Heuristic Algorithm**
  
  Success
  
  Schedule

  Failure
  
  - Not enough SW/HW performance
  - There is a schedule but the algorithm fails to find it.

- Optimal algorithm
  - Branch and bound search
  - Exponential complexity in the worst case
Latest Deadline Last - Example

Released messages \{m1, m2, m3\}

LDL scheduling \{m2, m1, m3\}
Latest Deadline Last

- Sorts the list of messages by:
  - Key1 = message deadline
  - Key2 = message release time
  - Key3 = producer deadline.

- Bus Scheduler is non-preemptive and just schedules the messages in the resulted order.
  - Starts from the end of the Bus Period and goes backwards.
  - Merges messages if they have to be sent by the same node, and are adjacent.
Search Scheduler - Example

LDL scheduling failure \{m_2, m_1\}  

Search scheduler \{m_1, m_2\}
Bus Properties as Constraints

- Relevant for:
  - Merging messages (min/max payload)
  - WCCT (Bps, protocol overhead)
  - Time alignment (inter frame gaps, clock resolution)
  - Control packets (time synchronization)

- Clock Resolution:
  - TDL time unit is microsecond (us).
  - Different platforms have a given clock resolution (e.g., 1ms or 100us).
  - Bus communication is computed in microseconds or even nanoseconds.
Merging Messages and Clock Resolution

Scheduled m1
Aligned m1
Scheduled m2
Aligned m2
Merged m1+m2
Aligned m1+m2

m1
m2
m1 + m2
m2 + m1

Scheduled m1 + Scheduled m2

Aligned m1 + Aligned m2

Merged m1 + m2

Aligned m1 + m2

m1
m2

Time
We do Various Measurements as Basis for Optimizations

Metrics relevant for efficient bus utilization:
- Throughput
- Bus utilization
- Average data efficiency
- Maximum and average sending rates
- Maximum and average receiving rates

Metrics relevant for flexibility in task scheduling:
- Minimum and average release-send intervals
- Minimum and average relative release-send intervals
TDL Run-time Environment
TDL Run-time Environment

Environment

Sensors/Actuators

Drivers

Ports

Message Buffers

Tasks

E-machine

TDL Scheduler (S-machine)

TDL Comm

Platform

r/w

r/w

r/w

r/w

r/w

calls

dispatche

releases

calls
E-Machine Operation

- Executes E-code instructions at logical time instances
- Implementation is platform dependent (OSEK, InTime, RTLinux, Java)
- It is fast and lightweight (e.g. 8KB for OSEK E-machine).

- Supports three kinds of module execution: local, push, and stub.
E-Machine

M2 imports M1
M4 imports M1, M3
E-Machine

M2 imports M1
M4 imports M1, M3

node 1

M1 push

node 2

M2 local
M3 push
M1 stub

node 3

M4 local
M1 stub
M3 stub

bus
- **release driver** - copies input arguments
- **terminate driver** - copies output arguments
- **start driver** - calls task impl. function
- **stop driver** - *noop*
- release driver - same as LOCAL
- terminate driver - same as LOCAL
- start driver - same as LOCAL
- stop driver - *copy results to TDLcomm*
- terminate driver - *copies from TDLcomm to output*
- uses special E-code that contains only terminate driver calls at appropriate time instances => stub mode
Transparent Distribution

node1

comm1

bus

comm2

node2

M1 stub

M2

LET1

stop

driver

let

terminate

driver

release

driver

LET2

time
Tool Chain
Tool Chain

- .tdl
- Compiler
- Bus Scheduler plugin
- Platform plugin
- .ecode
- E-machine
- functionality code
- busch
- platform specific
Development process and TDL module distribution
modeling the timing + functionality of a module

functionality model, i.e., control laws

eg, Simulink-Editor

Timing model (TDL)

eg, Simulink-Editor

TDL-Editor

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

functionality model, i.e., control laws

Task1Impl
Task2Impl
...

eg, Simulink-Editor

M1

TDL-Editor

timing model (TDL)

C-Code Generator

C Compiler

TDL Compiler

Linker

M1
independently developed TDL modules

M1
- requirements
- functional model
- application code
- test
- verification
- validation

eg, TDL+ Simulink

generated

M2
- requirements
- functional model
- application code
- test
- verification
- validation

eg, TDL+ Simulink

generated

M3
- requirements
- functional model
- application code
- test
- verification
- validation

eg, TDL+ Simulink

generated
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>

<table>
<thead>
<tr>
<th>module</th>
<th>@</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>ECU1</td>
</tr>
<tr>
<td>M2</td>
<td>ECU1</td>
</tr>
<tr>
<td>M3</td>
<td>ECU1</td>
</tr>
</tbody>
</table>
Thank you for your attention!