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

O.Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Pree

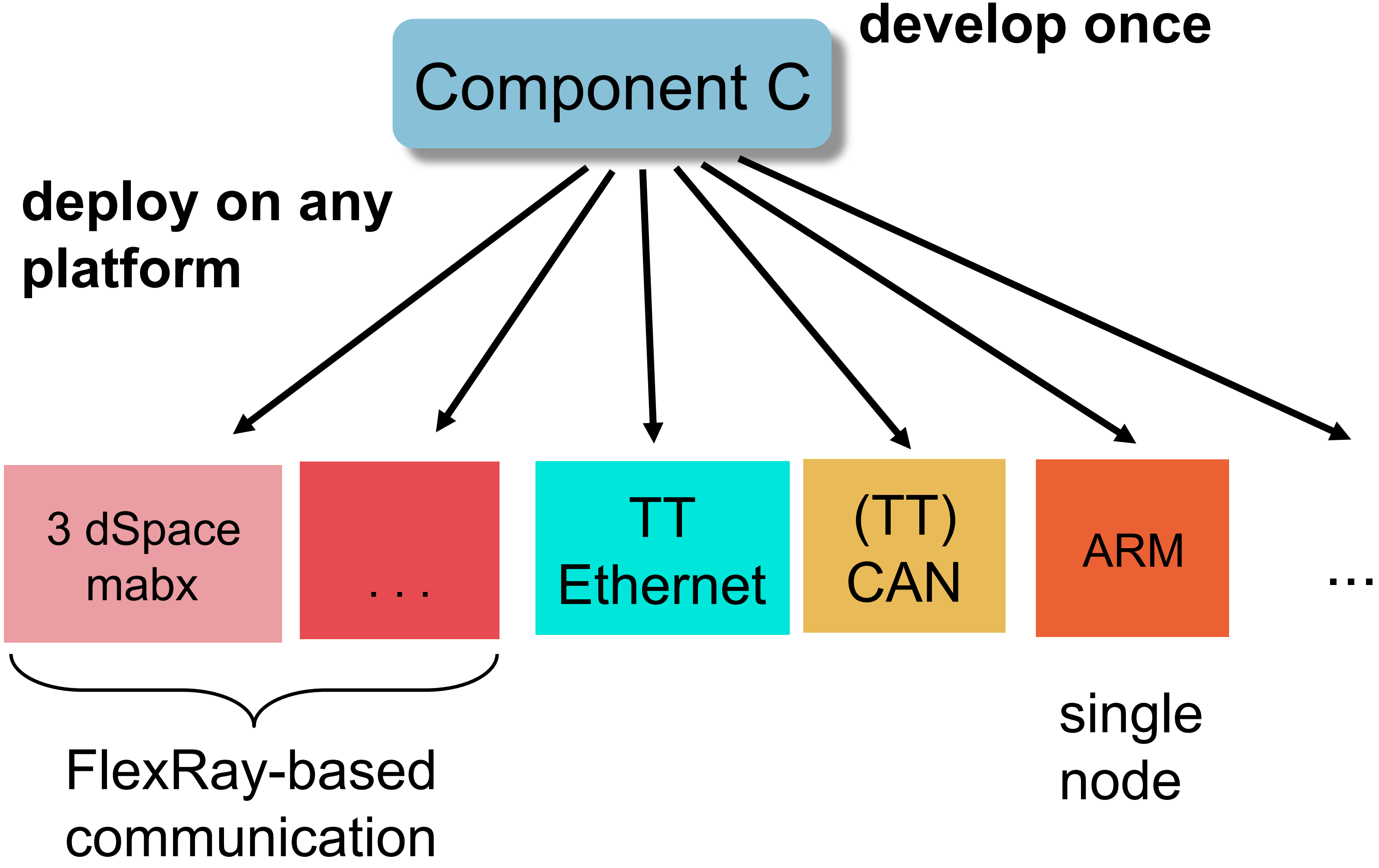
Univ. Salzburg

Overview

- Advantages of platform abstractions
- Timing Definition Language (TDL) in a nut shell
- TDL execution
- TDL extensions
- Transparent distribution of TDL components
- TDL development process
- TDL tools

Advantages of platform abstractions

Vision:



develop once

Component C

deploy on any platform

3 dSpace mabx

. . .

TT Ethernet

(TT) CAN

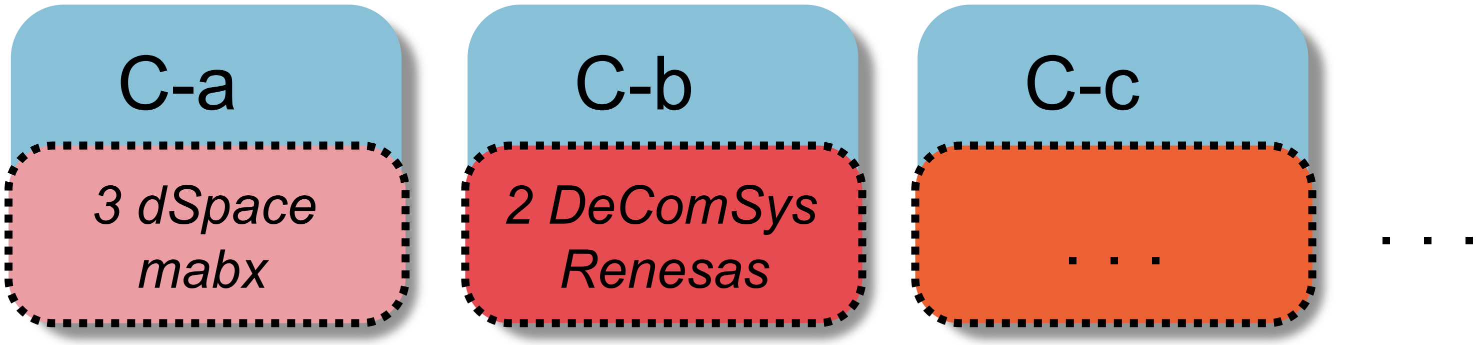
ARM

…

FlexRay-based communication

single node

State-of-the-art:



C-a

3 dSpace mabx

C-b

2 DeComSys Renesas

C-c

. . .

. . .

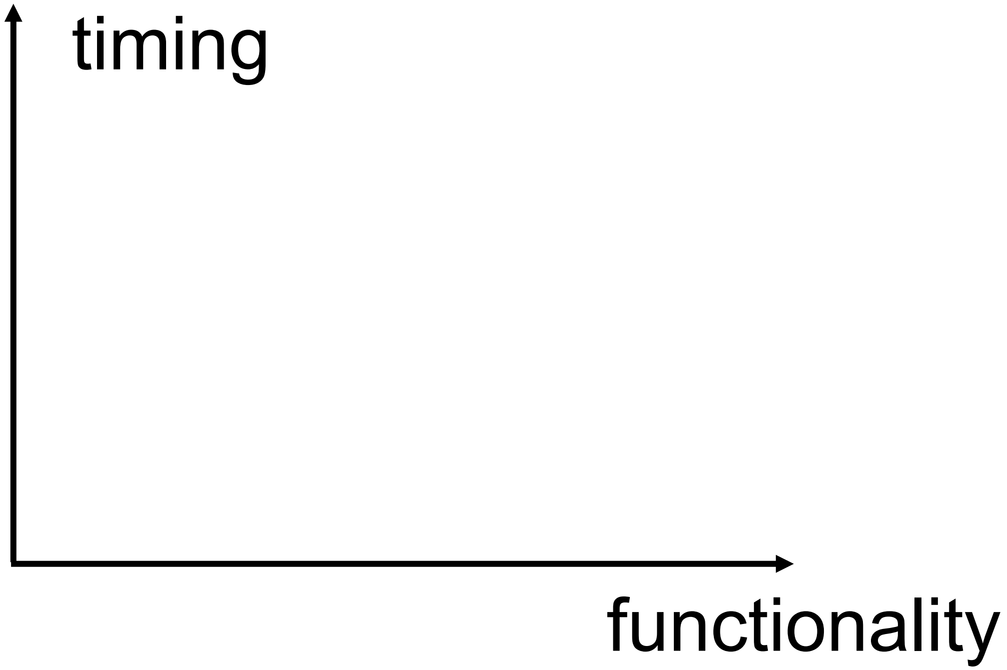
Timing Definition Language (TDL) advantages

- **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

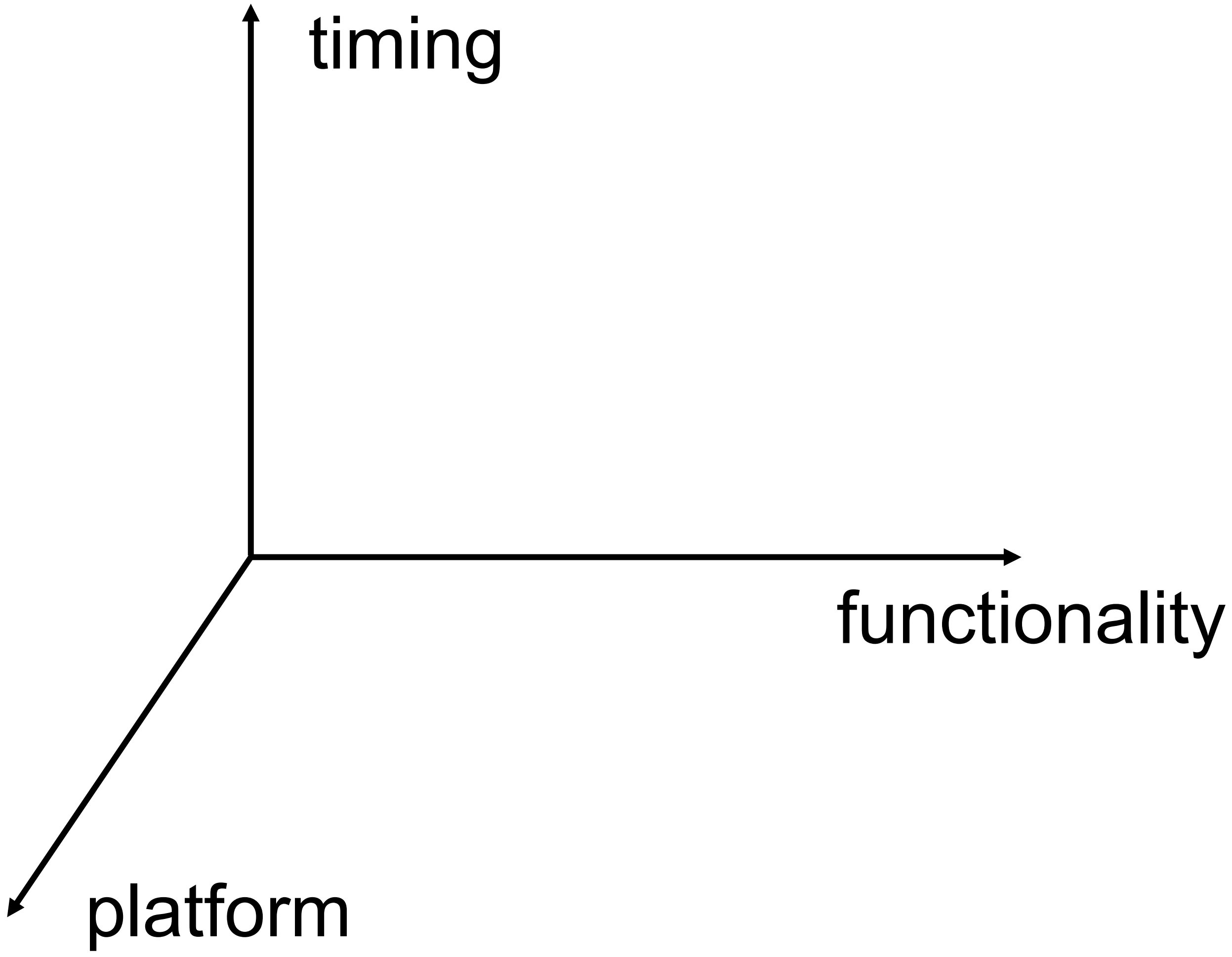
developers have to deal with 3 dimensions

functionality

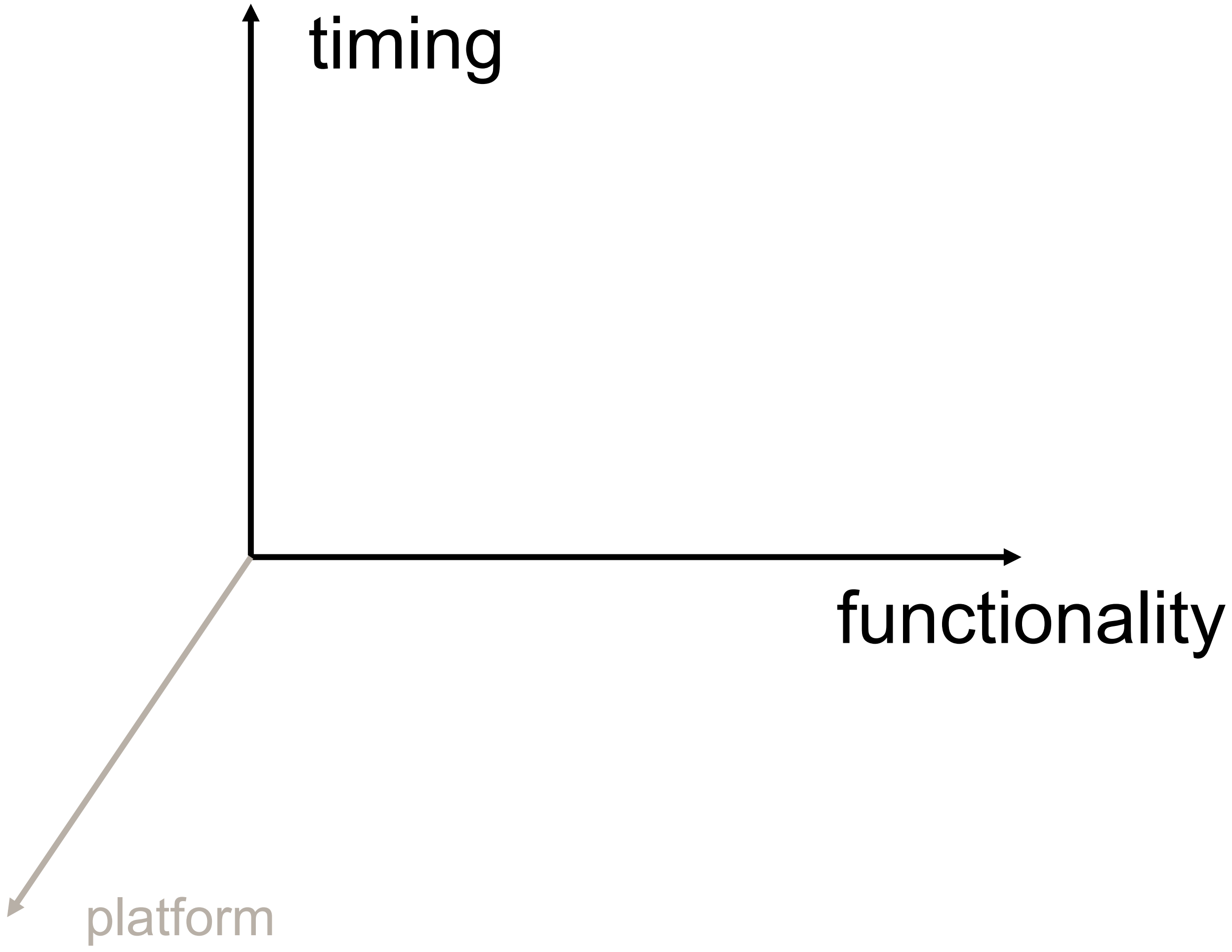
developers have to deal with 3 dimensions



