Model-based development of deterministic, portable real-time software components

Prof. Dr. Wolfgang Pree
Univ. Salzburg
Overview

- Timing Definition Language (TDL) in a nut shell
- TDL execution
- TDL extensions
- Transparent distribution of TDL components
- TDL development process
- TDL tools
- TDL advantages
TDL in a nut shell
What is TDL?

- A high-level textual notation for defining the timing behavior of a real-time application.

- TDL covers all aspects that are required to model safety-critical software as found, for example, in cars, airplanes, Unmanned Aerial Vehicles (UAVs), automation systems
  - seamless integration of time-triggered (synchronous) and event-triggered (asynchronous) activities
- TDL’s specification is public; could form the basis of an open standard
TDL is conceptually based on Giotto

Giotto project: 2000 – 2003, University of California, Berkeley

TDL = Giotto concepts
+ Syntax
+ Component Architecture
+ Tool Chain
+ Extensions
TDL tools

- TDL:Compiler
- TDL:VisualCreator
- TDL:VisualDistributor
- TDL:VisualAnalyzer

- requires Java 1.5 or later
- optional integration with MATLAB/Simulink from The MathWorks

- TDL:Machine (alias E-Machine)
  - platform-specific, typically in C
TDL tool chain

* Simulink, OSEK, dSpace, ARM, AES, INtime, RTLinux, ...
TDL tool chain

* Simulink, OSEK, dSpace, ARM, AES, INtime, RTLinux, ...
TDL tool chain

TDL:VisualCreator

TDL:Compiler

.TDL

.TDL:Compiler

.TDL:Machine

.functionality

code

Platform

plugin*

platform

specific

platform

specific

platform

specific

Modeling with TDL
TDL tool chain

TDL:VisualCreator

TDL:Compiler

.TDL

.functionality code

.TDL:Machine

.TDL:VisualDistributor

Platform plugin*

AST

.platform specific

.platform specific
TDL tool chain
TDL programming model: multi-rate, multi-mode systems (I)
TDL programming model: multi-rate, multi-mode systems (II)

LET-semantics

![Diagram of LET-semantics with tasks and modes]

© 2009, W. Pree
Logical Execution Time (LET) abstraction

ET <= WCET <= LET

results are internally available at ‘stop (ET)’
results are externally visible at ‘terminate’
spare time between ‘stop’ and ‘terminate’
LET advantages

- observable (logical) timing is identical on all platforms
- allows for simulation
- allows for composition
- allows for distribution
Periodic execution in TDL modes

- Every mode has a fixed period.
- A task \( t \) has a frequency \( f \) within a mode.
- The mode period is filled with \( f \) task invocations.
- The LET of a task invocation is \( modePeriod / f \).
TDL module: modes, sensors and actuators form a unit
Motivation for TDL modules

- e.g. modern cars have up to 80 electronic control units (ECUs = nodes)
- ECU consolidation is a topic
- run multiple programs on one ECU
- leads to TDL modules
TDL modules

- ProgramX is called a *module*
- modules may be independent
- modules may also refer to each other
- modules can be used for multiple purposes
Example: Receiver imports from Sender module

module Sender

mode 1
- task 1 [10 ms]
- task 2 [20 ms]

mode 2
- task 1 [5 ms]
- task 3 [1 ms]

module Receiver

mode 1
- task 1 [5 ms]
  - task 2 [10 ms]
  - task 3 [5 ms]

mode 2
- task 3 [10 ms]

mode 3
- task 3 [5 ms]
  - task 4 [1 ms]
Example: Receiver imports from Sender module

module Sender

module Receiver

© 2009, W. Pree
Modeling with TDL
Example: Receiver imports from Sender module
TDL syntax by example

module Sender {

    sensor boolean s1 uses getS1;
    actuator int a1 uses setA1;

    public task inc {
        output int o := 10;
        uses incImpl(o);
    }

    start mode main [period=5ms] {
        task
            [freq=1] inc(); // LET = 5ms / 1 = 5ms
        actuator
            [freq=1] a1 := inc.o; // update every 5ms
        mode
            [freq=1] if exitMain(s1) then freeze;
    }

    mode freeze [period=1000ms] {} 
}
module Receiver {

    import Sender;
    ...
    task clientTask {
        input int i1;
        ...
    }
    mode main [period=10ms] {
        task [freq=1] clientTask(Sender.inc.o); // LET = 10ms / 1 = 10ms
        ...
    }
}
LET-behavior (independent of component deployment)

Sender

inc inc inc inc

5 ms

communication of inc’s output to clientTask

Receiver

clientTask clientTask

10 ms
TDL execution
TDL run-time environment

- based on a **virtual machine**, called TDL:Machine
- executes virtual instruction set, called **E-code** (embedded code)
- E-code is executed at logical time instants
- synchronized logical time for all components
- E-code generated by TDL compiler from TDL source
- covers one mode period
- contains one E-code block per logical time instant
one TDL:Machine per node

TDL:Machine

single node
one TDL:Machine per node
TDL extensions
TDL slot selection

- $f = 6$

<table>
<thead>
<tr>
<th>Mode Start</th>
<th>Mode Period</th>
<th>Mode End</th>
</tr>
</thead>
<tbody>
<tr>
<td>slot 1</td>
<td>slot 2</td>
<td>slot 3</td>
</tr>
<tr>
<td>slot 4</td>
<td>slot 5</td>
<td>slot 6</td>
</tr>
</tbody>
</table>
TDL slot selection

- $f = 6$
- task invocation 1 covers slots 1 – 2
- task invocation 2 covers slots 4 – 5
TDL slot selection allows the specification of ...

