UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

isp

Hardware-based Runtime Verification with Embedded Tracing Units and Stream Processing

Lukas Convent Sebastian Hungerecker Torben Scheffel
Malte Schmitz **Daniel Thoma** Alexander Weiss

Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany

The 18th International Conference on Runtime Verification

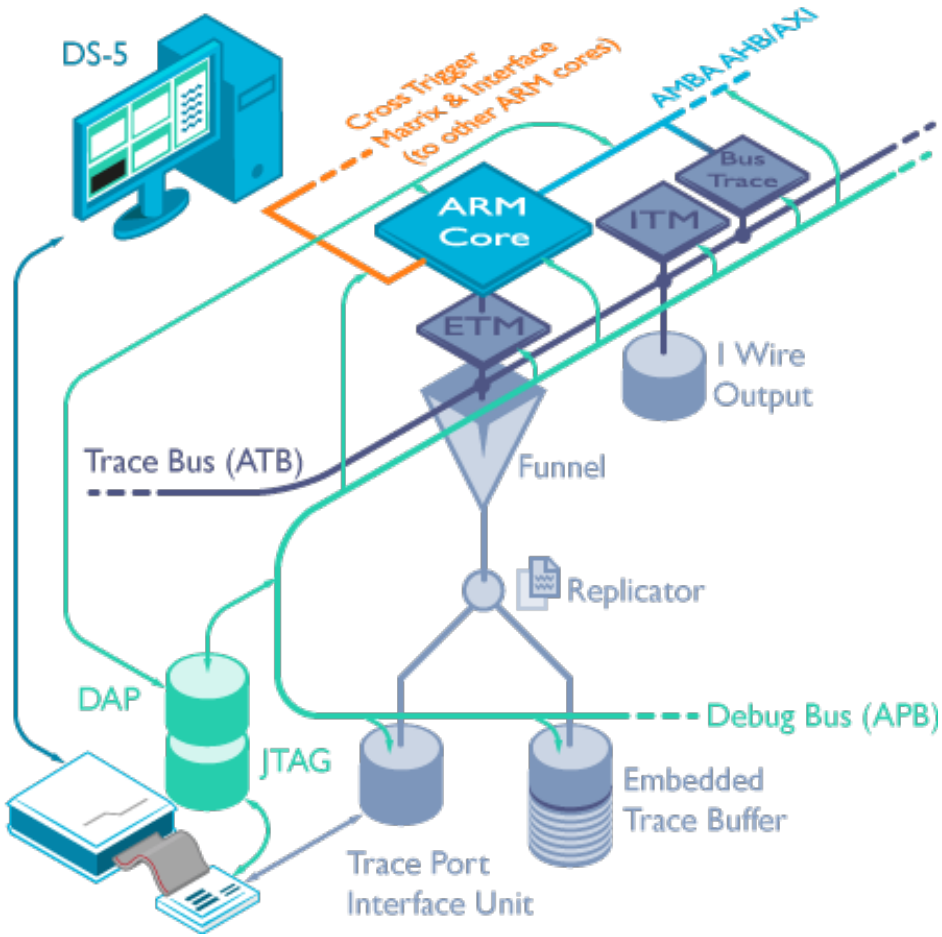
Outline

Interactive Hardware Monitoring Workflow

State of the Art Hardware Monitoring / Debugging

- Embedded, hybrid and cyber-physical systems have tight time and resource constraints.
- Comprehensive logging output in the software (e.g. via instrumentation) decreases the performance significantly.
- Breakpoint-based debugging features of the processor are slow due to the potentially high number of interruptions.
- Logging and breakpoints are highly intrusive. Problematic for concurrent programs or real-time applications.

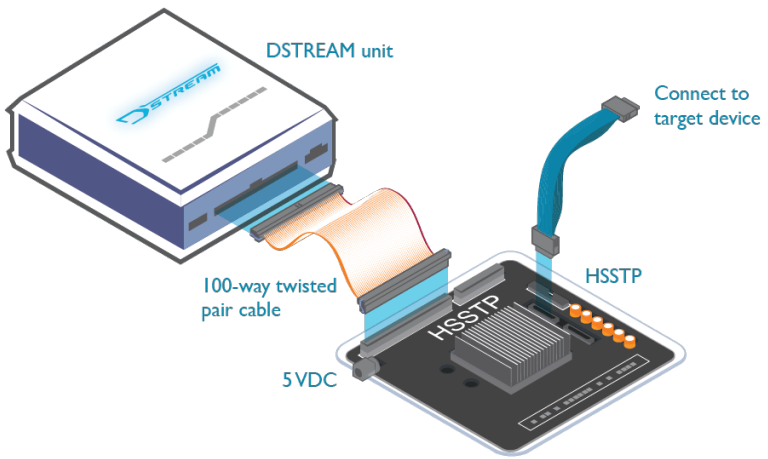
Embedded Tracing Unit: ARM CoreSight

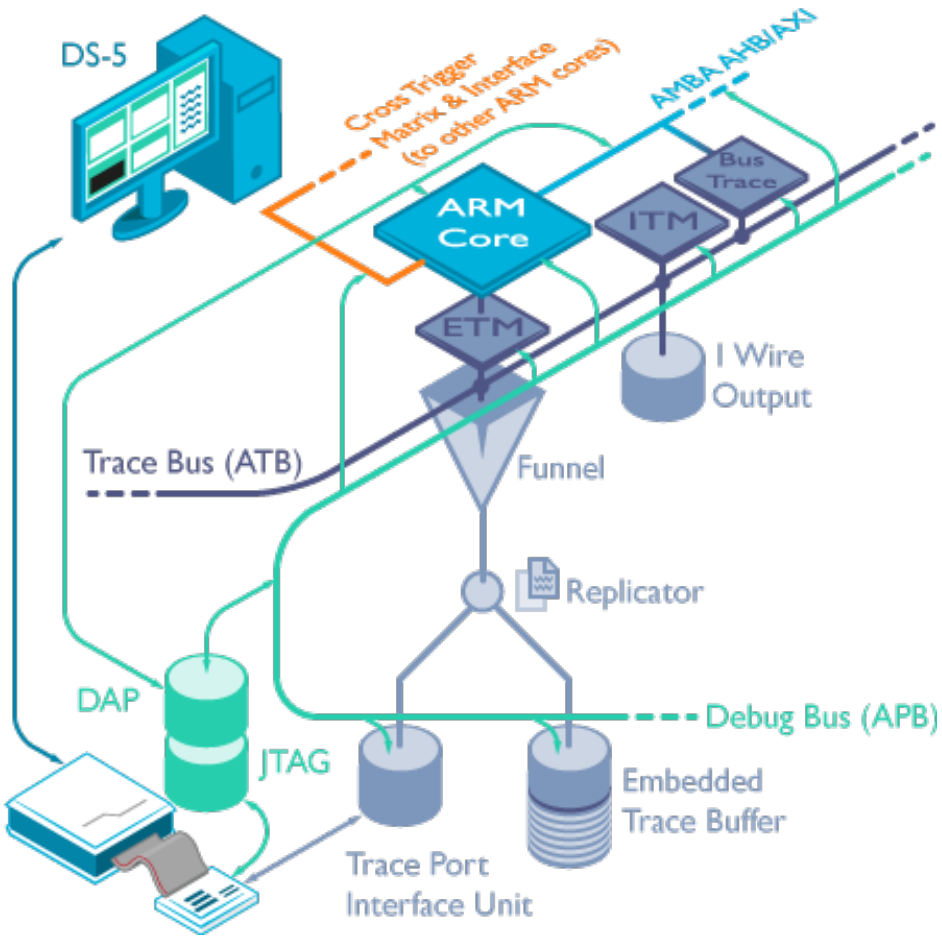


Embedded Tracing Unit: ARM CoreSight

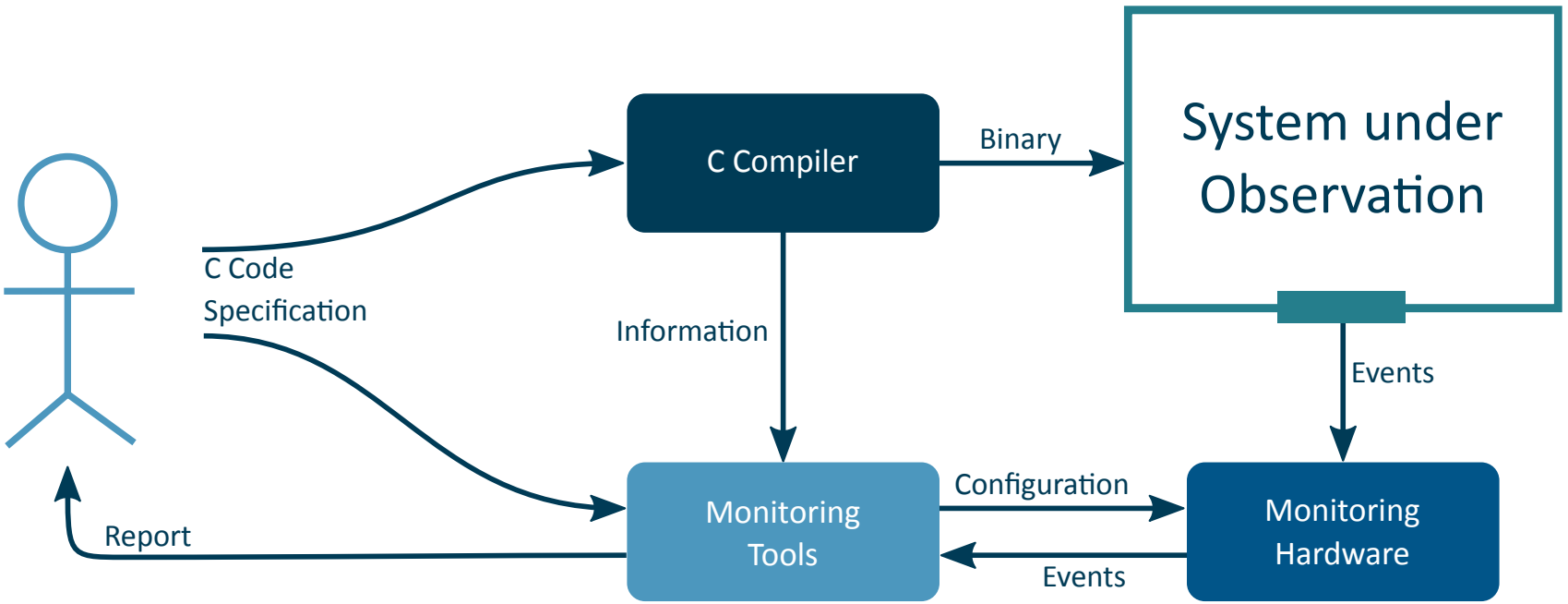
DSTREAM

- Record trace for offline reconstruction and analysis.
- Traces can be recorded for at most a few seconds.
- Trace buffer of 4GB for a recording speed of 10 Gbit/s.

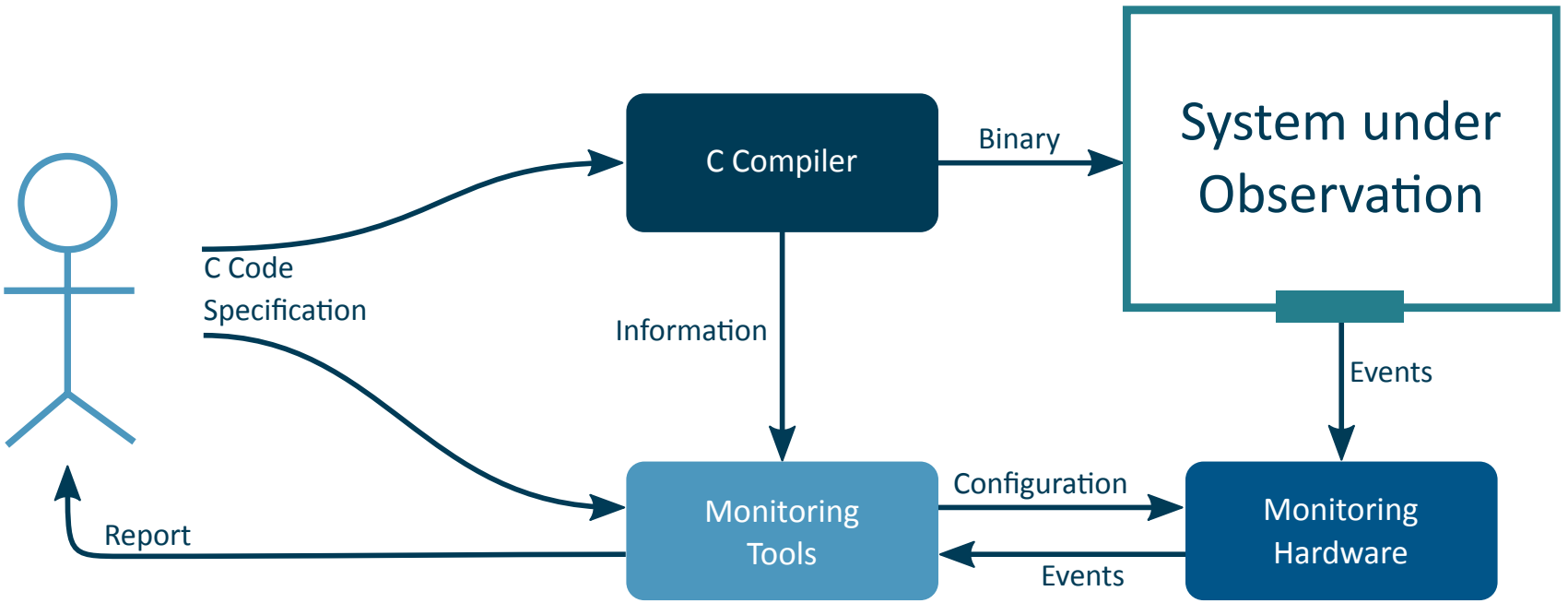




Interactive Hardware Monitoring Workflow



Interactive Hardware Monitoring Workflow



Fast reconfigurablility for interactive debugging sessions

Synthesized FPGA design

- Trace Reconstruction
- Monitoring Engine

configured through memory with

- program binary (LUT of jumps)
- monitoring specification

ICT-10-2016

COEMS

Continuous Observation of Embedded Multicore Systems

UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING AND PROGRAMMING LANGUAGES

