Stream Runtime Verification

Martin Leucker

Together with the whole TeSSLa Team
(Lukas Convent, Hannes Kallwies, Martin Sachenbacher, Malte Schmitz, Daniel Thoma, Volker Stolz, Cesar Sanchez, and many others)

UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING AND PROGRAMMING LANGUAGES

ACCEMIC TECHNOLOGIES

Western Norway University of Applied Sciences

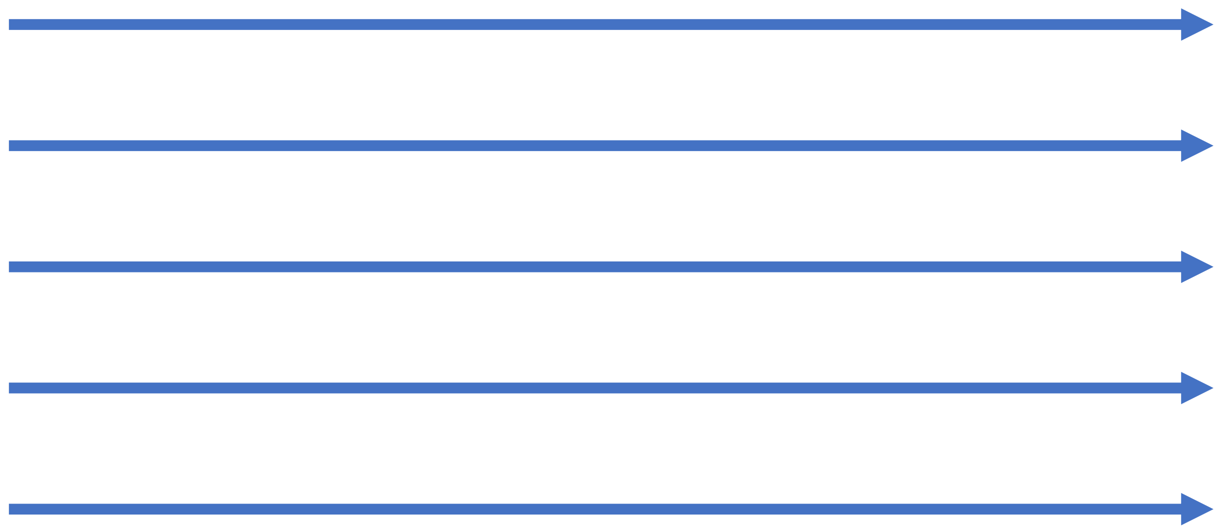
institute IMDEA software

Plan

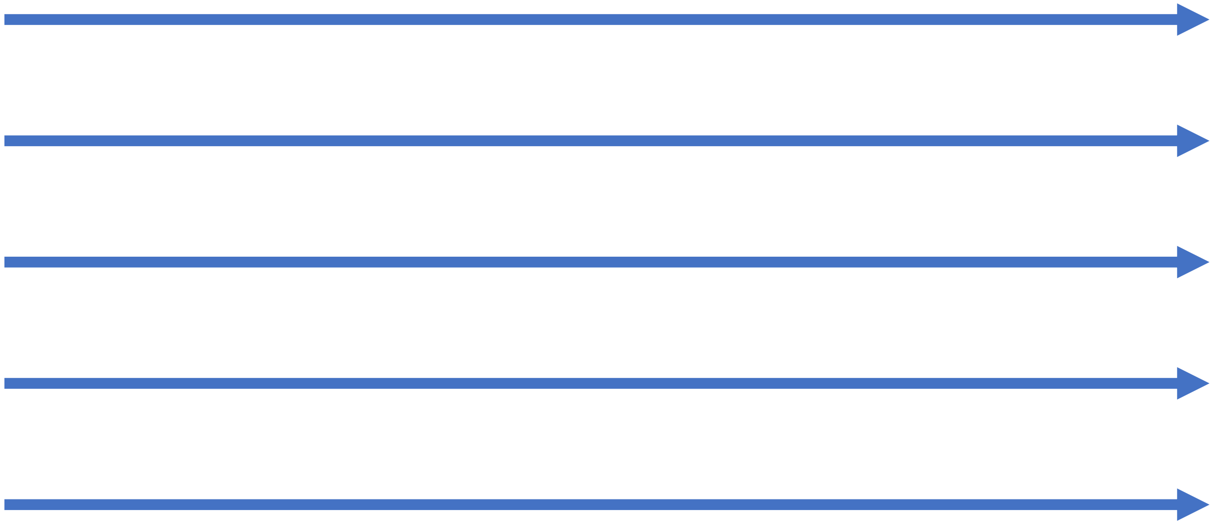
- Stream Runtime Verification
- LOLA
- TeSSLa
  - Language
  - Eco-System
- Control
  - Cyber-Physical Systems
  - Controllers
  - TeSSLa/ROS bridge

Motivation

Streams

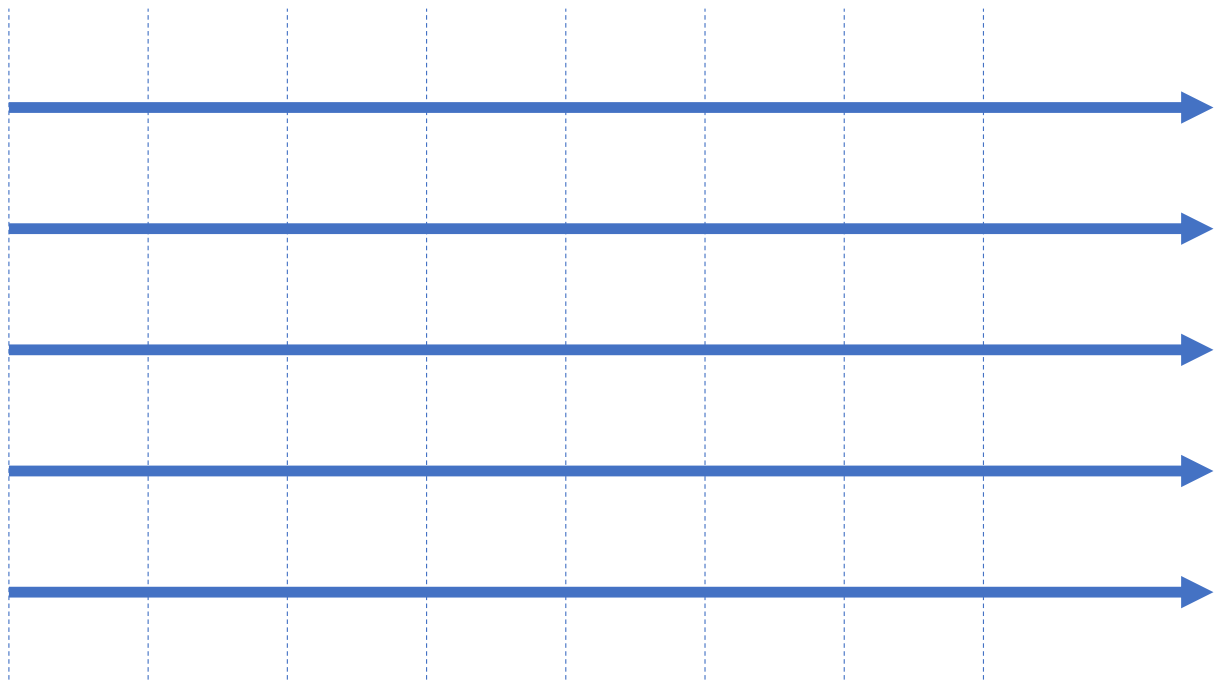


Streams

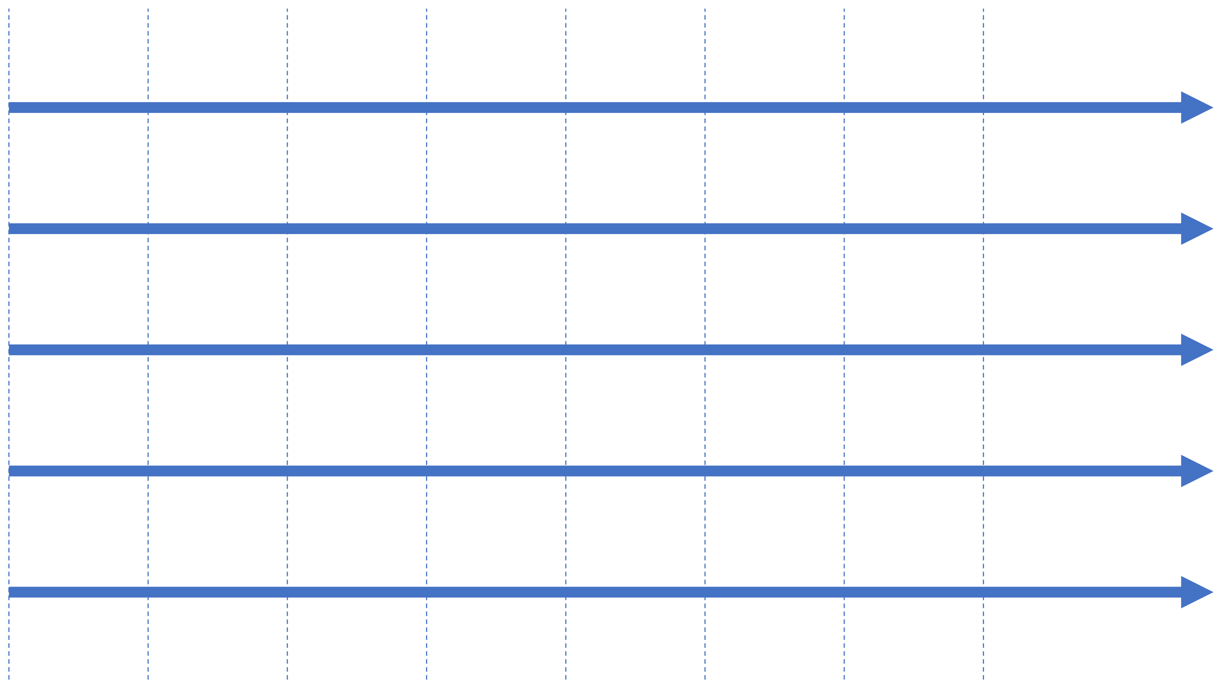


Concurrency/Distribution

Streams



Streams



Time? Synchrony/Ticks

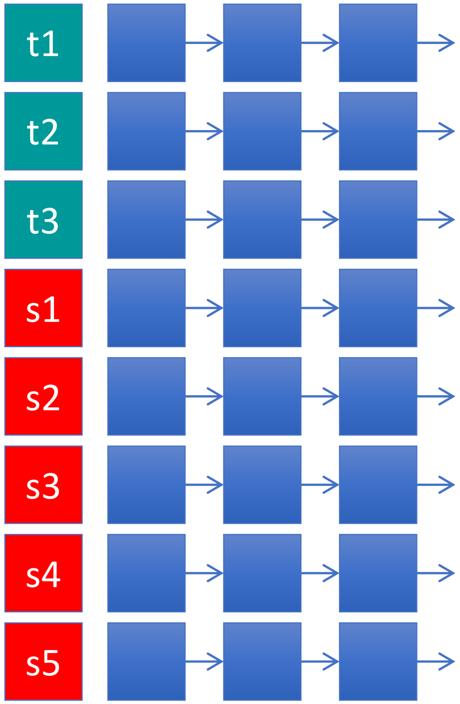
Equational specifications, data, time, concurrency

LOLA

[D'Angelo et al.]

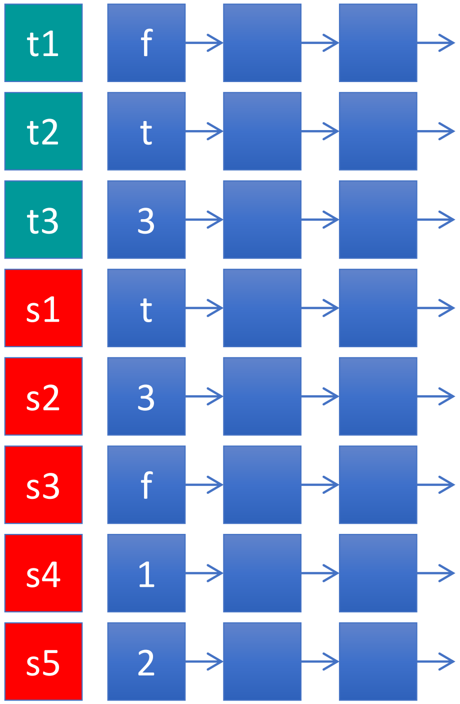
$$
\begin{aligned}
s_1 &= \mathbf{true}\\
s_2 &= t_3\\
s_3 &= t_1 \vee (t_3 \leq 1)\\
s_4 &= ((t_3)^2 + 7) \ mod \ 15\\
s_5 &= \mathtt{ite}(s_3, s_4, s_4 + 1)\\
s_6 &= \mathtt{ite}(t_1, t_3 \leq s_4, \neg s_3)\\
s_7 &= t_1[+1, \mathbf{false}]\\
s_8 &= t_1[-1, \mathbf{true}]\\
s_9 &= s_9[-1, 0] + (t_3 \ mod \ 2)\\
s_{10} &= t_2 \ \vee \ (t_1 \ \wedge \ s_{10}[1, \mathbf{true}])
\end{aligned}
$$

Example



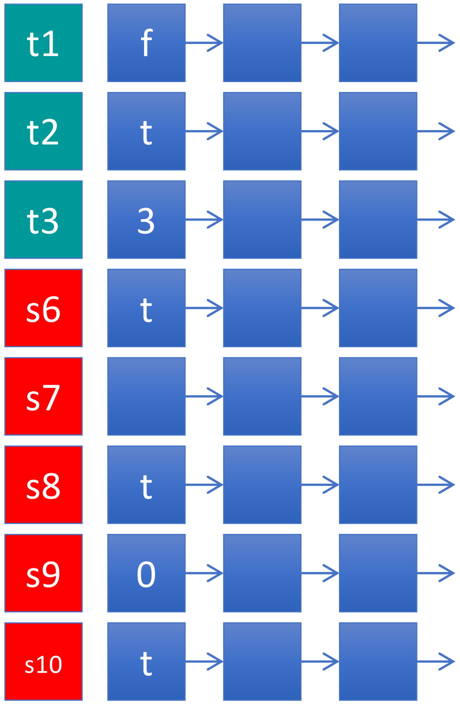
$$
\begin{aligned}
s_1 &= \mathbf{true} \\
s_2 &= t_3 \\
s_3 &= t_1 \vee (t_3 \leq 1) \\
s_4 &= ((t_3)^2 + 7) \ \mathit{mod} \ 15 \\
s_5 &= \mathtt{ite}(s_3, s_4, s_4 + 1) \\
s_6 &= \mathtt{ite}(t_1, t_3 \leq s_4, \neg s_3) \\
s_7 &= t_1[+1, \mathbf{false}] \\
s_8 &= t_1[-1, \mathbf{true}] \\
s_9 &= s_9[-1, 0] + (t_3 \ \mathit{mod} \ 2) \\
s_{10} &= t_2 \ \vee \ (t_1 \ \wedge \ s_{10}[1, \mathbf{true}])
\end{aligned}
$$

Example



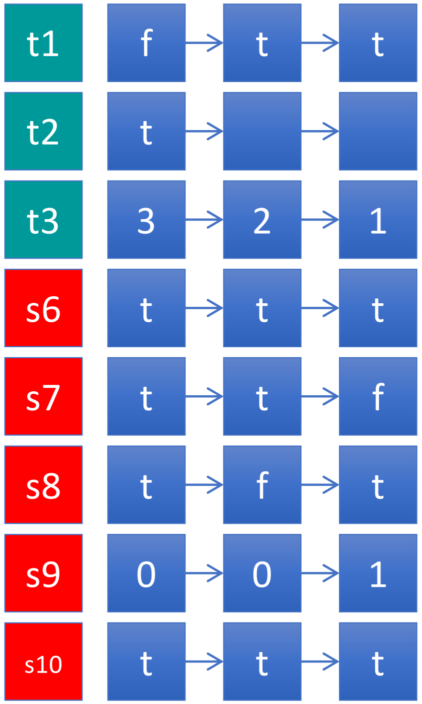
$$
\begin{aligned}
s_1 &= \mathbf{true} \\
s_2 &= t_3 \\
s_3 &= t_1 \lor (t_3 \leq 1) \\
s_4 &= ((t_3)^2 + 7)\ mod\ 15 \\
s_5 &= \texttt{ite}(s_3, s_4, s_4 + 1) \\
s_6 &= \texttt{ite}(t_1, t_3 \leq s_4, \neg s_3) \\
s_7 &= t_1[+1, \mathbf{false}] \\
s_8 &= t_1[-1, \mathbf{true}] \\
s_9 &= s_9[-1, 0] + (t_3\ mod\ 2) \\
s_{10} &= t_2 \ \lor\ (t_1\ \land\ s_{10}[1, \mathbf{true}])
\end{aligned}
$$

Example



$$
\begin{aligned}
s_1 &= \mathbf{true} \\
s_2 &= t_3 \\
s_3 &= t_1 \vee (t_3 \leq 1) \\
s_4 &= ((t_3)^2 + 7)\ \mathit{mod}\ 15 \\
s_5 &= \mathtt{ite}(s_3, s_4, s_4 + 1) \\
s_6 &= \mathtt{ite}(t_1, t_3 \leq s_4, \neg s_3) \\
s_7 &= t_1[+1, \mathbf{false}] \\
s_8 &= t_1[-1, \mathbf{true}] \\
s_9 &= s_9[-1, 0] + (t_3\ \mathit{mod}\ 2) \\
s_{10} &= t_2 \ \vee \ (t_1 \ \wedge \ s_{10}[1, \mathbf{true}])
\end{aligned}
$$

Example – Reaching the end of the Trace



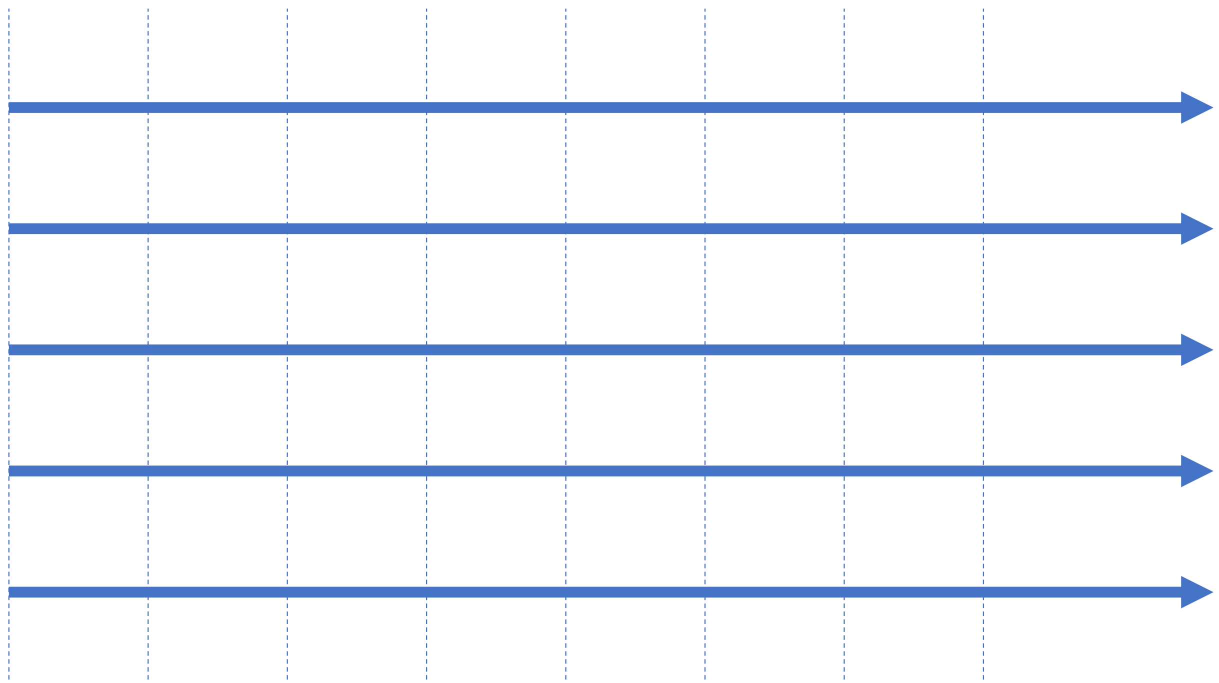
$$
\begin{aligned}
s_1 &= \mathbf{true} \\
s_2 &= t_3 \\
s_3 &= t_1 \vee (t_3 \leq 1) \\
s_4 &= ((t_3)^2 + 7)\ mod\ 15 \\
s_5 &= \mathtt{ite}(s_3, s_4, s_4 + 1) \\
s_6 &= \mathtt{ite}(t_1, t_3 \leq s_4, \neg s_3) \\
s_7 &= t_1[+1, \mathbf{false}] \\
s_8 &= t_1[-1, \mathbf{true}] \\
s_9 &= s_9[-1, 0] + (t_3\ mod\ 2) \\
s_{10} &= t_2\ \vee\ (t_1\ \wedge\ s_{10}[1, \mathbf{true}])
\end{aligned}
$$