- an arbitrary repetition pattern
- the LET more explicitly
- gaps
- task invocation sequences
- optional task invocations
Physical layer / E-code blocks

- E-Code block follows fixed pattern:
  1. task terminations
  2. actuator updates
  3. mode switches
  4. task releases
E-code compression

- E-code blocks may be identical
- compression feature would be welcome
- new instruction:
  
  \[
  \text{REPEAT } \langle \text{targetPC} \rangle, \langle N \rangle
  \]
- jumps \( N \) times to \( \text{targetPC} \), then to \( PC + 1 \).
- uses a counter per module
- counter is reset upon mode switch
Adding asynchronous activities

Priority levels

- black: highest priority (E-code)
Adding asynchronous activities

Priority levels
- black: highest priority (E-code)
- red: lower priority (synchronous tasks)
Adding asynchronous activities

Priority levels
- black: highest priority (E-code)
- red: lower priority (synchronous tasks)
- blue: lowest priority (asynchronous activities)
Asynchronous activities rationale

- event-driven background tasks
- may be long running
- not time critical
- could be implemented at platform level, but:
  - platform-specific
  - unsynchronized data-flow to/from E-machine

- support added to TDL
- **Goal**: avoid complex synchronization constructs and the danger of deadlocks and priority inversions
Kinds of asynchronous activities

- task invocation
  - similar to synchronous task invocations except for timing
  - input ports are read just before physical execution
  - output ports are visible just after physical execution
  - data flow is synchronized with E-machine

- actuator updates
  - similar to synchronous actuator updates except for timing
  - data flow is synchronized with E-machine
Trigger Events

- hardware and software interrupts
- periodic asynchronous timers
- port updates

Use a registry for later execution of the async activities.

Parameter passing occurs at execution time.

Registry functions as a priority queue.
Transparent distribution
TDL module-to-node-assignment (example)

Sender
ECU1

FlexRay bus

ECU2
Receiver
Transparent distribution of TDL components:

- Firstly, at runtime a set of **TDL components behaves exactly the same**, no matter if all components are **executed on a single node or if they are distributed across multiple nodes**. The logical timing is always preserved, only the physical timing, which is not observable from the outside, may be changed.

- Secondly, **for the developer of a TDL component, it does not matter where the component itself and any imported component are executed.**
sample physical execution times on ECU1/ECU2

Sender

ECU1

inc inc inc inc

5 ms

Receiver

ECU2

clientTask clientTask

10 ms
Constraints for automatic schedule generation

Sender

ECU1

Receiver

ECU2

communication window

communication window

inc

inc

inc

inc

5 ms

10 ms

clientTask

clientTask

stop

(stop (WCET))

(stop (WCET))

© 2009, W. Pree

Modeling with TDL
Bus schedule generation

Sender

ECU1

FlexRay bus

Receiver

ECU2

communication window

5 ms

clientTask

10 ms

local buffer

communication window

local buffer
TDL: VisualDistributor maps TDL modules to nodes
TDL-based development process
preeTEC tools in the V model

TDL:VisualCreator
in Matlab®/Simulink®

requirements
+ timing

functional model

C

application code

generated for
platform 1

for
platform 2

generated for
platform 2

. . .

test

verification

© 2009, W. Pree
TDL tools: status quo
Status quo

- ready
  - TDL:VisualCreator (stand-alone or in Matlab®/Simulink®)
  - TDL:VisualDistributor (extensible via plugins; currently a plugin for FlexRay is available as product, together with plug-ins for various cluster nodes such as the MicroAutoBox, and Renesas–AES)

The TDL:VisualDistributor is available as stand-alone tool or in Matlab®/Simulink® and provides the following features:
  - Communication Schedule Generator
  - TDL:CommViewer
  - automatic generation of all node-, OS- and cluster-specific files
  - TDL:Compiler
  - TDL:Machine for Simulink, mabx, AES, ARM, INtime, OSEK
  - seamless integration of asynchronous events with TDL
  - multiple slot selection (decoupling of LET and period; eg, for event modeling)
  - harnessing existing FlexRay communication schedules (via FIBEX) for their incremental extension
  - TDL:VisualAnalyzer (recording and debugging tool)

- work in progress
  - ‘intelligent’ FlexRay parameter configuration editor
  - TDL:Machine for further platforms (ARM, etc.)
TDL advantages
The TDL way:

- Develop once
- Deploy on any platform

Components:
- 3dSpace mabx
- TT Ethernet
- (TT) CAN
- ARM

FlexRay-based communication

Single node
State-of-the-art:

C-a
3 dSpace mabx

C-b
2 DeComSys Renesas

C-c
...

...
TDL advantages

- **transparent distribution**: developers do not have to consider the target platform (processor, OS, communication bus, etc.), which could be a single node or a distributed system.

- **time and value determinism**: same inputs imply corresponding same outputs
  - significantly improved reliability
  - simulation = behavior on execution platform
developers have to deal with 3 dimensions

functionality
developers have to deal with 3 dimensions

timing

functionality
developers have to deal with 3 dimensions

- timing
- functionality
- platform
TDL reduces this to 2 dimensions

timing

functionality

platform
TDL reduces this to 2 dimensions

- **timing**
  - significantly simplified

- platform
  - functionality
TDL allows your developers to focus on the functionality.
TDL allows your developers to focus on the functionality

3D → 1,5D
TDL leads to enormous gains in efficiency and quality

eg, FlexRay development reduced by a factor of 20
- 1 person year => 2 person weeks

deterministic system:
- simulation and executable on platform always exhibit equivalent (observable) behavior
- time and value determinism guaranteed

flexibility to change topology, even platform
- automatic code generators take care of the details
Thank you for your attention!