Timing Definition Language (TDL)—Concepts, Code Generation and Tools

Wolfgang Pree

Embedded Software & Systems Research Center
Department of Computer Sciences
Univ. Salzburg
Overview

- Motivation
- Timing Definition Language (TDL) and TDL tools
- Communication schedule generation
embedded systems characteristics


- embedded system := an engineering artifact involving computation that is subject to physical constraints
  - reaction to a physical environment (control theory)
  - execution on a physical platform (computer engineering)
- => the powerful separation of
  - computation (software)
  - from physicality (platform and environment), which has been one of the central ideas enabling the science of computing, **does not work for embedded systems.**
Scientific challenge: appropriate abstractions

Component C

deploy on any platform

develop once

3 dSpace mabx

... FlexRay-based communication

TT Ethernet

(TT) CAN

ARM

single node

...
Scientific challenge: appropriate abstractions

Component C

develop once

compiler, automatic code generation

3 dSpace mabx

... FlexRay-based communication

TT Ethernet

(TT) CAN

ARM

... single node
State-of-the-art:

- C-a: 3 dSpace mabx
- C-b: 2 DeComSys Renesas
- C-c: ...

...
Computing needs time
(Edward Lee, UC Berkeley)

- define a computing model that explicitly considers time & concurrency
- achieve a paradigm shift from platform-centric to platform-independent embedded software development
  - put time into programming languages
  - find adequate abstractions for concurrency
  - define component models that deliver software properties such as determinism and portability and that allow transparent distribution
  - rethink the hardware/software split
  - make memory management predictable regarding timing
  - make network communication deterministic
  - define a computational dynamic systems theory
  - etc.
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 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

![Diagram of TDL tool chain]

- **.tdl** → **TDL:Compiler** → **.ecode** → **TDL:Machine**

  - **TDL:Compiler** processes **.tdl** to generate **.ecode**.
  - **Platform plugin** is platform-specific.
  - **TDL:Machine** produces **functionality code**.

* Platform-specific plugins include:
  - Simulink, OSEK, dSpace, ARM, AES, INtime, RTLinux, ...

---

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

TDL:VisualCreator

TDL:Compiler

Platform plugin*

Platform specific

AST

.functionality code

.TDL:Machine

.code

.TDL:Compiler

.TDL:VisualCreator

.platform specific
TDL tool chain

TDL:VisualCreator

TDL:Compiler

TDL:Machine

platform specific

Platform plugin*

functionality code

platform specific

.TDL

AST

.ecode

.src, presented by W. Pree
modeling and platform mapping

TDL:VisualCreator

Component C

TDL:VisualDistributor

...
run-time analysis
TDL programming model: multi-rate, multi-mode systems (I)
TDL programming model: multi-rate, multi-mode systems (II)

LET-semantics

(mode 1
  task 1 [10 ms]
  task 2 [20 ms]
  mode 2
  task 1 [5 ms]
  task 3 [1 ms]

S1 ↔ S2
A1 ↔ A2 ↔ A3)
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 \( \text{modePeriod} / f \).
TDL component model: modes, sensors and actuators form a unit, the TDL module
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 allows specification of mulit-rate, multi-mode, multi-program systems
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
Example: Receiver imports from Sender module
Example: Receiver imports from Sender module

module Sender

module Receiver

Example: Receiver imports from Sender module

public

private
TDL syntax by example

```java
tdl 

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] {}
}
```

Sender (mode main)
module Receiver {

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

- **Sender**
  - Inc
  - Inc
  - Inc
  - Inc

  5 ms

- **Receiver**
  - ClientTask
  - ClientTask

  10 ms

Communication of Inc's output to clientTask
Selected TDL constructs
TDL slot selection

- $f = 6$
TDL slot selection

- \( f = 6 \)
- Task invocation 1 covers slots 1 – 2
- Task invocation 2 covers slots 4 – 5

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

**Logical**

- Task invocation 1
- Task invocation 2

**Modes**

- Mode Start
- Mode Period
- Mode End

**Time**
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
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 TDL:Machine

- **actuator updates**
  - similar to synchronous actuator updates except for timing
  - data flow is synchronized with TDL: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.
Sample TDL module

```java
module GPS {
    import INS;

    public output INS.Vector pos;
    INS.Vector vel;
    long timeStamp;

    public task receiveGPS {
        input long time;
        uses getGPSData(time, pos, vel, timeStamp);
    }

    asynchronous {
        [interrupt = iGPS, priority = 2] receiveGPS(INS.time);
    }
}
```
Transparent distribution and automatic schedule generation
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
Constraints for automatic schedule generation

Sender
ECU1

inc
inc
inc
inc

5 ms
communication window

stop (WCET)

Receiver
ECU2

clientTask
clientTask

10 ms
communication window

stop (WCET)
Sample bus schedule
TDL: VisualDistributor maps TDL modules to nodes and generates communication schedule for distributed systems.
Thank you for your attention!