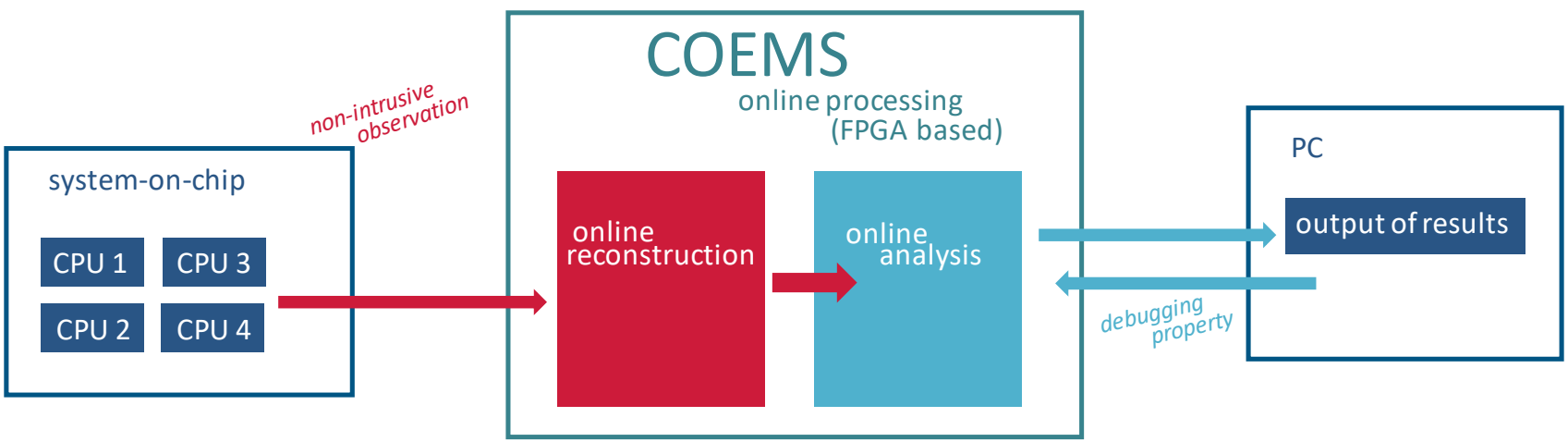
ACCEMIC TECHNOLOGIES

AIRBUS

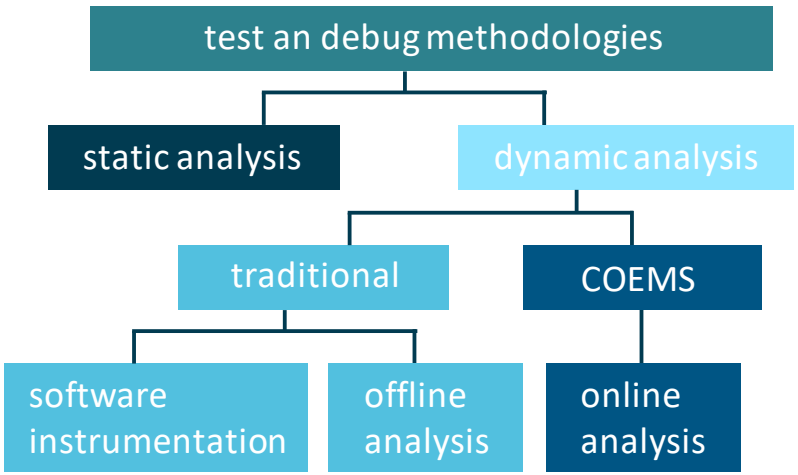
Høgskulen på Vestlandet

THALES

COEMS Hardware



COEMS Advantages

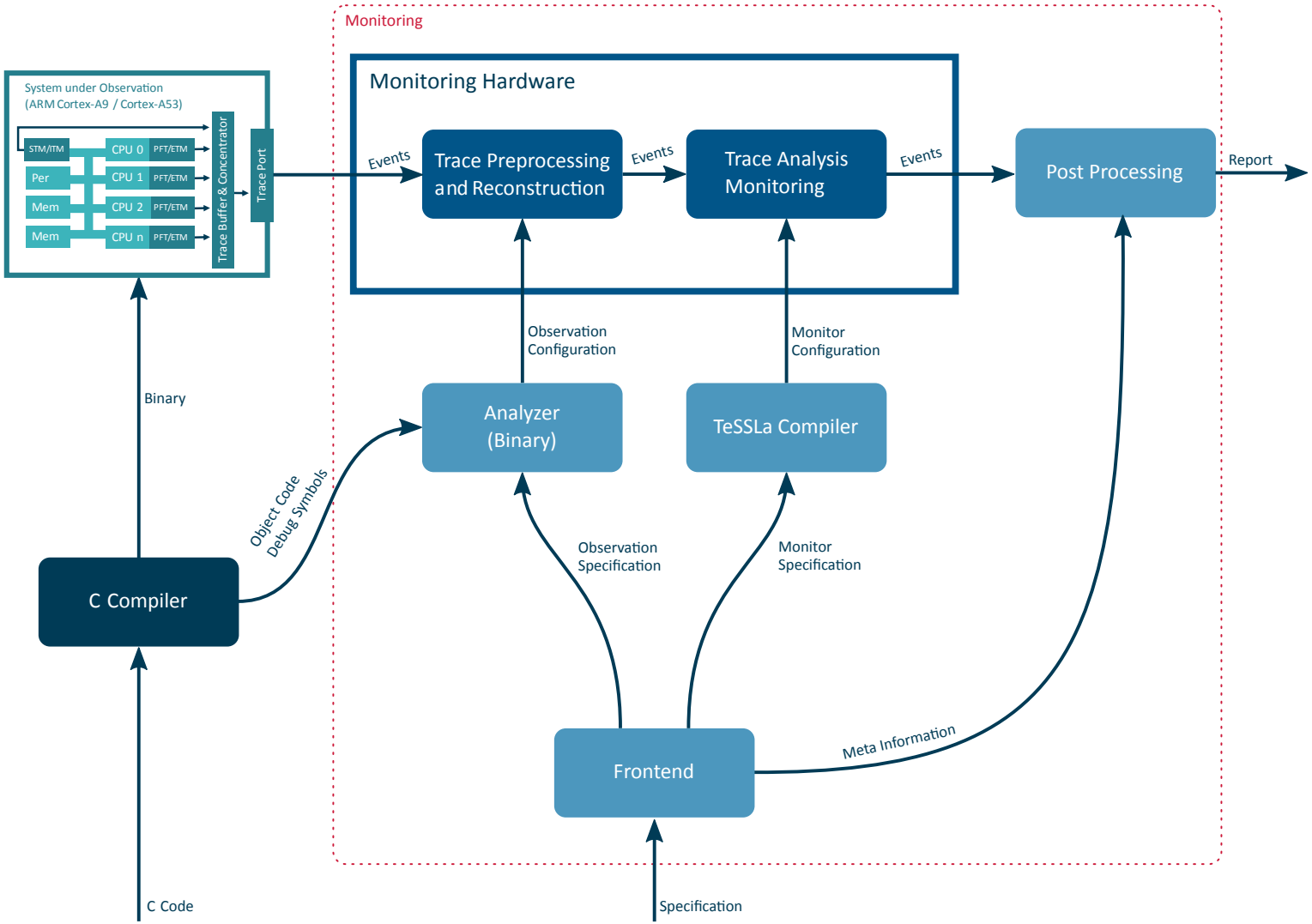


	software instrumentation	offline analysis	online analysis
long observation	✓	✗	✓
non-intrusiveness	✗	✓	✓
multicore	✗	!	✓
multiple focuses	✓	!	✓
autonomous operation	!	!	✓
time and cost efficiency	!	!	✓
results immediately	✓	✗	✓

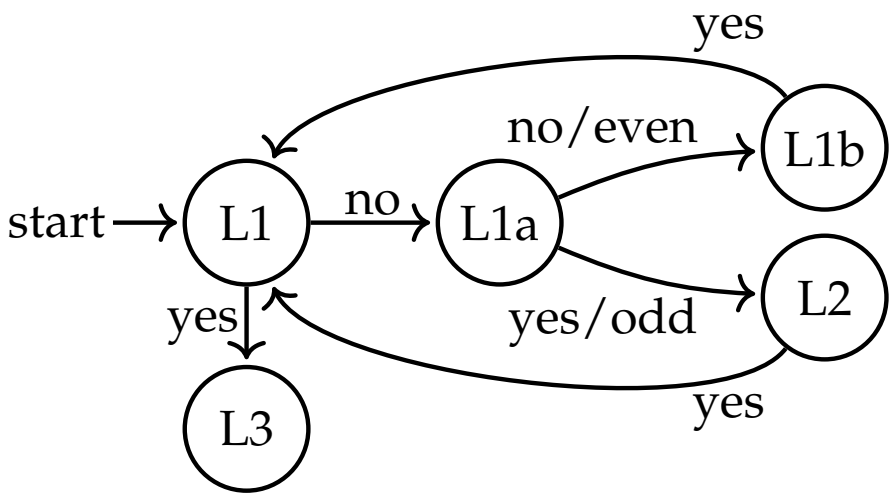
✓ well suited solution
! significant limitations
✗ unsuitable in most cases

Monitoring Program Flow Trace

Monitoring Program Flow Trace: Tools



Trace Reconstruction



C Code

```
while (n > 1) {


  if (n % 2 == 0) {



    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

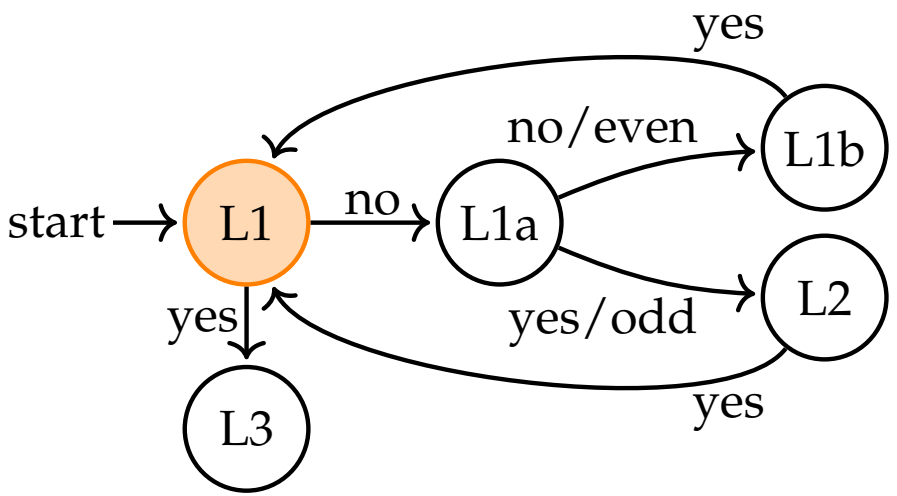
Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?

Events

Trace Reconstruction



C Code

```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

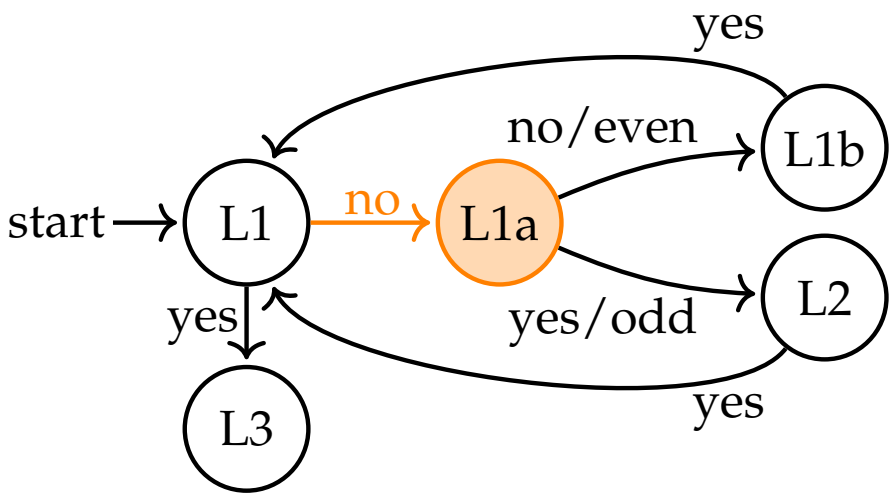
Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?