Defining new Streams

Defining new Streams

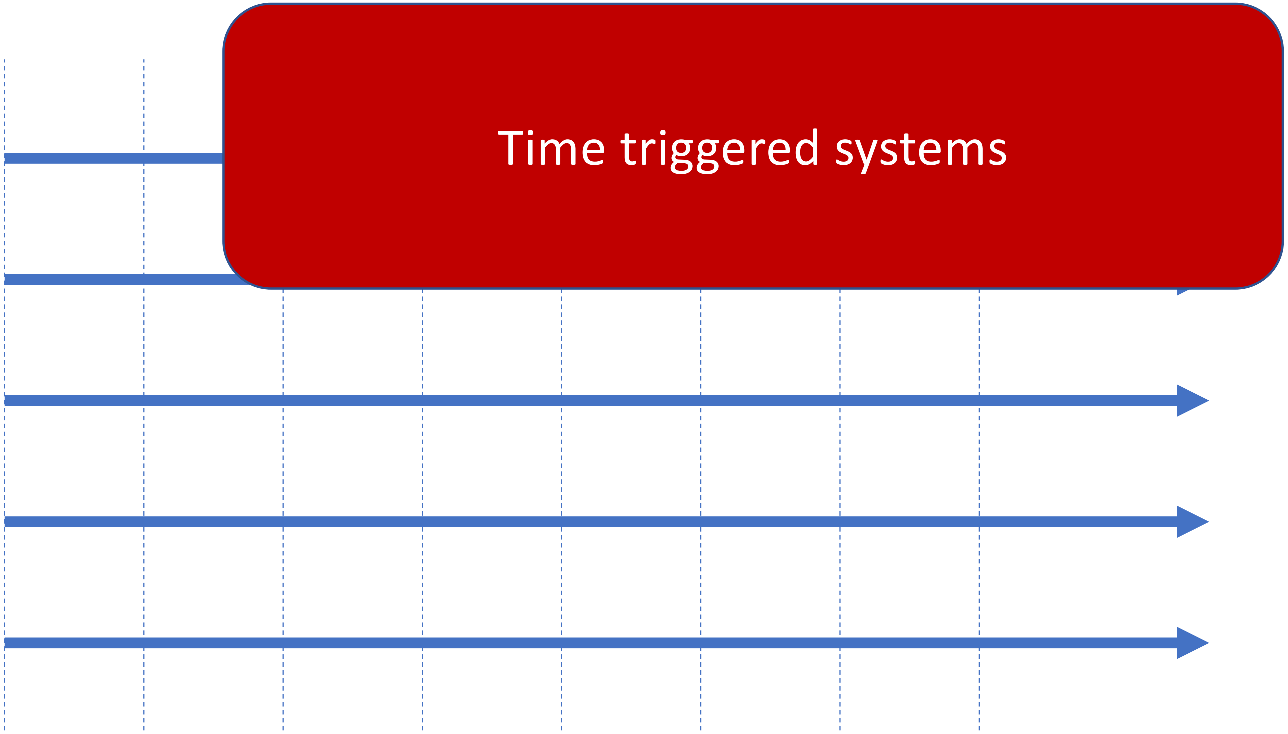
Runtime Verification as Stream Transformation