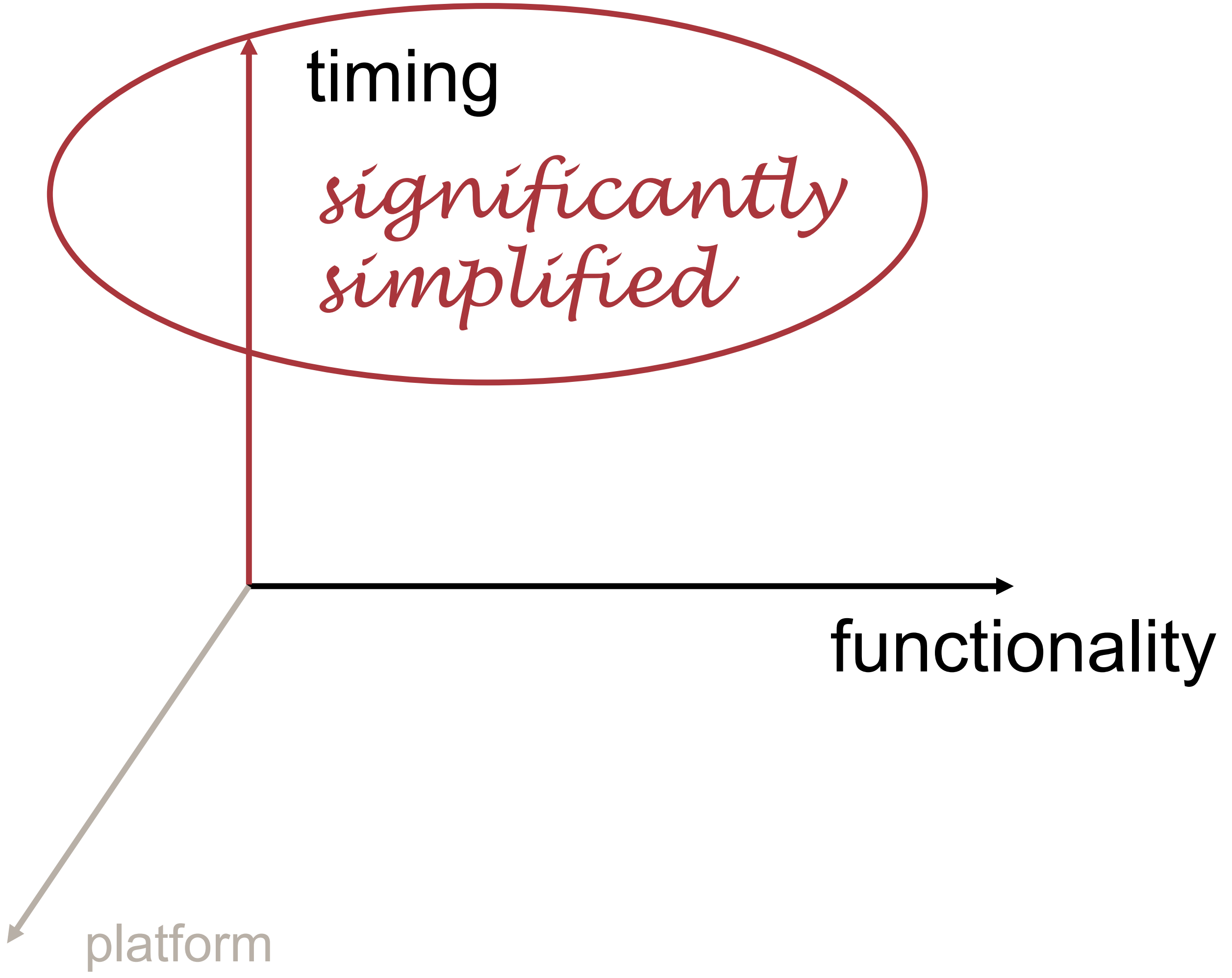
developers have to deal with 3 dimensions



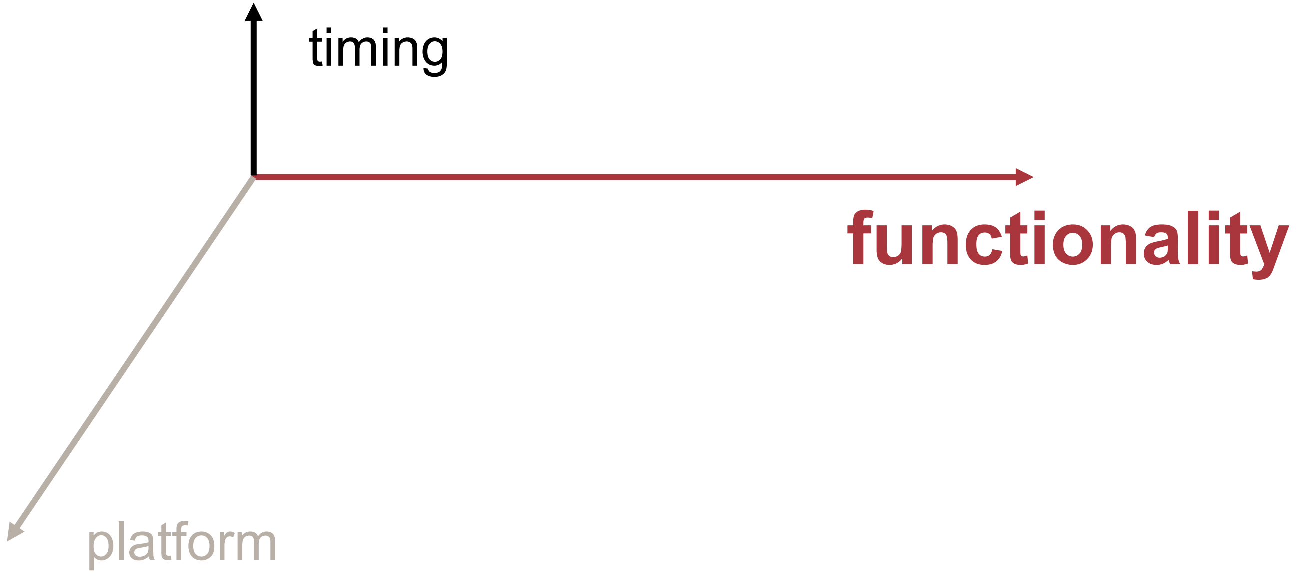
TDL reduces this to 2 dimensions



TDL reduces this to 2 dimensions

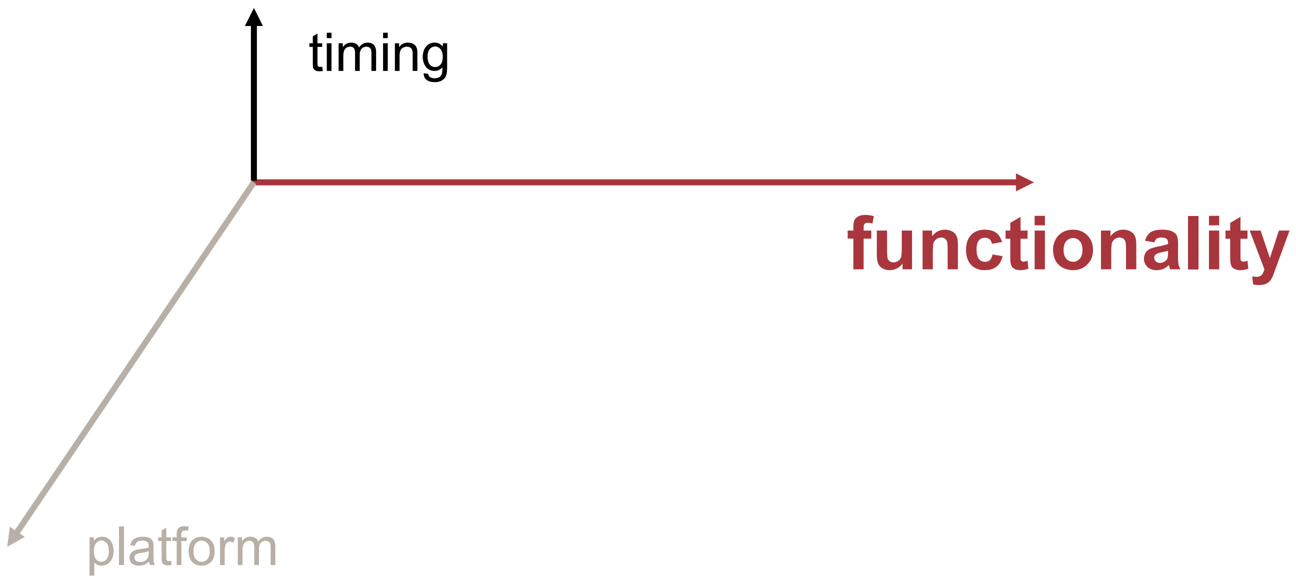


TDL allows your developers to focus on the functionality



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

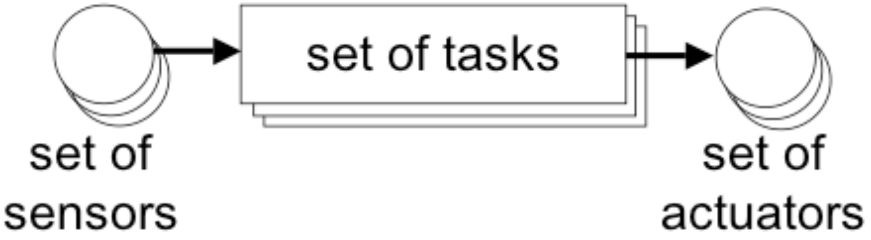
flexibility to change topology, even platform

- automatic code generators take care of the details

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

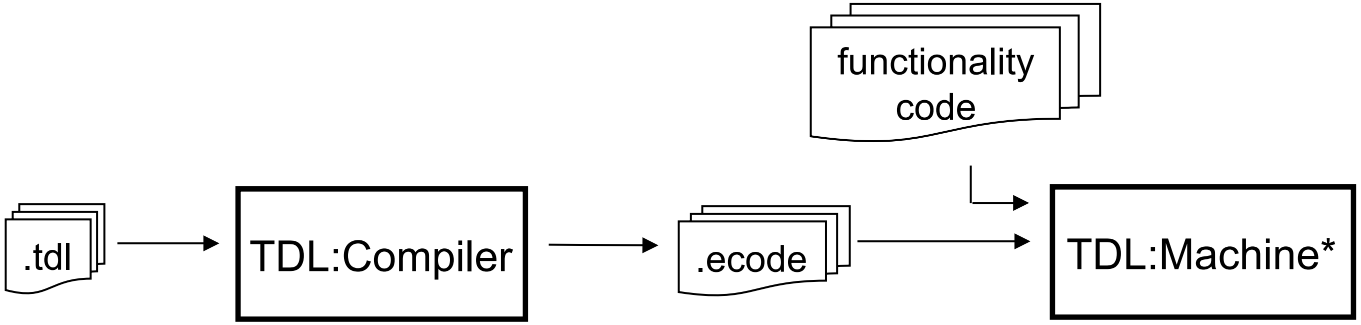
TDL tools

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

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

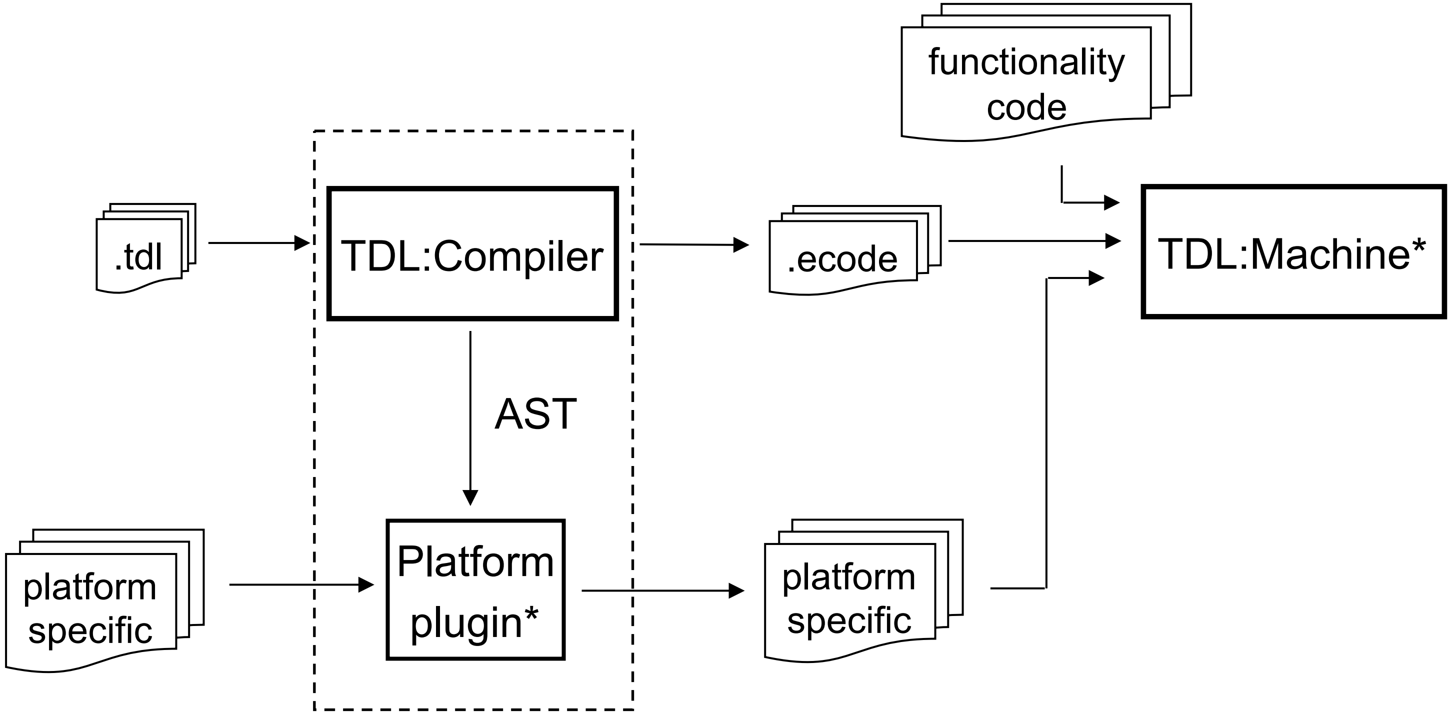
- TDL:Machine (alias E-Machine)
  - platform-specific, typically in C