Events

Trace Reconstruction



C Code

```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

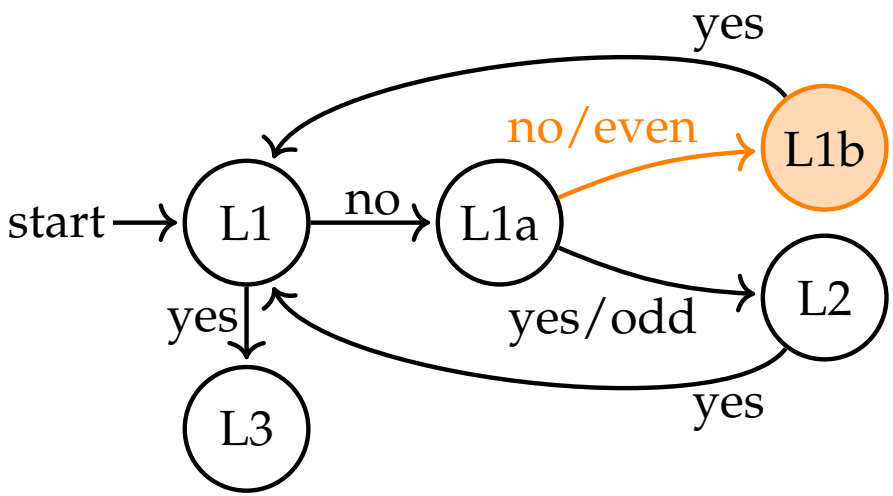
```
.L1:
        cmp $n, 1
        ble .L3
.L1a:
        $tmp = $n % 2
        cmp $tmp, 0
        bne .L2
.L1b:
        $n = $n / 2
        b .L1
.L2:
        $n = 3 * $n
        $n = $n + 1
        b .L1
.L3:
```

Branch taken?

no

Events

Trace Reconstruction



C Code

```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?

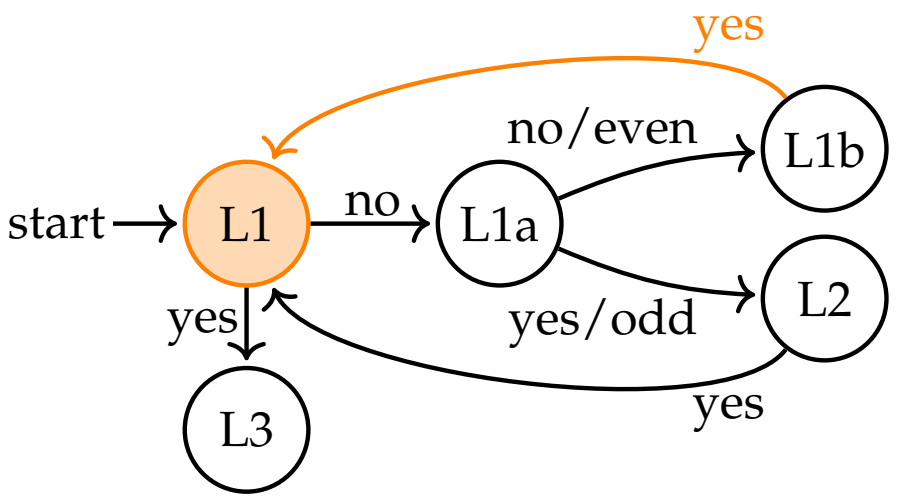
no

no

Events

even

Trace Reconstruction



C Code

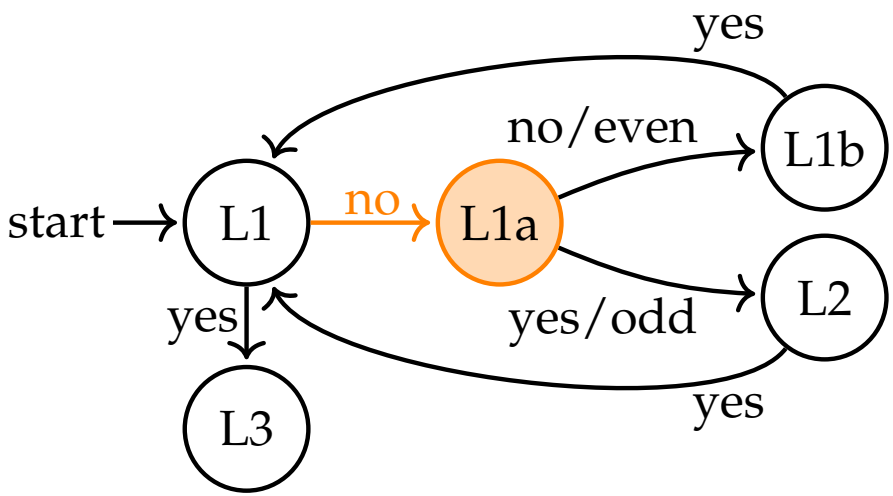
```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	

Trace Reconstruction



C Code

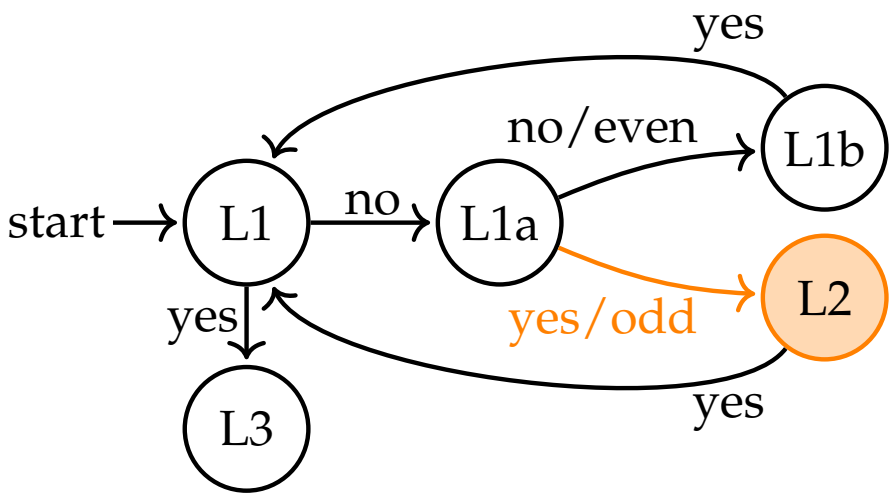
```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	

Trace Reconstruction



C Code

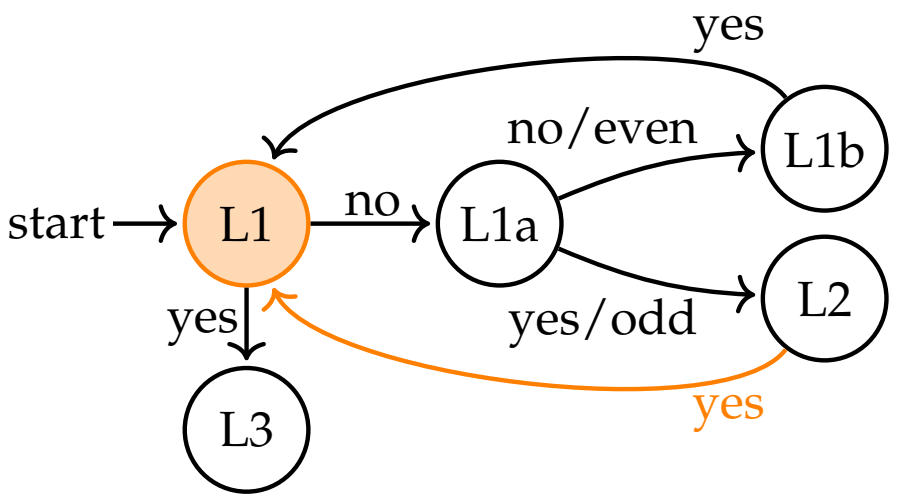
```
while (n > 1) {


  if (n % 2 == 0) {



    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	
yes	odd

Trace Reconstruction



C Code

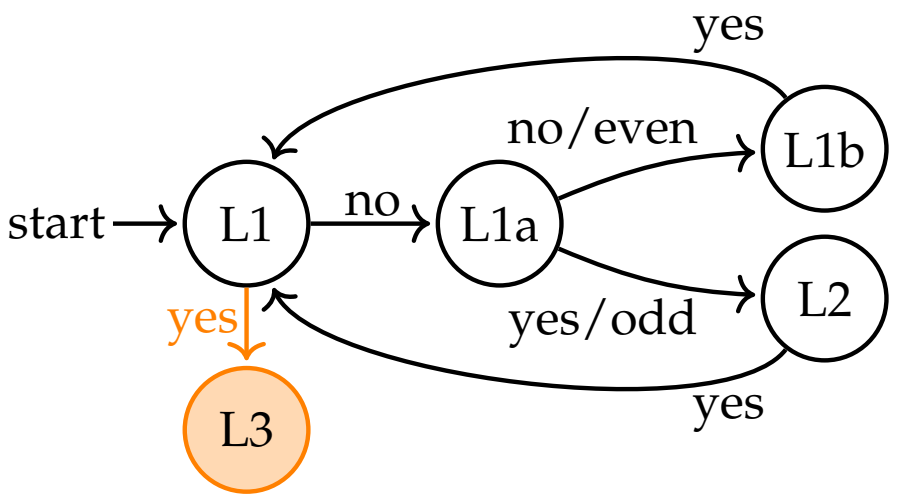
```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	
yes	odd
yes	

Trace Reconstruction



C Code

```
while (n > 1) {

  if (n % 2 == 0) {

    // even
    n = n / 2;
  } else {
    // odd
    n = 3 * n + 1;
  }
}
```

Assembler

```
.L1:
      cmp $n, 1
      ble .L3
.L1a:
      $tmp = $n % 2
      cmp $tmp, 0
      bne .L2
.L1b:
      $n = $n / 2
      b .L1
.L2:
      $n = 3 * $n
      $n = $n + 1
      b .L1
.L3:
```

Branch taken?	Events
no	
no	even
yes	
no	
yes	odd
yes	
yes	

Multithreading

Problem: Distinguish multiple threads

- We can only distinguish instructions traced from the different cores.
- Scheduler can execute multiple threads on the same core.
- Context switch reconfigures MMU
  $\Longrightarrow$ Same logical addresses used in different threads.

Solution: Context ID Register

1. OS writes thread ID to context ID register.
2. Tracing unit generates context ID message.
3. Trace reconstruction changes lookup table
   for the program flow reconstruction information.

Monitoring Properties with Stream Processing

Stream Processing: Technical Prerequisites

- Multi-core CPUs generate large amounts of trace data.
  $\Longrightarrow$ Perform monitoring in hardware.
- FPGAs have limited amount of memory.
  $\Longrightarrow$ Explicit memory usage. Constant memory usage per operator.
- Properties and analyses might become very complex.
  $\Longrightarrow$ Combined monitoring on hardware and in software.
- Timing is crucial in embedded and cyber-physical systems.
  $\Longrightarrow$ Support time as first-class citizen.
- Properties and analyses might require data.
  $\Longrightarrow$ Support analyses and aggregation of data values.

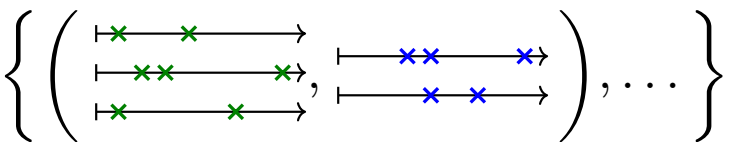
Temporal Specification Languages

LTL formula, regular expression

$$\left\{ a_1 a_2 a_3 \ldots, b_1 b_2 b_3 \ldots, \ldots \right\}$$

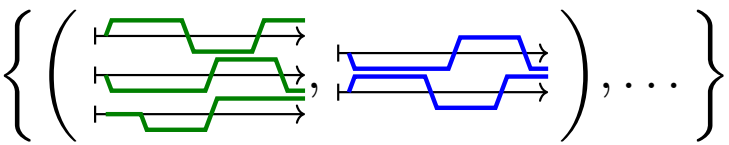
Set of traces

Stream / event processor



Function from streams to streams

Functional reactive program



Function from signals to signals

TeSSLa Design Goals

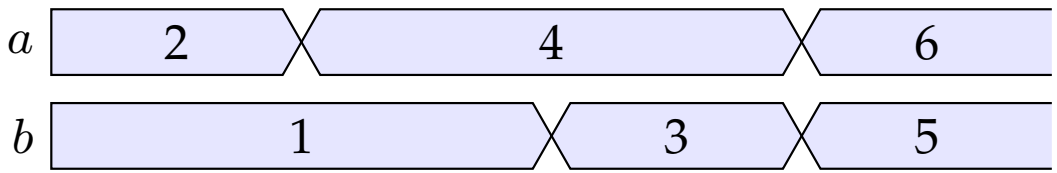
- **Declarative** style: Specification rather than implementation
- **Modularity**: Allowing abstractions based on **few primitives**
- **Time** as first-class citizen
- Abstractions for both **events** and **signals**
- **Recursion** to reason about past
- Implementable with **limited memory**

Stream Processing with TeSSLa

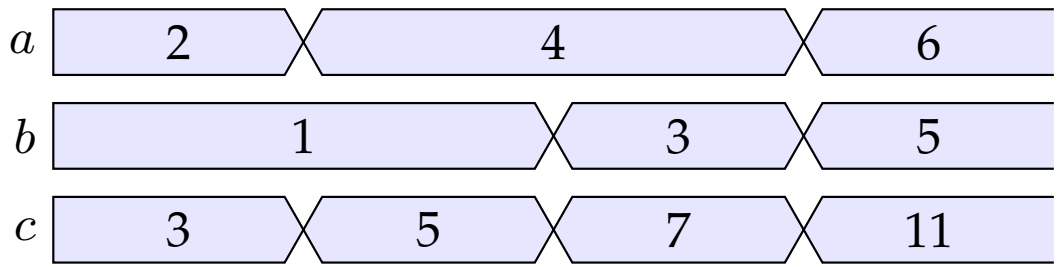
www.tessla.io

Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, Daniel Thoma.
TeSSLa: Temporal Stream-based Specification Language.
SBMF 2018, CoRR abs/1808.10717.

