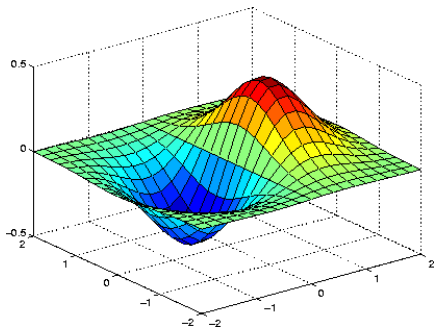
Predictability and Robustness in Embedded Systems

Tom Henzinger
EPFL

Based on joint work with Alberto Sangiovanni-Vincentelli.

Complexity Management in Engineering



Calculate

Model

Abstract

Predict

Artifact

Mathematics

Build & test

Bridge
Aircraft
Software
etc.

Mathematical Modeling: A Tale of Two Cultures

Bridge, Aircraft, etc.:
Physics-based Models

Differential Equations
Linear Algebra
Probability Theory

Software:
Systems-based Models

Logics
Discrete Structures
Automata Theory

Plane makers are accustomed to testing metals and plastics under almost every conceivable kind of extreme stress, but it's impossible to run a big computer program through every scenario to detect the bugs that invariably crop up.

In extreme cases, foul-ups can lead to sudden loss of control, sometimes not showing up until years after aircraft are introduced into service. Malaysia Airlines Flight 124 is a case in point. Boeing's 777 jets started service in 1995 and had never experienced a similar emergency before.

[Wall Street Journal; May 30, 2006]

What's wrong with our models?

Bridge, Aircraft, etc.:
Physics-based Models

Theories of estimation.
Theories of sensitivity.

Software:
Systems-based Models

Theories of correctness.

What's wrong with our models?

Bridge, Aircraft, etc.: Physics-based Models

Theories of estimation.
Theories of sensitivity.

Artifacts are physical objects; we want to make them predictable and robust.

Software: Systems-based Models

Theories of correctness.

Fallacy:
systems are non-physical, pseudo-mathematical objects; we want to prove properties.

Current State of Affairs

Most of Computer Science has systematically removed real time and resource consumption from its programming abstractions.

Current State of Affairs

Most of Computer Science has systematically removed real time and resource consumption from its programming abstractions.

Most of Electrical Engineering pretends there is no choice between (i) automatically synthesizing code from high-level, resource-aware models and (ii) low-level (assembly) coding.

Current State of Affairs

Most of Computer Science has systematically removed real time and resource consumption from its programming abstractions.

Most of Electrical Engineering pretends there is no choice between (i) automatically synthesizing code from high-level, resource-aware models and (ii) low-level (assembly) coding.

All three approaches (high-level programming, code synthesis, low-level coding) lead to "build & test & tweak":

Software is the most costly, least flexible, and most error-prone part of an embedded system.

Complexity Control

Traditional Answer:

We need to divide-and-conquer: components, contracts, interfaces, modularity, assume-guarantee, separation of concerns, etc.

Complexity Control

Traditional Answer:

We need to divide-and-conquer: components, contracts, interfaces, modularity, assume-guarantee, separation of concerns, etc.

It has not yet been demonstrated that these approaches simplify the problem; in fact, they often make it more complicated [Lamport].

Complexity Control

Traditional Answer:

We need to divide-and-conquer: components, contracts, interfaces, modularity, assume-guarantee, separation of concerns, etc.

It has not yet been demonstrated that these approaches simplify the problem; in fact, they often make it more complicated [Lamport].

Let's try something simpler first:

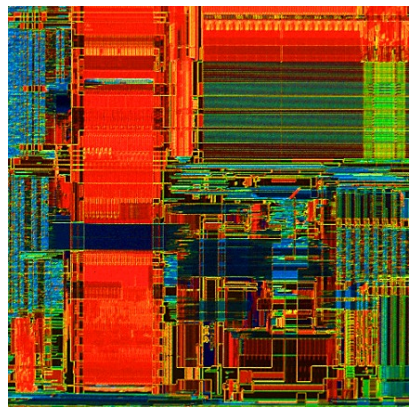
1. Let's restrict ourselves to deterministic designs.
2. Let's restrict ourselves to continuous designs.

Where have all the smart guys gone?

Where have all the smart guys gone?

Software truly is the most complex artifacts mankind builds. It's not surprising we rarely get it right.

Between 10^{69} and 10^{81} atoms in the universe.



10 MB cache > $10^{20,000,000}$ states.

Challenge 1:
Build Predictable Systems

Challenge 1:
Build Predictable Systems

Predictable = Deterministic

A system is *deterministic* if
for every input behavior, the output behavior is unique.

Challenge 1:
Build Predictable Systems

Predictable = Deterministic

A system is *deterministic* if
for every input behavior, the output behavior is unique.