TDL tool chain



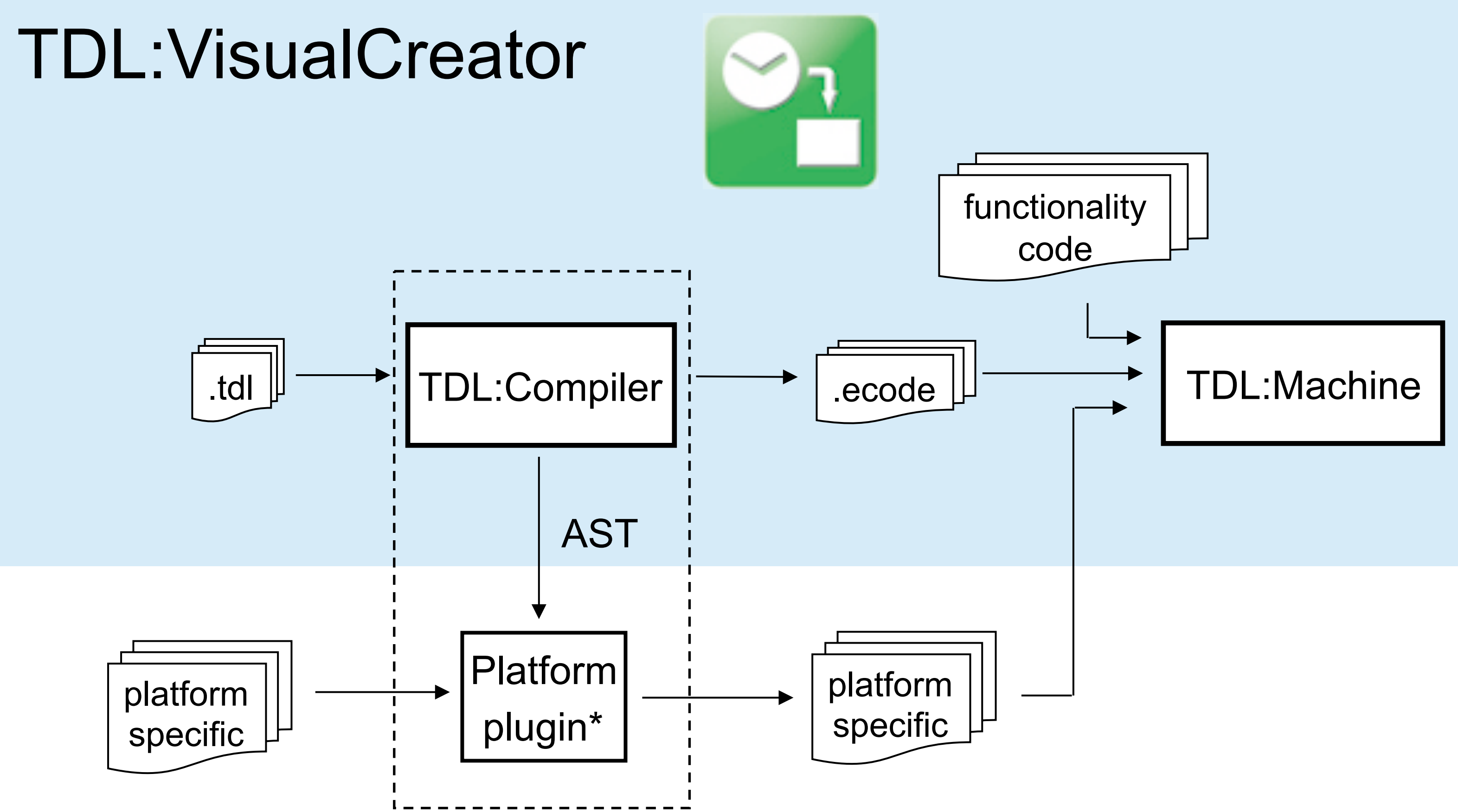
* Simulink, OSEK, dSpace, ARM, AES, INtime, RTLinux, ...

TDL tool chain

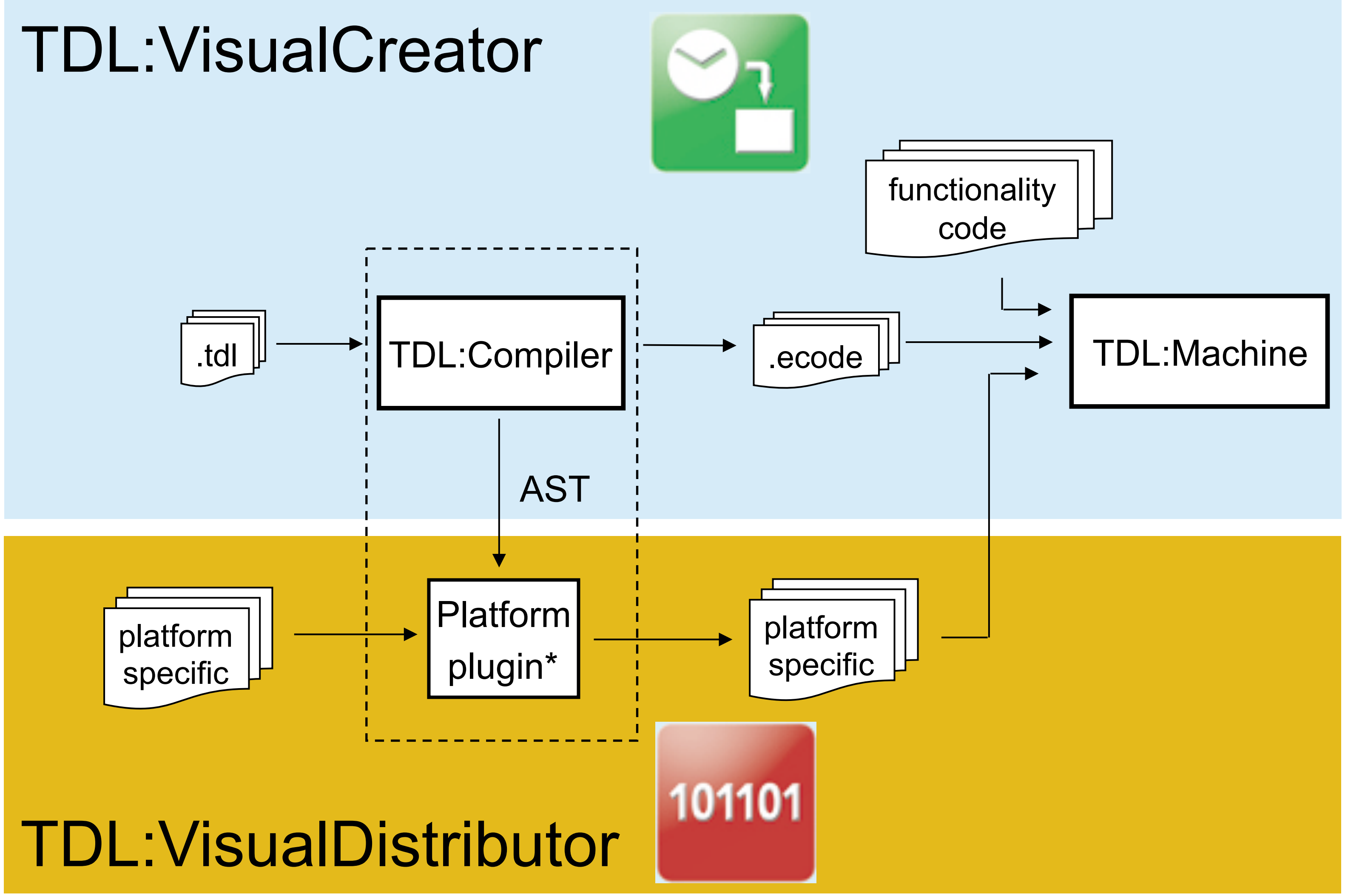


* Simulink, OSEK, dSpace, ARM, AES, INtime, RTLinux, ...