Streams



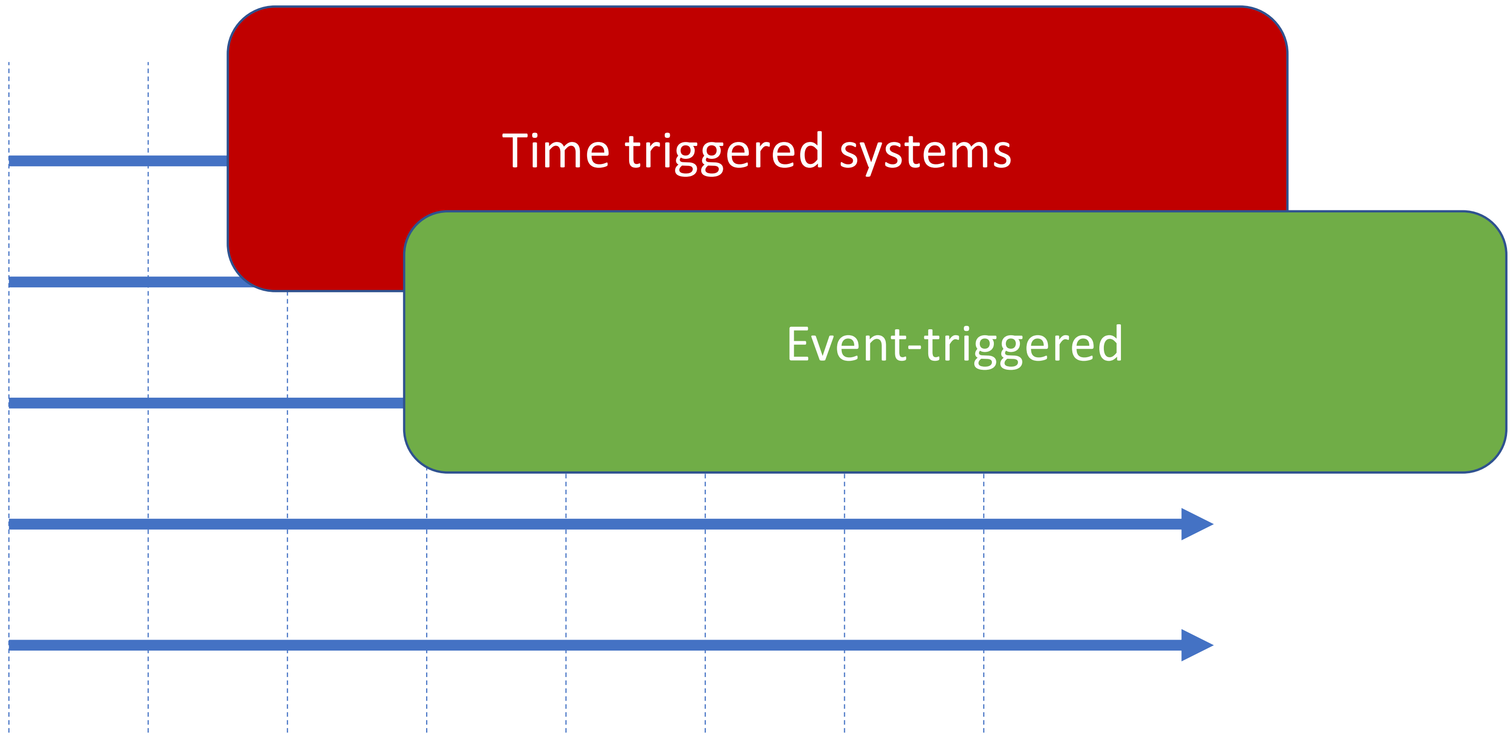
Time? Synchrony/Ticks

Streams



Time? Synchrony/Ticks

Streams

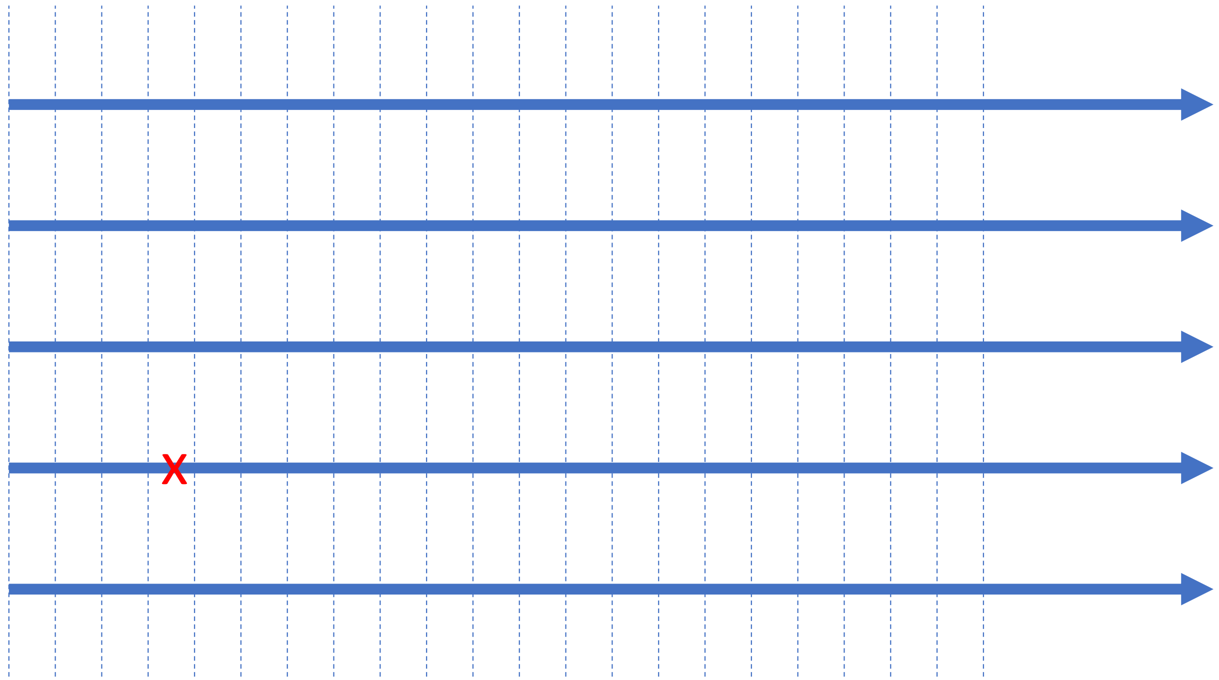


Time? Synchrony/Ticks

Tessla’s Streams

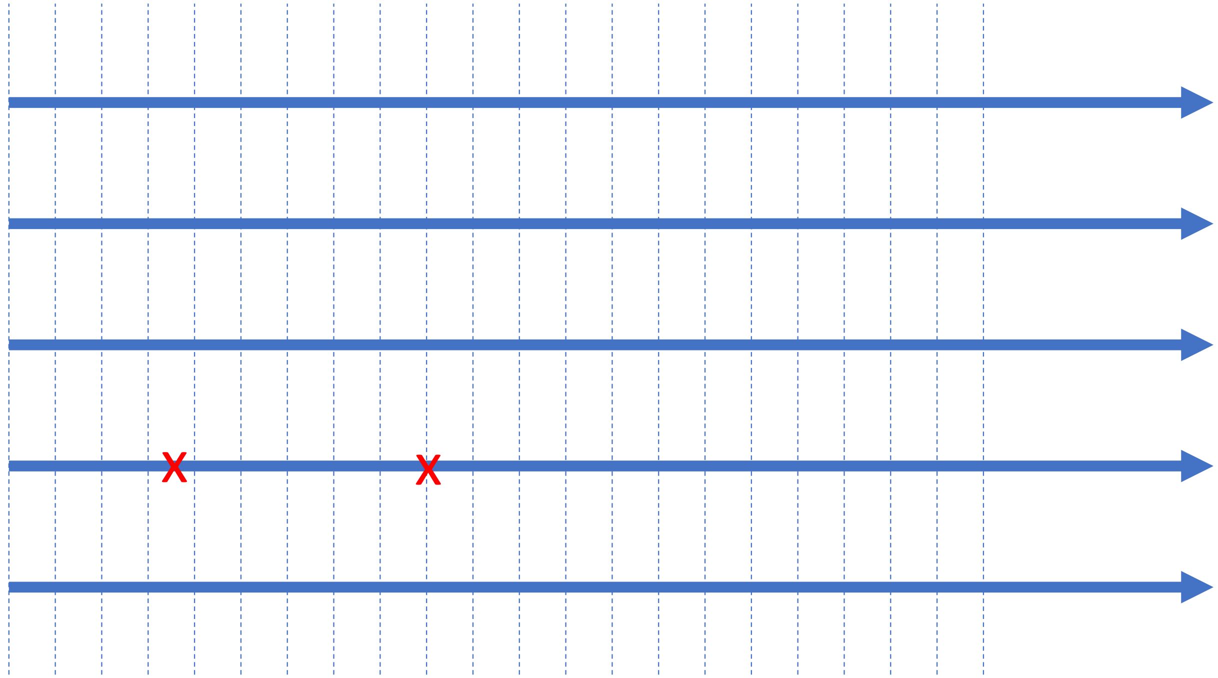
Time? Events

Tessla's Streams



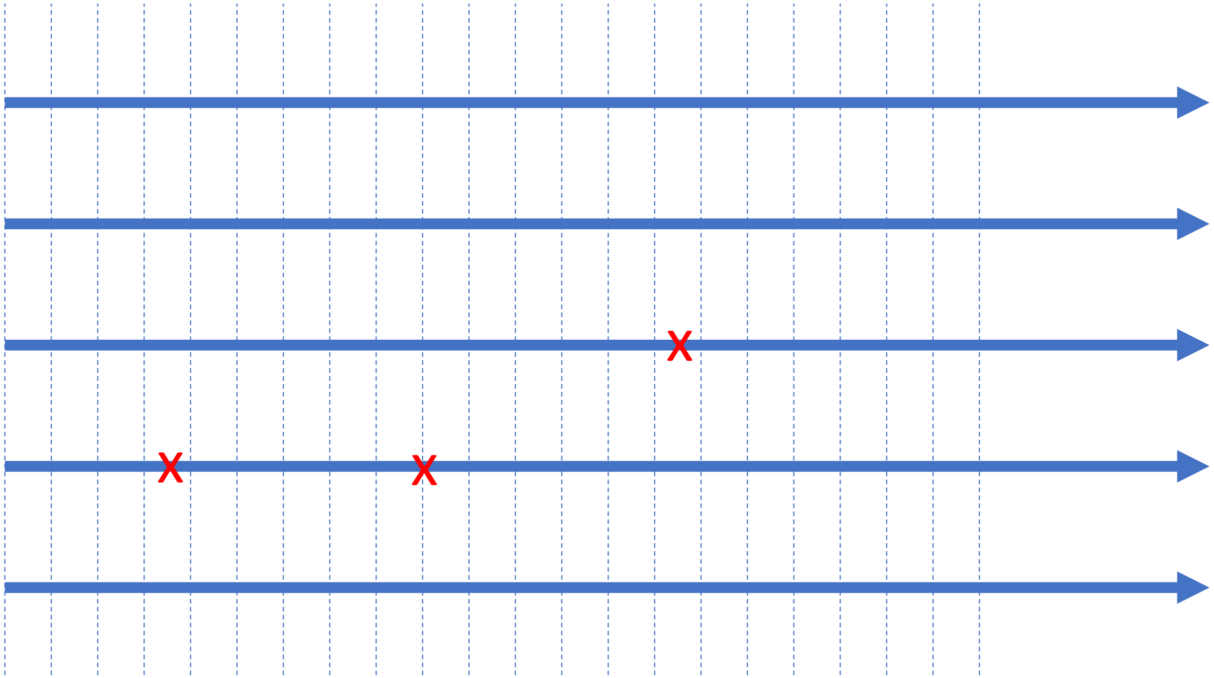
Time? Events

Tessla's Streams



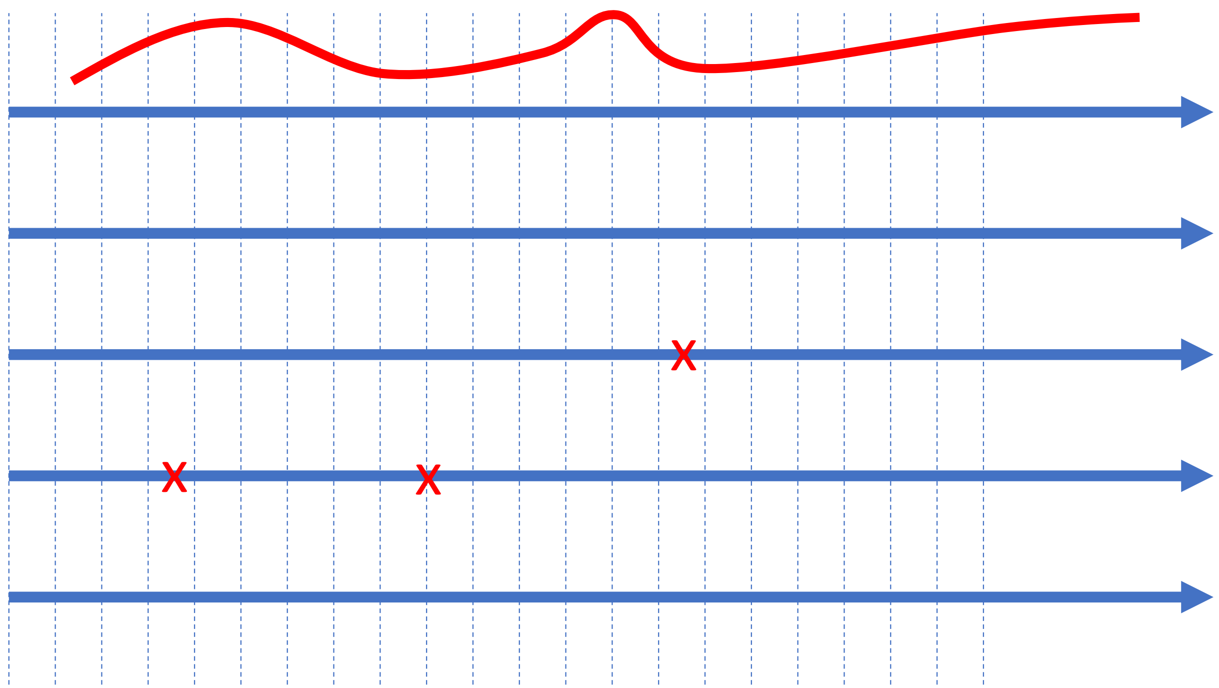
Time? Events

Tessla’s Streams



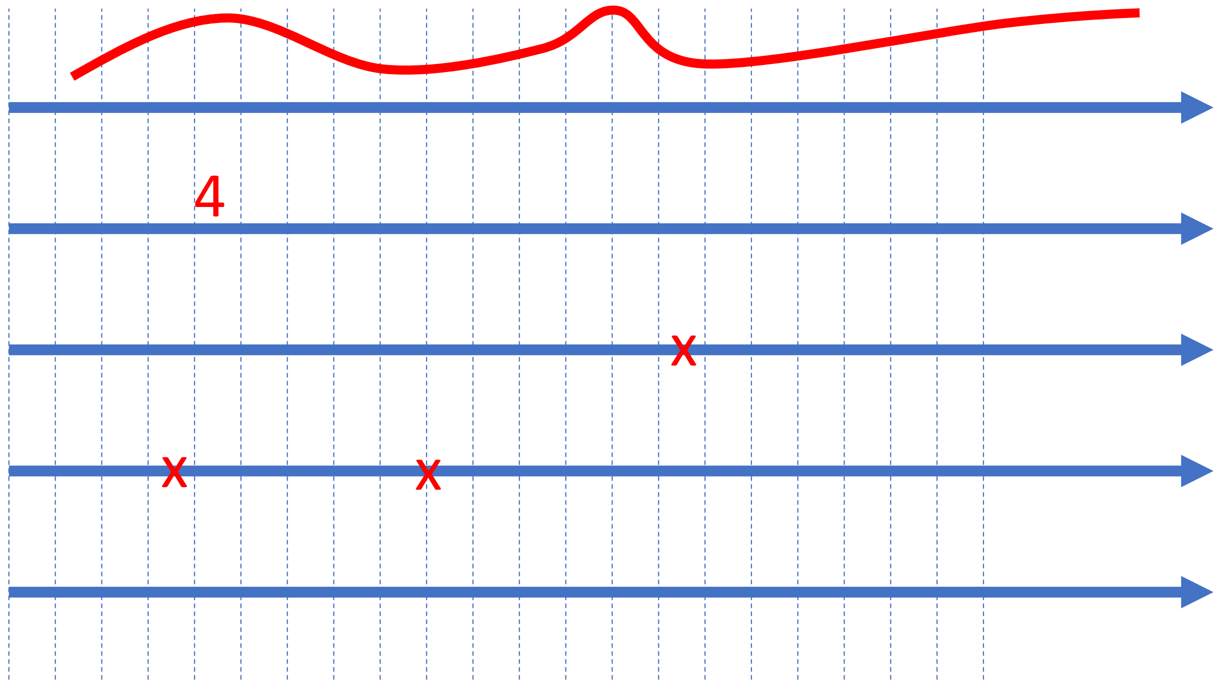
Time? Events

Tessla's Streams



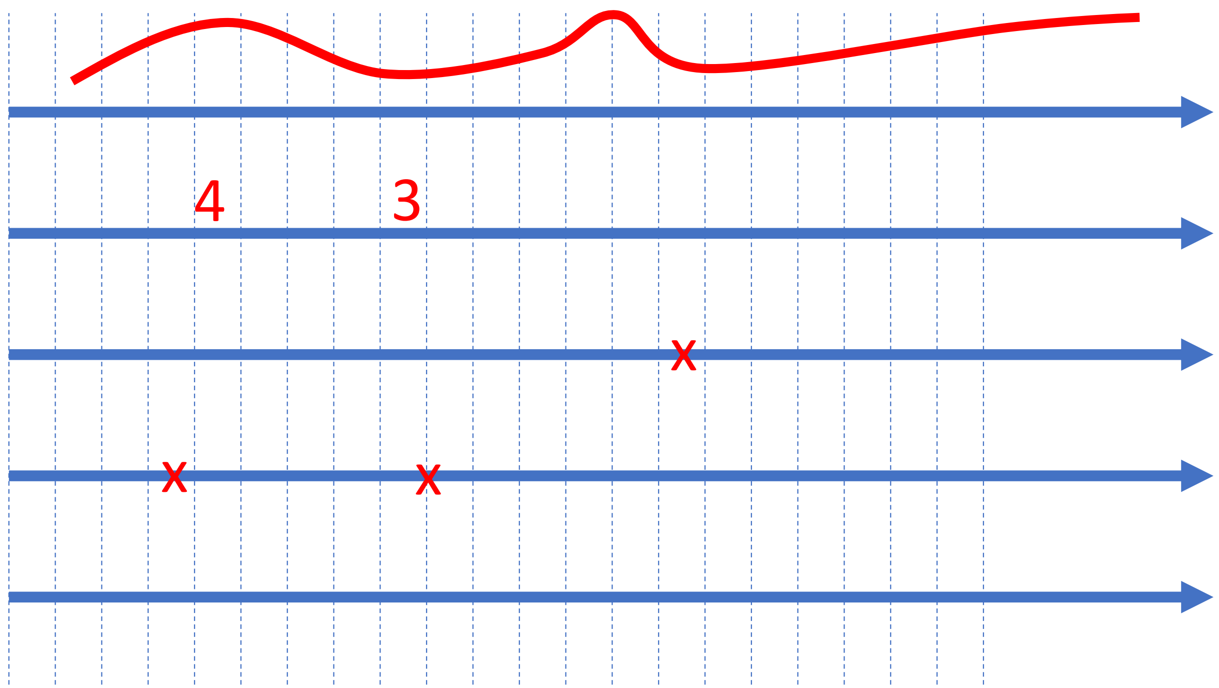
Time? Events

Tessla’s Streams



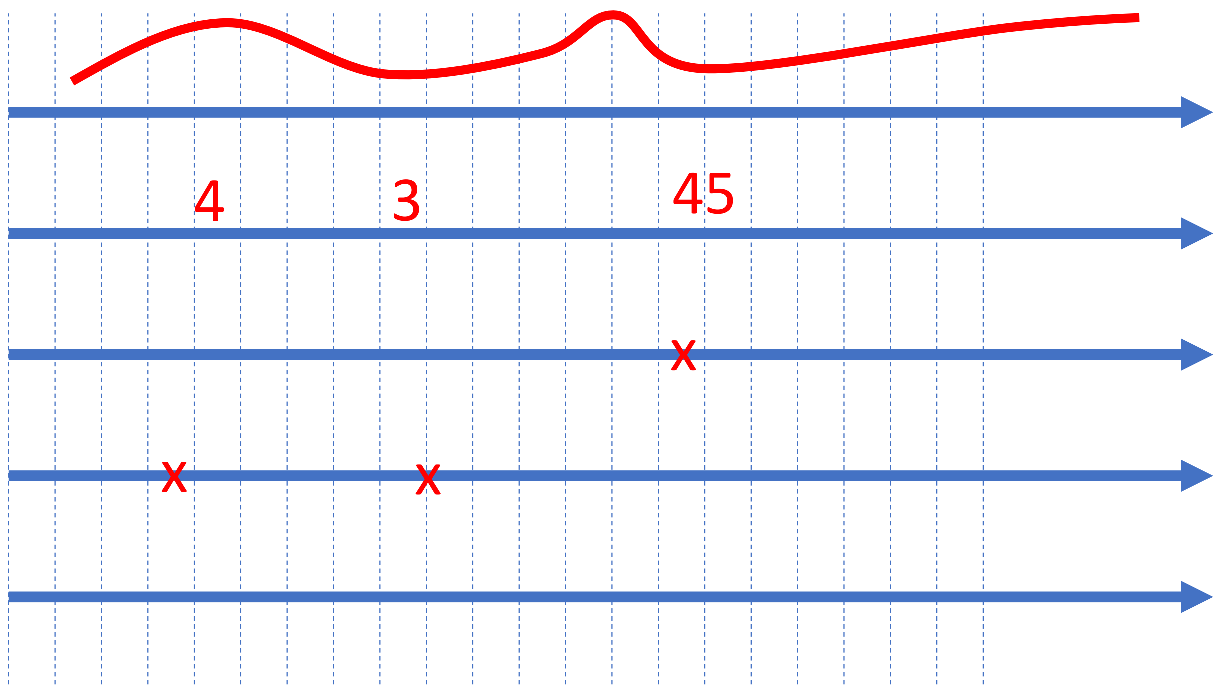
Time? Events

Tessla’s Streams



Time? Events

Tessla's Streams

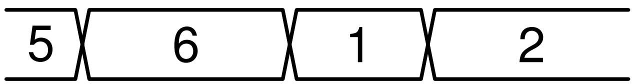


Time? Events

Streams of Programs - After Discretization

Values

e.g., of a program variable `x`



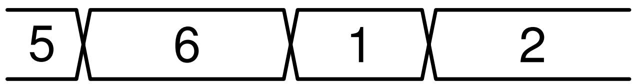
Program events

e.g., call to `my_func()`

Streams

Values

e.g., of a program variable `x`



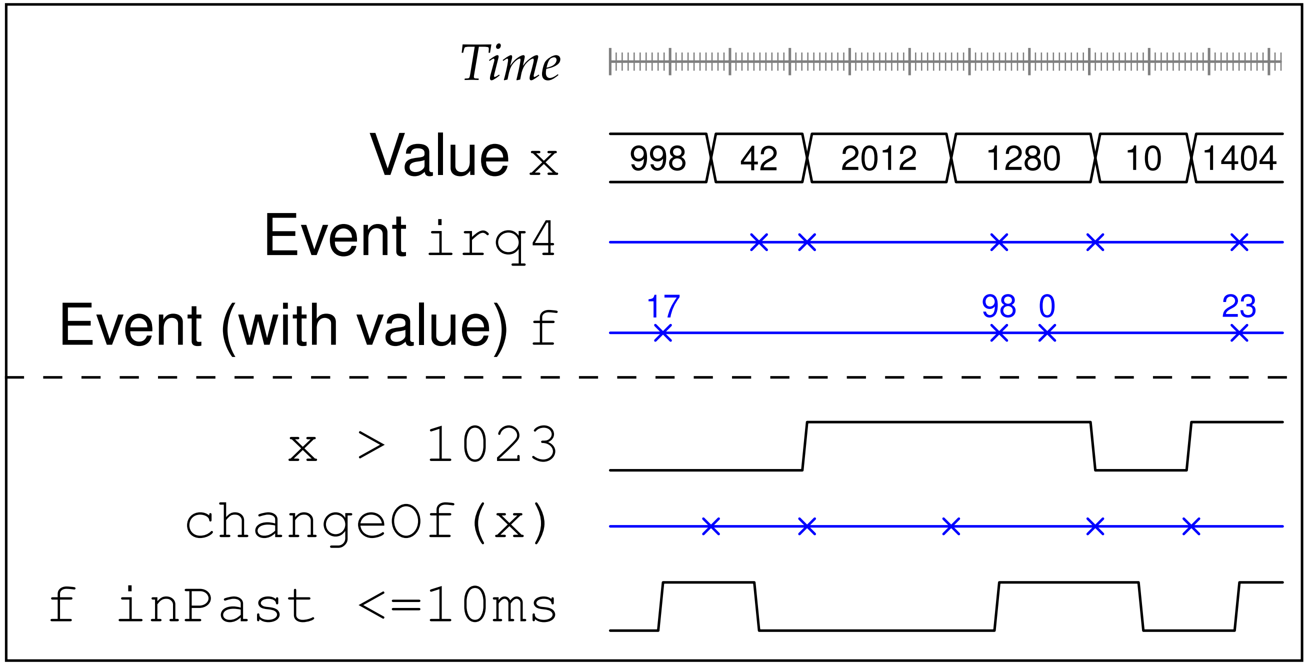
Program events

e.g., call to `my_func()`

Defining new Streams

Observations
(Input streams)

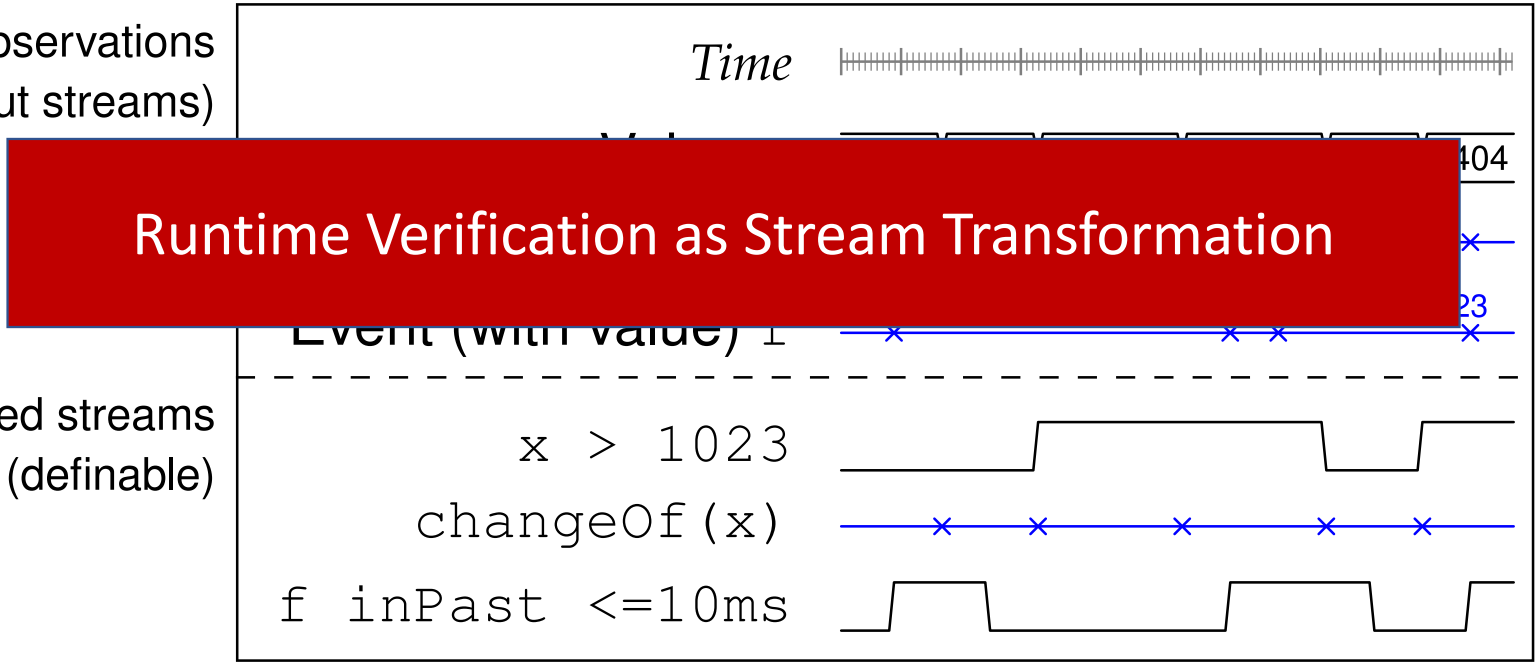
Derived streams
(definable)



Defining new Streams

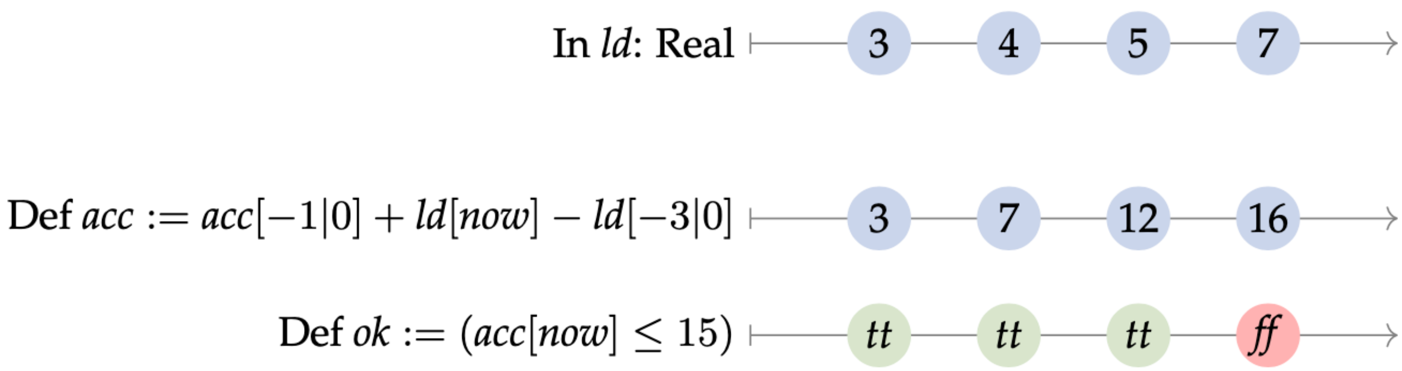
Observations
(Input streams)

Derived streams
(definable)



Runtime Verification with Uncertainties

Lola Example

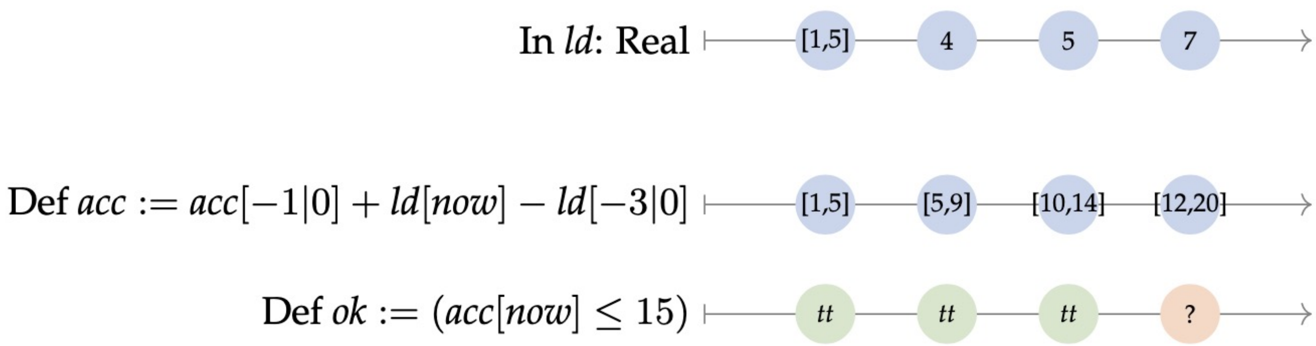


Three basic LOLA stream expressions:

- Constant streams
- Offset operators $s[o|c]$
  $\Rightarrow$: We restrict our self to the past fragment here (i.e. $o \leq 0$)
- Function applications

Using Abstract Domains

In 2019 Leucker et al. presented approach with intervals as abstract domain.

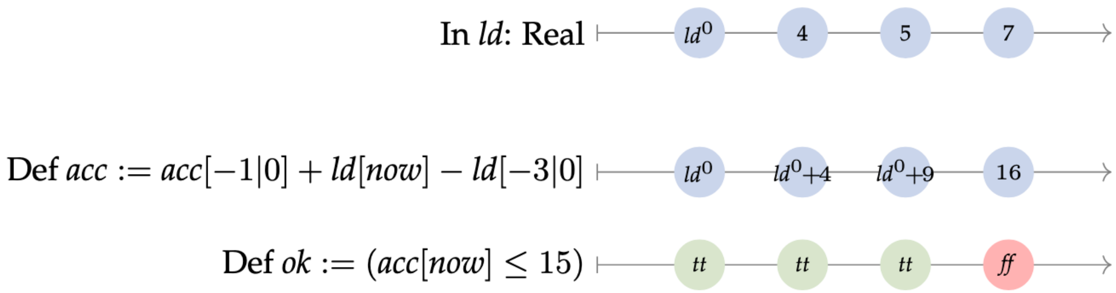


- Approach is sound, but not perfect.
- Handling of complex assumptions in general not possible.

Symbolic Evaluation

Idea: Use symbolic formulas for representation of unknown values and additional logical constraints (e.g. assumptions).

$\Rightarrow$ Use SMT solver for queries on possible values.



Additional constraints: $\{1 \leq ld^0 \leq 5\}$

- Approach in principle perfect.
- Assumptions can be added as propositions to constraint set.