TeSSLa by Example

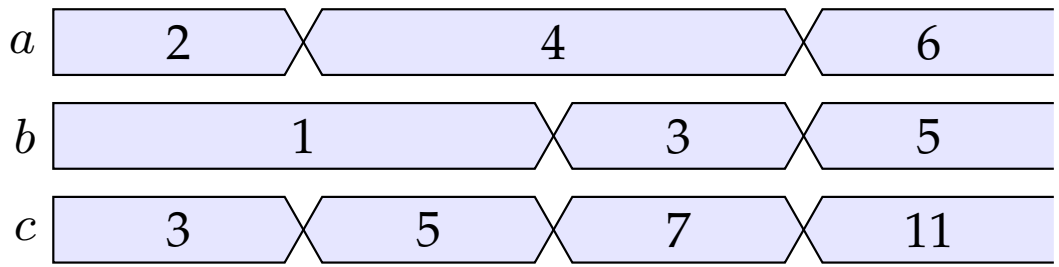


TeSSLa by Example

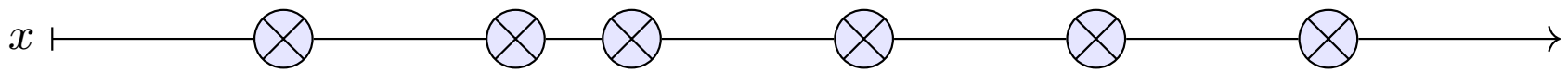



```
def c := a + b
```

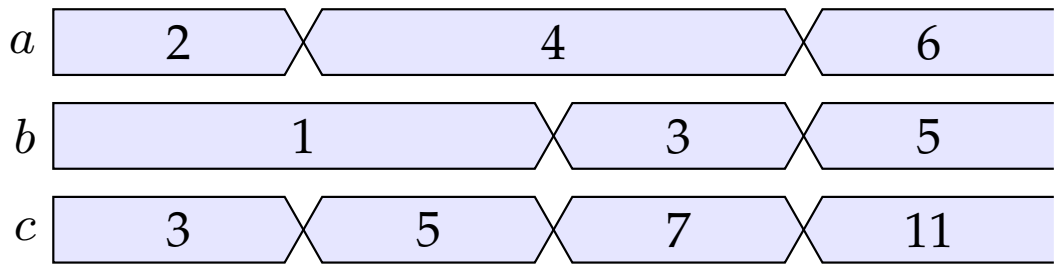
TeSSLa by Example



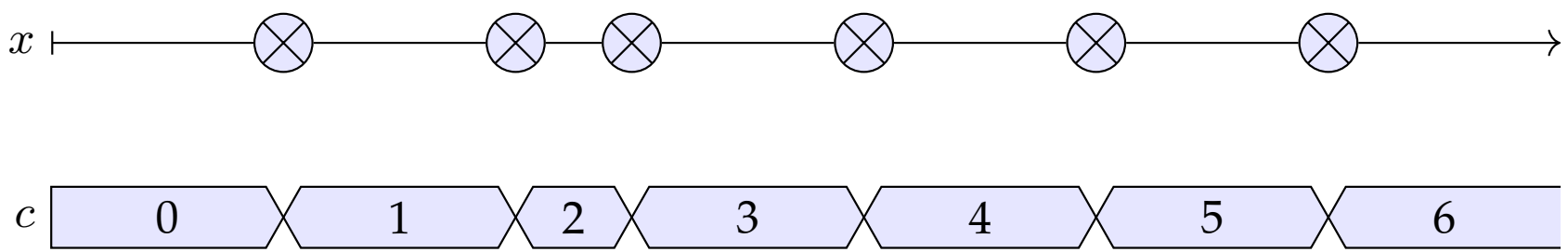

```
def c := a + b
```



TeSSLa by Example

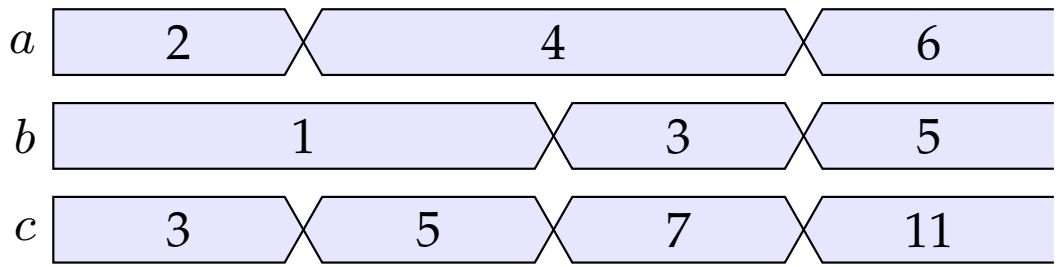



```
def c := a + b
```

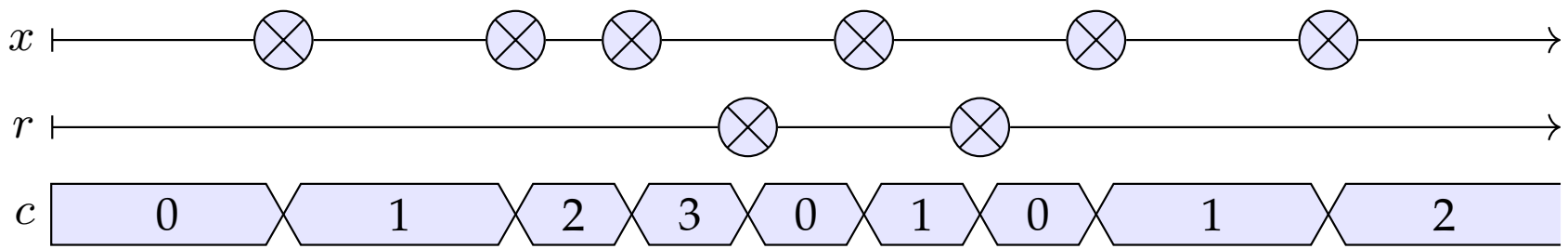



```
def c := eventCount(x)
```

TeSSLa by Example




```
def c := a + b
```




```
def c := eventCount(x, reset = r)
```

Why TeSSLa?

TeSSLa vs. timed LTL

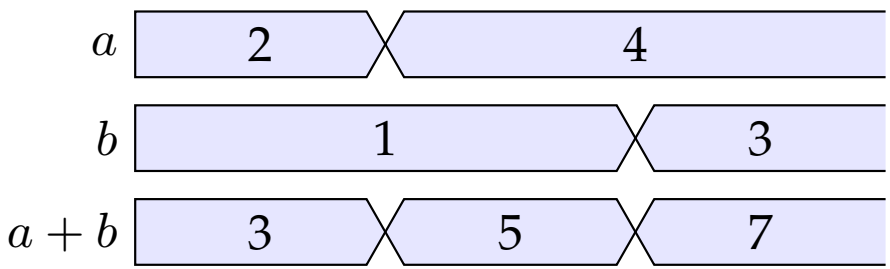
- timed TeSSLa fragment is equivalent to timed finite state transducers
- timed LTL is a logic
- TeSSLa is more than a logic
  - TeSSLa can check traces (like a logic)
  - TeSSLa can aggregate information
  - TeSSLa can compute statistics

TeSSLa vs. LOLA

- LOLA needs synchronous events on all input streams
- TeSSLa supports asynchronous (sparse) events on input streams with a global clock
- Time is first-class citizen in TeSSLa, every event has a timestamp

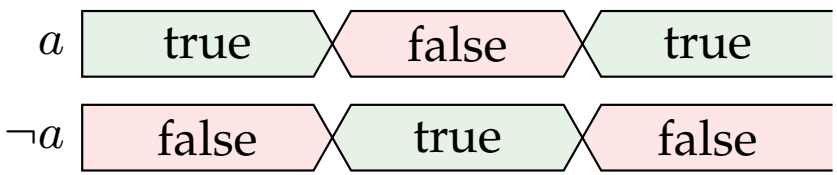
TeSSLa Operators: Signal Lift of Addition

- *Signal lift* allows to lift operations on arbitrary data types to streams.
- E.g. the *addition* on integer numbers can be lifted to streams of integers.



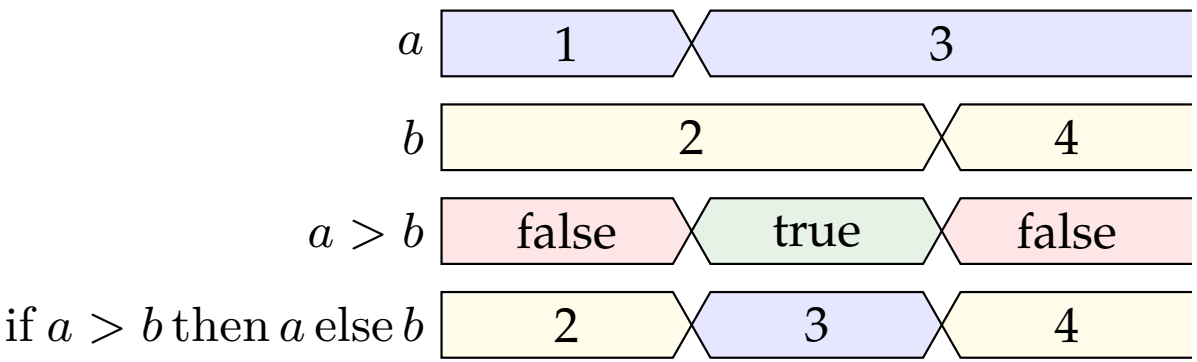
TeSSLa Operators: Signal Lift of Negation

- *Signal lift* allows to lift operations on arbitrary data types to streams.
- E.g. the *negation* of booleans can be lifted to a stream of booleans.



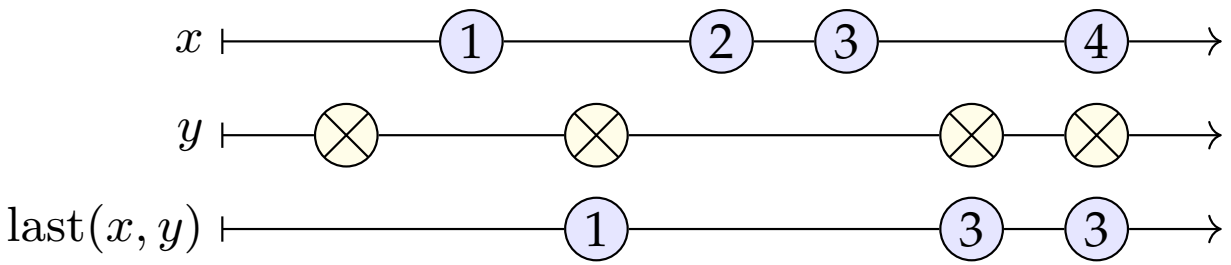
TeSSLa Operators: Signal Lift of If-Then-Else

- *Signal lift* allows to lift operations on arbitrary data types to streams.
- E.g. the ternary *if-then-else* function can be lifted to a stream of booleans and two streams of identical type.



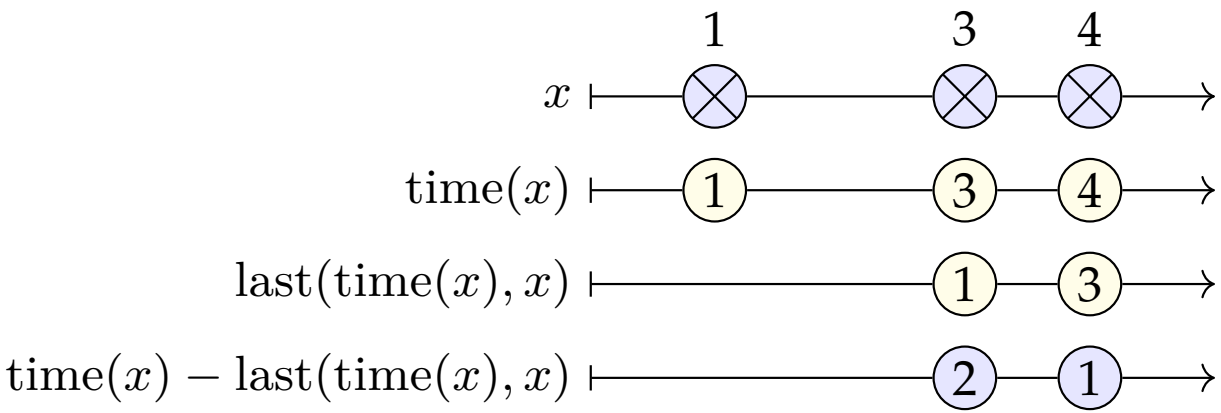
TeSSLa Operators: Last

- Needed to define properties over sequences of events.
- Last allows to refer to the values of events on one stream that occurred strictly before the events on another stream



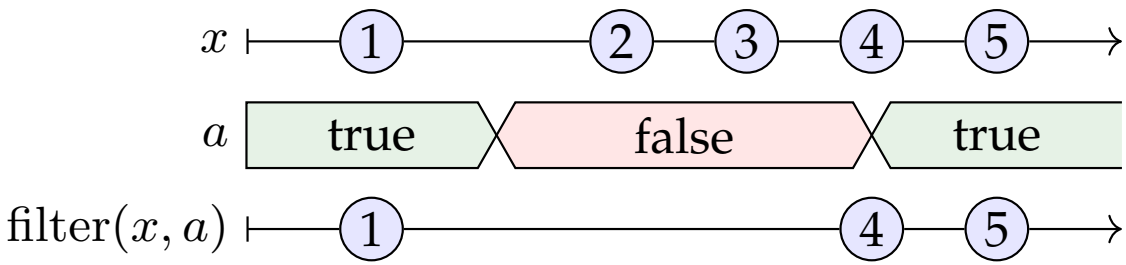
TeSSLa Operators: Time

- Provides access to the *timestamps* of events
- Produces events carrying their *timestamps as data value*
- Hence *all operators* for data values can be applied to timestamps.