TDL tool chain



TDL tool chain



TDL:VisualCreator

functionality code

.tdl

TDL:Compiler

.ecode

TDL:Machine

AST

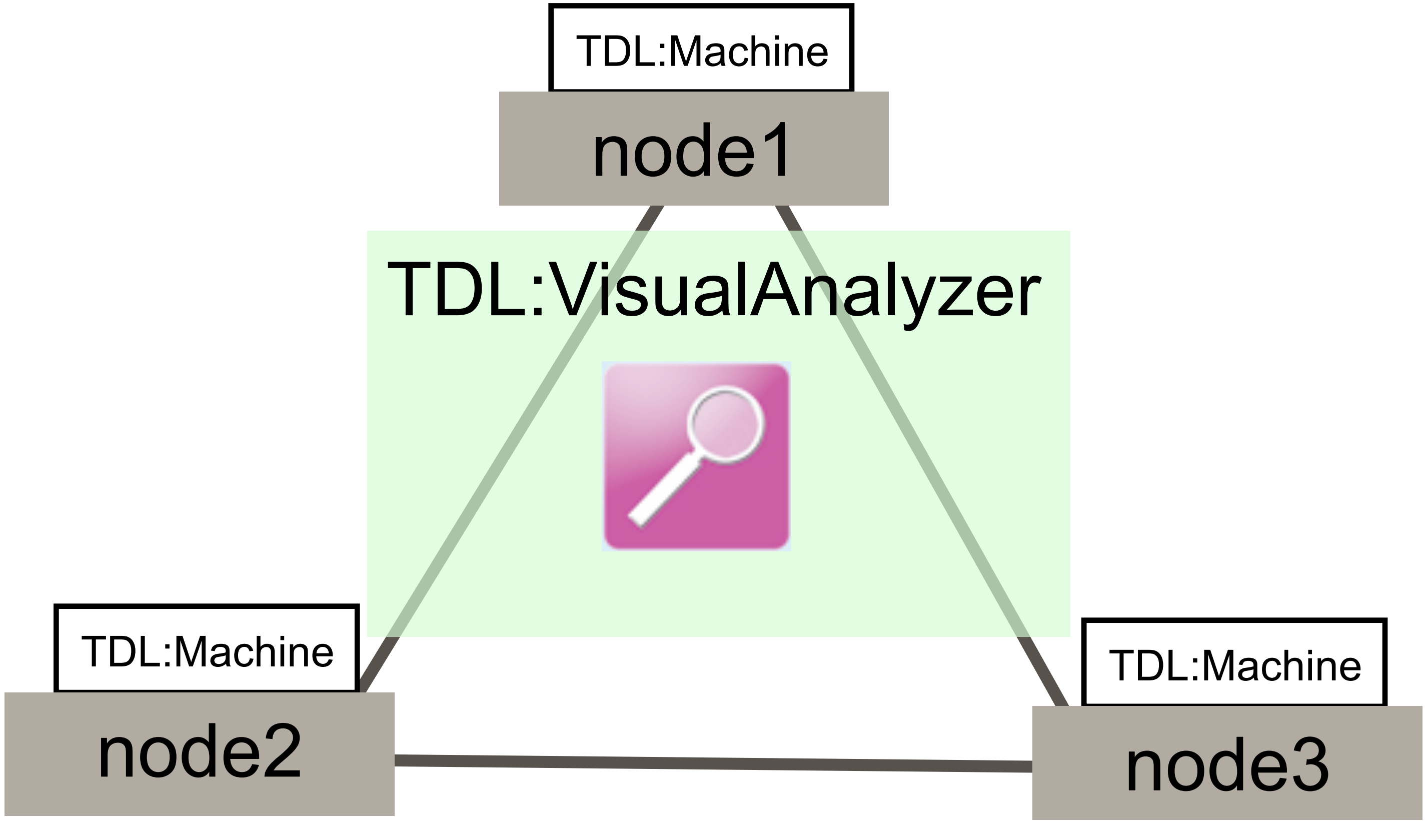
platform specific

Platform plugin*

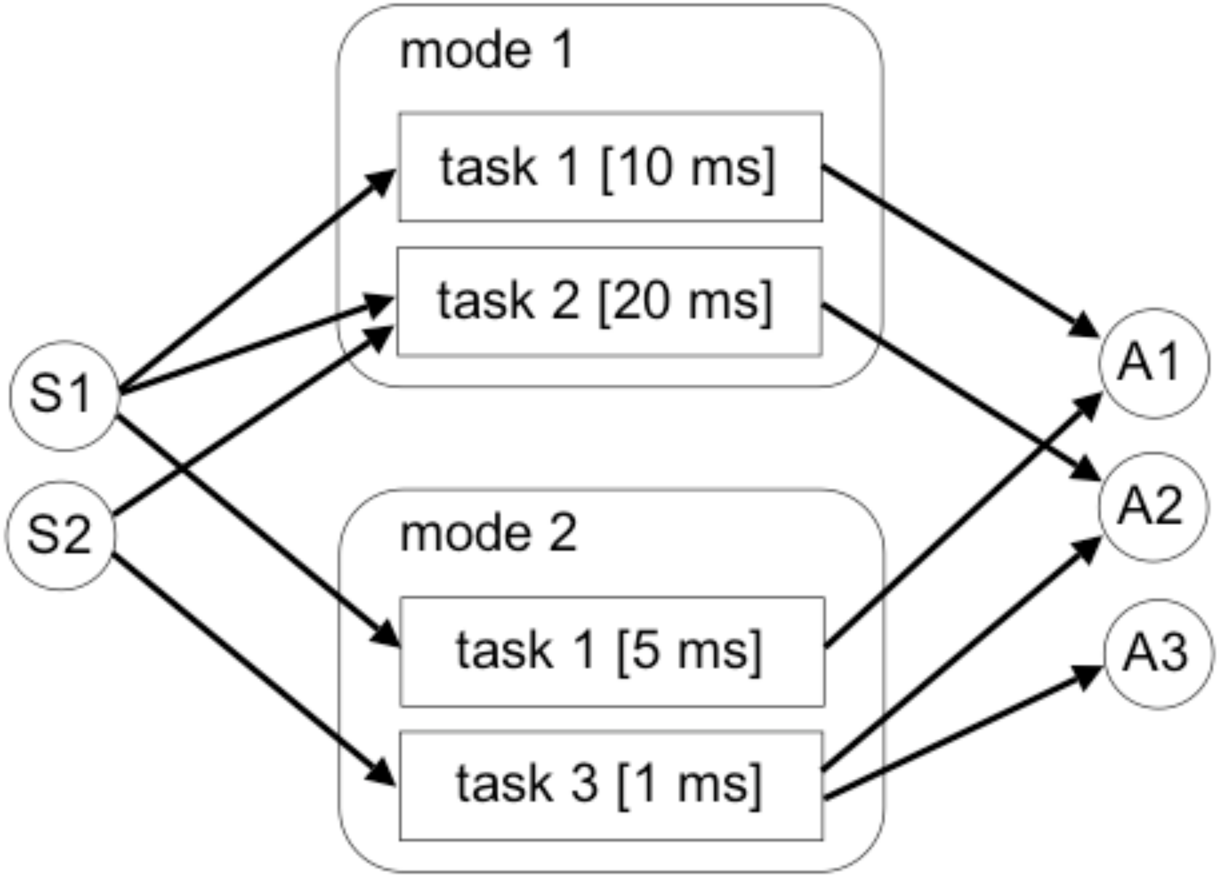
platform specific

TDL:VisualDistributor

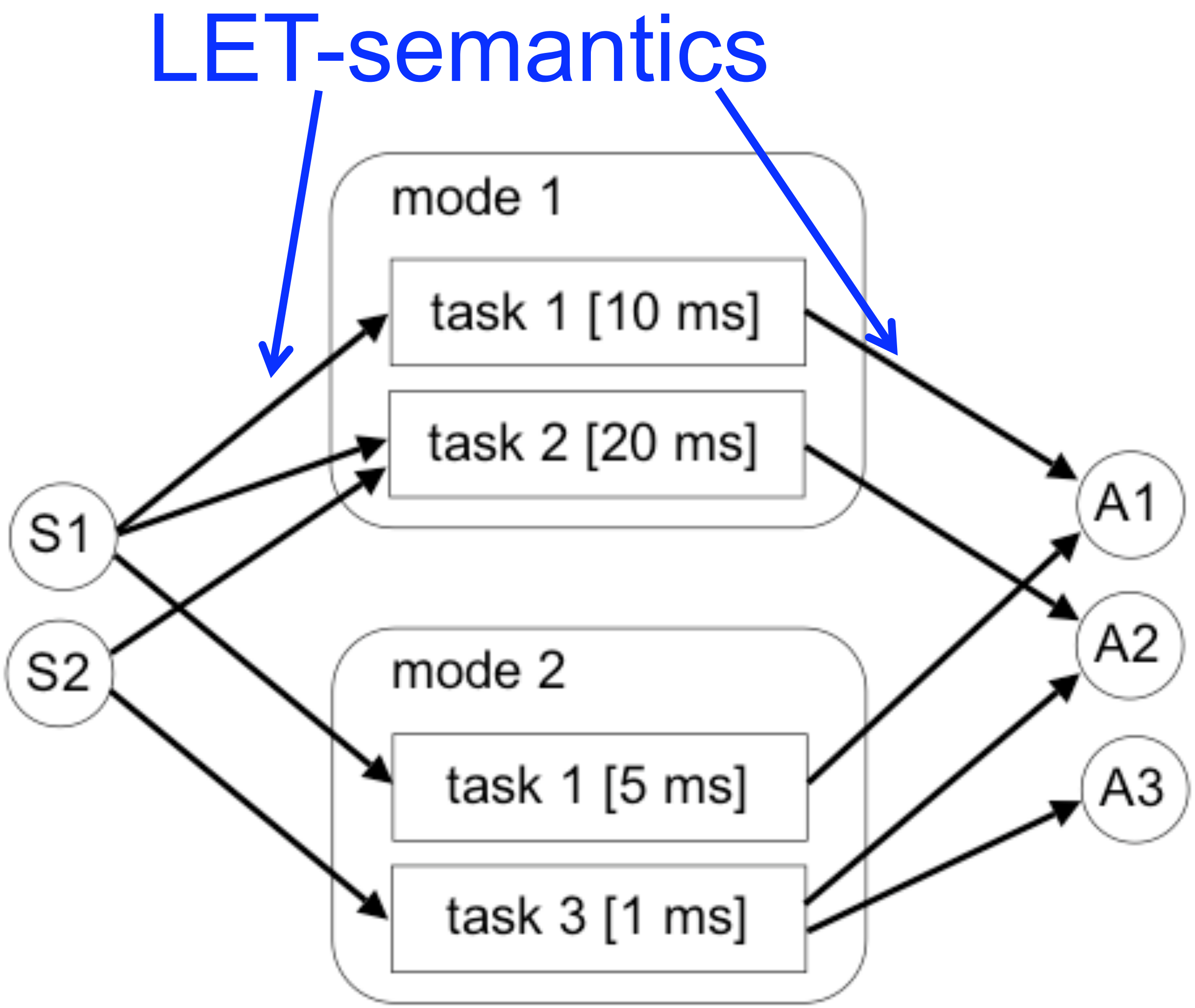
TDL tool chain



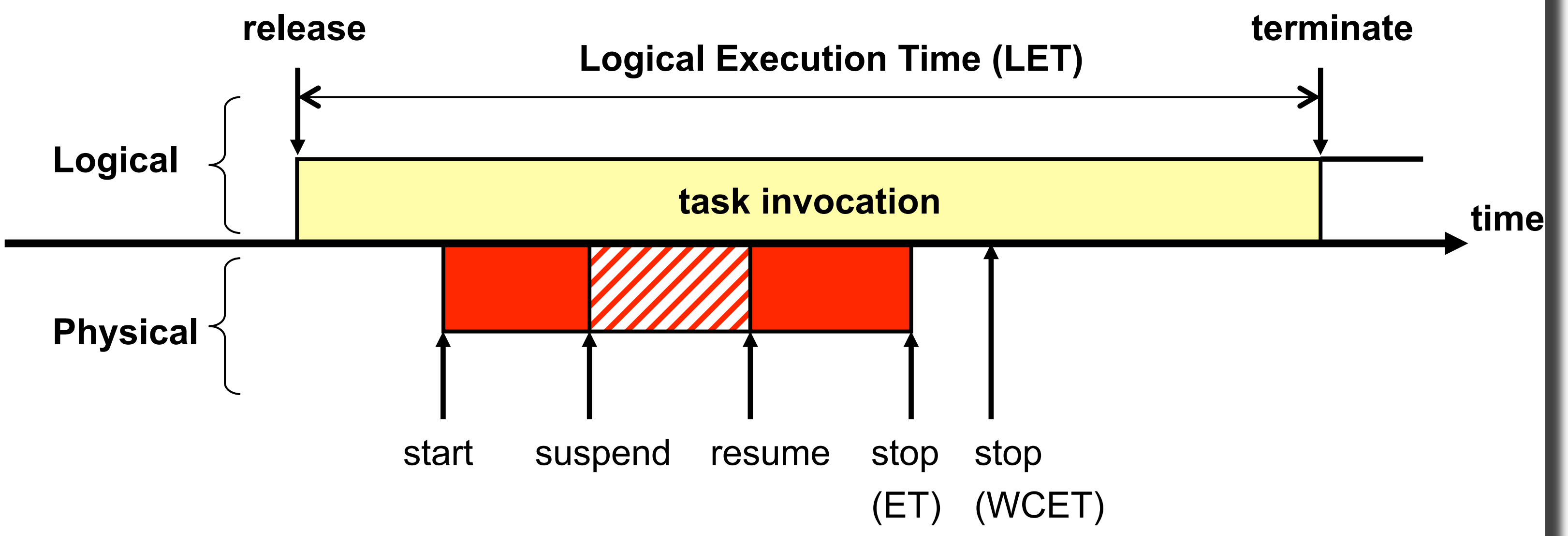
TDL programming model: multi-rate, multi-mode systems (I)



TDL programming model: multi-rate, multi-mode systems (II)



Logical Execution Time (LET) abstraction



ET <= WCET <= LET

results are internally available at 'stop (ET)'

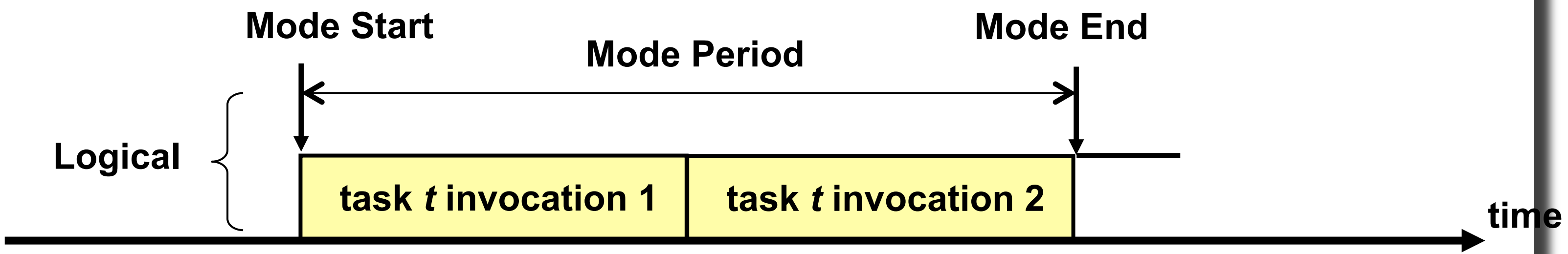
results are externally visible at 'terminate'

spare time between 'stop' and 'terminate'

LET advantages

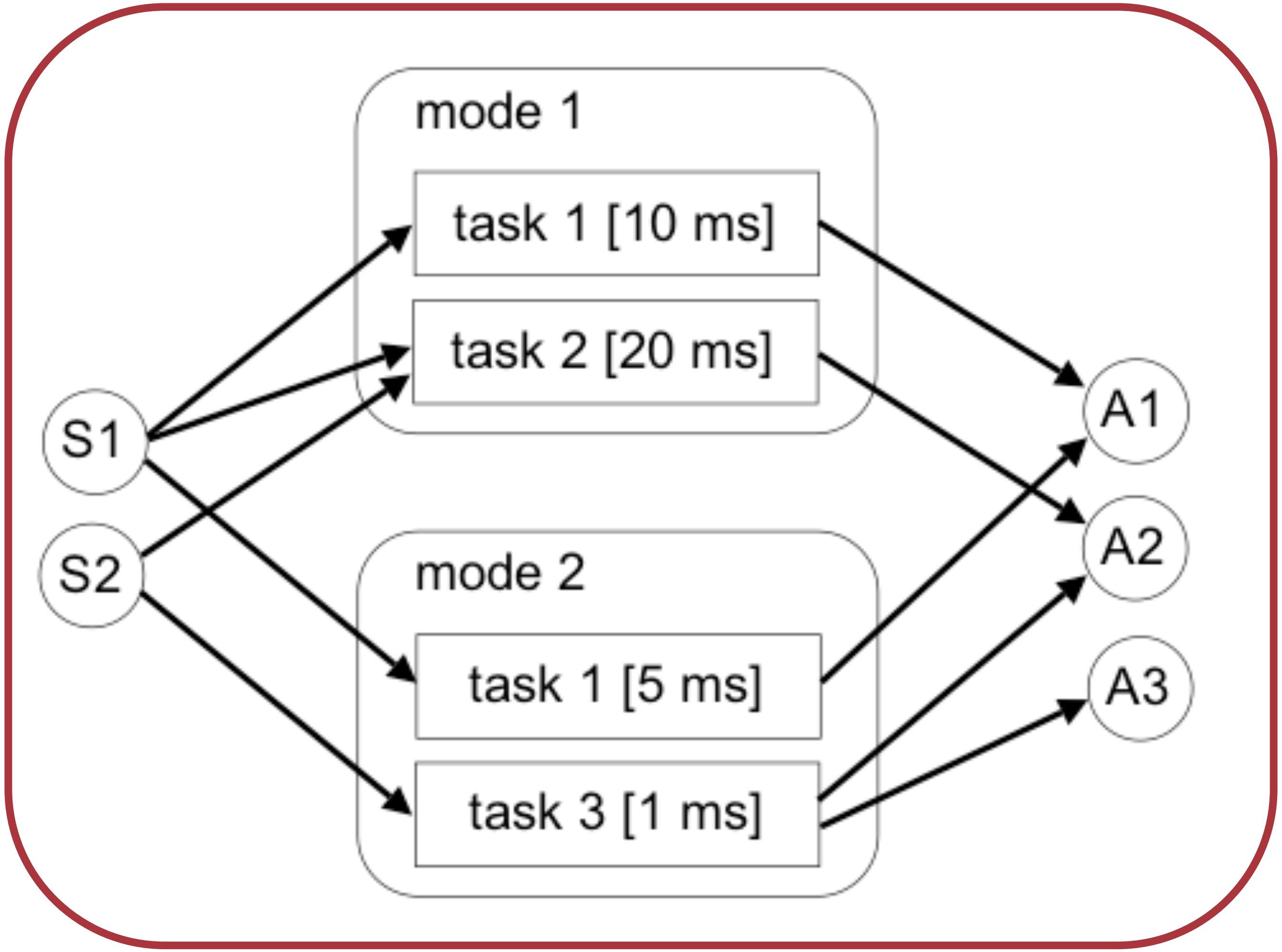
- observable (logical) timing is identical on all platforms
- allows for simulation
- allows for composition
- allows for distribution

Periodic execution in TDL modes

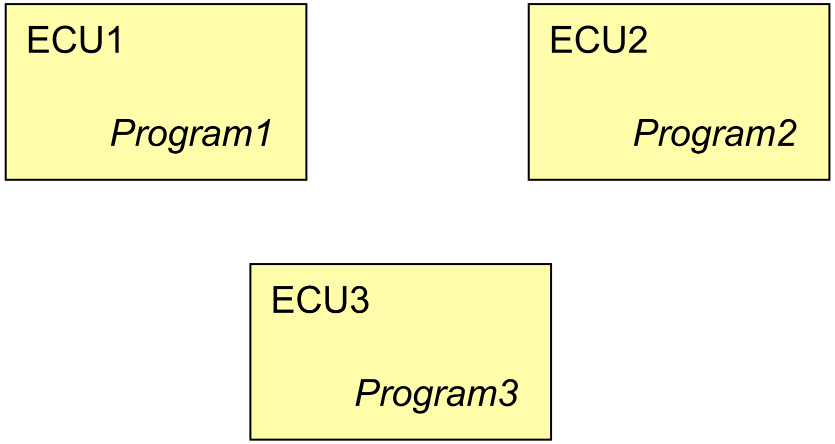


- Every mode has a fixed period.
- A task *t* has a frequency *f* within a mode.
- The mode period is filled with *f* task invocations.
- The LET of a task invocation is *modePeriod / f.*