Hannes Kallwies, Martin Leucker, César Sánchez: Symbolic Runtime Verification for Monitoring Under Uncertainties and Assumptions. ATVA 2022: 117-134

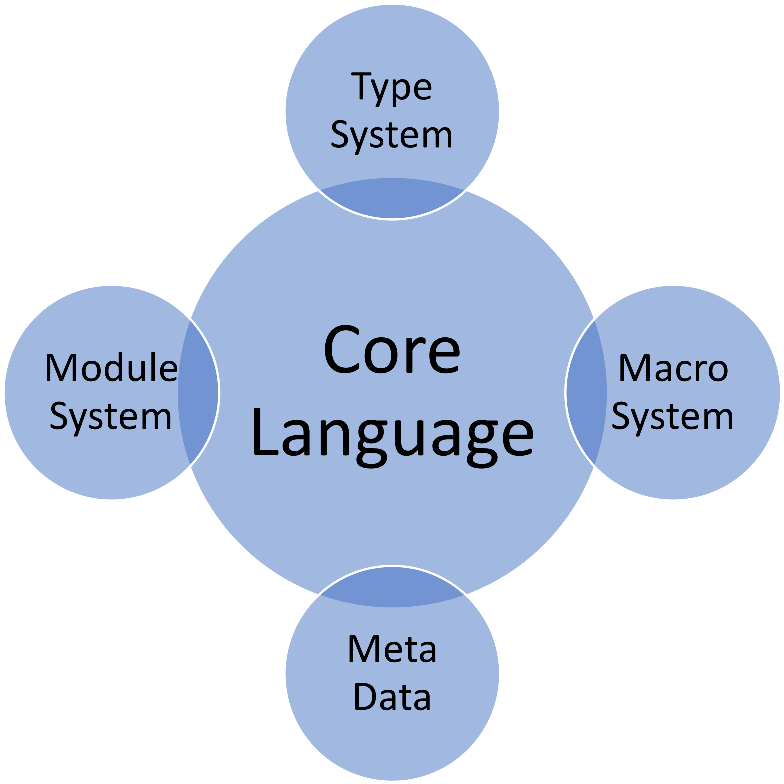
TeSSLa

TeSSLa

- **T**emporal
- **S**tream-based
- **S**pecification
- **La**nguage

- Specifying the (expected) behavior of a system's execution

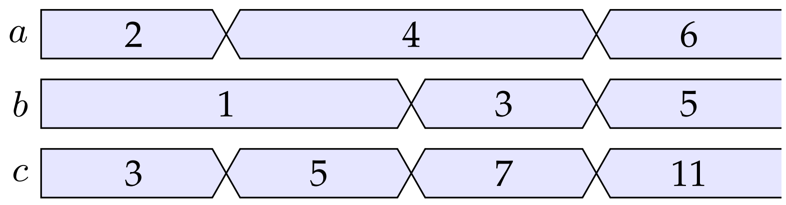
Language - Overview



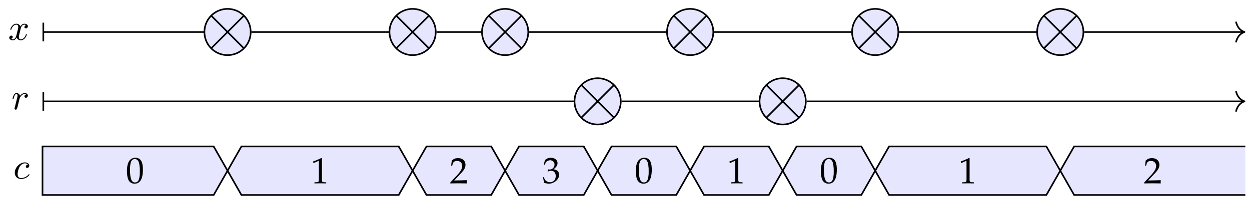
Design Goals – Core Language

- Declarative style: Specification rather than implementation
- Modularity: Allowing abstractions based on few primitives
  (6 operators: `unit, nil, lift, last, delay, time`)
- Time as first-class citizen
- Abstractions for both events and signals
- Recursion to reason about past
- Implementable with limited memory
  (For a restricted fragment)

TeSSLa by example



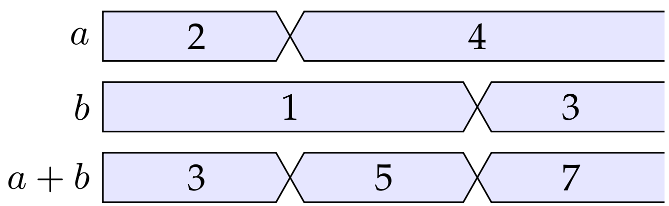

```
def c := a + b
```




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

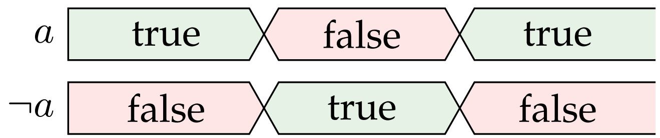
TeSSLa operators: Signal Lift (of Addition)

- *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.



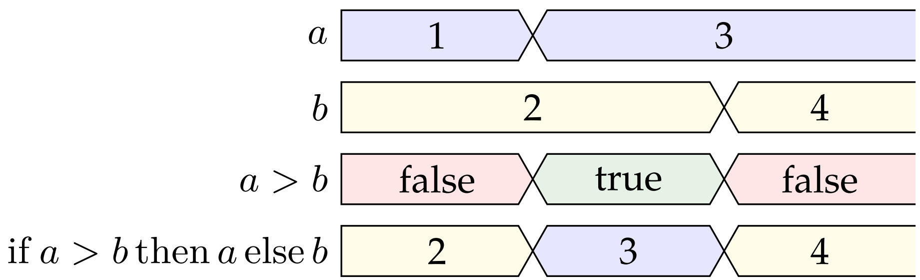
TeSSLa operators: Signal Lift (of Negation)

- *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.



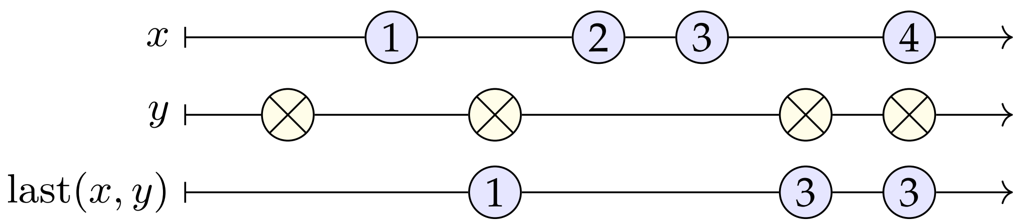
TeSSLa operators: Signal Lift (of If-Then-Else)

- *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

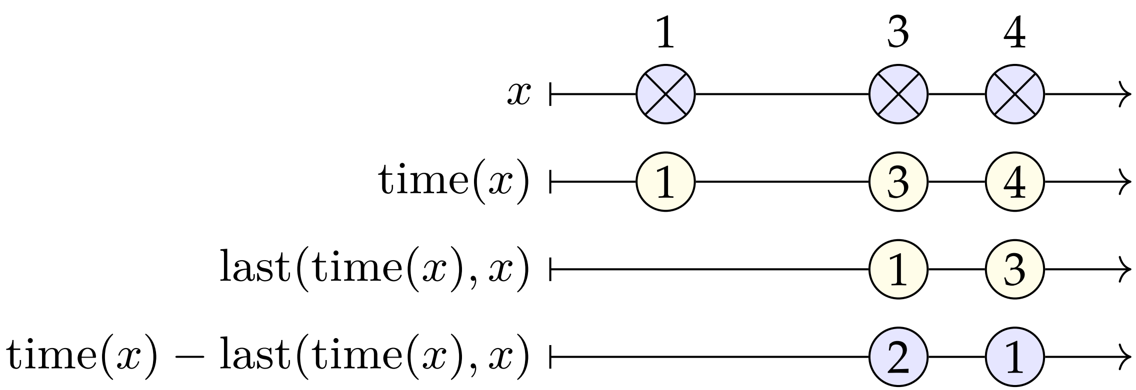
- Needed to define properties over sequences of events.
- Last allows to refer to the values of events on one stream that occurred strictly before the events on another stream



Read last(*x,y*) as last of *x* when event on *y*

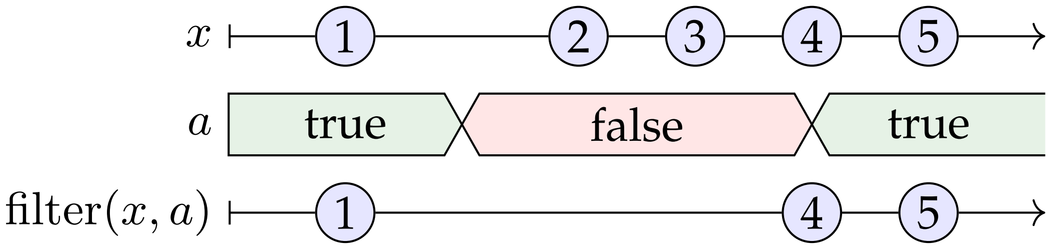
TeSSLa operators: Time

- 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.



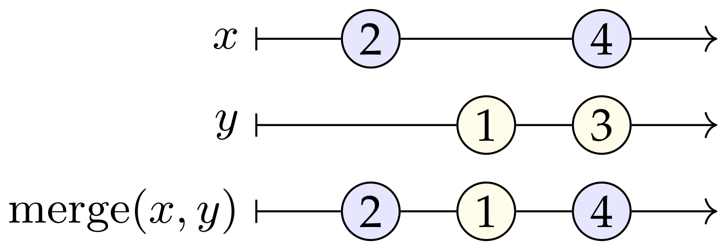
TeSSLa operators: Filter

- 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: Nil and Cons

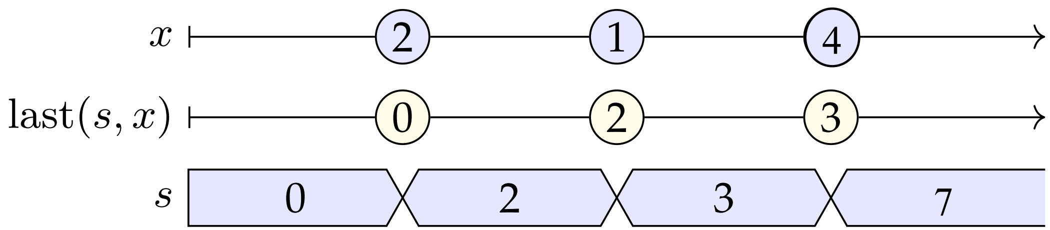
- The constant *nil* for the empty stream
- The operator *const* converting a value to a stream starting with that value at timestamp 0.

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)
```

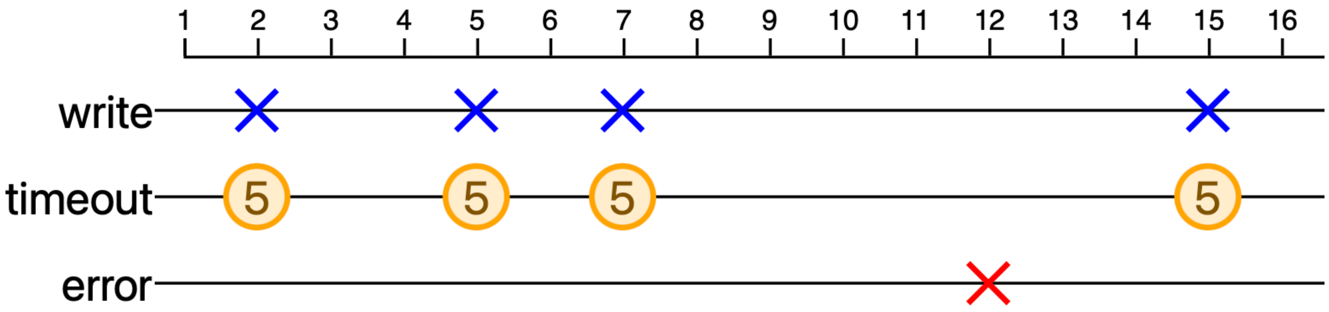
Create Events

in write: Events[Unit]

def **timeout** := const(5, write)

def **error** := delay(timeout, write)

out error



Data types in TeSSLa

- TeSSLa strongly typed, generic types
- TeSSLa agnostically wrt 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

Macros in TeSSLa

- Few primitive operators
- Readable specifications via Macros
- TeSSLa Standard Library for common useful stuff
- Domain specific libraries for application areas/domains (anticipated)
  - Timex/Autosar library
  - PastLTL
  - Petri nets (under development)

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
}
```

Modules in TeSSLa

- Sets of Macros can be grouped to modules/libraries
- TeSSLa Standard Library for common useful stuff
- Domain specific libraries for application areas/domains (anticipated)
  - Timex/Autosar library
  - PastLTL
  - Petri nets (under development)

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
}
```

Meta Data / Annotations

- TeSSLa allows annotations similar like @interface in Java
- Several categories for annotations
  - Documentation
  - Correspondence to Source Code (C-Code)
  - Graphical presentation of streams / dashboard support
  - Directives for Example Generator
  - Directives for bridging to frameworks (ROS)

TeSSLa by Example

```
# Inputs
@InstFunctionCall("read_brake_sensor")
in read_brake_sensor: Events[Unit]
@InstFunctionCall("activate_brakes")
in activate_brakes: Events[Unit]

# Trace Processing
def latency = measureLatency(read_brake_sensor,
                             activate_brakes)

def error = latency > 4ms
def high = filter(latency, error) - 4ms
def is_critical = count(high) > 10
def critical = filter(high, is_critical)

# Output
@VisDots out high
@VisEvents out critical

# Macro
def measureLatency[A, B](a: Events[A],
                         b: Events[B]) =
  time(b) - last(time(a), b)
```

Input decl. & annotations

Monitoring property

Output decl. & annotations

Macro definitions

TeSSLa by Example

```
# Inputs
@InstFunctionCall("read_brake_sensor")
in read_brake_sensor: Events[Unit]
@InstFunctionCall("activate_brakes")
in activate_brakes: Events[Unit]

# Trace Processing
def latency = measureLatency(read_brake_sensor,
                             activate_brakes)
def error = latency > 4ms
def high = filter(latency, error) - 4ms
def is_critical = count(high) > 10
def critical = filter(high, is_critical)

# Output
@VisDots out high
@VisEvents out critical

# Macro
def measureLatency[A, B](a: Events[A],
                         b: Events[B]) =
  time(b) - last(time(a), b)
```

Input decl. & annotations

Monitoring property

Output decl. & annotations

Macro definitions

TeSSLa by Example

```
# Inputs
@InstFunctionCall("read_brake_sensor")
in read_brake_sensor: Events[Unit]
@InstFunctionCall("activate_brakes")
in activate_brakes: Events[Unit]

# Trace Processing
def latency = measureLatency(read_brake_sensor,
                             activate_brakes)
def error = latency > 4ms
def high = filter(latency, error) - 4ms
def is_critical = count(high) > 10
def critical = filter(high, is_critical)

# Output
@VisDots out high
@VisEvents out critical

# Macro
def measureLatency[A, B](a: Events[A],
                         b: Events[B]) =
  time(b) - last(time(a), b)
```

Input decl. & annotations

Monitoring property

Output decl. & annotations

Macro definitions

TeSSLa by Example

```
# Inputs
@InstFunctionCall("read_brake_sensor")
in read_brake_sensor: Events[Unit]
@InstFunctionCall("activate_brakes")
in activate_brakes: Events[Unit]

# Trace Processing
def latency = measureLatency(read_brake_sensor,
                             activate_brakes)
def error = latency > 4ms
def high = filter(latency, error) - 4ms
def is_critical = count(high) > 10
def critical = filter(high, is_critical)

# Output
@VisDots out high
@VisEvents out critical

# Macro
def measureLatency[A, B](a: Events[A],
                         b: Events[B]) =
  time(b) - last(time(a), b)
```

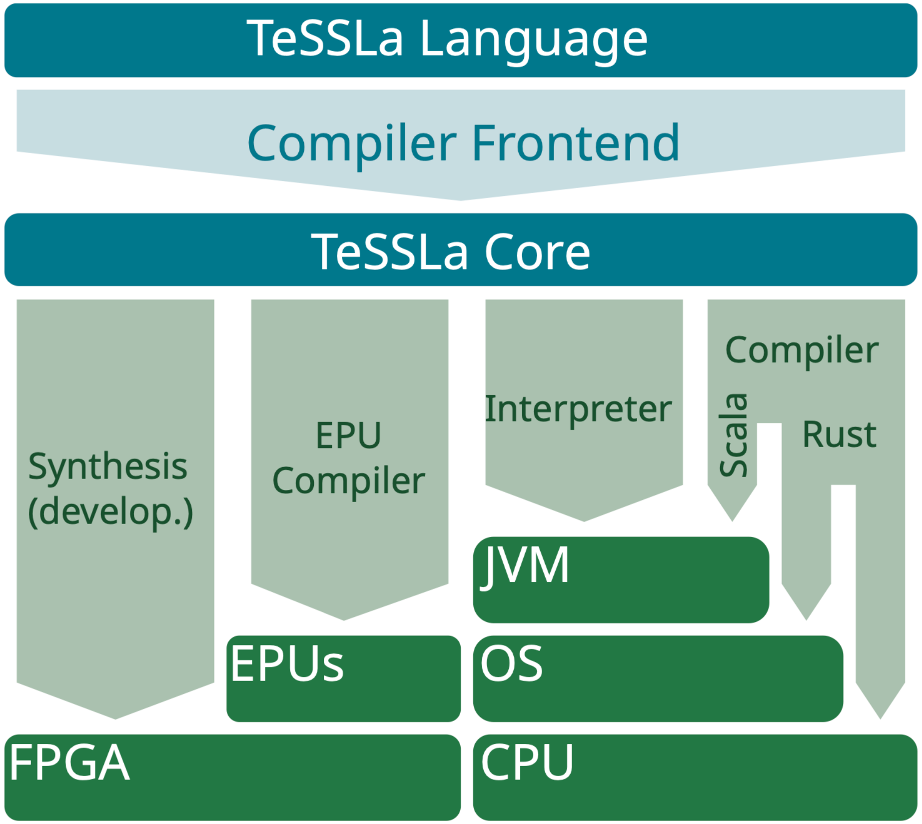
Input decl. & annotations

Monitoring property

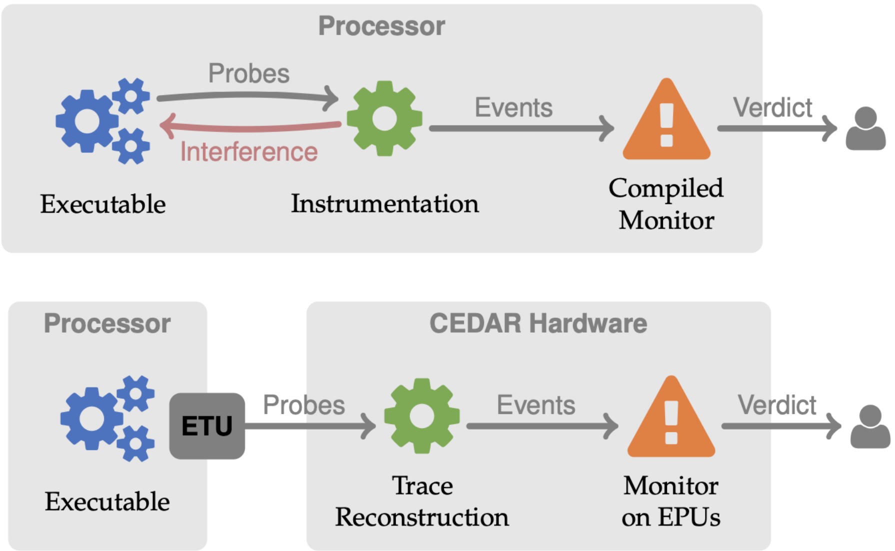
Output decl. & annotations

Macro definitions

TeSSLa compilers

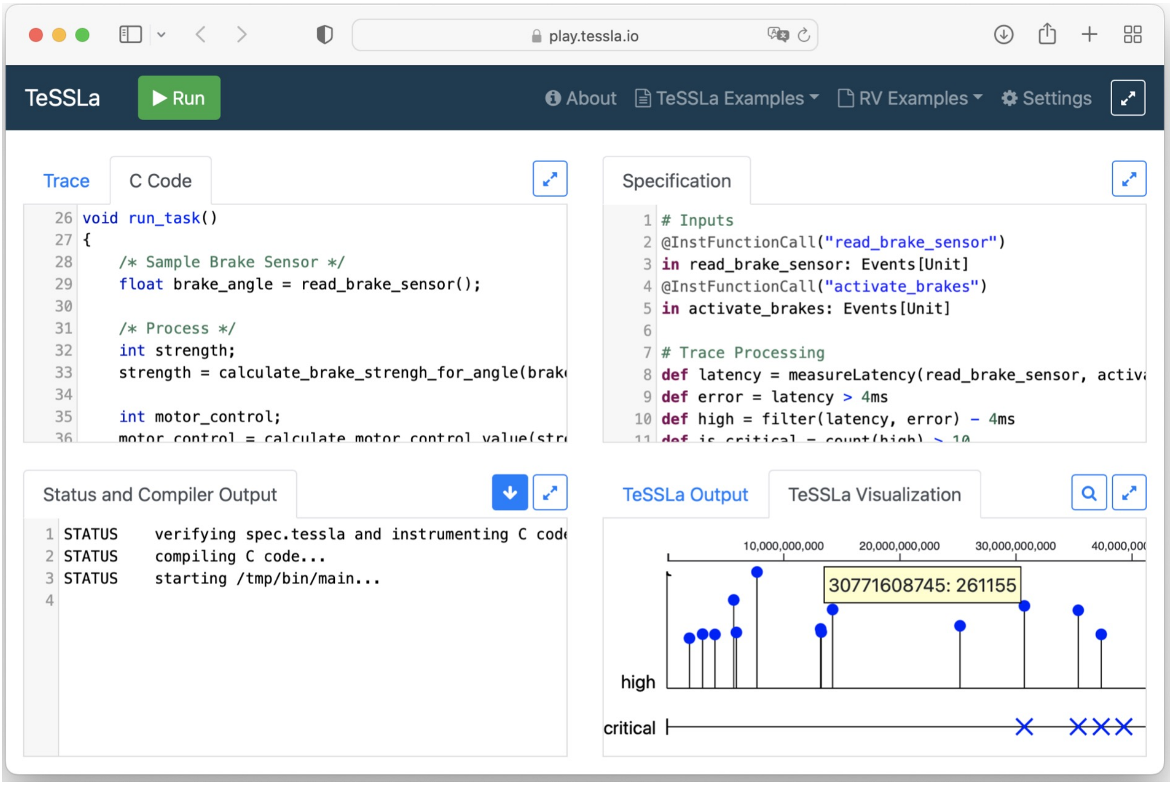


Observation/Instrumentation



- Instrumenter for C code integrated in compiler
- Accemic's CEDARtools for non-intrusive hardware monitoring
- Connection to other instrumentation tools via generic annotation system

Supporting Web IDE



Supporting Online Documentation

constIf

`constIf[T](value: T, condition: Events[Bool]): Events[T]`

Produce an event with the given value every time that the condition is met

Usage example:

```
in condition: Events[Bool]
def result = constIf(42, condition)
out result
```

Trace example:

Source

```
def constIf[T](value: T, condition: Events[Bool]): Events[T] =
  filter(const(value, condition), condition)
```

count

`count[T](x: Events[T]): Events[Int]`

Count the number of events on `x`. Provides for every input event an output event whose value is the number of events seen so far. See `resetcount` for a counting macro with an external reset.

TeSSLa Ecosystem

- User Libraries
  - Macro system allows definition of application-specific libraries
  - E.g. AUTOSAR Timex, Past LTL libraries...
- Tutorials
  - Extensive tutorials about the usage of the TeSSLa language and tools.
- Open-Source availability
  - Free availability of most parts of the tool chain.
  - Community-driven project.

TeSSLa for professional usage

- Clear definition of license
- Separation of
  - Language,
  - Compilers, and
  - Tools
- Language specification
  - TeSSLa and TeSSLa Core
- Reference Compiler (Interpreter)

Resources

- TeSSLa Website:

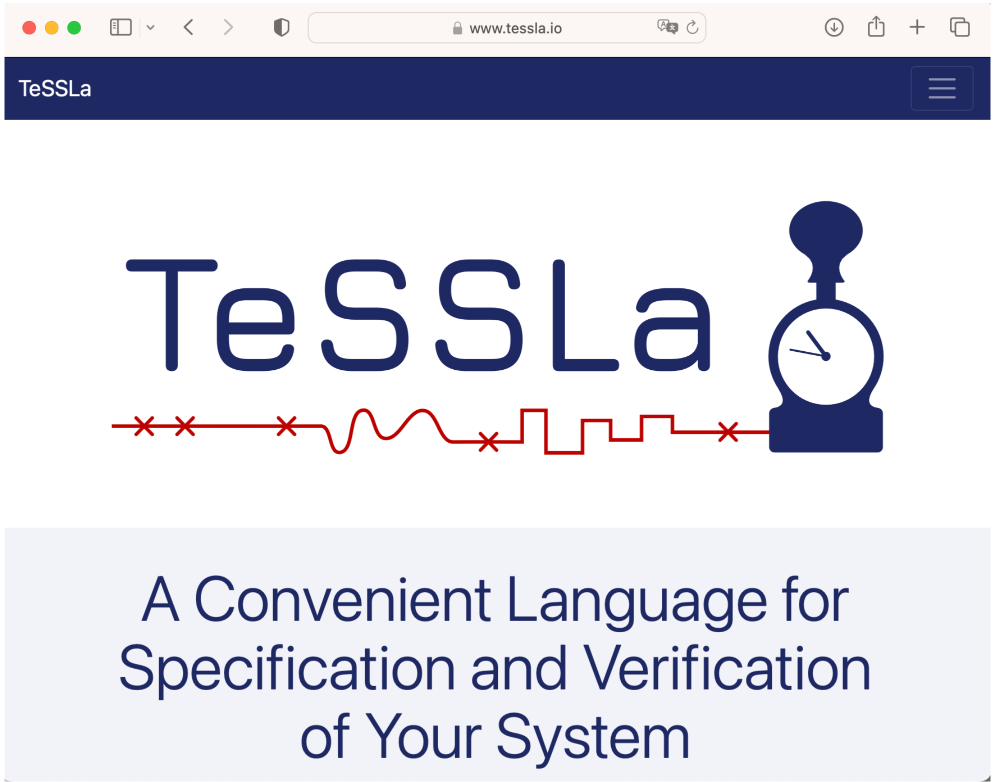
  https://www.tessla.io/
- TeSSLa Playground:

  https://play.tessla.io/
- TeSSLa Sourcecode:

  https://git.tessla.io/
- Contact:

  info@tessla.io

tessla.io


www.tessla.io
TeSSLa
TeSSLa
A Convenient Language for Specification and Verification of Your System

TeSSLa Installation and First-Steps

Installation – TeSSLa Bundle

- contains a compiler, interpreter and other useful tools for executing TeSSLa specifications
- written in Scala and available as a single JAR archive.
- The TeSSLa bundle is licensed under Apache 2.0 license.
  Run java -jar tessla.jar -h for information on the usage of the TeSSLa command line tool.

https://git.tessla.io/tessla/tessla/builds/artifacts/master/raw/target/scala-3.2.2/tessla-assembly-2.0.0.jar?job=deploy

Logging Library

- For instrumenting C-Code

https://www.tessla.io/logging.zip

TeSSLa libraries

Futher libraries

https://www.tessla.io/usrLibs/overview/

TeSSLa TADL2/Autosar-Timex Library
TeSSLa version: 1.2.2-1.2.4, **License:** Apache 2.0
TeSSLa library with functions for checking TADL 2 Constraints
Download, Documentation Project Page

TeSSLa TDDL Library
TeSSLa version: 1.2.2-1.2.4, **License:** Apache 2.0
TeSSLa implementation of Timed Dyadic Deontic Logic
Download, Documentation Additional information

TeSSLa/ROS Bridge
TeSSLa version: 1.2.2+, **License:** Apache 2.0
TeSSLa library and tooling for integration with the Robot Operating System (ROS)
Download, Documentation Project Page

TeSSLa Telegraf Connector
TeSSLa version: 1.2.3+, **License:** Apache 2.0
TeSSLa library and tooling for integration with the Telegraf framework
Download, Documentation Project Page

A simple specification

- specification.tessla

```
in x: Events[Int]
in y: Events[Int]

def diff = sum(x) - sum(y)

liftable
def abs(x: Int) = if x < 0 then -x else x
def tooBig = abs(diff) >= 10

out diff
out tooBig
```

Input trace

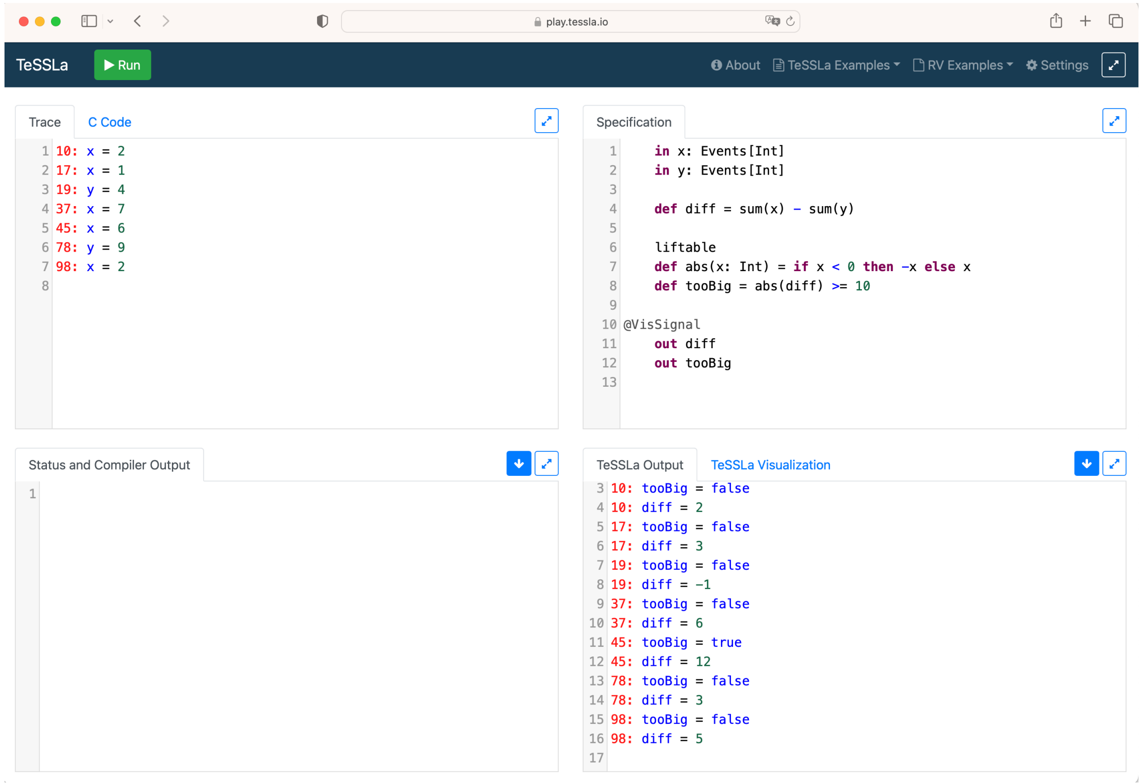
- trace.input

10: x = 2
17: x = 1
19: y = 4
37: x = 7
45: x = 6
78: y = 9
98: x = 2

In the playground

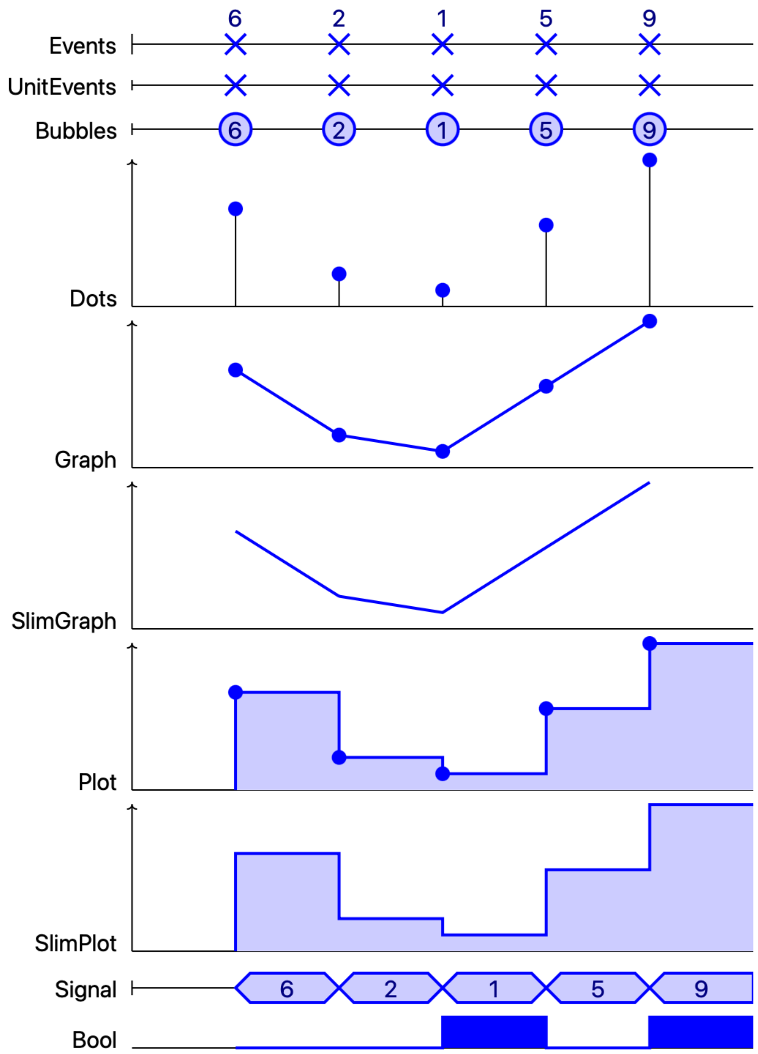
https://play.tessla.io

Playground



play.tessla.io
TeSSLa
Run
About
TeSSLa Examples
RV Examples
Settings
Trace
C Code
10: x = 2
17: x = 1
19: y = 4
37: x = 7
45: x = 6
78: y = 9
98: x = 2
Specification
in x: Events[Int]
in y: Events[Int]
def diff = sum(x) - sum(y)
liftable
def abs(x: Int) = if x < 0 then -x else x
def tooBig = abs(diff) >= 10
@VisSignal
out diff
out tooBig
Status and Compiler Output
TeSSLa Output
TeSSLa Visualization
10: tooBig = false
10: diff = 2
17: tooBig = false
17: diff = 3
19: tooBig = false
19: diff = -1
37: tooBig = false
37: diff = 6
45: tooBig = true
45: diff = 12
78: tooBig = false
78: diff = 3
98: tooBig = false
98: diff = 5

@VisSTYLE



Events
UnitEvents
Bubbles
Dots
Graph
SlimGraph
Plot
SlimPlot
Signal
Bool
6
2
1
5
9

Running

- java -jar tessla.jar interpreter specification.tessla trace.input

0: tooBig = false
0: diff = 0
10: tooBig = false
10: diff = 2
17: tooBig = false
17: diff = 3
19: tooBig = false
19: diff = -1
37: tooBig = false
37: diff = 6
45: tooBig = true
45: diff = 12
78: tooBig = false
78: diff = 3
98: tooBig = false
98: diff = 5

TeSSLa Scala/Rust Compiler

- **Scala compiler**
  - allows compilation to Scala code or a JAR file executable on the Java JVM.

    java -jar tessla.jar compile-scala -j monitor.jar specification.tessla
  - creates an executeable Jar-File **monitor.jar** which receives inputs and produces outputs via stdio in the same format as the interpreter
- **Rust compiler**

    java -jar tessla.jar compile-rust -b monitor specification.tessla
  - creates an executable **monitor** which receives inputs and produces outputs via stdio in the same format as the interpreter

Instrumenting C-Code

- Instrument the C source code using the observation annotations defined in the TeSSLa specification:

```
java -jar tessla.jar instrumenter spec.tessla main.c
        /usr/lib/gcc/x86_64-linux-gnu/9/include/
```

- Instrumentation is done on the LLVM level and specific setup for your machine is needed

For convenience

- As long as it works

```
docker run -v $(pwd):/wd -w /wd --rm registry.isp.uni-luebeck.de/tessla/tessla-docker:2.0.0 rv spec.tessla main.c
```

TeSSLa Language in Detail

Let‘s work through the tutorial

https://www.tessla.io/tutorial/

RV with TeSSLa

main.c

```c
void foo() {
	int x = 42;
}

int main() {
	for (int i = 0; i < 5; i++) {
		foo();
	}
	return 0;
}
```

spec.tessla

```
@InstFunctionCall("foo")
in foo: Events[Unit]
out foo
def num := count(foo)
out num
```

Explore

- Instrument the C source

  ```
  java -jar tessla.jar instrumenter spec.tessla main.c
  /usr/lib/gcc/x86_64-linux-gnu/9/include/
  ```
- Compile the instrumented C code

  ```
  gcc main.c.instrumented.c -llogging -pthread -ldl -o main
  ```
- Execute the compiled program, creating the file trace.log

  ```
  ./main
  ```
- Monitor the trace

  ```
  java -jar tessla.jar interpreter --base-time 1ns spec.tessla trace.log
  ```
- Alternatively

  ```
  docker run -v $(pwd):/wd -w /wd --rm registry.isp.uni-luebeck.de/tessla/tessla-
  docker:2.0.0 rv spec.tessla main.c
  ```

Measuring a Function's Runtime

```c
#include <stdlib.h>
#include <unistd.h>

void compute() {
  int duration = 40000;
  duration += (rand() % 10) * 1000;
  usleep(duration);
}

int main() {
  for (int i = 0; i < 10; i++) {
    compute();
  }
}
```

```
@InstFunctionCall("compute")
in call: Events[Unit]

@InstFunctionReturn("compute")
in ret: Events[Unit]

def duration := runtime(call, ret)
out duration

out maximum(duration) as max
out average(duration) as avg
```

Checking Correctness of Values

```c
#include <stdio.h>
#include <unistd.h>

int add(int a, int b) {
  return a + b;
}

int main() {
  printf("%i\n", add(2,3));
  printf("%i\n", add(17,4));
  printf("%i\n", add(2000000000,1000000000));
}
```

```
@InstFunctionCallArg("add", 0)
in a: Events[Int]
@InstFunctionCallArg("add", 1)
in b: Events[Int]

@InstFunctionReturnValue("add")
in r: Events[Int]

def should = last(a + b, r)
def ok = r == should

out a
out b
out r
out should
out ok
```

Multiple Threads

```c
#include <pthread.h>

void foo() {}

void *task () {
  foo();
  foo();
  foo();
  return NULL;
}

int main ()
{
  pthread_t t1, t2;
  pthread_create(&t1, NULL, &task, NULL);
  pthread_create(&t2, NULL, &task, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  return 0;
}
```

```
@InstFunctionCall("foo")
in foo: Events[Unit]

@ThreadId
in tid: Events[Int]

out foo
out tid
```

Checking Correct Locking

```c
#include <pthread.h>
#include <unistd.h>

int shared_memory[4] = {0};
pthread_mutex_t locks[4] = {
  PTHREAD_MUTEX_INITIALIZER, PTHREAD_MUTEX_INITIALIZER,
  PTHREAD_MUTEX_INITIALIZER, PTHREAD_MUTEX_INITIALIZER,
};

void use(int index) {
  shared_memory[index]++;
}

void *task1 () {
  for (int i = 0; i < 4; i++) {
    pthread_mutex_lock(&locks[i]);
    use(i);
    pthread_mutex_unlock(&locks[i]);
  }
  return NULL;
}
```

```c
void *task2 () {
  for (int i = 3; i >= 0; i--) {
    pthread_mutex_lock(&locks[i]);
    use(i);
    pthread_mutex_unlock(&locks[i]);
  }
  return NULL;
}

int main ()
{
  pthread_t t1, t2;
  pthread_create(&t1, NULL, &task1, NULL);
  pthread_create(&t2, NULL, &task2, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  return 0;
}
```

Checking Correct Locking (2)

```
@InstFunctionCallArg("pthread_mutex_lock", 0)
in lock: Events[Int]

@InstFunctionCallArg("pthread_mutex_unlock", 0)
in release: Events[Int]

@InstFunctionCallArg("use", 0)
in access: Events[Int]

@ThreadId
in tid: Events[Int]

def locksOfThread = {
  def oldMap = last(map, tid)
  def oldLocks = Map.getOrElse(oldMap, tid, Set.empty[Int])
  def map: Events[Map[Int, Set[Int]]] = merge3(
    on(lock, Map.add(oldMap, tid, Set.add(oldLocks, lock))),
    on(release, Map.add(oldMap, tid, Set.remove(oldLocks, release))),
    Map.empty[Int, Set[Int]])
  map
}
```