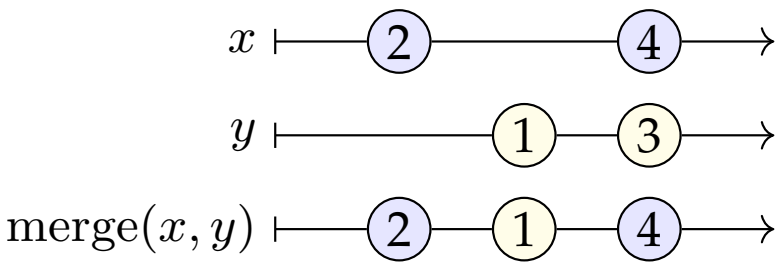
TeSSLa Operators: Filter

- Process streams in an *event-oriented fashion*
- *Filter* the events of one stream based on a second boolean stream interpreted as piecewise constant signal.



TeSSLa Operators: Merge

- Process streams in an *event-oriented fashion*
- *Merge* combines two streams into one,
  giving preference to the first stream when both streams contain identical timestamps.



TeSSLa Operators: Const and Nil

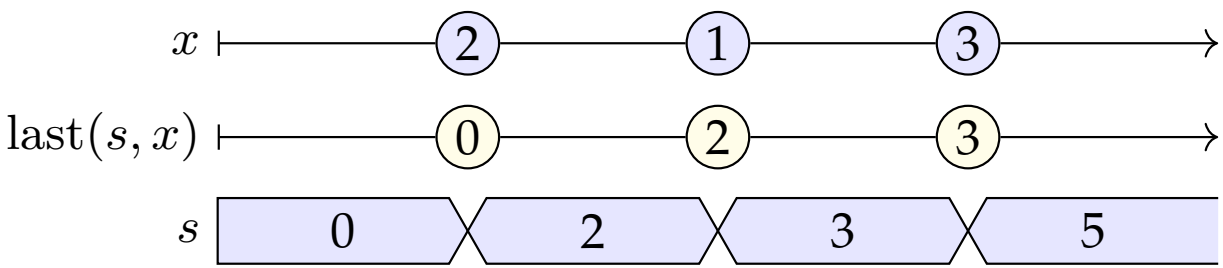
- The constant *nil* for the empty stream
- The operator *const* converting a value to a stream starting with that value at timestamp 0.

Implicit Conversions

- Integer and Boolean constants are converted to streams via *const*.
- Build-in operators on integers and Booleans are lifted to streams via *signal lift*.

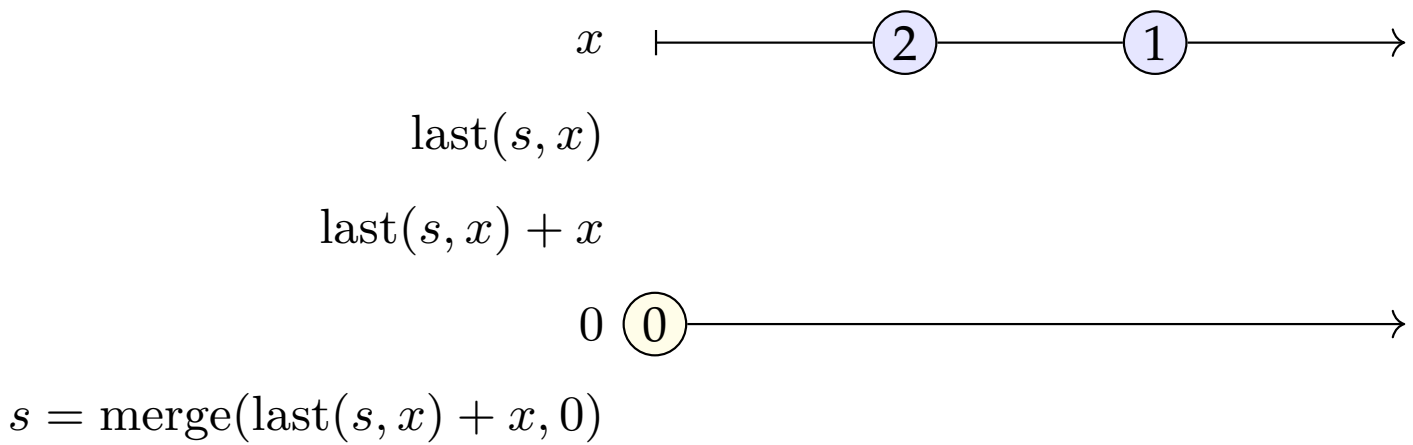
Recursive Equations in TeSSLa

- The *last* operator allows to write *recursive equations*
- The *merge* operation allows to *initialize* recursive equations with an initial event from an other stream.
- Express *aggregation* operations like the *sum* over all values of a stream.

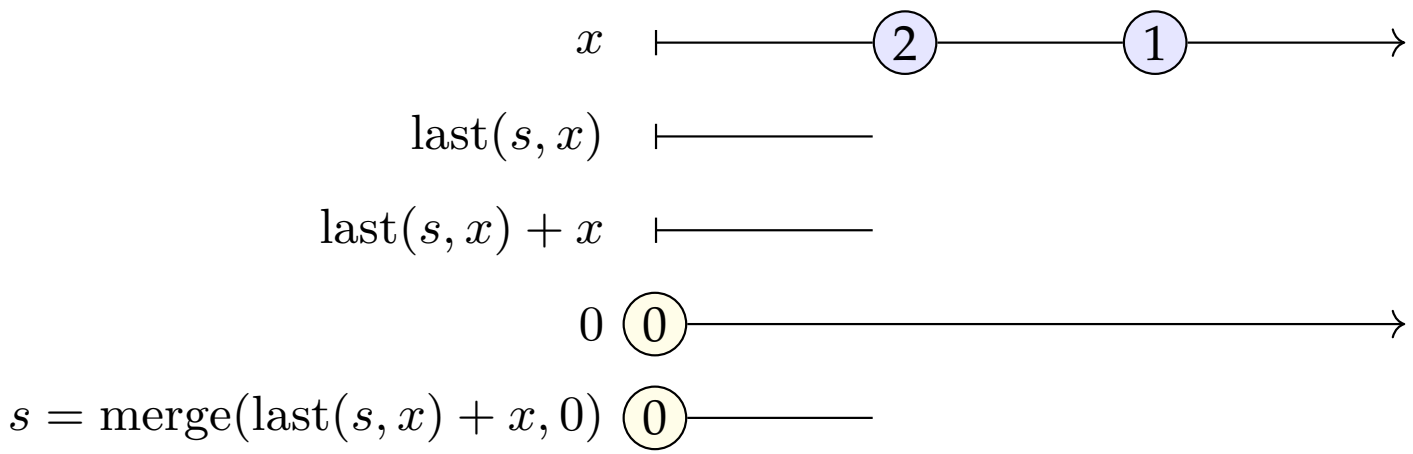



```
def s := merge(last(s, x) + x, 0)
```

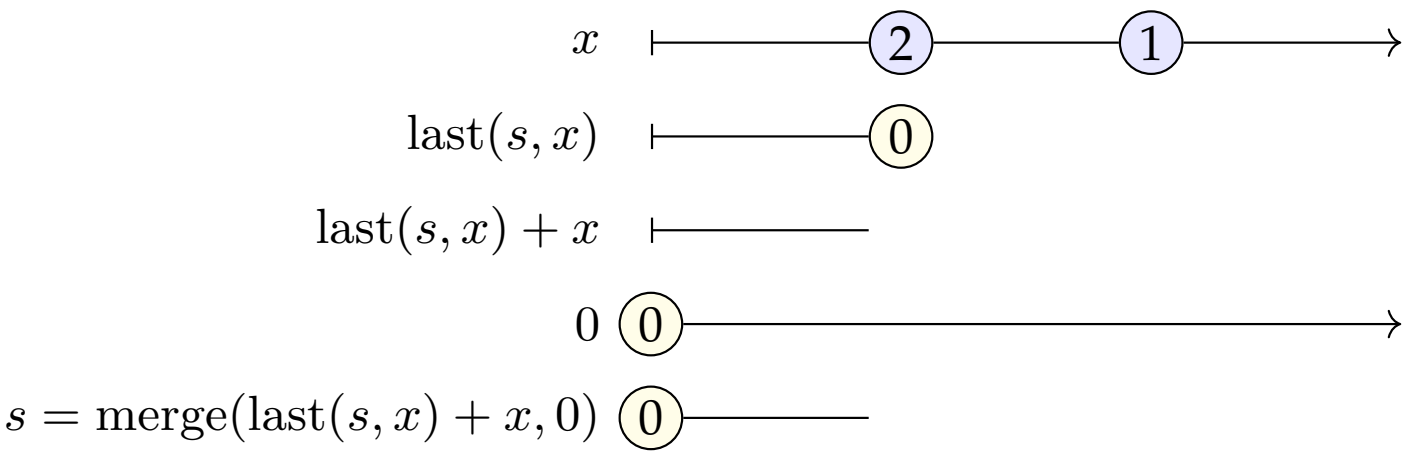
Recursive Equations in TeSSLa: How It Works



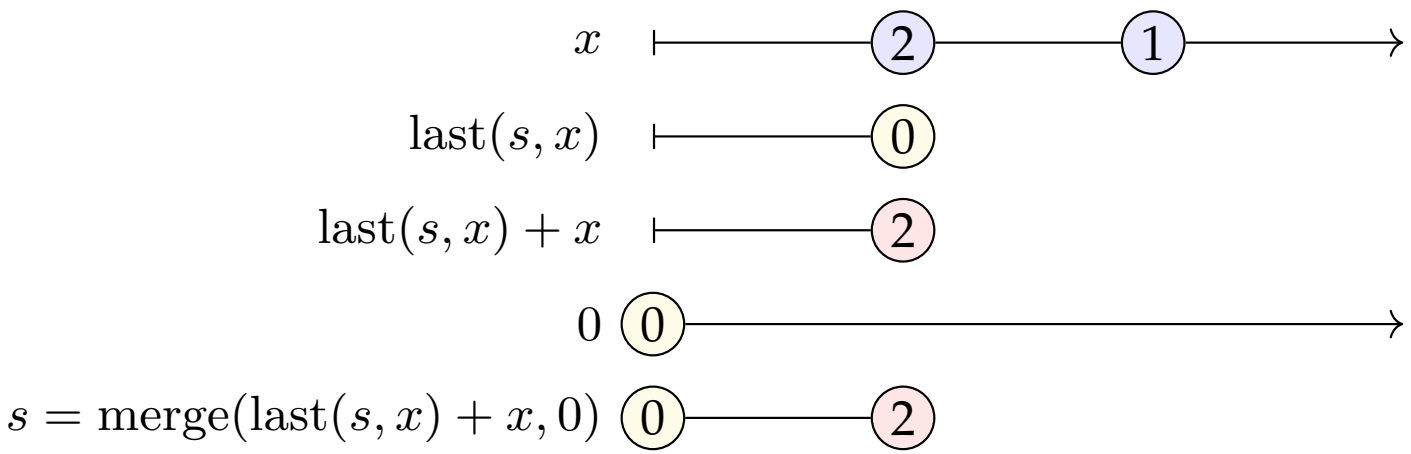
Recursive Equations in TeSSLa: How It Works



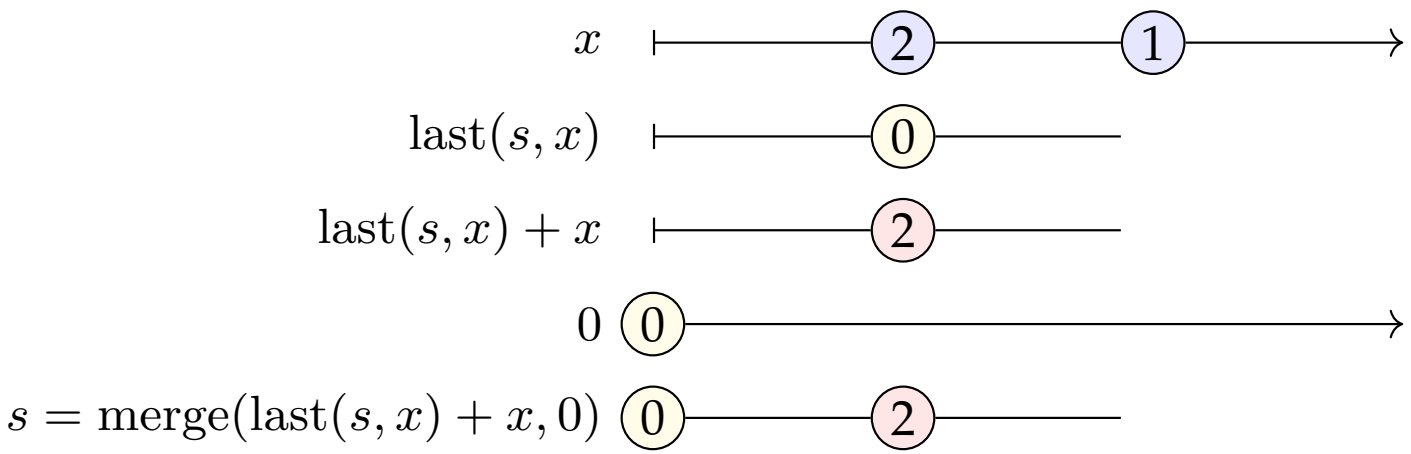
Recursive Equations in TeSSLa: How It Works