TDL module: modes, sensors and actuators form a unit

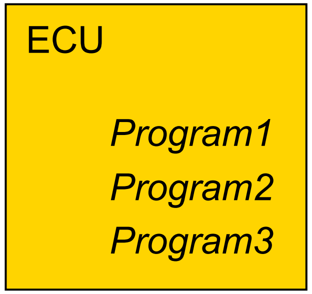


Motivation for TDL modules



- 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



ECU

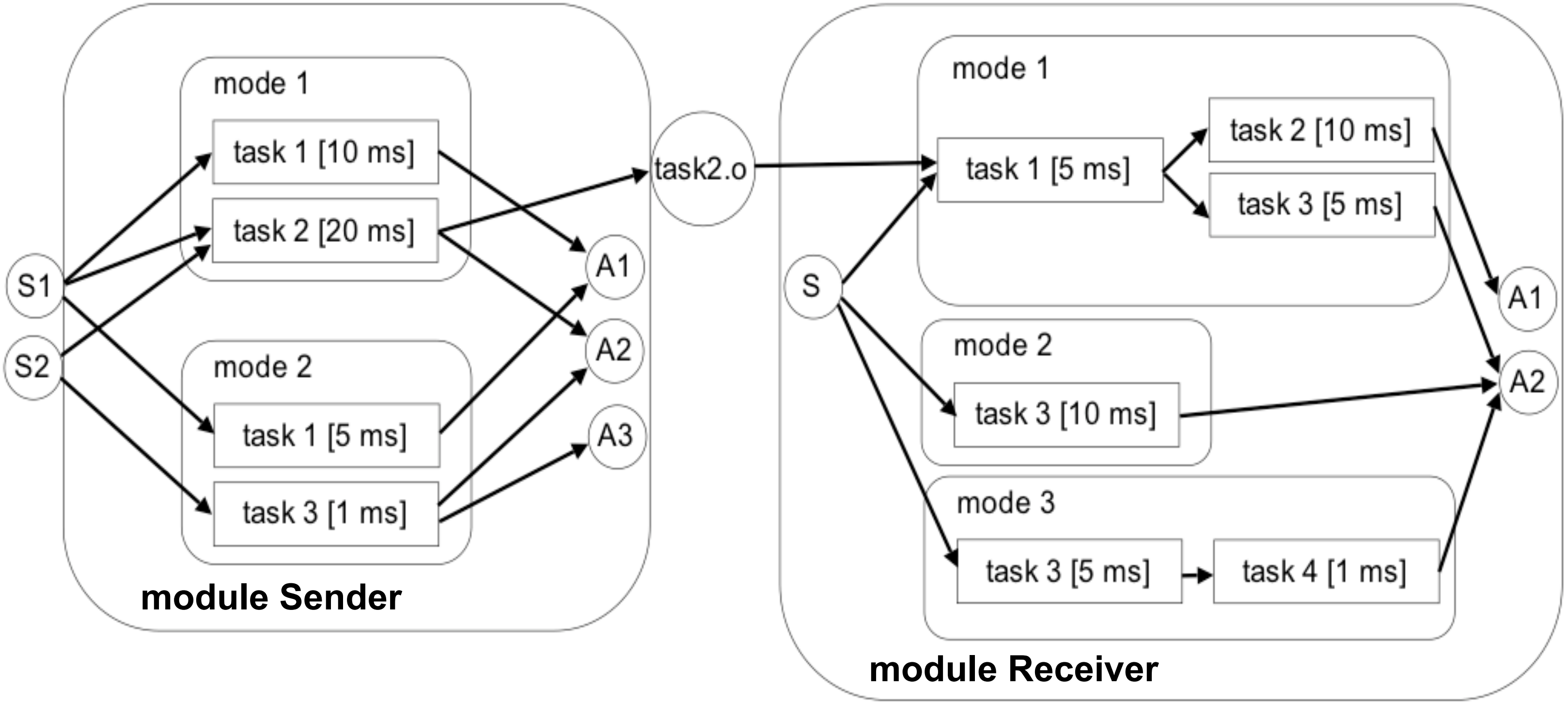
Program1

Program2

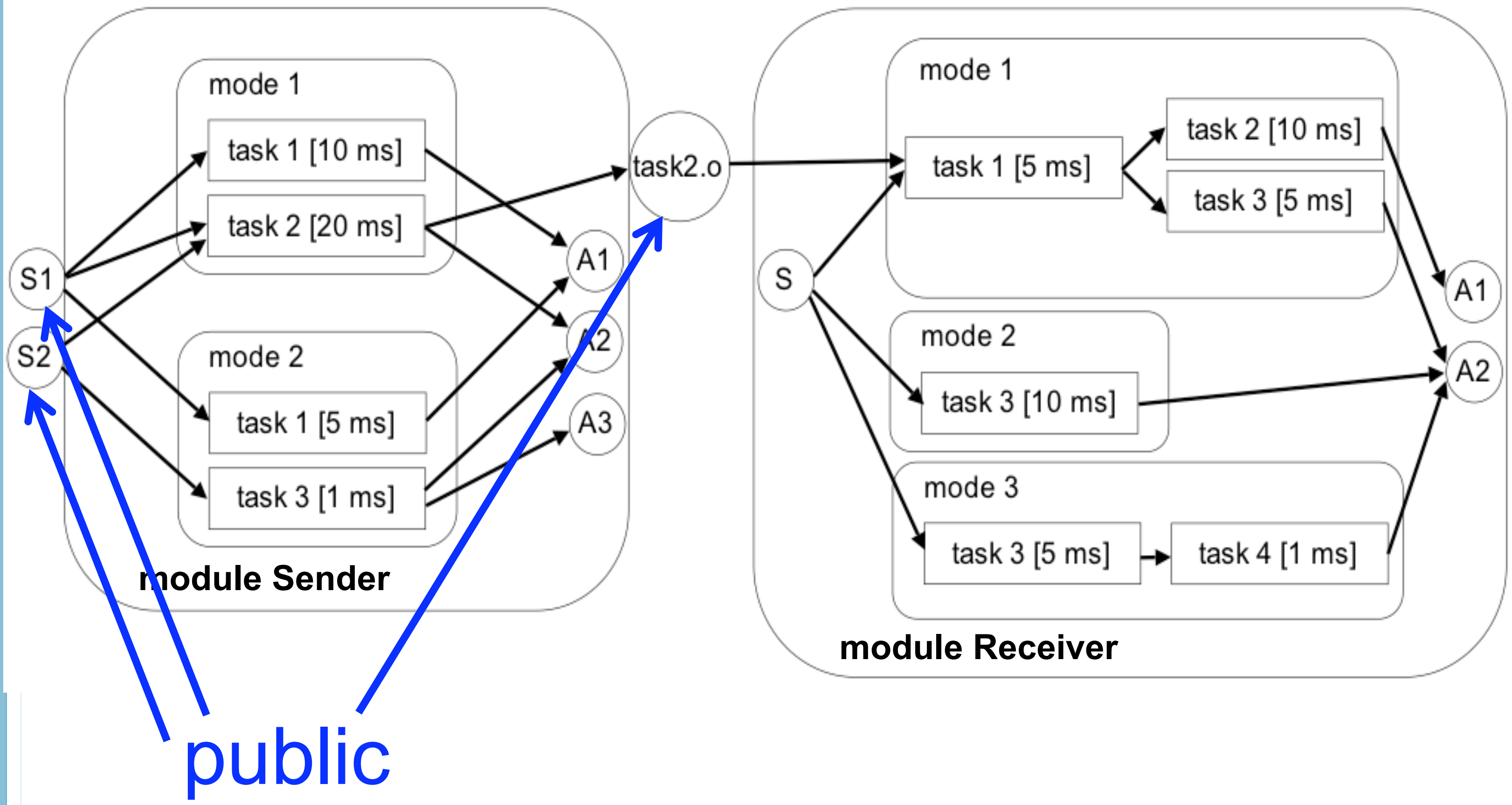
Program3

- ProgramX is called a *module*
- modules may be independent
- modules may also refer to each other
- modules can be used for multiple purposes

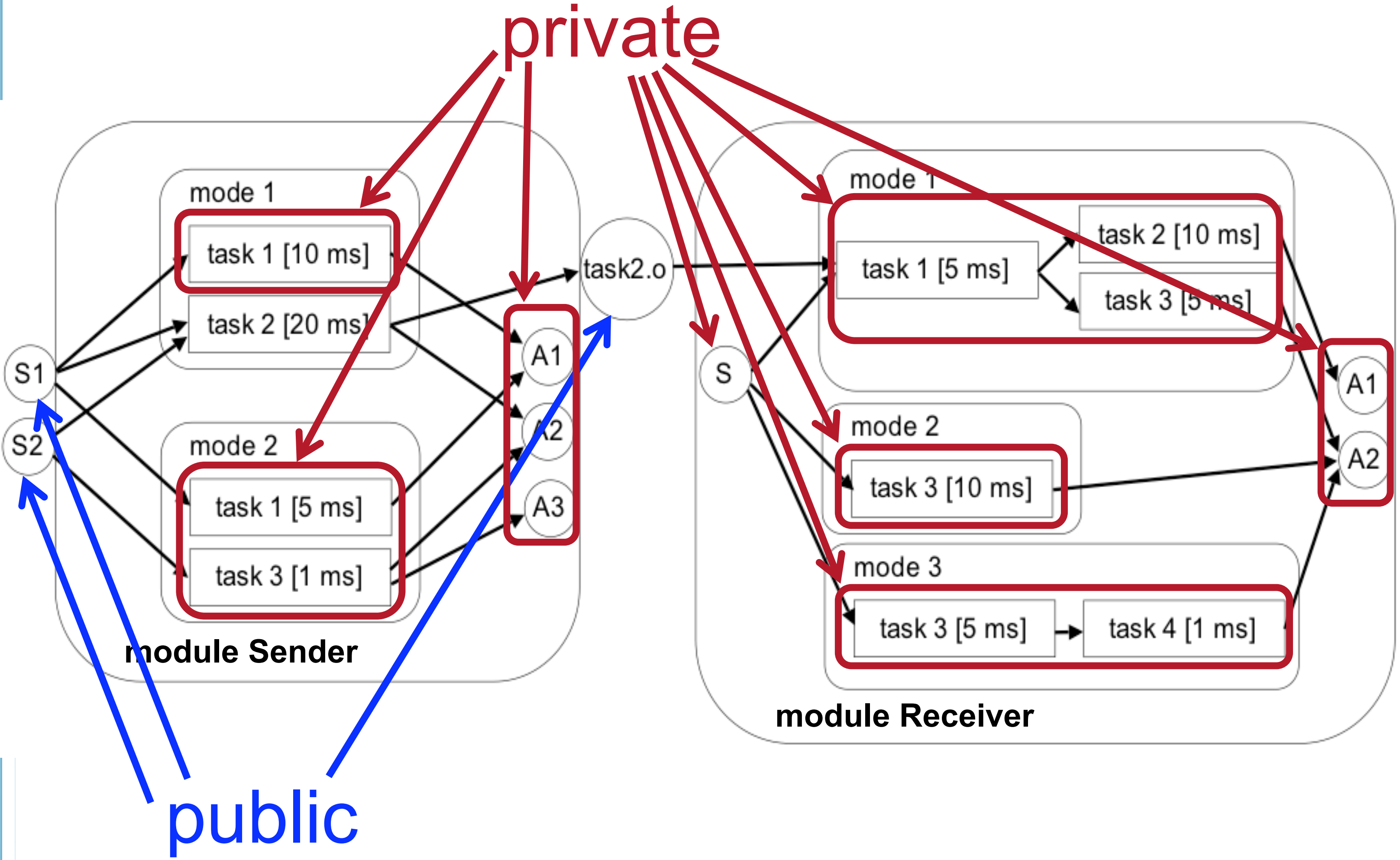
Example: Receiver imports from Sender module



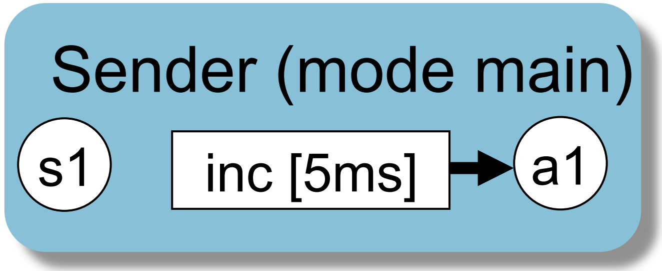
Example: Receiver imports from Sender module