```
def locksForResource = {
  def old = last(map, access)
  def currentLocks = Map.get(locksOfThread, tid)
  def map: Events[Map[Int, Set[Int]]] = merge(
    on(access, Map.add(old, access, Set.intersection(
      currentLocks,
      Map.getOrElse(old, access, currentLocks)))),
    Map.empty[Int, Set[Int]])
  map
}

def error = unitIf(Set.size(Map.get(locksForResource, access)) == 0)
out error
```

Cyber-Physical Systems

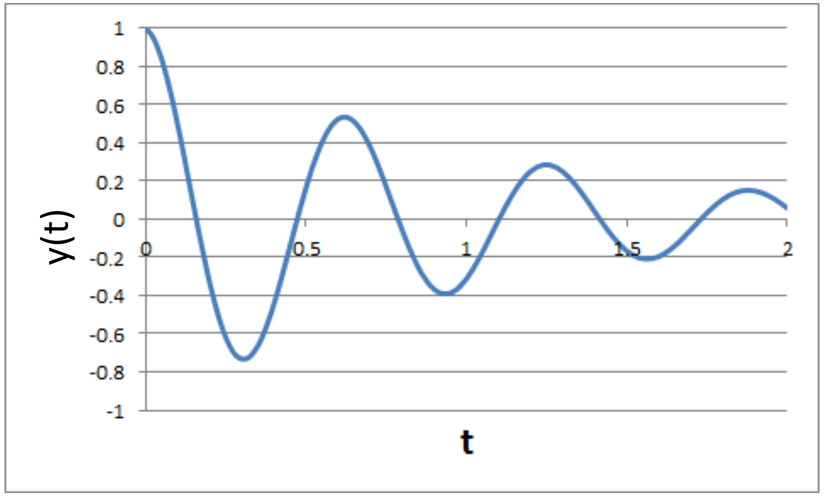
Cyber-Physical System

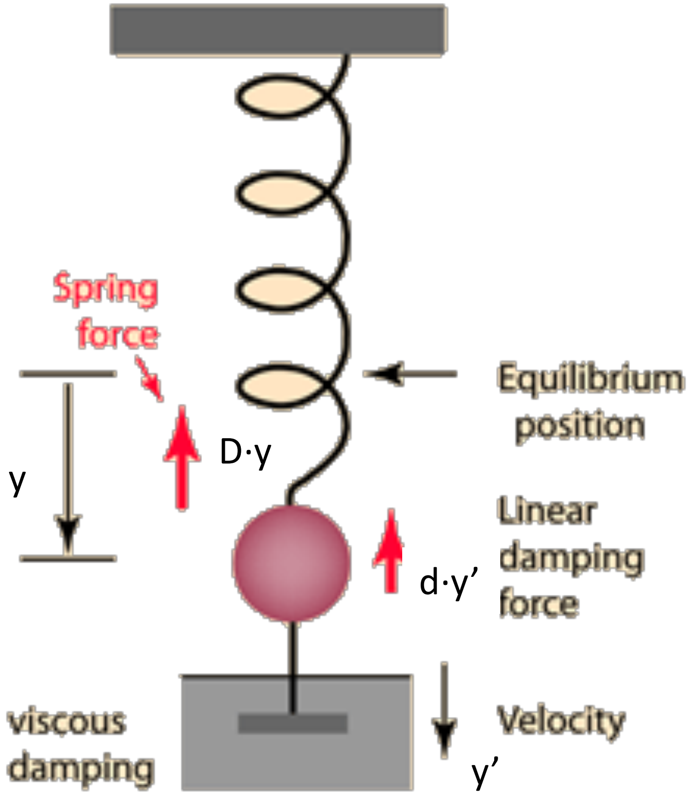
- Communicating hybrid systems
- Communicating embedded systems interacting with the physical world

- Discrete Math, Events, Propositions
- Continuous Math, Signals

Damped Harmonic Oscillator

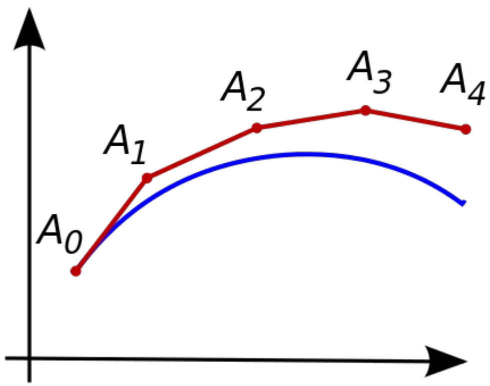
$$m \cdot y'' = -D \cdot y - d \cdot y'$$



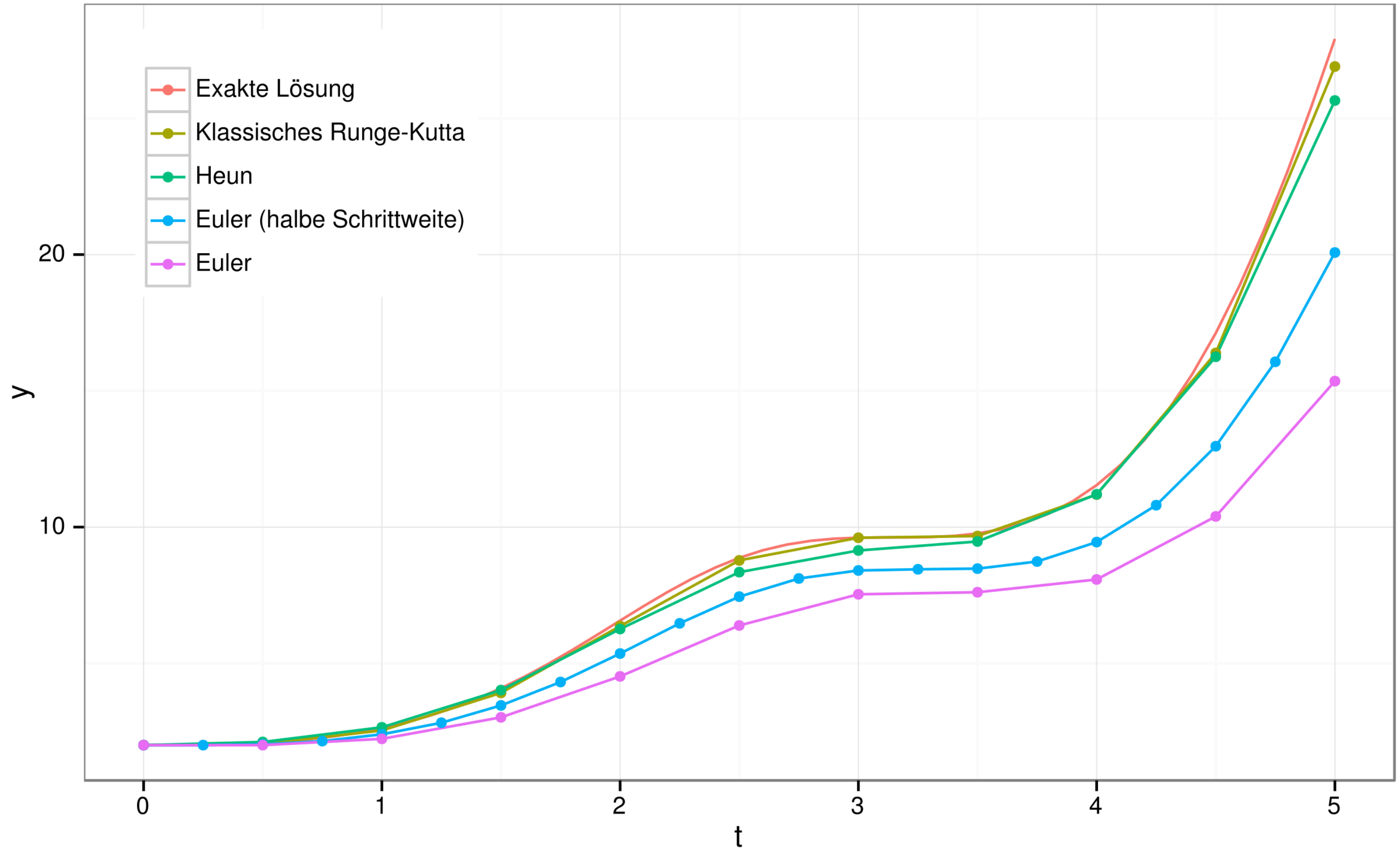


Solving of ODE – Numerical Approximations

- Euler's method



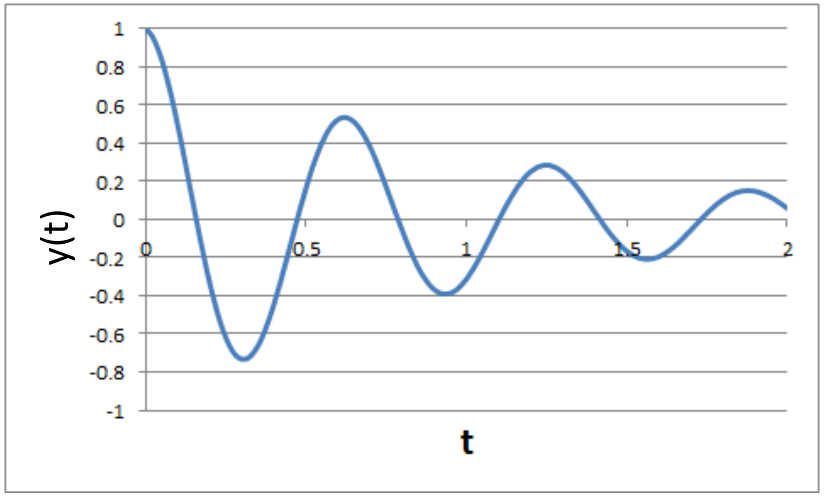
Solving of ODE – A Variety of Methods

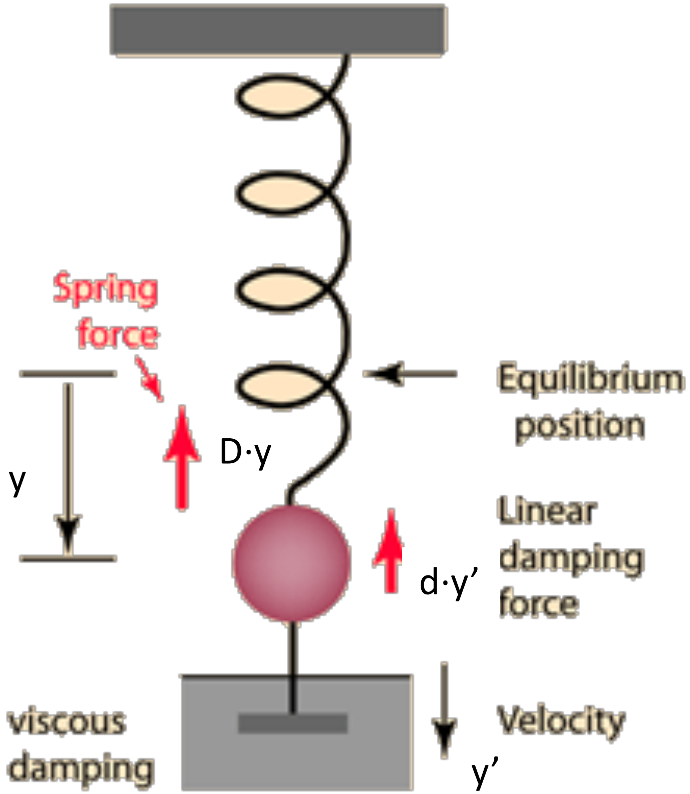


ODEs in TeSSLa

Damped Harmonic Oscillator

$$m \cdot y'' = -D \cdot y - d \cdot y'$$

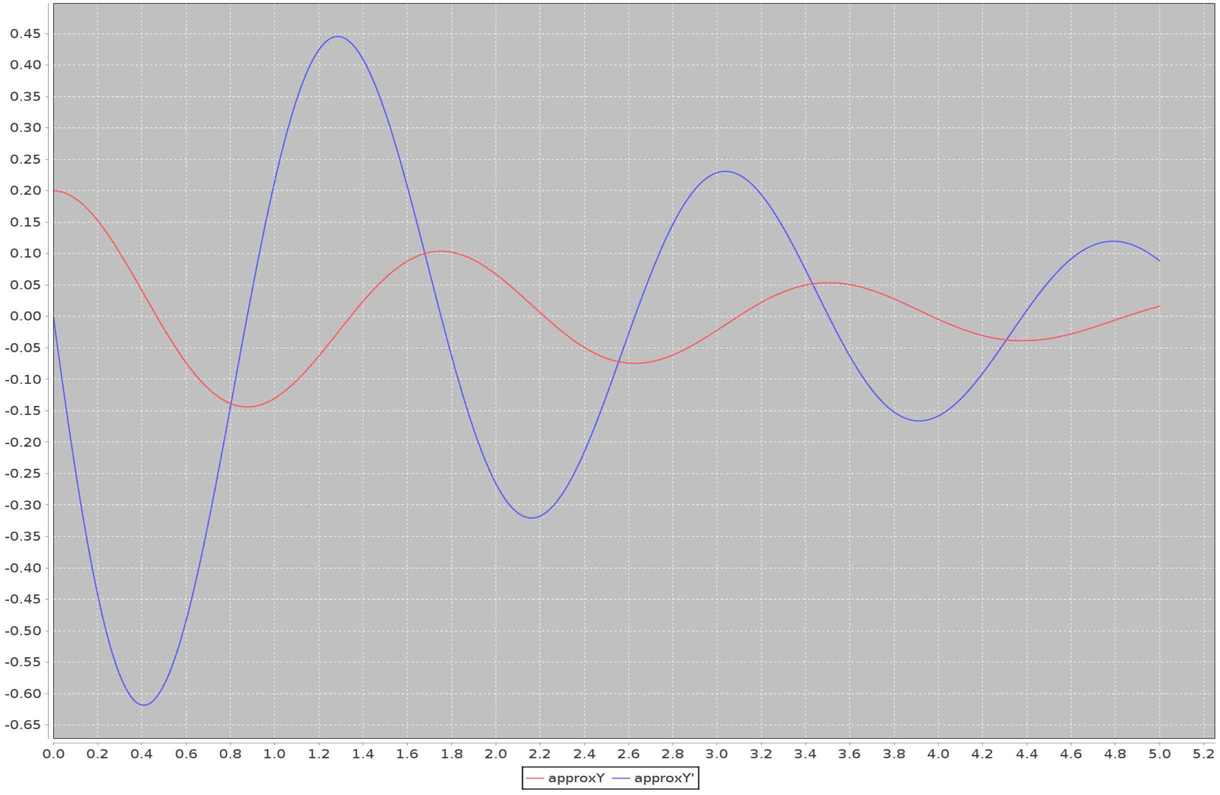




The Spring Example

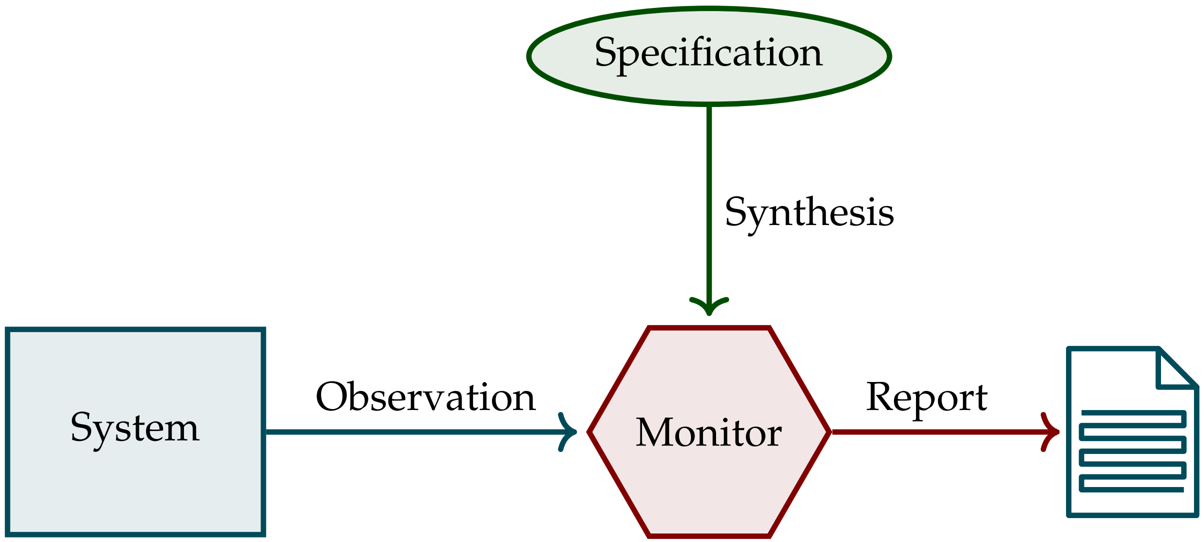
```
in sensor: Events[Float]

def m: Float = 0.2       # kg
def D: Float = 2.6       # N/m
def d: Float = 0.15      # kg/s
def y''(t: Float, y: Float, y': Float): Float =
  -D / m * y - d / m * y')
def y_0  = 0.2   # m
def y'_0 = 0.0   # m/s