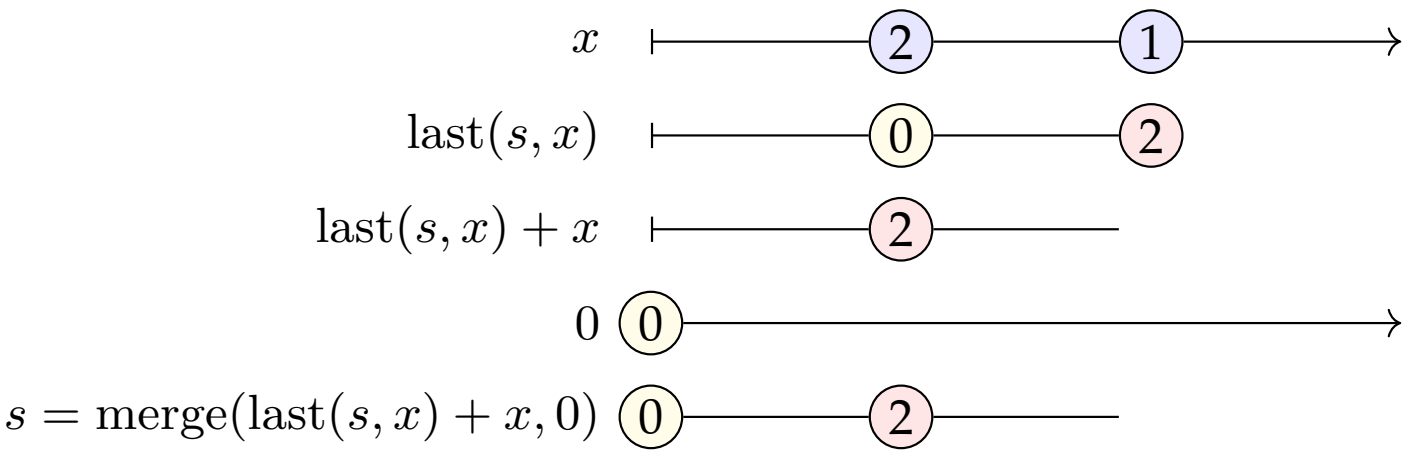
Recursive Equations in TeSSLa: How It Works



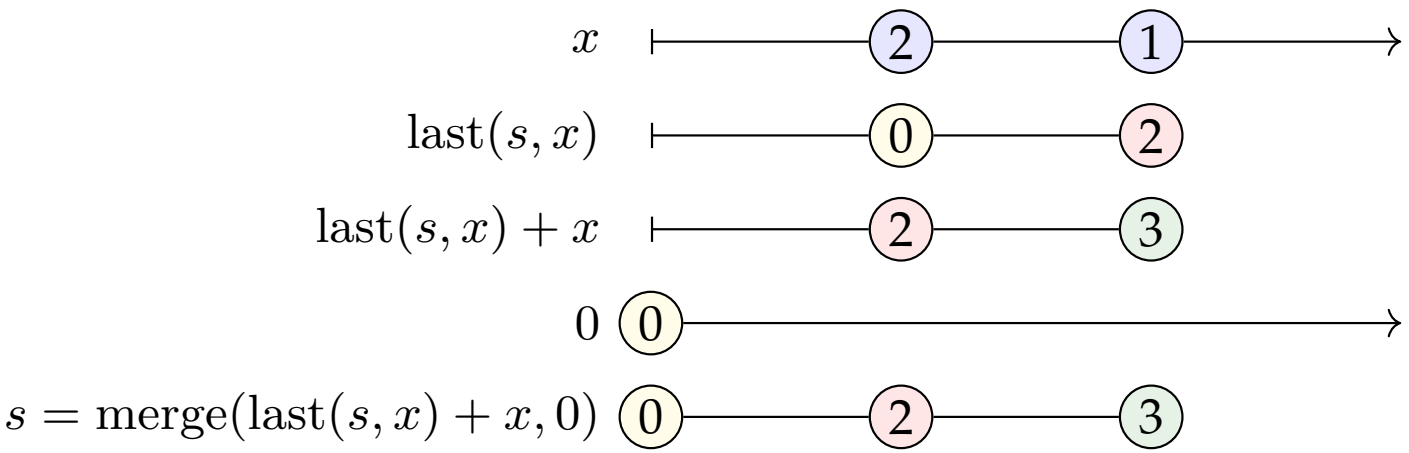
Recursive Equations in TeSSLa: How It Works



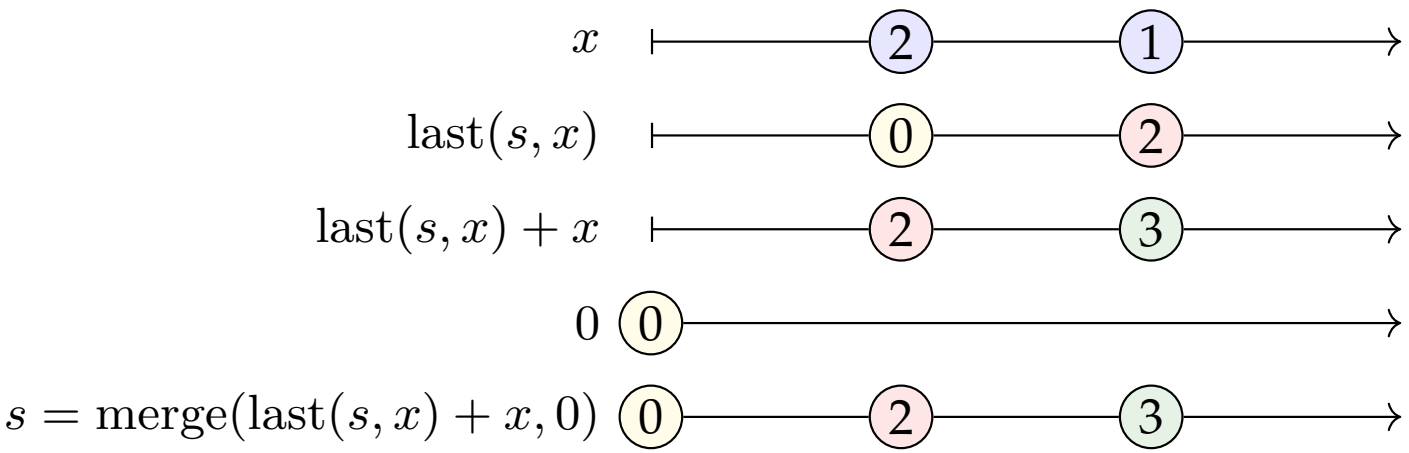
Recursive Equations in TeSSLa: How It Works



Recursive Equations in TeSSLa: How It Works



Recursive Equations in TeSSLa: How It Works



Macros in TeSSLa: eventCount

```
# Count the number of events on `values`.
def eventCount[A,B](values: Events[A]) := {
  def count: Events[Int] := merge(
    # increment counter
    last(count, values) + 1
  , 0)
  count
}
```

Macros in TeSSLa: eventCount With Reset

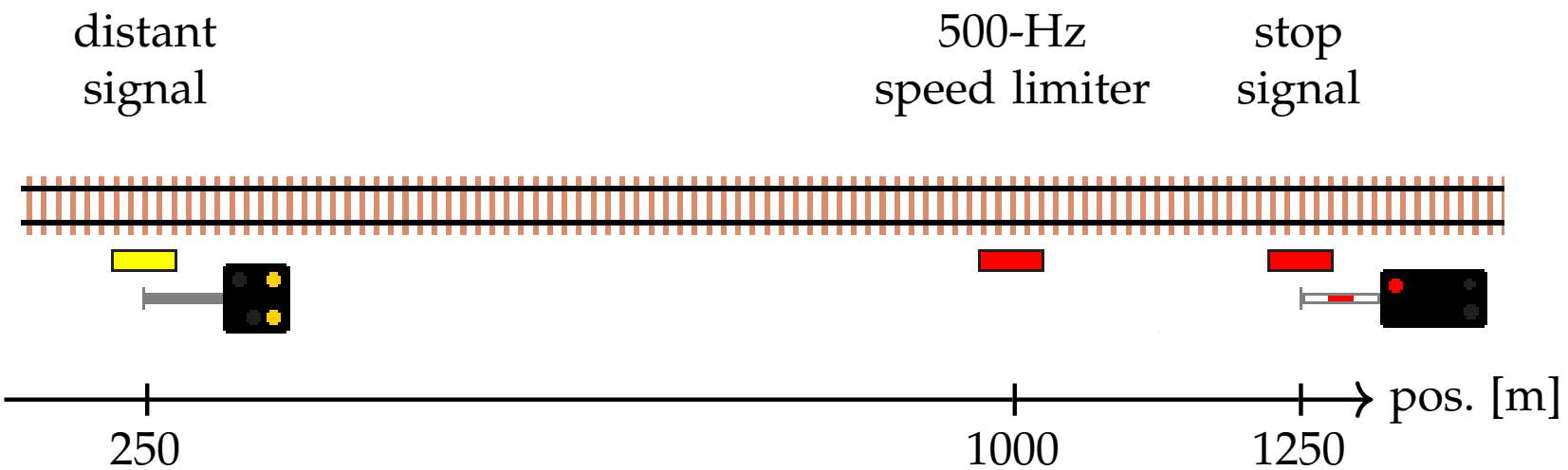
```
# Count the number of events on `values`. Reset the output to 0
# on every event on `reset`.
def eventCount[A,B](values: Events[A], reset: Events[B]) := {
  def count: Events[Int] := merge(

    # `reset` contains the latest event
    if merge(time(reset) > time(values), false)
    then 0

    # `reset` and `values` latest event happen simultaneously
    else if merge(time(reset) == time(values), false)
    then 1

    # `values` contains the latest event --> increment counter
    else last(count, values) + 1

  , 0)
  count
}
```

Data Types in TeSSLa

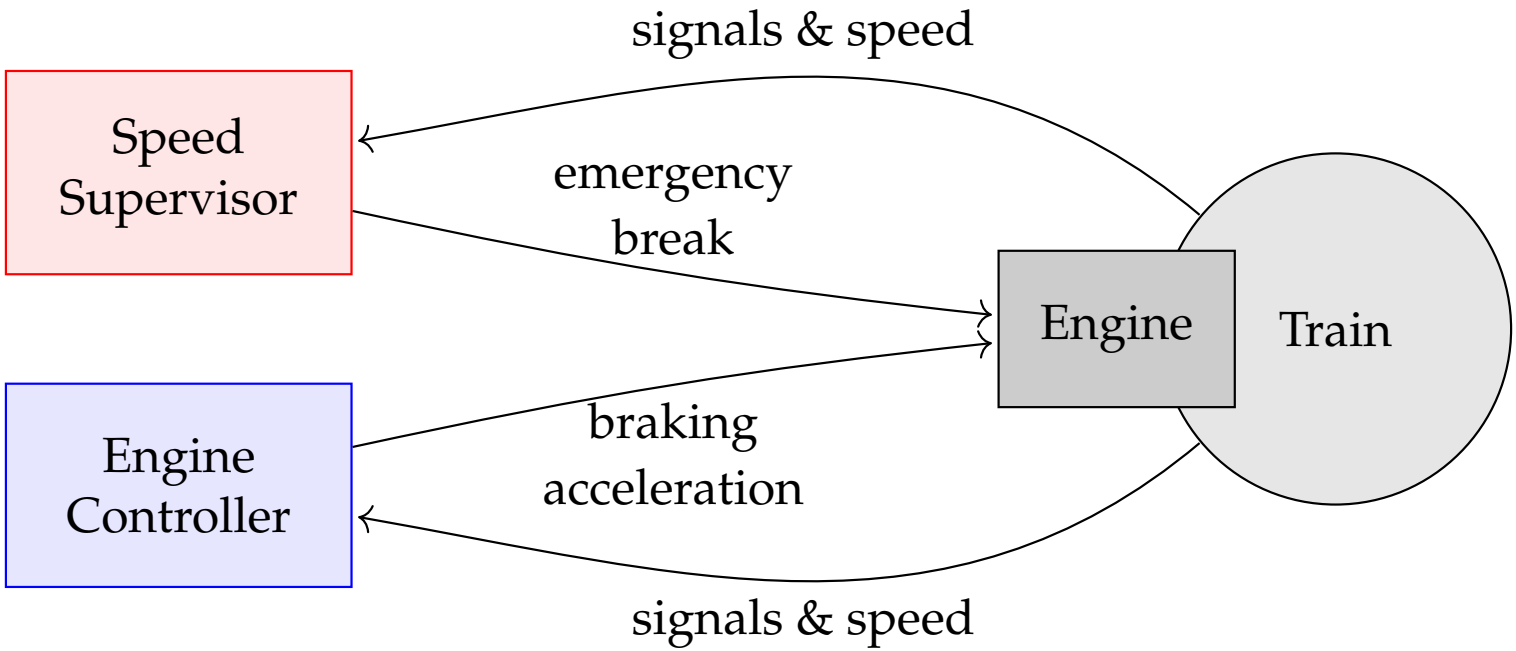
- TeSSLa is defined agnostically with respect to any time or data domain.
- Different data structures can be used to represent time and data.
- Monitoring in hardware: atomic data types, e.g. int or float.
- Monitoring in software: complex data structures like lists, trees and maps.