Example: Receiver imports from Sender module



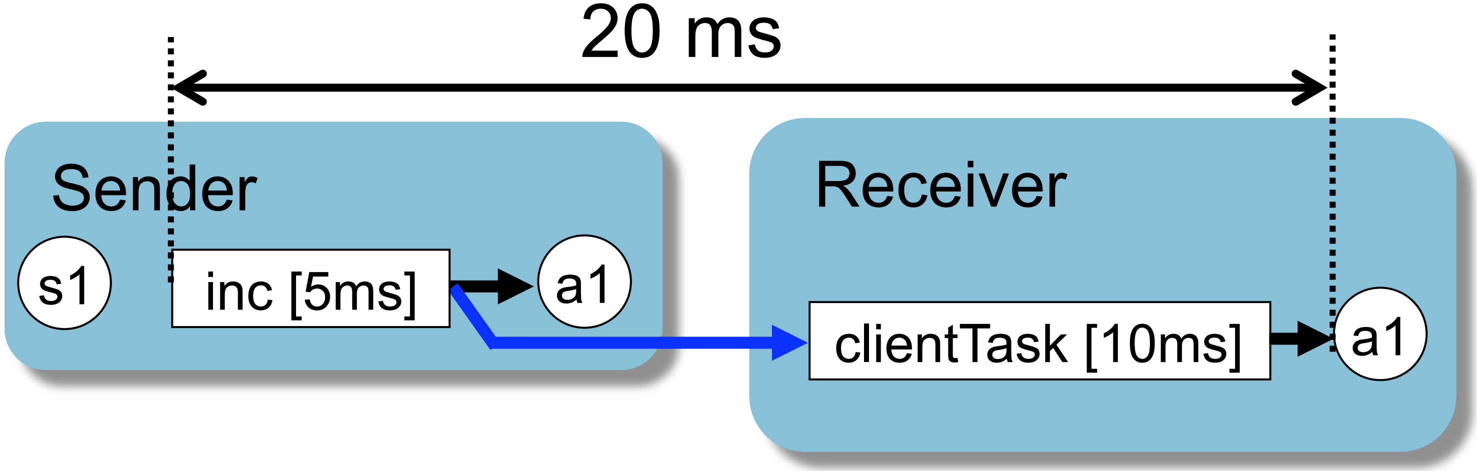
TDL syntax by example



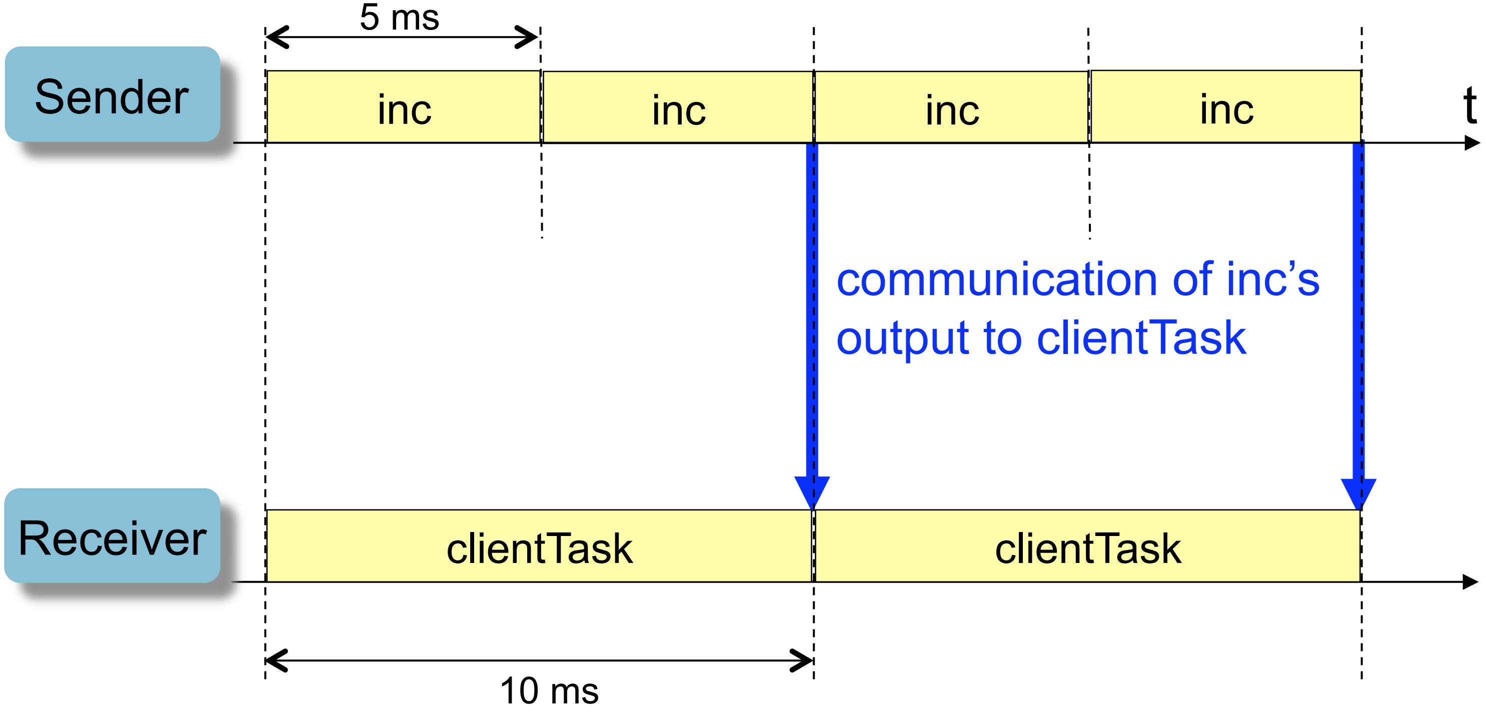
```
module Sender {

  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 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)

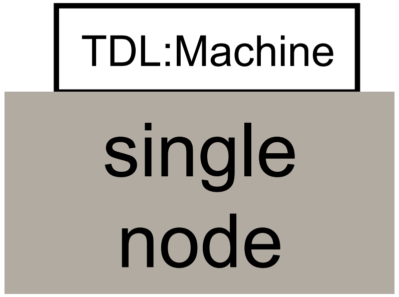


TDL execution

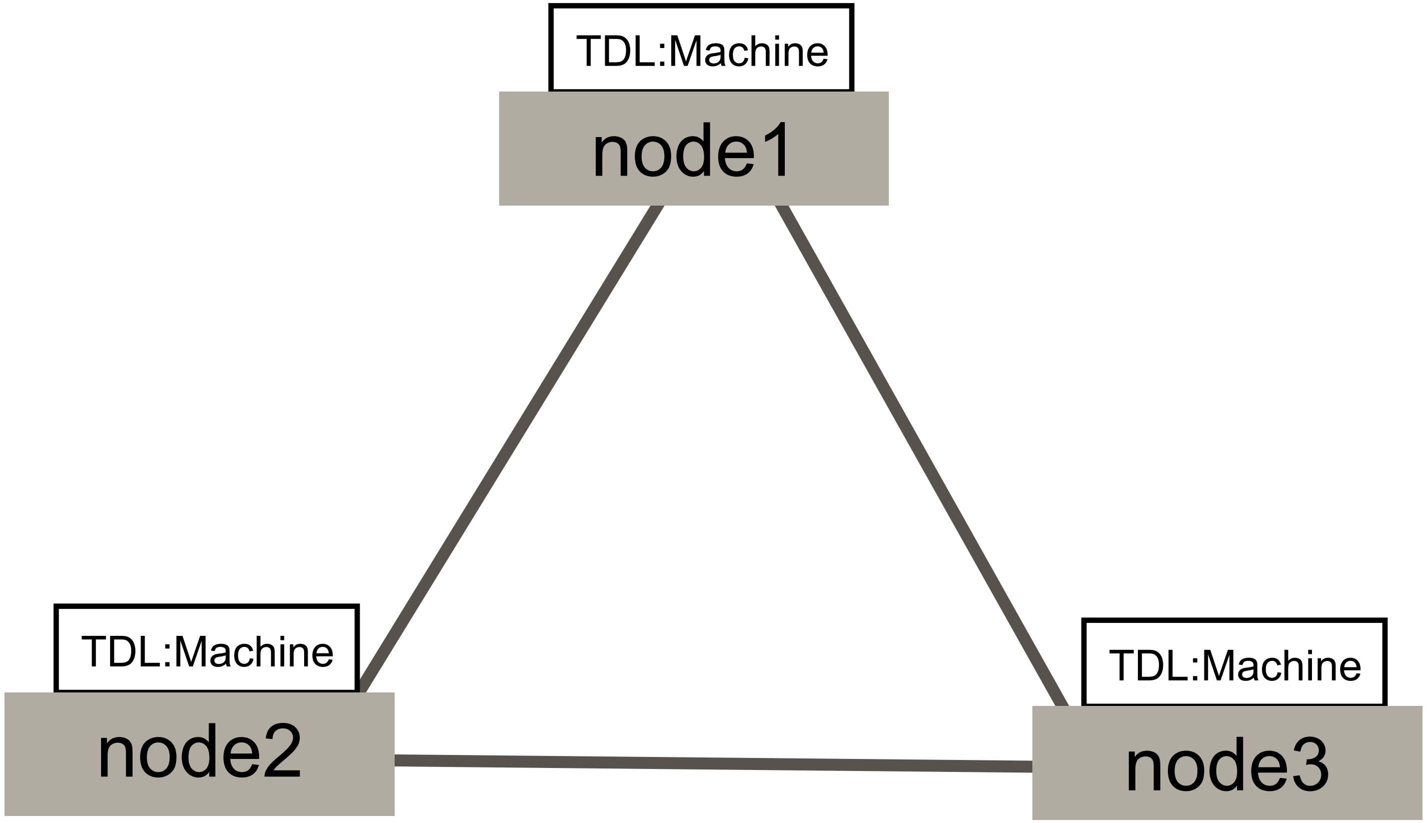
TDL run-time environment

- 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

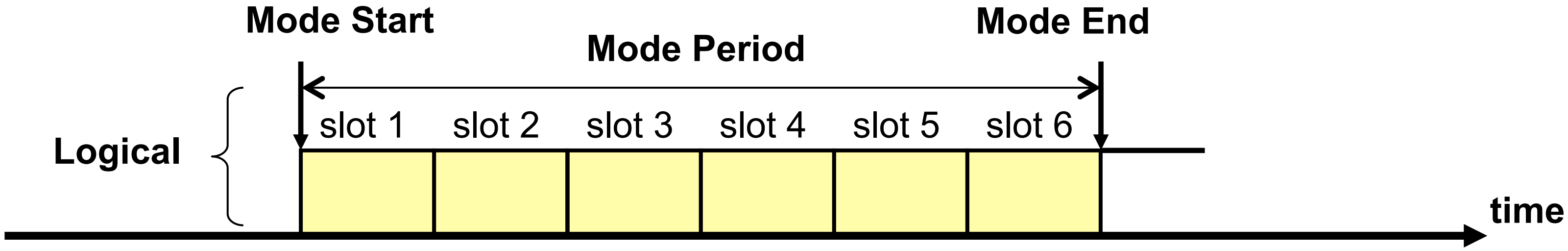


one TDL:Machine per node



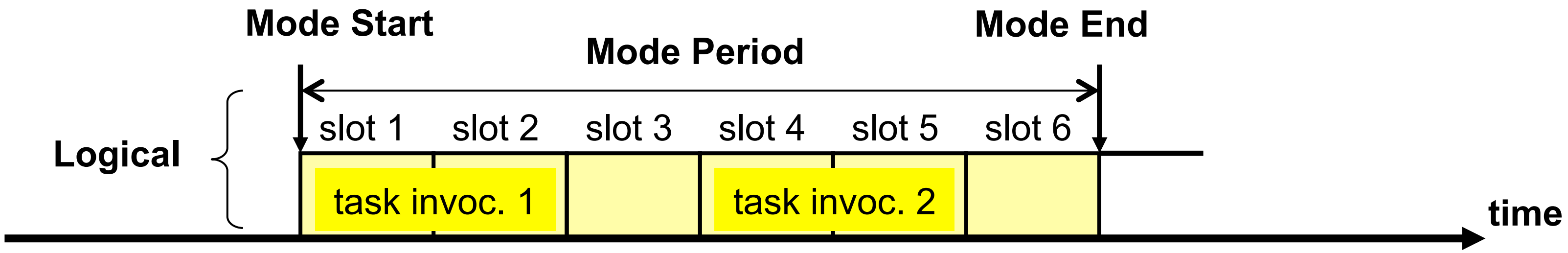
TDL extensions

TDL slot selection



- f = 6

TDL slot selection

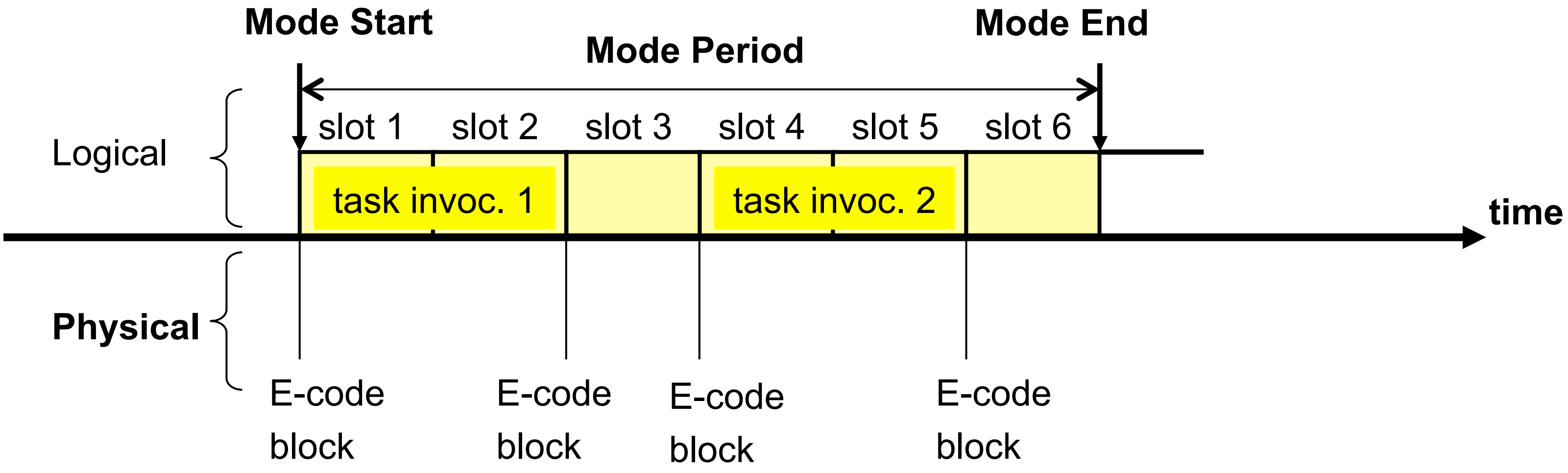


- f = 6
- task invocation 1 covers slots 1 – 2
- task invocation 2 covers slots 4 – 5

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

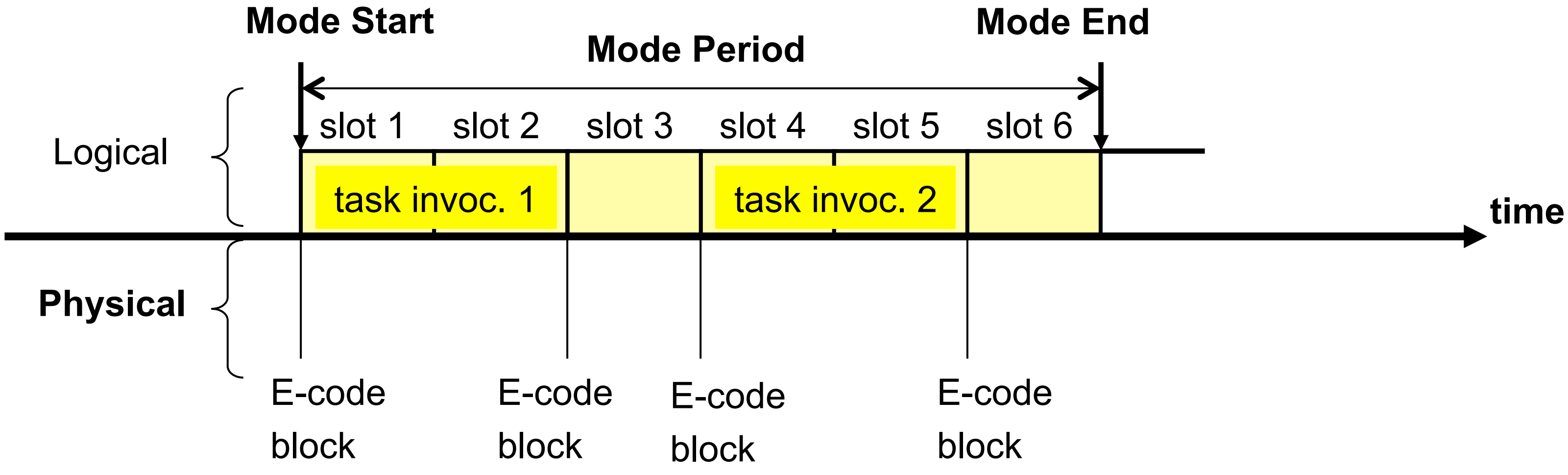
E-code compression

- E-code blocks may be identical
- compression feature would be welcome
- new instruction:

```
REPEAT <targetPC>, <N>
```

- jumps *N* times to *targetPC*, then to *PC* + 1.
- uses a counter per module
- counter is reset upon mode switch

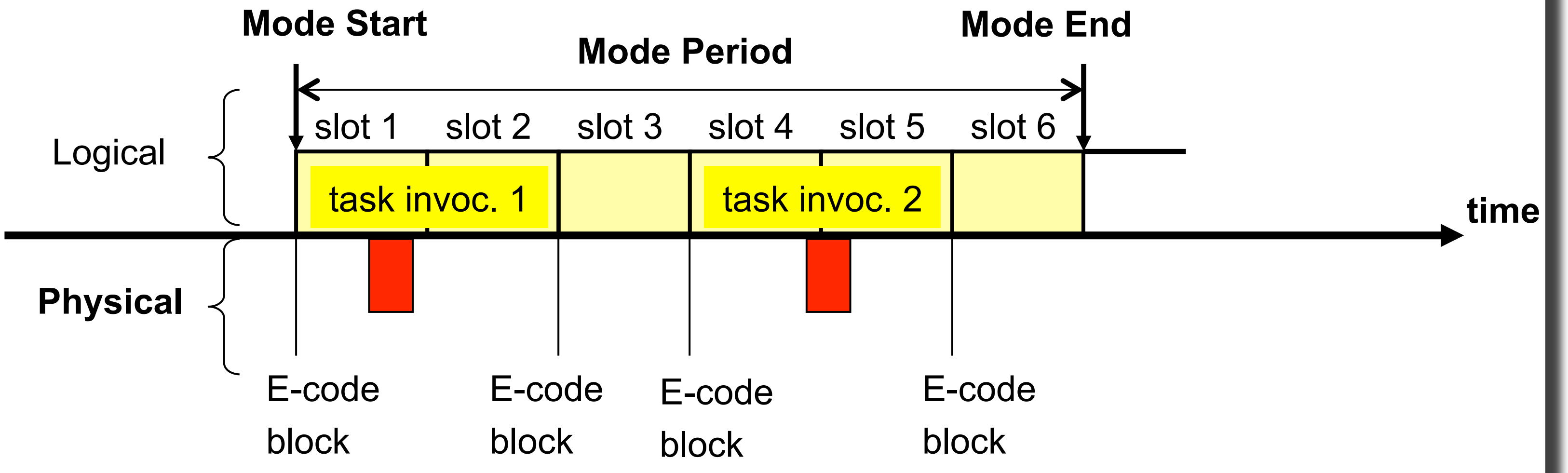
Adding asynchronous activities



Priority levels

- black: highest priority (E-code)

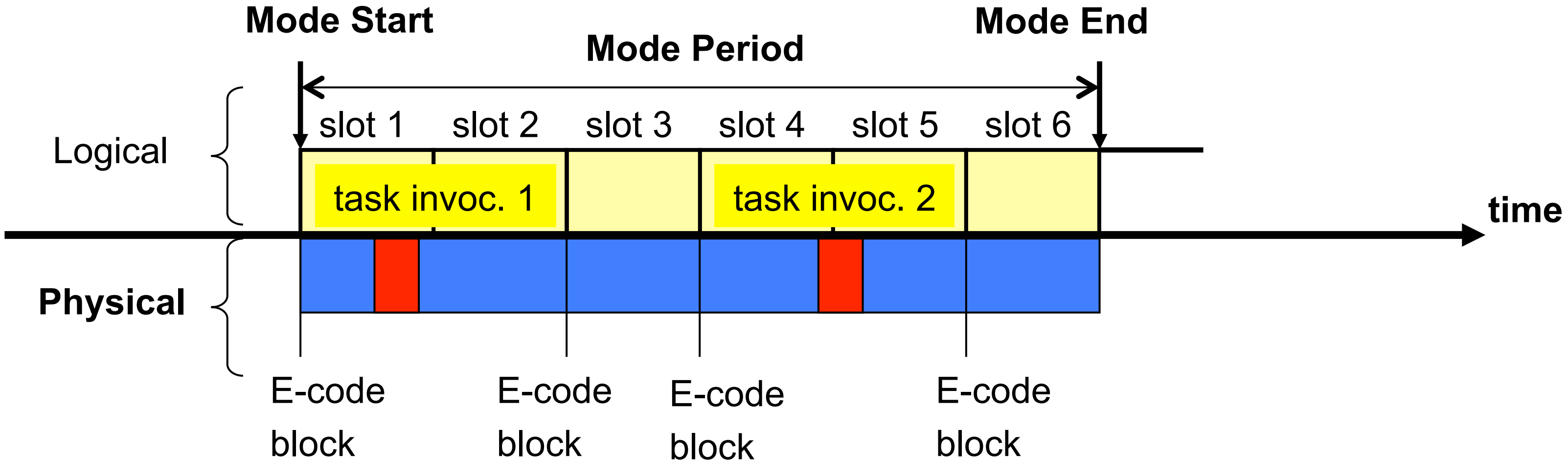
Adding asynchronous activities



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 toTDL
- **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

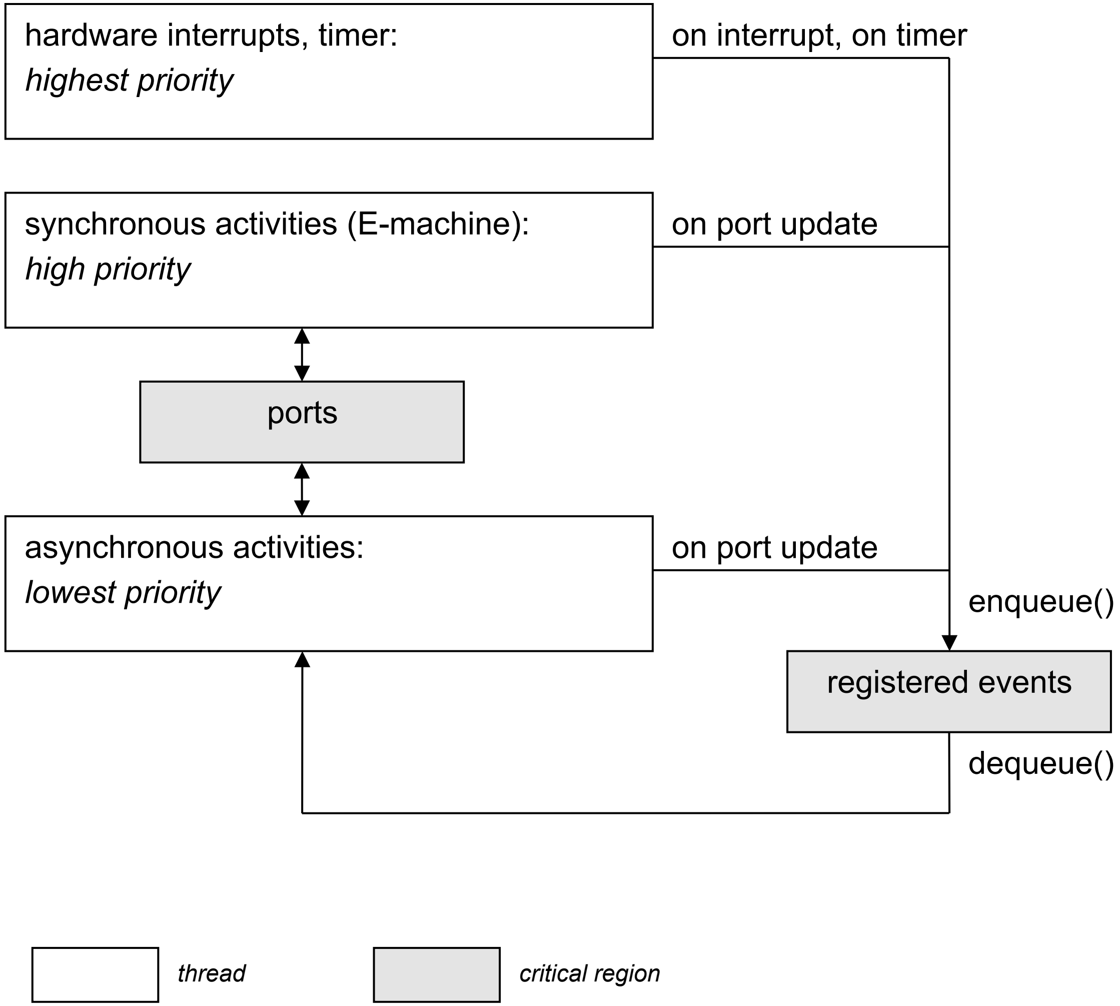
- hardware and software interrupts
- periodic asynchronous timers
- port updates

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

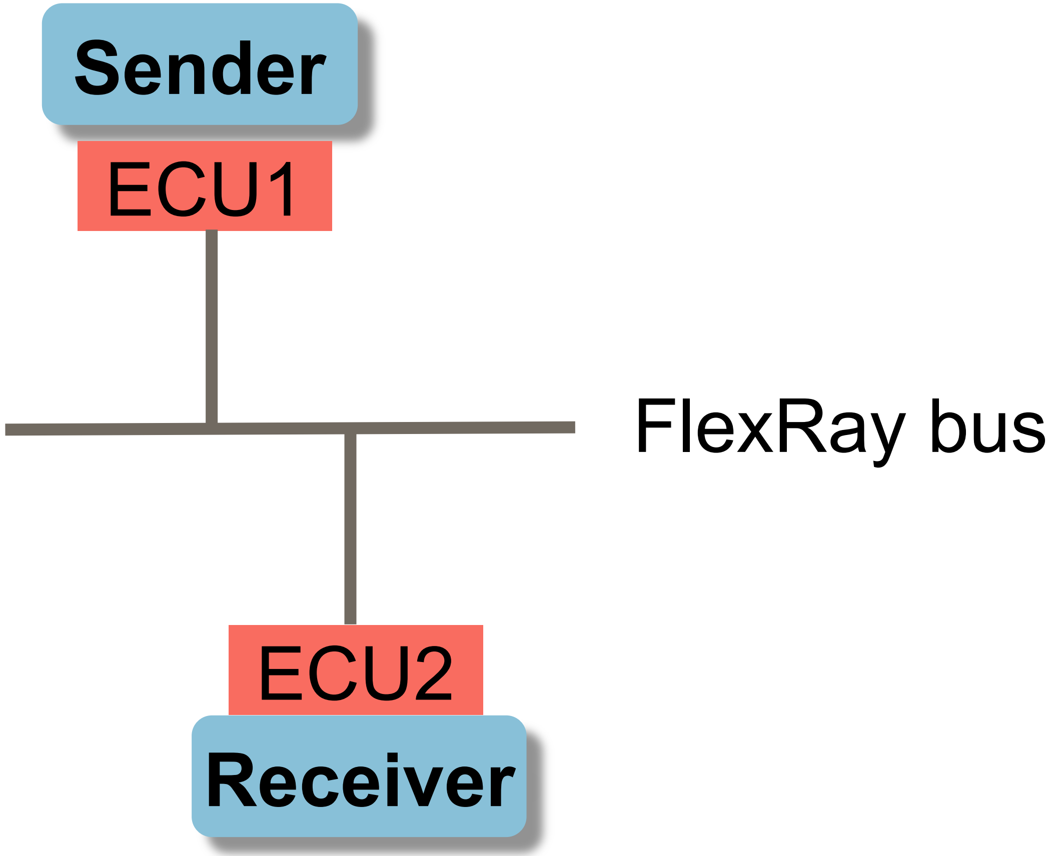


Synchronization requirements

- 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

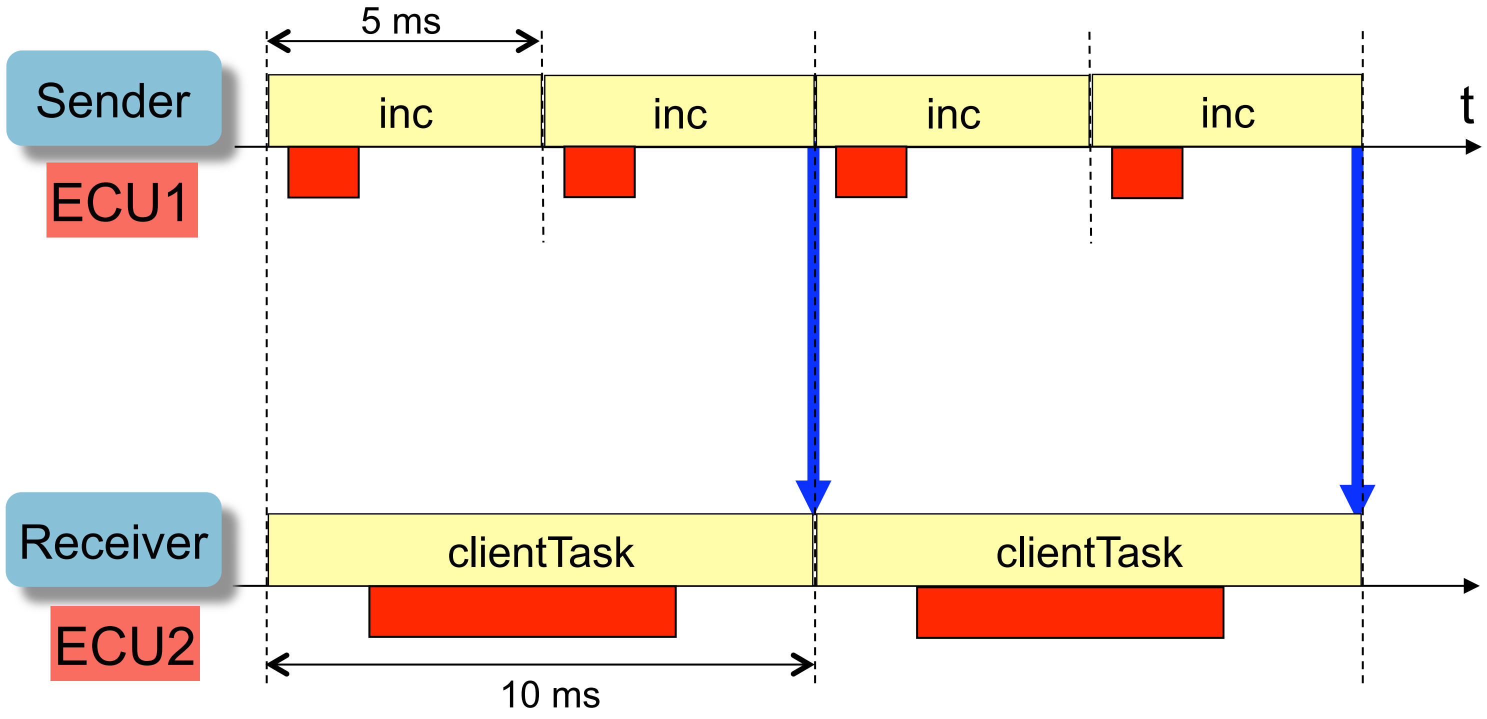
TDL module-to-node-assignment (example)



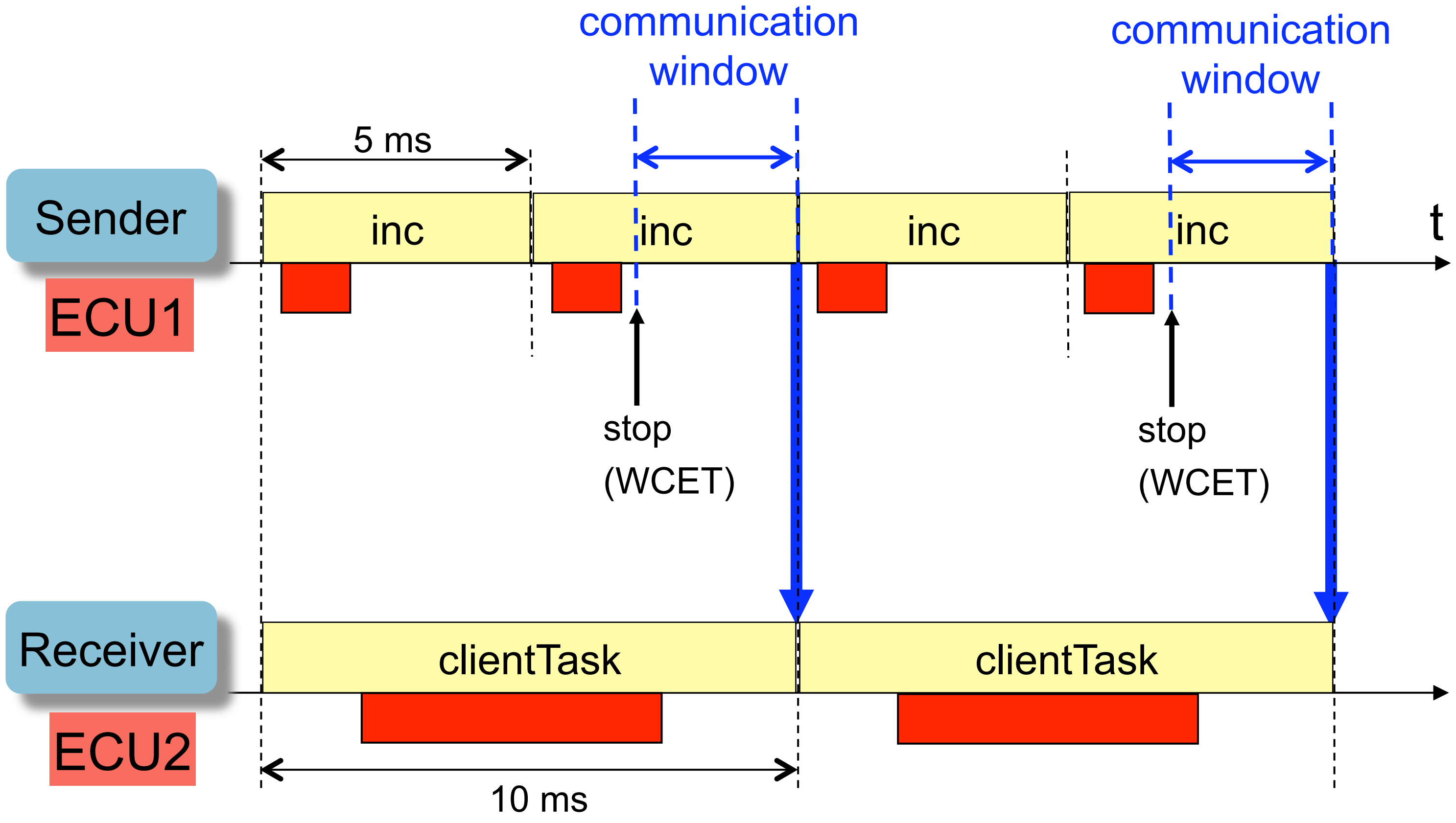
Transparent distribution of TDL components:

- 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**.

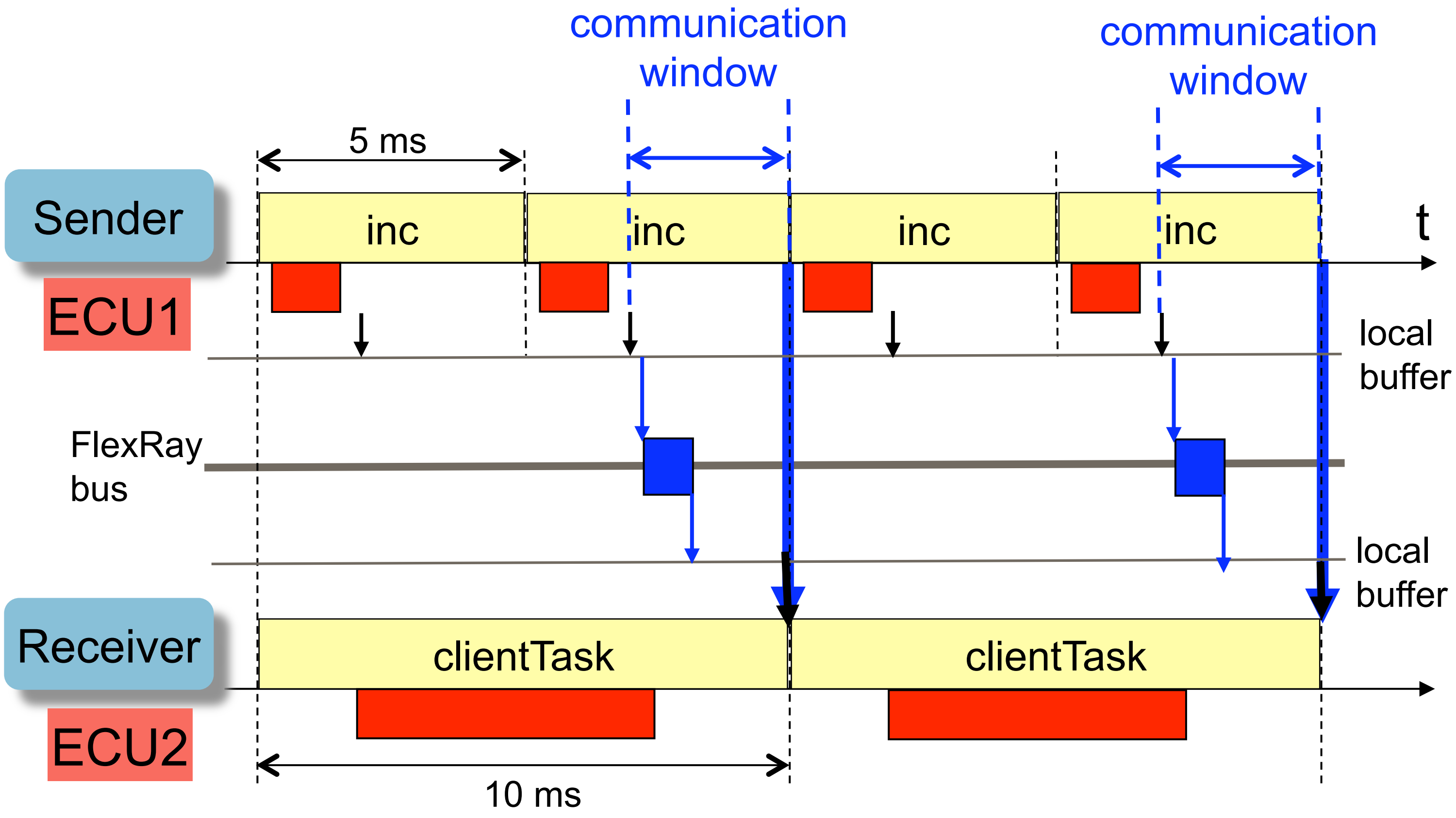
sample physical execution times on ECU1/ECU2 🟥



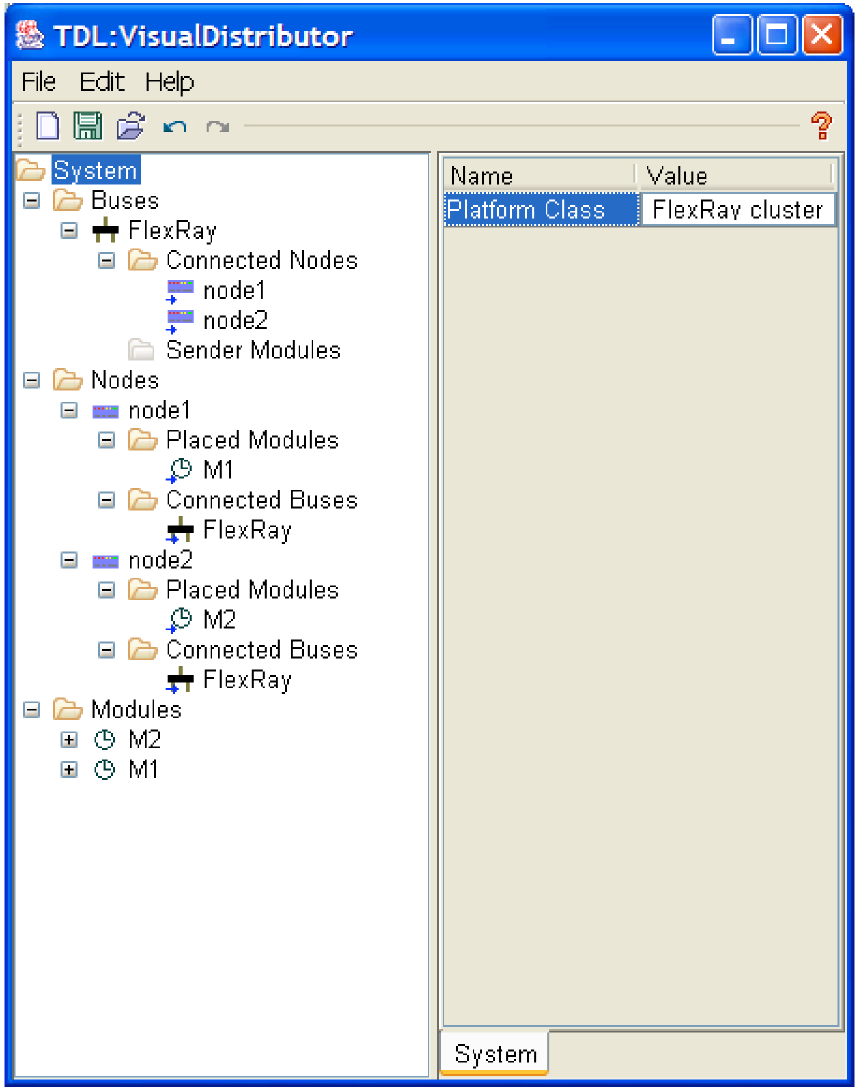
Constraints for automatic schedule generation



Bus schedule generation



TDL:VisualDistributor maps TDL modules to nodes

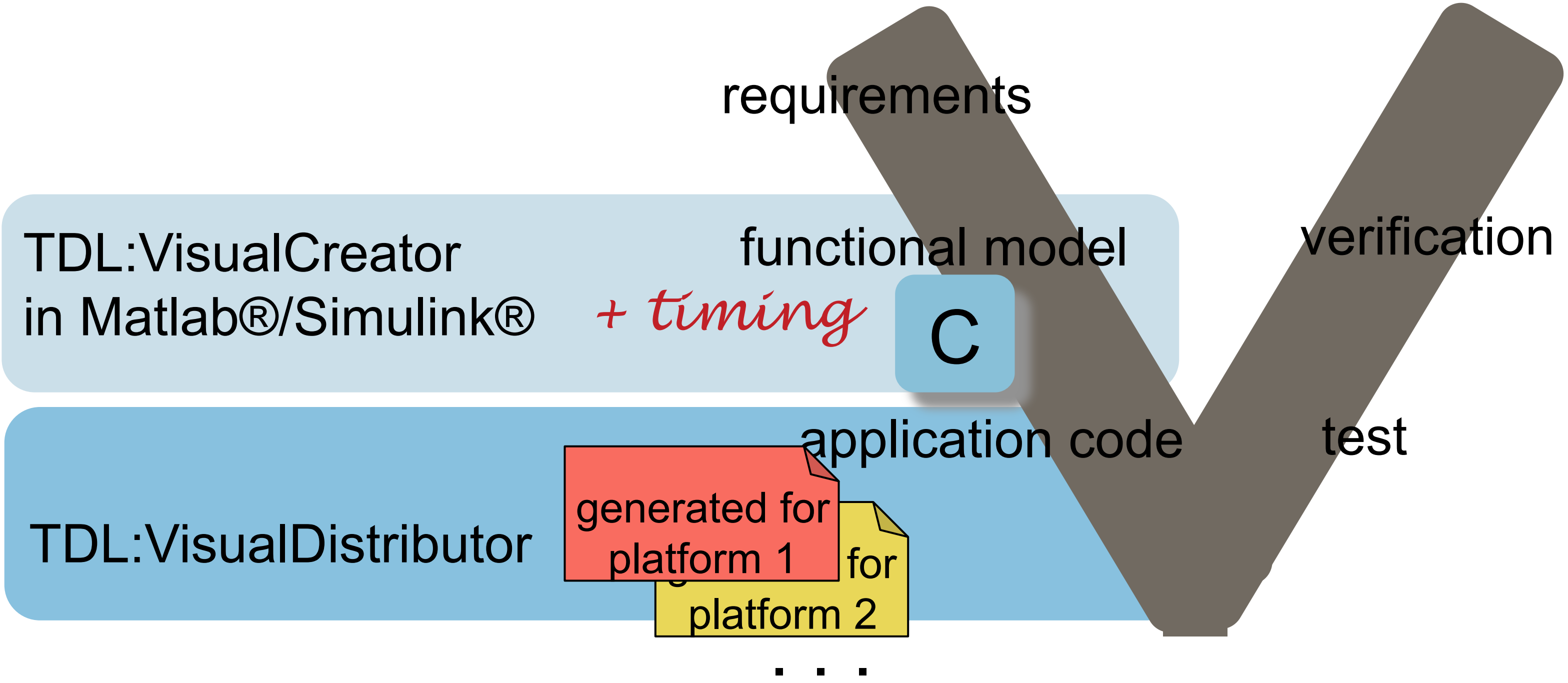


```
System
⊟ Buses
    ⊟ FlexRay
        ⊟ Connected Nodes
              node1
              node2
          Sender Modules
⊟ Nodes
    ⊟ node1
        ⊟ Placed Modules
              M1
        ⊟ Connected Buses
              FlexRay
    ⊟ node2
        ⊟ Placed Modules
              M2
        ⊟ Connected Buses
              FlexRay
⊟ Modules
    ⊞ M2
    ⊞ M1
```

Name	Value
Platform Class	FlexRay cluster

TDL-based development process

preeTEC tools in the V model



TDL:VisualCreator in Matlab®/Simulink®

TDL:VisualDistributor

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.)

Thank you for your attention!