def approx:   Events[(Float, Float)] = rk4(y'', y_0, y'_0)
def approxY:  Events[Float] = approx._1
def approxY': Events[Float] = approx._2
def alarm = |sensor - approxY| > ϵ
```

Plot of the Damped Spring



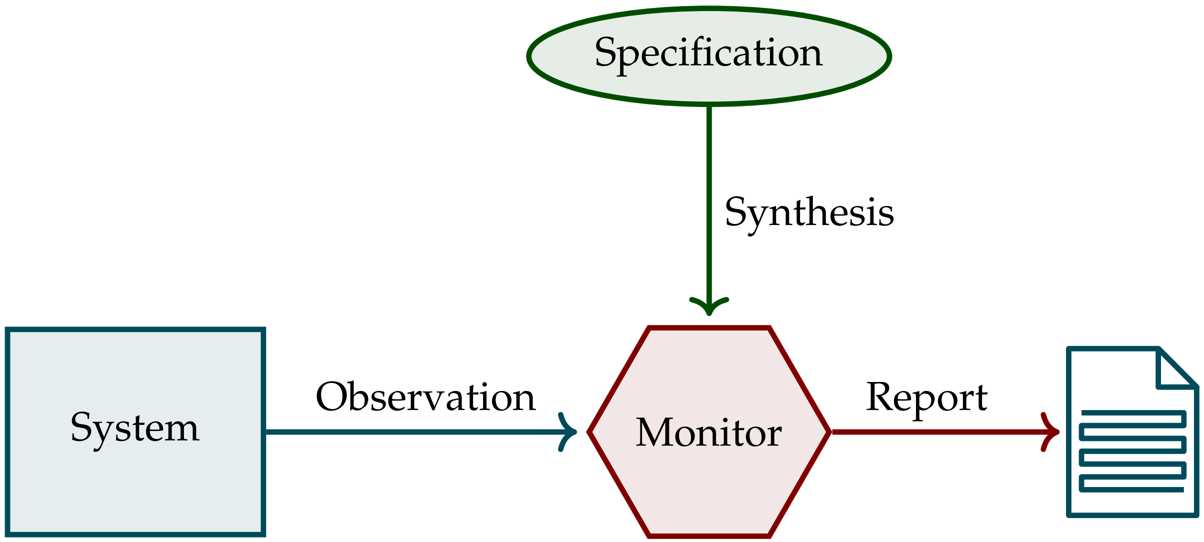
Control

Runtime Verification

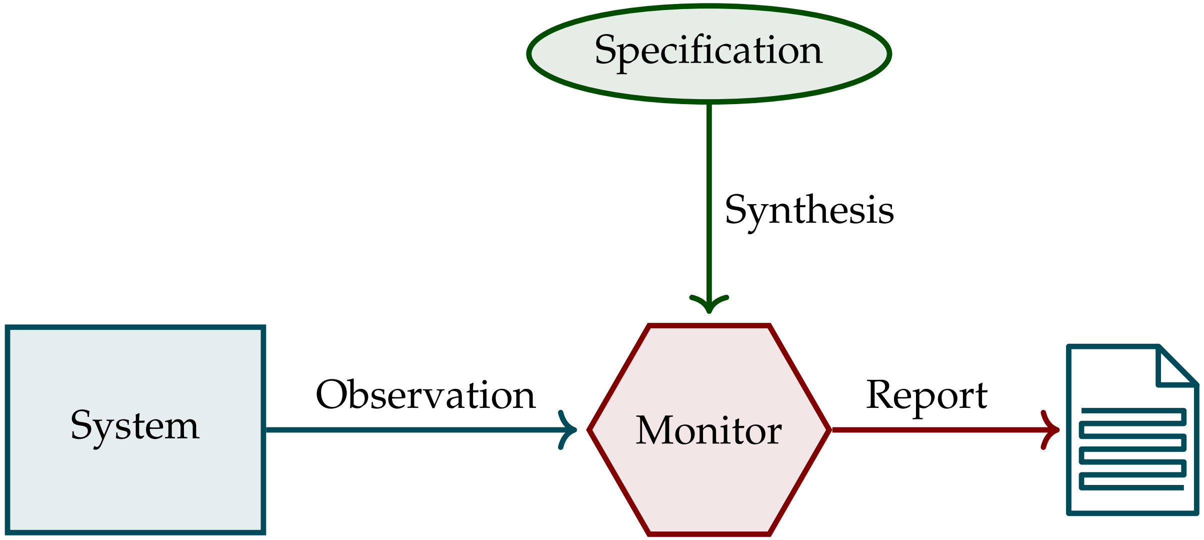


Runtime Verification

- Partial Verification

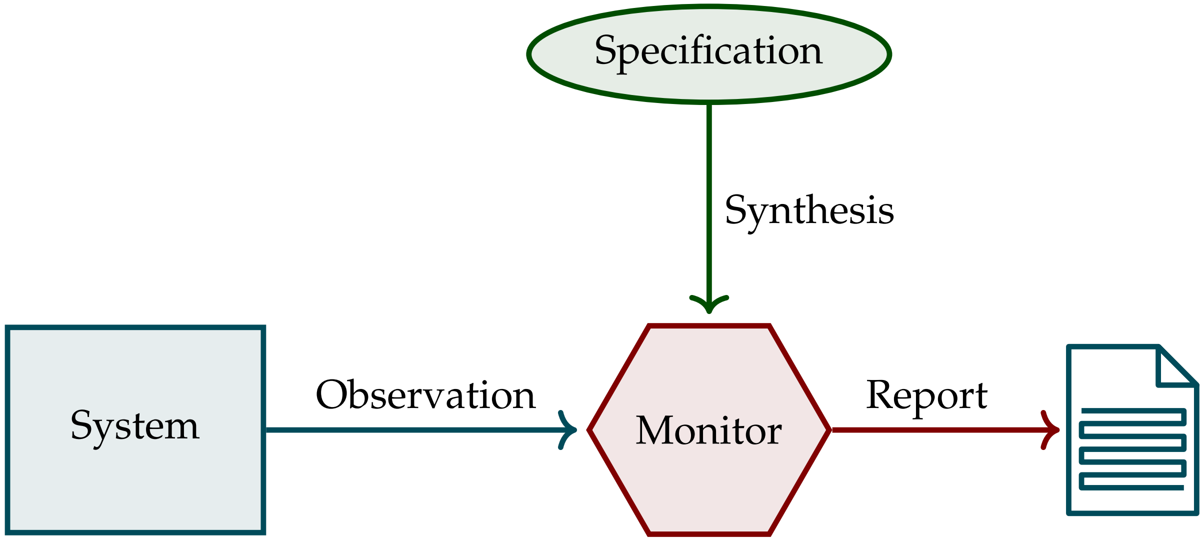


Runtime Verification



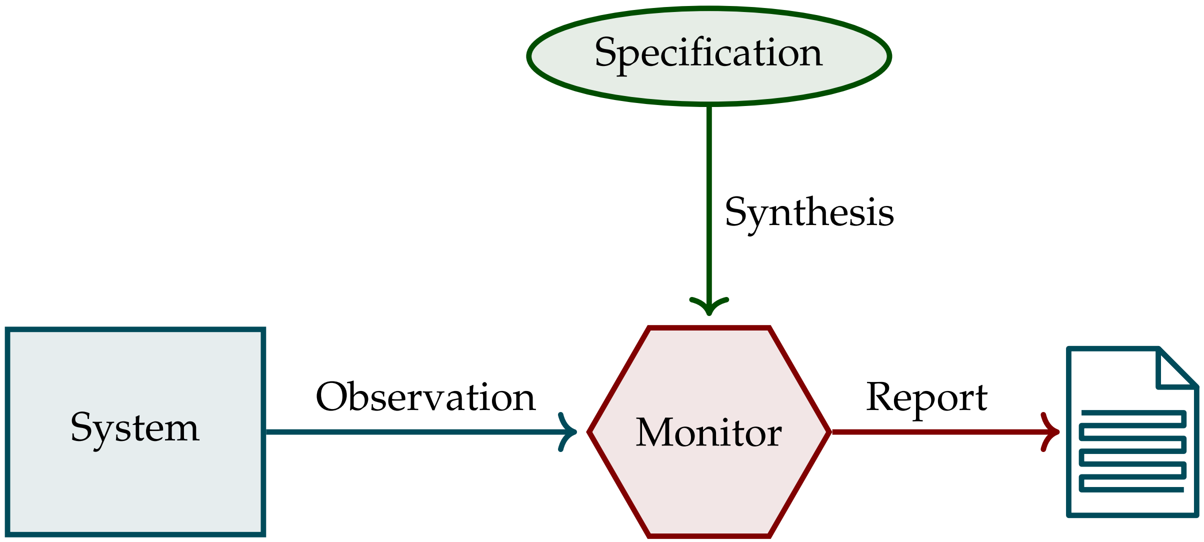
- Partial Verification
- Testing Temporal Assertions

Runtime Verification



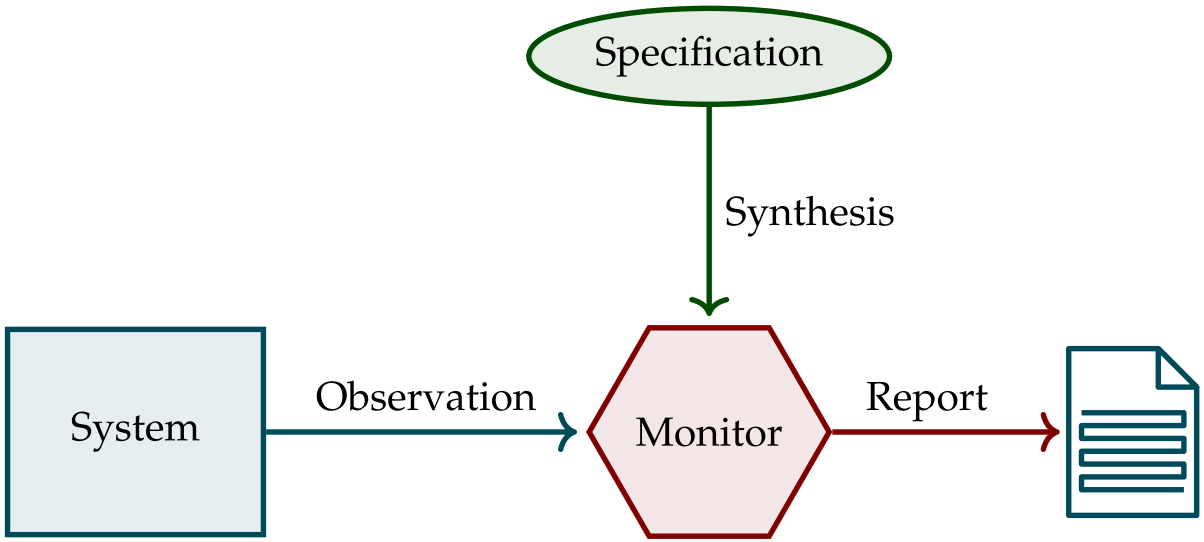
- Partial Verification
- Testing Temporal Assertions
- Test Cases as Input Sequences checked by Monitors

Runtime Verification



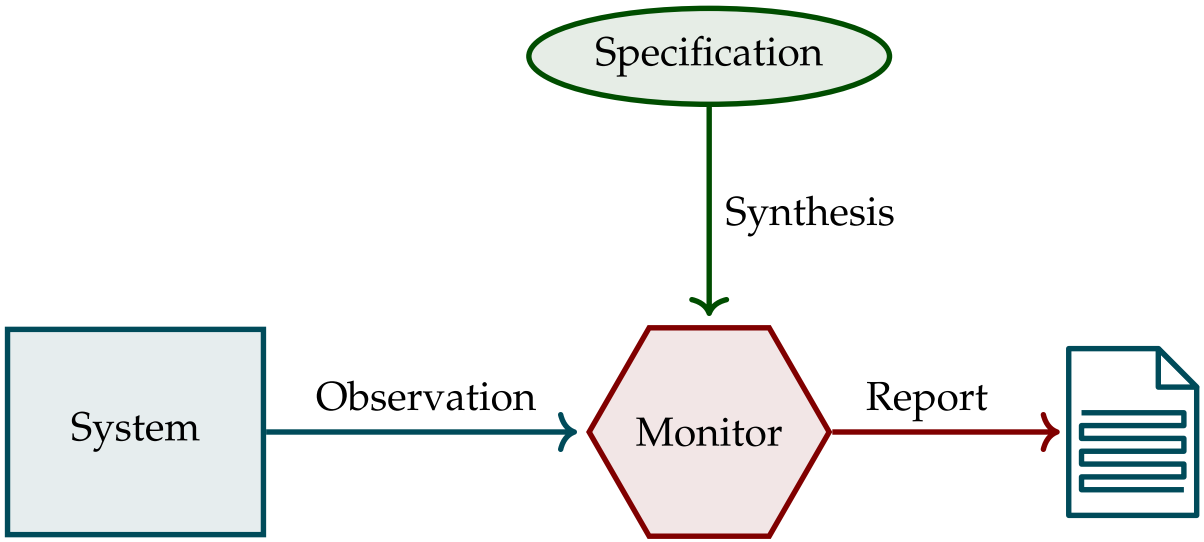
- Partial Verification
- Testing Temporal Assertions
- Test Cases as Input Sequences checked by Monitors
- Debugging

Runtime Verification

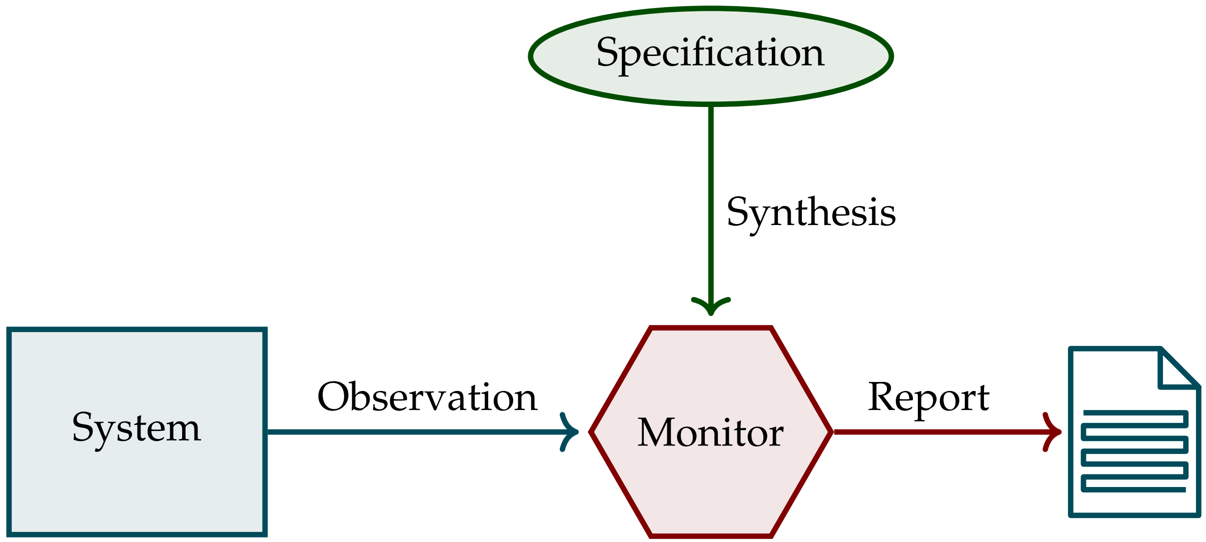


- Partial Verification
- Testing Temporal Assertions
- Test Cases as Input Sequences checked by Monitors
- Debugging
- Control?

Control from an RV Point of View

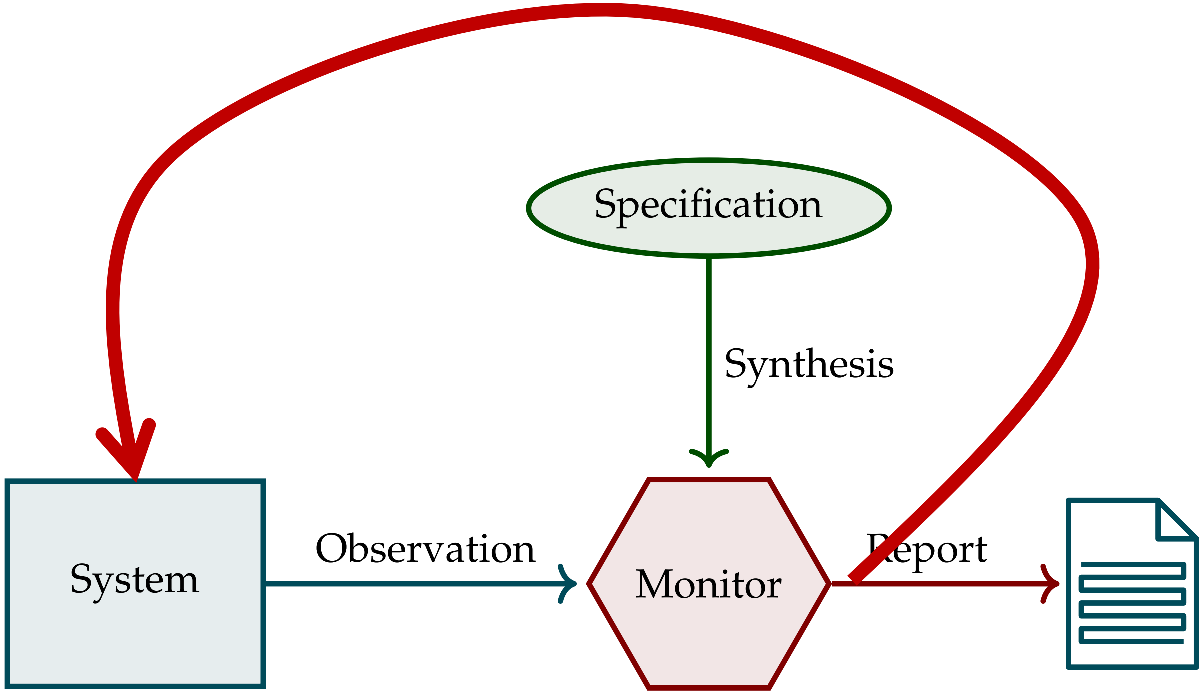


Control from an RV Point of View



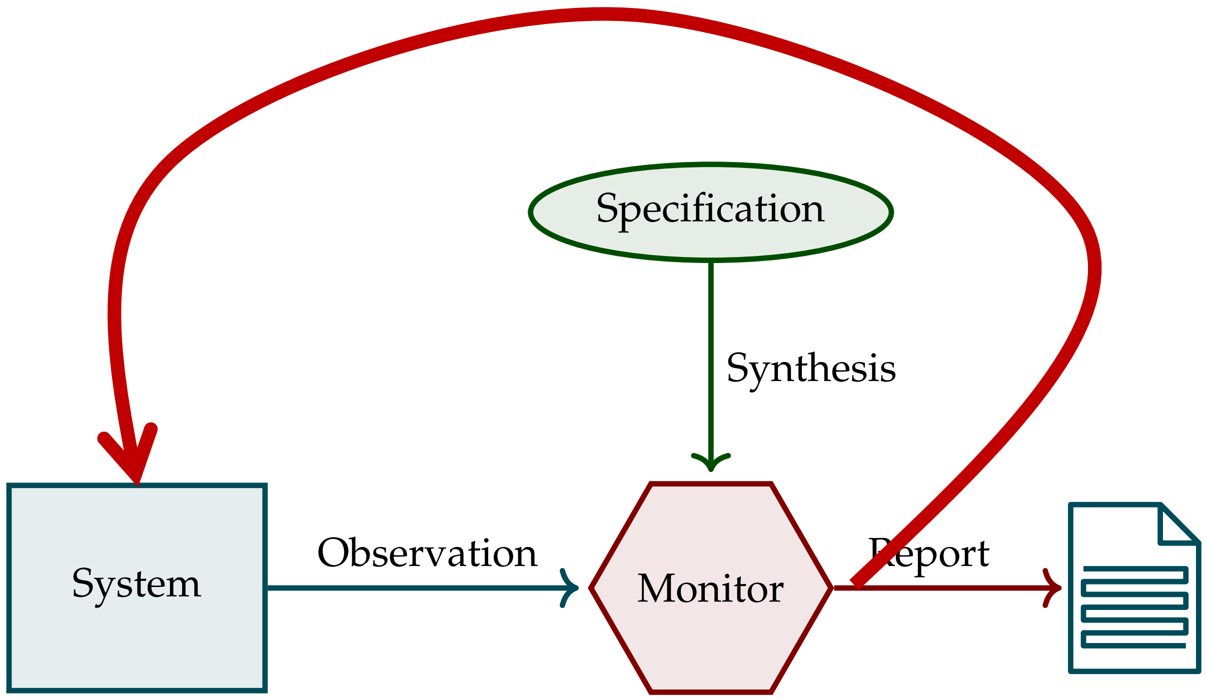
- Monitor Output as Feedback/ Intervention to System

Control from an RV Point of View



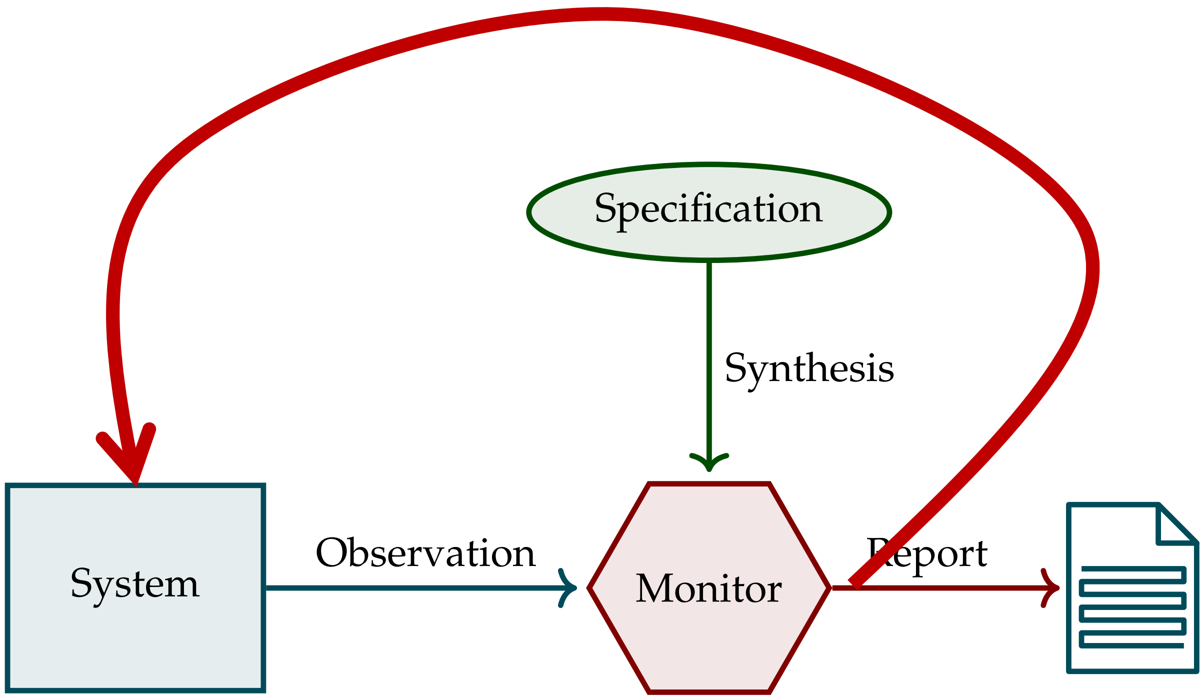
- Monitor Output as Feedback/ Intervention to System

Control from an RV Point of View



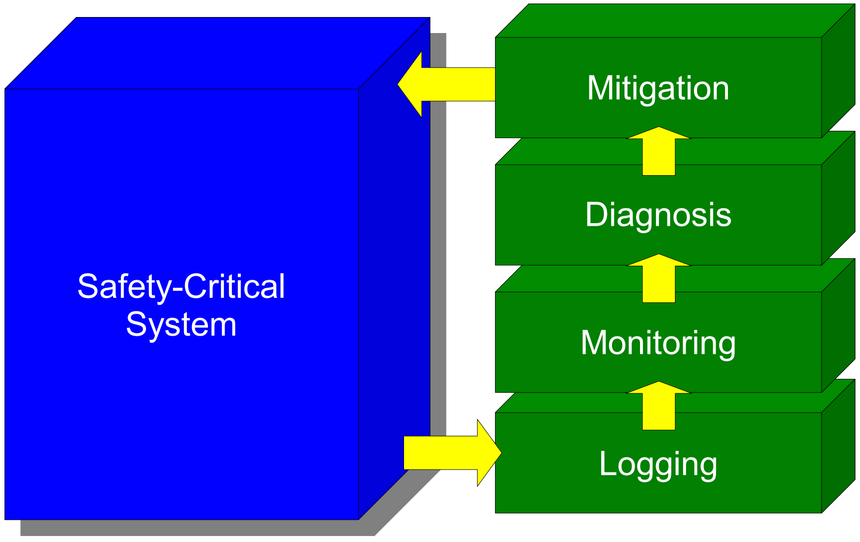
- Monitor Output as Feedback/ Intervention to System
- Monitor has to give more specific Output

Control from an RV Point of View

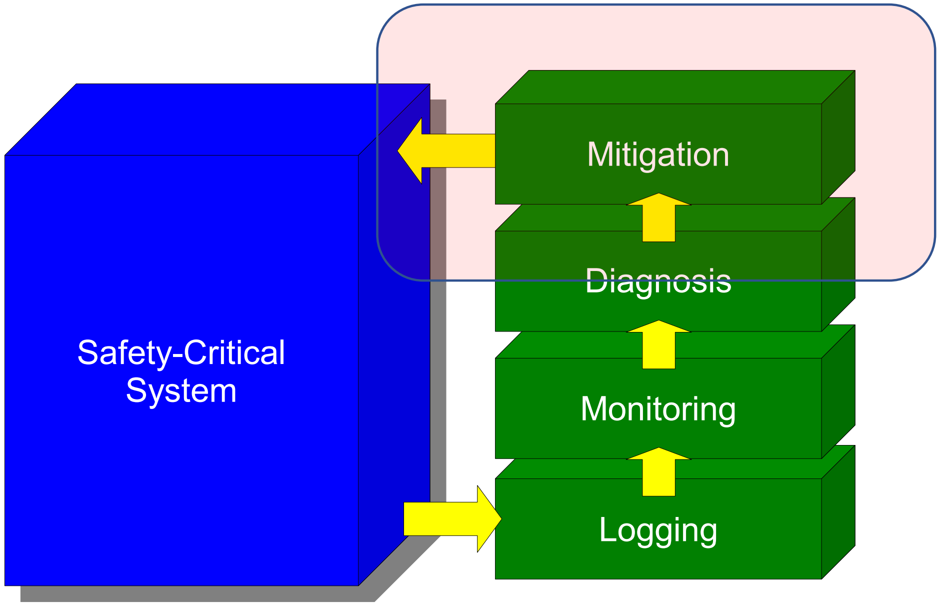


- Monitor Output as Feedback/ Intervention to System
- Monitor has to give more specific Output
- Here: Monitor actually computes control values

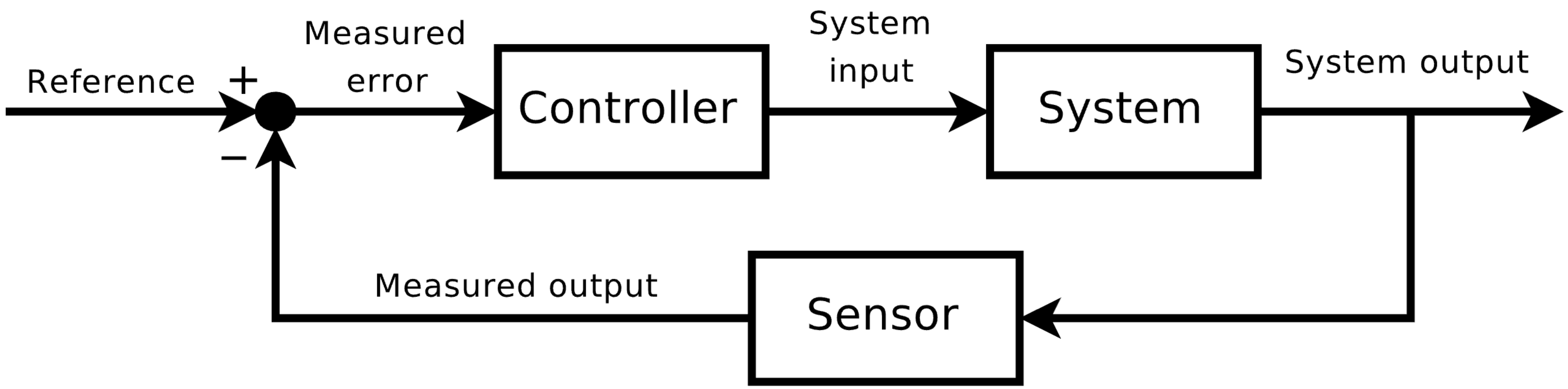
Self-Healing System (FDIR with RV)



Self-Healing System (FDIR with RV)



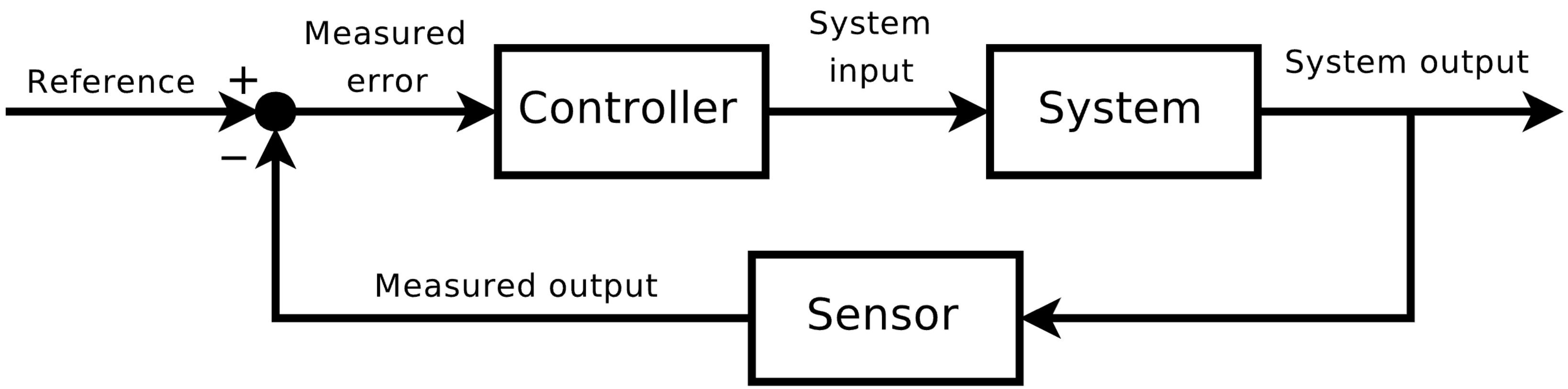
Control



Reference
+
−
Measured
error
Controller
System
input
System
System output
Measured output
Sensor

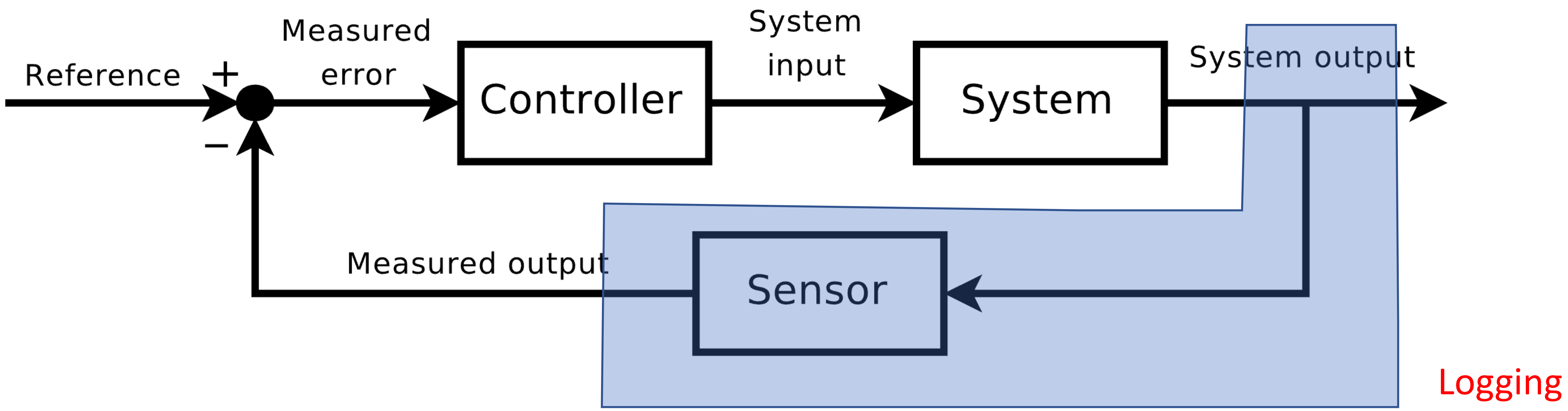