Example Scenario: Braking a Train With Punktförmige Zugbeeinflussung

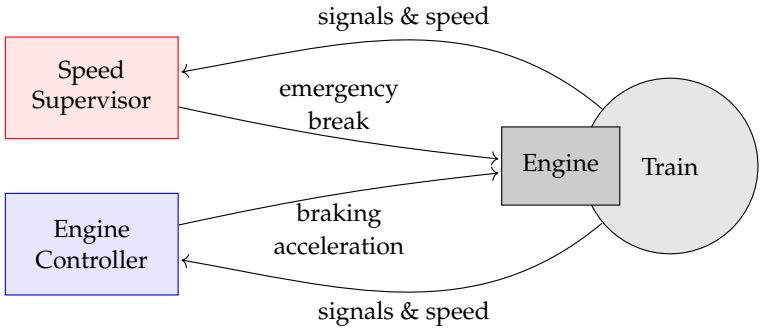
Example Scenario: Punktförmige Zugbeeinflussung

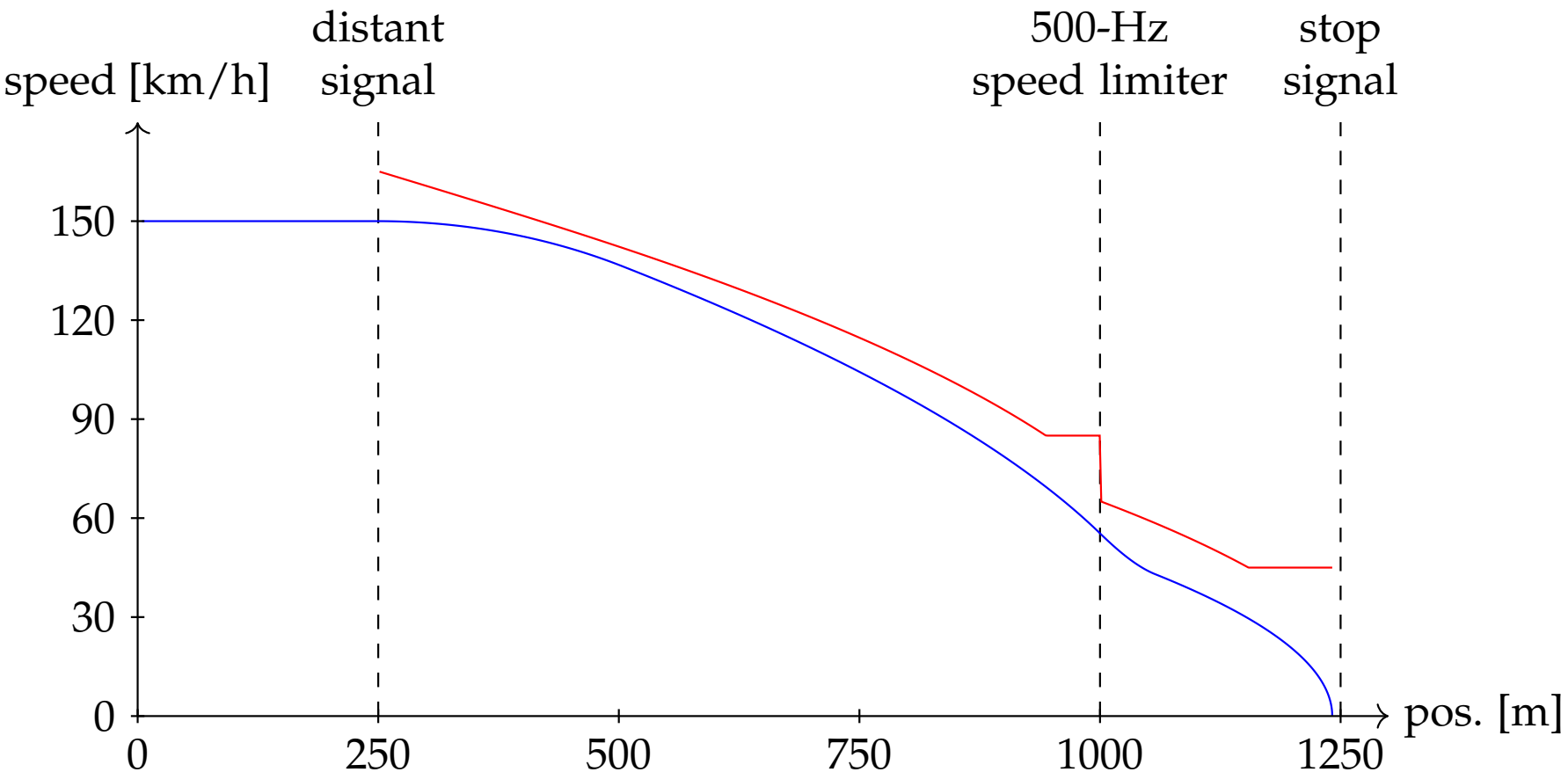


Example Scenario: Braking a Train



Example Scenario





Measuring Timing Constraints

Timing Constraints: Measuring Runtime

Property

Measure the runtime of one execution of the speed supervisor.

```
# specify events of interest
def call := function_call("supervisor")
def return := function_return("supervisor")

# specify property over the events
def supervisorRuntime := runtime(call, return)
```

Timing Constraints: Measuring Runtime

Property

Measure the runtime of one execution of the speed supervisor.

```
# specify events of interest
def call := function_call("supervisor")
def return := function_return("supervisor")

# specify property over the events
def supervisorRuntime := runtime(call, return)
```

TeSSLa standard library

```
def runtime(call: Events[Unit], return: Events[Unit]) :=
  at(return, time(return) - time(call))

def at[A,B](trigger: Events[A], values: Events[B]) :=
  filter(values, time(trigger) == time(values))
```

Timing Constraints: Aggregate Statistics

Property

Measure the *maximal* runtime of the speed supervisor.

```
def maxSupervisorRuntime := max(supervisorRuntime)
```

Timing Constraints: Aggregate Statistics

Property

Measure the *maximal* runtime of the speed supervisor.

```
def maxSupervisorRuntime := max(supervisorRuntime)
```

TeSSLa standard library

```
# Return the event-wise maximum of two integer streams
def maximum(a: Events[Int], b: Events[Int]) :=
  if a > b then a else b

# Aggregate the maximal event value of a given stream
def max(x: Events[Int]) := {
  def result := merge(maximum(last(result, x), x), 0)
  result
}
```

Alternative Definition of Max Using Fold

```
# Aggregate all events' values of a given stream
def fold[T,R](f: (Events[R], Events[T]) => Events[R],
    stream: Events[T], init: R) := {
  def result: Events[R] :=
    merge(f(last(result, stream), stream), init)
  result
}

# Return the event-wise maximum of two integer streams
def maximum(a: Events[Int], b: Events[Int]) :=
  if a > b then a else b

# Aggregate the maximal event value of a given stream
def max(x: Events[Int]) := fold(maximum, x, 0)
```

Checking Event Ordering Constraints

Event Ordering Constraints: Order of Function Calls

Property

Every call to `getAllowedSpeed` leads to a call of `computeAllowedSpeedDistant` or `computeAllowedSpeedMagnet`.

```
# specify events of interest
def call := function_call("getAllowedSpeed")
def return := function_return("getAllowedSpeed")
def computeDistant :=
  function_call("computeAllowedSpeedDistant")
def computeMagnet :=
  function_call("computeAllowedSpeedMagnet")

# specify property over the events
def computation := on(return,
  time(computeDistant) > time(call) ||
  time(computeMagnet) > time(call))
```

Event Ordering Constraints: B Not Allowed After A

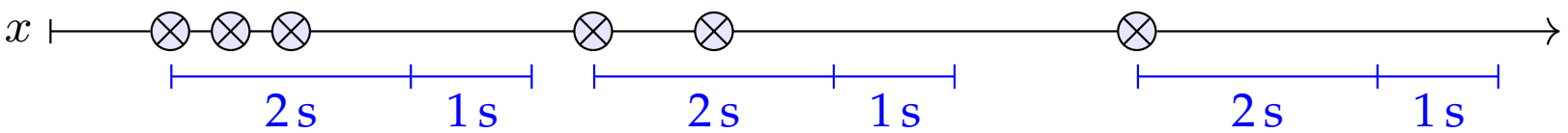
Property

Once the function `computeAllowedSpeedMagnet` was called for the first time we must be past the 500 Hz inductor and hence the function `computeAllowedSpeedDistant` must not be called any more.

```
# specify events of interest
def computeDistant :=
  function_call("computeAllowedSpeedDistant")
def computeMagnet :=
  function_call("computeAllowedSpeedMagnet")

# specify property over the events
def magnetAfterDistant :=
  time(computeMagnet) > time(computeDistant)
```

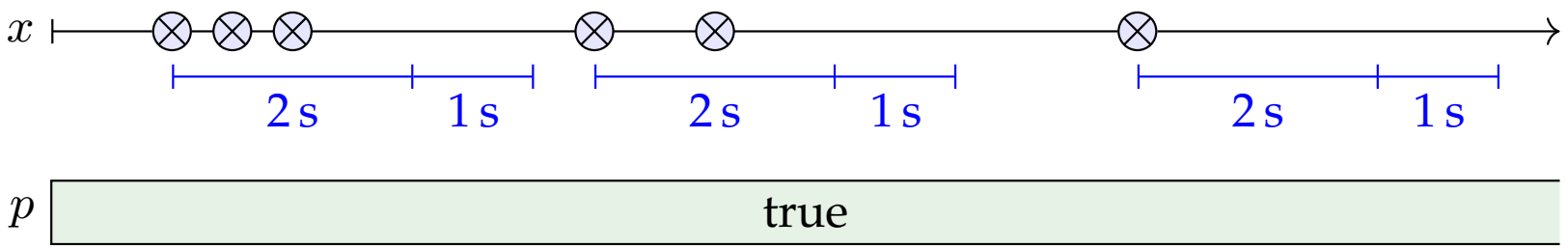
Burst Pattern

- Combination of timing and event ordering.
- AUTOSAR Timing Extension and EAST-ADL2 timing extension TADL2
- Pattern checks if events happen in bursts.



Burst Pattern

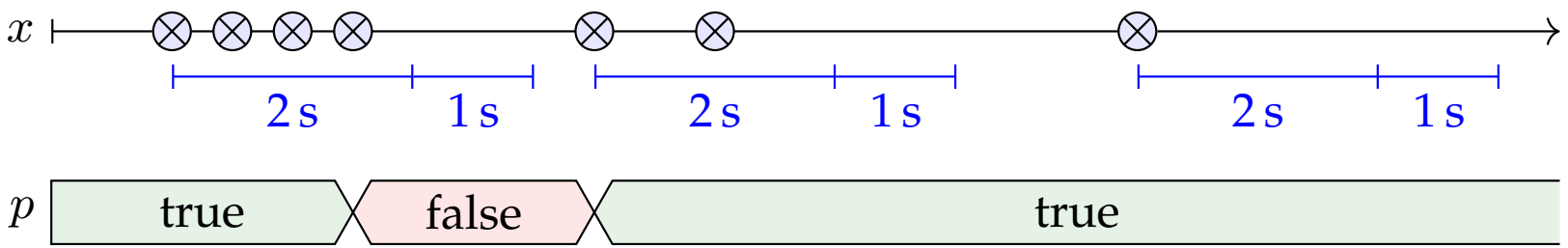
- Combination of timing and event ordering.
- AUTOSAR Timing Extension and EAST-ADL2 timing extension TADL2
- Pattern checks if events happen in bursts.




```
def p := bursts(x, burstLength = 2s,
                   waitingPeriod = 1s,
                   burstAmount = 3)
```

Burst Pattern

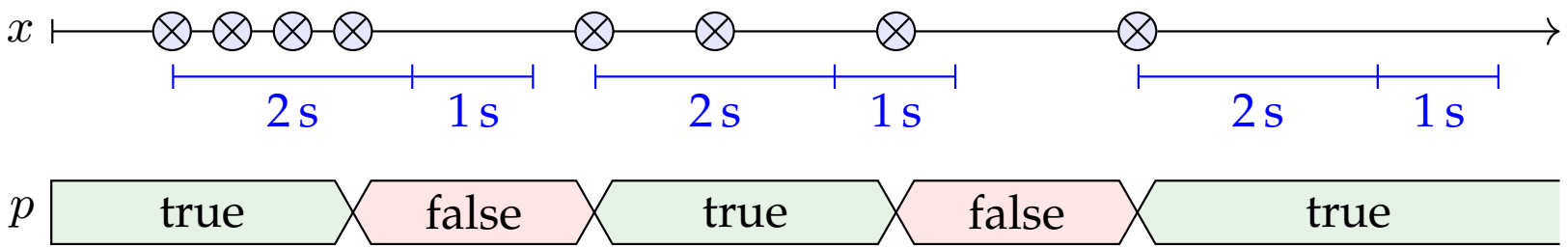
- Combination of timing and event ordering.
- AUTOSAR Timing Extension and EAST-ADL2 timing extension TADL2
- Pattern checks if events happen in bursts.




```
def p := bursts(x, burstLength = 2s,
                   waitingPeriod = 1s,
                   burstAmount = 3)
```

Burst Pattern

- Combination of timing and event ordering.
- AUTOSAR Timing Extension and EAST-ADL2 timing extension TADL2
- Pattern checks if events happen in bursts.




```
def p := bursts(x, burstLength = 2s,
                   waitingPeriod = 1s,
                   burstAmount = 3)
```

