Timing Definition Language
TDL

Wolfgang Pree
Founder
chrona.com
Overview

- Timing Definition Language (TDL) in a nut shell
- TDL development process
- TDL tools
- Benefits of TDL
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 programming model: multi-rate, multi-mode systems (I)
TDL programming model: multi-rate, multi-mode systems (II)

LET-semantics

Diagram showing a system with two modes: mode 1 and mode 2. Mode 1 contains two tasks: task 1 [10 ms] and task 2 [20 ms]. Mode 2 contains tasks: task 1 [5 ms] and task 3 [1 ms]. The system transitions between modes S1 and S2, and actions A1, A2, and 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 $modePeriod / f$. 
TDL module: modes, sensors and actuators form a unit
Example: Receiver imports from Sender module

module Sender

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

S1
S2
A1
A2
A3

module Receiver

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

S
S
A1
A2

mode 2
  task 3 [10 ms]

mode 3
  task 3 [5 ms]
  task 4 [1 ms]
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]

mode 2
- task 3 [10 ms]

mode 3
- task 3 [5 ms]
- task 4 [1 ms]

S1 → S2
S1 → A1
S1 → A2
S1 → A3
A1 → S
S2 → S
S2 → A1
S2 → A2
S2 → A3
A2 → S
A2 → A1
A2 → A3
A3 → S
A3 → A1
A3 → A2

public
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] {}
}

Sender (mode main)
Module import

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
one TDL:Machine per node

node1

TDL:Machine

node2

TDL:Machine

node3

TDL:Machine
Some additional TDL constructs
TDL slot selection

- $f = 6$

Diagram showing the relationship between Mode Start, Mode Period, and Mode End, with slots labeled 1 to 6.
Modeling with the Timing Definition Language (TDL)

- \( f = 6 \)
- 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
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)

Mode Start

- slot 1
- slot 2
- slot 3
- slot 4
- slot 5
- slot 6

Mode Period

- task invoc. 1
- task invoc. 2

Mode End

Logical

Physical

E-code block
E-code block
E-code block
E-code block
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.
Threads and critical regions

- Hardware interrupts, timer: highest priority
  - On interrupt, on timer

- Synchronous activities (E-machine): high priority
  - On port update

- Asynchronous activities: lowest priority
  - On port update

- Ports

- Registered events
  - Enqueue()
  - Dequeue()
Synchronization requirements

- Async activities don’t preempt anything.
- E-machine may preempt async activities.
- Hardware interrupts (incl. timers) may preempt everything incl. other hardware interrupts.
- We need a very robust thread safe registry.
- We need a very efficient `enqueue` operation
  - for serving hardware interrupts quickly
  - for efficient synchronous port update triggers
- `dequeue` is done asynchronously and may be slower.
Transparent distribution
TDL module-to-node-assignment (example)

Sender
ECU1

ECU2
Receiver

FlexRay bus
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

Receiver

ECU2

clientTask

clientTask

communication window

communication window

5 ms

10 ms

stop

(WCET)

stop

(WCET)

t
Bus schedule generation

Sender

ECU1

FlexRay bus

Receiver

ECU2

Modeling with the Timing Definition Language (TDL)
Chrona’s VisualDistributor maps TDL modules to nodes
TDL-based development process
Chrona tools in the V model

- Chrona VisualCreator
  - in Matlab®/Simulink®
  - + timing
  - generated for platform 1
  - for platform 2

- Chrona VisualDistributor
  - generated for platform 1
  - for platform 2

- requirements
- functional model
- test
- verification
TDL tools
Modeling with the Timing Definition Language (TDL)

**TDL tools**

- Chrona TDL-Compiler
- Chrona VisualCreator
- Chrona VisualDistributor
- Chrona VisualAnalyzer

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

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

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

- .tdl
  - TDL:Compiler
    - AST
      - Platform plugin*
        - platform specific
      - platform specific
  - .ecode
    - TDL:Machine*
      - functionality code

* Simulink, OSEK, dSpace, ARM, AES, INtime, RTLinux, ...
Chrona VisualCreator

TDL tool chain

- .tdl
- TDL:Compiler
- .ecode
- TDL:Machine
- AST
- Platform plugin*
- platform specific
- functionality code
- platform specific
- platform specific
Modeling with the Timing Definition Language (TDL)

Chrona VisualCreator

TDL tool chain

Chrona VisualDistributor

TDL:Compiler

functionality
code

.TDL

.code

.TDL:Machine

Platform plugin*

AST

platform specific

platform specific

platform specific

platform specific
TDL tool chain

Tacna VisualAnalyzer: communication analysis

node1

node2

node3
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, RT-Linux
  - 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 (NEC, etc.)
TDL advantages
The TDL way:

- Develop once
- Deploy on any platform

Component C

- 3 dSpace mabx
- ... FlexRay-based communication
- TT Ethernet
- (TT) CAN
- ARM

Single node
State-of-the-art: variant for each platform

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
embedded software developers have to deal with 3 dimensions
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

Significantly simplified
TDL allows your developers to focus on the functionality
TDL allows your developers to focus on the functionality

3D $\rightarrow$ 1,5D

![Diagram with axes: timing, platform, and functionality]
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!