Control

Closed-Loop Controller - Feedback



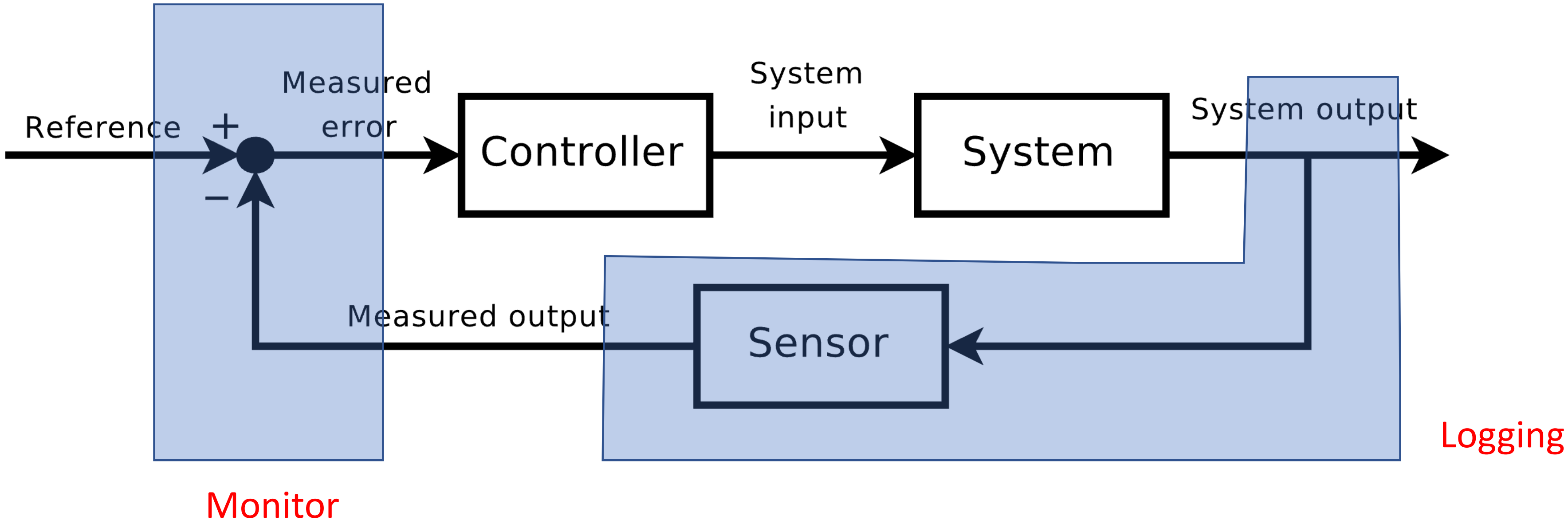
Control

Closed-Loop Controller - Feedback



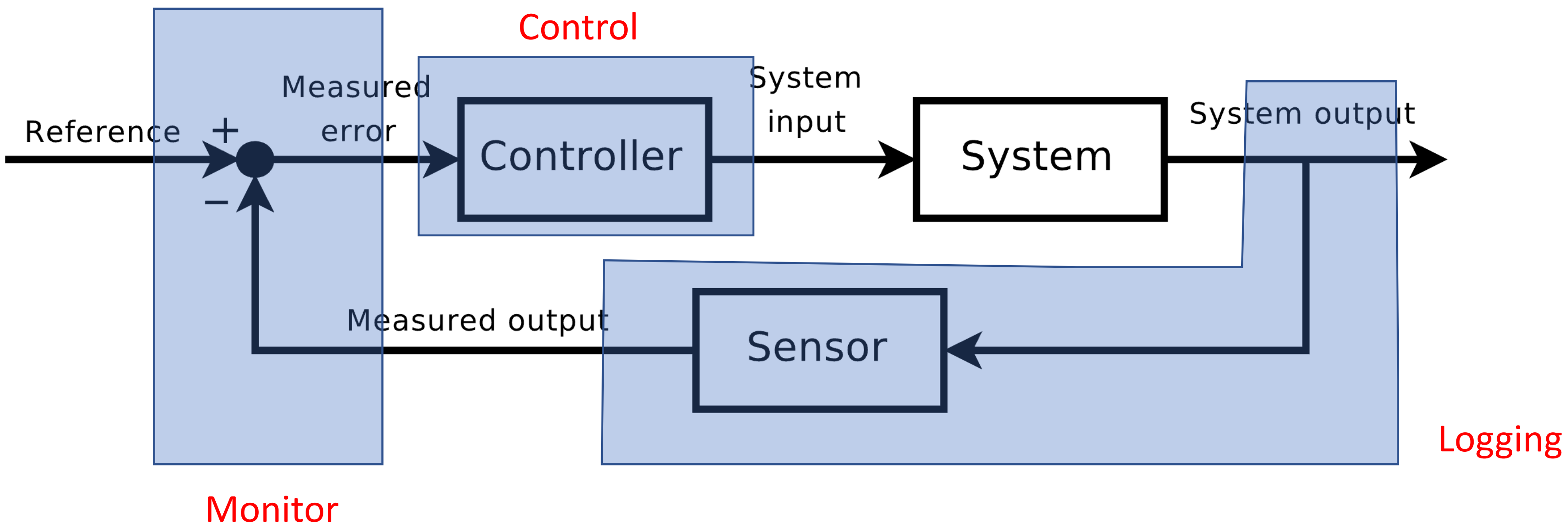
Control

Closed-Loop Controller - Feedback

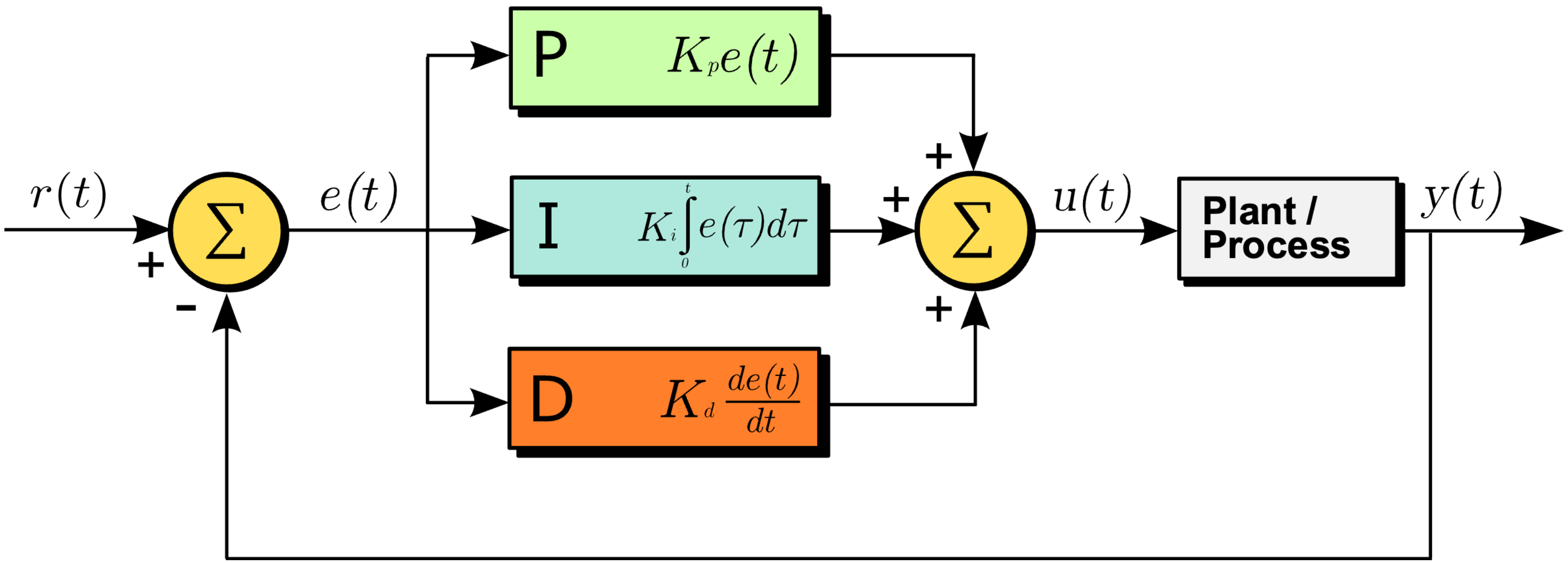


Control

Closed-Loop Controller - Feedback



PID-Controller



Controller Combinations

P
Proportional controller to reduce the transient period.
Changes the magnitude only.

I
Integral controller to reduce the time invariant error
Lags the output phase.

D
Derivative controller to minimize the transient errors like overshoot, oscillatory response.
Leads the output phase.

PI
Reduces rise time and steady state errors
Changes the magnitude as well as lags the output.

PD
Reduces rise time and transient errors such as overshoot, oscillations in output.
Changes both the magnitude as well as adds a leading phase to the output.

PID
General case of a controller. Can be used to control the magnitude and lead/ lag phase problems.
Changes the magnitude and can add positive or negative phase to the output as per the requirements.

https://medium.com/@svm161265/when-and-why-to-use-p-pi-pd-and-pid-controller-73729a708bb5

Code of Controller in TeSSLa

- *See* tessla.io

Controlling Robots

TeSSLa/ROS Bridge

```
include "TesslaROSBridge.tessla“

@RosSubscription("/reduced_scan_to_tessla", "int64", "10")
in scan: Events[Int]


# Stop if there are short rays detected

def stop = scan < 20


@RosPublisher("/result_from_tessla_to_ros", "bool", "10")
out stop
```

Example

https://tessla.io/blog/rosBridge/rosBridge_jackal_video.m4v

Conclusions

Conclusions

- Stream-based Runtime Verification makes sense
- TeSSLa one approach in this setting
- Supports handling of data

- Monitoring CPS makes sense
- Controlling using RV techniques makes sense
- Separation of concerns

Future Work

- Controller module in TeSSLa?
- More concrete examples?
- Gain more experiences?
- Programming (safety aspects) of robots?
- Better use Modellica and FMUs?
- Add continuous functions symbolically to perform algebraic simplifications?