Checking Complex Events Patterns

Property

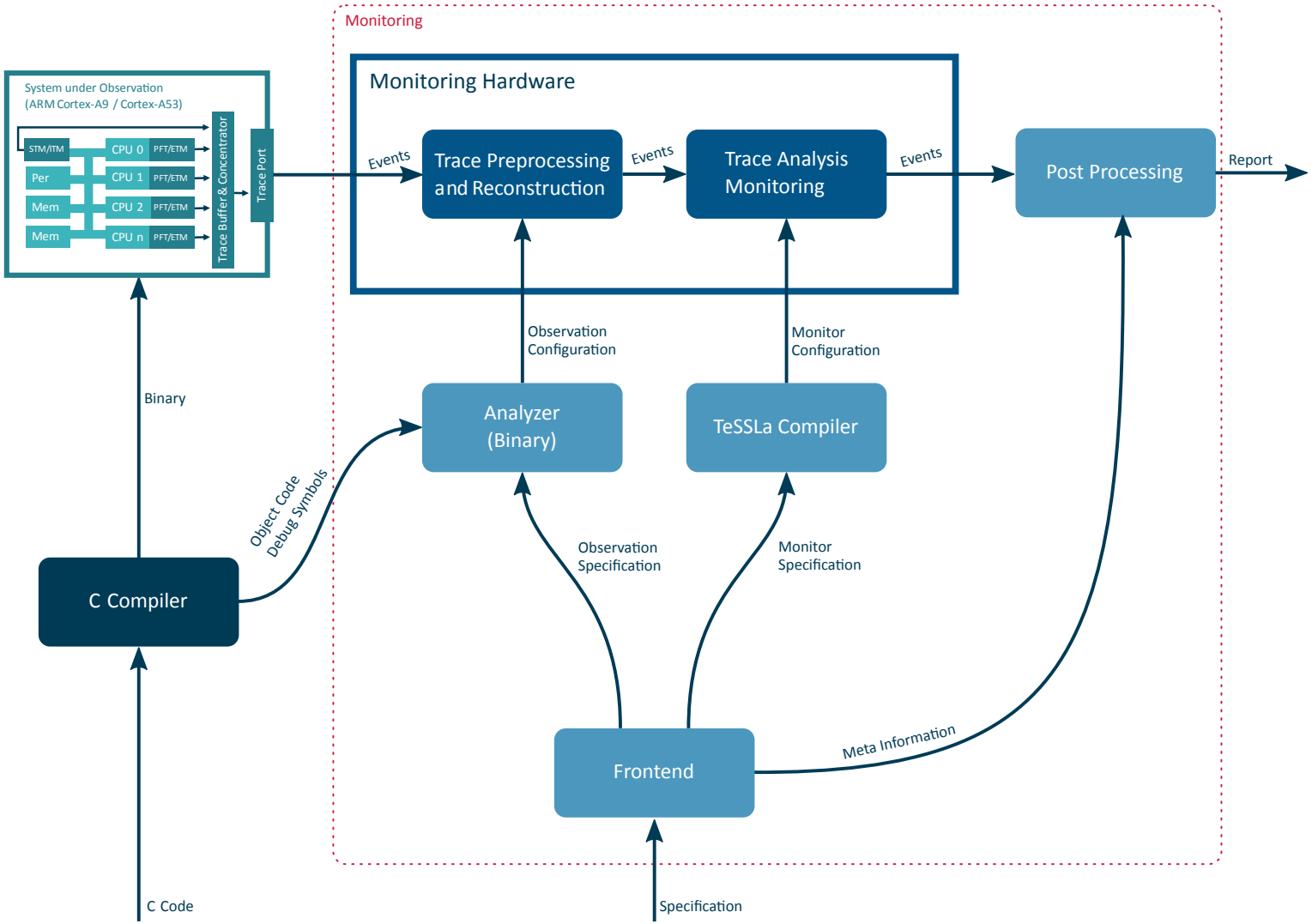
There are maximal 3 function calls in a period of 100 ms
with at least half a second of silence between the bursts.

```
# specify events of interest
def calls := merge(
  function_call("getAllowedSpeed"),
  function_call("computeAllowedSpeedDistant"),
  function_call("computeAllowedSpeedMagnet"))

# specify property over the events
def b := bursts(calls, burstLength = 100ms,
                       waitingPeriod = 500ms,
                       burstAmount = 3)
```

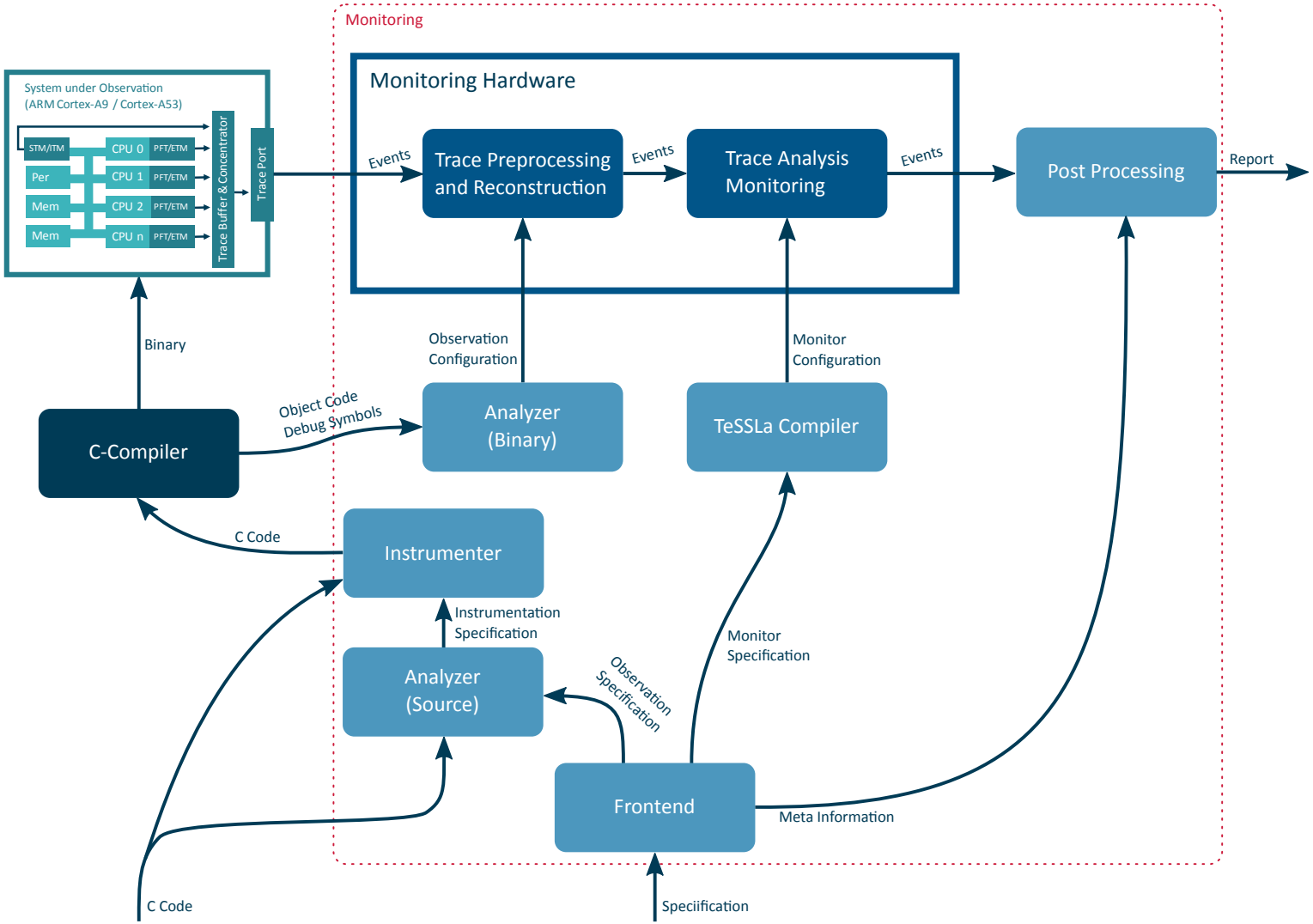
Such pattern specifications can spot abnormal behaviour which helps detecting interesting parts of traces of partially unknown systems.

Monitoring Data Values

Recall: Monitoring Program Flow Trace



Monitoring Data Values



Monitoring the Speed Supervisor

Property

23 s after the distant signal the allowed speed
computed by the supervisor must be equal to 85 km/h.

```
# Specify values we are interested in
def signal := function_argument("getAllowedSpeed", 1)
def allowed_speed := function_result("getAllowedSpeed")

# Changes where signal becomes DISTANT_SIGNAL_CAUTION
def caution := filter(changes(signal),
                      signal == DISTANT_SIGNAL_CAUTION)

def valid :=
  # Either we are not yet 23s after the distant signal
  time(allowed_speed) - time(caution) <= 23s
  # or the allowed speed must be below 85~km/h
  || allowed_speed * 36 <= 85 * 1000
```

Monitoring Data Value: How It Works

```
# Specify values we are interested in
def signal := function_argument("getAllowedSpeed", 1)
def allowed_speed := function_result("getAllowedSpeed")
```

Instrumented C Code

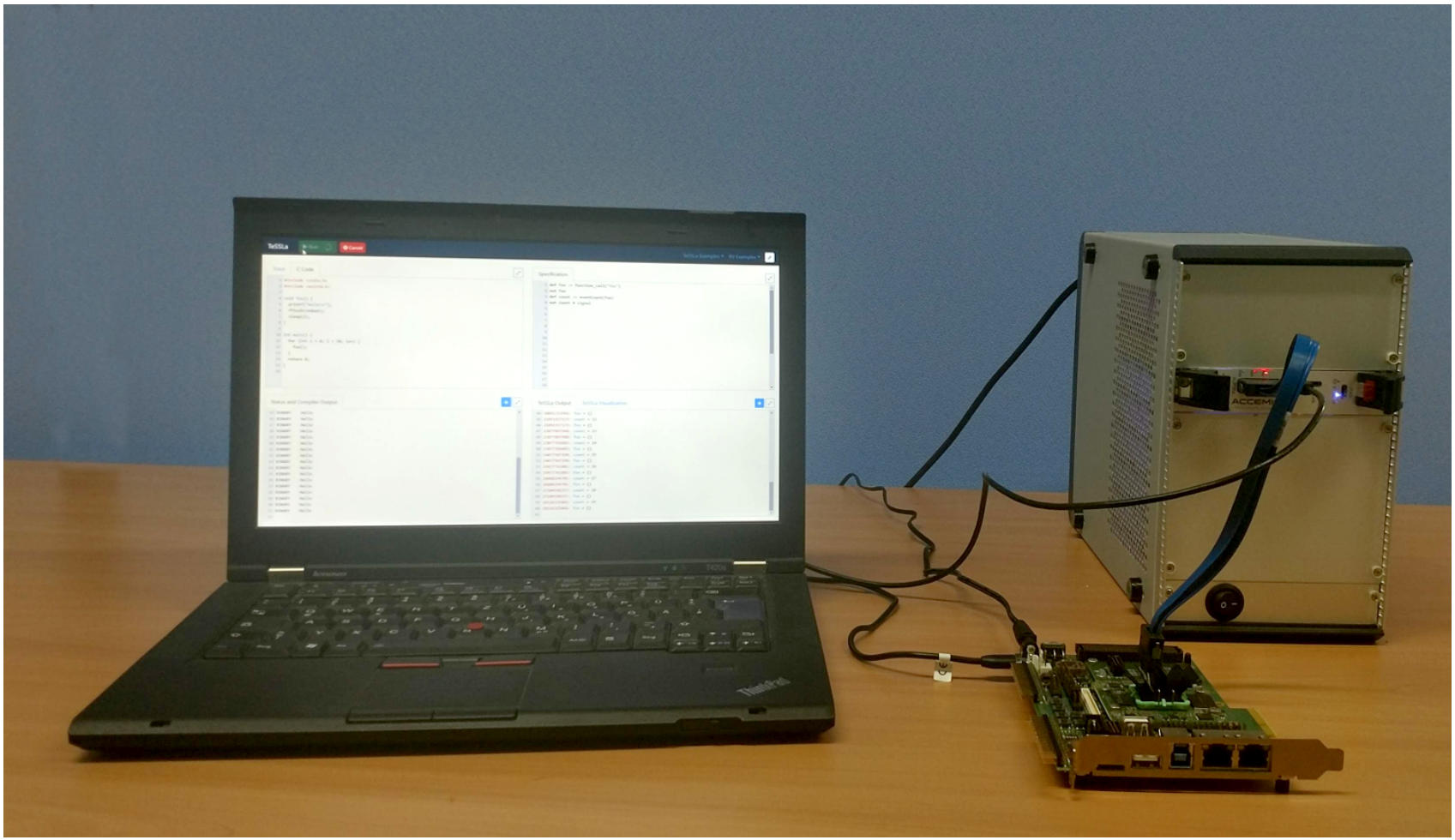
```
double getAllowedSpeed(
    int signal, ...) {
  tessla_debug(1, (int64_t)
    signal);
  double result = ...
  tessla_debug(2, (int64_t)
    (result * 1000));
  return result;
}
```

Updated Specification

```
in debug_slot: Events[Int]
in debug_value: Events[Int]
def signal :=
  filter(debug_value,
    debug_slot == 1)
def allowed_speed :=
  filter(debug_value,
    debug_slot == 2)
```

Practical Hardware Setup and Demonstration

Practical Hardware Setup and Demonstration



Conclusion

1. COEMS provides interactive debugging and continuous monitoring for embedded, hybrid and cyber-physical systems.
2. Continuous monitoring of processor traces provided by embedded tracing units (ETU) requires
   - online trace reconstruction in hardware and
   - monitoring in hardware.
3. Using TeSSLa we can check
   - event ordering constraints,
   - timing constraints and
   - complex event patterns like the burst pattern.
4. TeSSLa can be used to aggregate data and compute statistical data.
5. Monitoring data values is possible with lightweight instrumentation.
6. Specifications for data values can be naturally written in TeSSLa.