-internal (invisible) behavior need not be unique

-visible behavior includes all aspects of interest:
for real-time systems, behavior includes time stamps

Nondeterminism

Central to complexity theory: P v. NP

Central to abstraction:

-high-level programming languages: e.g. memory management

-algorithm design: as long as there exists an $0 \leq i < n$ such that $a[i] > a[i+1]$, swap $a[i]$ and $a[i+1]$

-don't cares: if input=0, then output=$\perp$

Central to concurrency: $a||b = ab + ba$

Nondeterminism

Central to complexity theory: P v. NP

invisible

Central to abstraction:

-high-level programming languages: e.g. memory management

-algorithm design: as long as there exists an $0 \leq i < n$ such that $a[i] > a[i+1]$, swap $a[i]$ and $a[i+1]$

-don't cares: if input=0, then output=$\perp$

Central to concurrency: $a||b = ab + ba$

Nondeterminism

Central to complexity theory: P v. NP

invisible

Central to abstraction:

-high-level programming languages: e.g. memory m

invisible

-algorithm design: as long as there exists an $0 \leq i < n$ such that $a[i] > a[i+1]$, swap $a[i]$ and $a[i+1]$

-don't cares: if input=0, then output=$\perp$

Central to concurrency: a||b = ab + ba

Nondeterminism

Central to complexity theory: P v. NP

invisible

Central to abstraction:

-high-level programming languages: e.g. memory m

invisible

-algorithm design: as long as there exists an $0 \leq i < n$

a[i]>a[i+1], swap a[i] and a[i+1]

invisible

-don't cares: if input=0, then output=$\perp$

Central to concurrency: a||b = ab + ba

Nondeterminism

Central to complexity theory: P v. NP

invisible

Central to abstraction:

-high-level programming languages: e.g. memory m

invisible

-algorithm design: as long as there exists an $0 \leq i < n$ a[i]>a[i+1], swap a[i] and a[i+1]

invisible

-don't cares: if input=0, then output=$\perp$

deterministic

Central to concurrency: a||b = ab + ba

Output domain = {0,1.$\perp$}

Nondeterminism

Central to complexity theory: P v. NP

invisible

Central to abstraction:

-high-level programming languages: e.g. memory m

invisible

-algorithm design: as long as there exists an $0 \leq i < n$ a[i]>a[i+1], swap a[i] and a[i+1]

invisible

-don't cares: if input=0, then output=$\perp$

deterministic

Central to concurrency: a||b = ab + ba

visible

a: x := x+1
b: x := 2x

Nondeterminism

Central to complexity theory: P v. NP

invisible

Central to abstraction:

-high-level programming languages: e.g. memory m

invisible

-algorithm design: as long as there exists an $0 \leq i < n$ a[i]>a[i+1], swap a[i] and a[i+1]

invisible

-don't cares: if input=0, then output=$\perp$

deterministic

Central to concurrency: a||b = ab + ba

visible

Alternatives to threads:

actors; transactions

less nondeterministic

Nondeterminism

1. Input nondeterminism: OK

   for every observable input behavior,
   unique observable output behavior

2. Unobservable implementation nondeterminism: OK

   deterministic abstraction layer over nondeterministic components

3. Don’t care nondeterminism: OK

   use don’t care (or probability distribution) as output observation

4. Observable implementation nondeterminism: AVOID

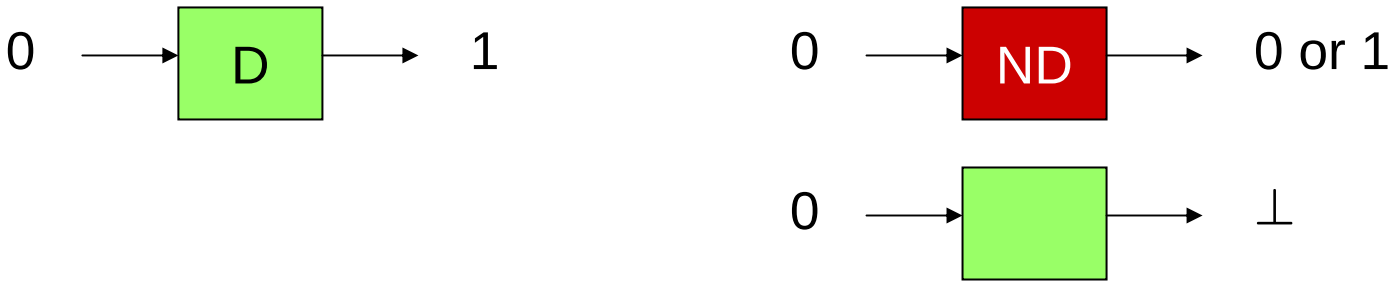
   e.g. multi-threaded systems, scheduled systems

What is an Observation?

Traditional software: input and output values

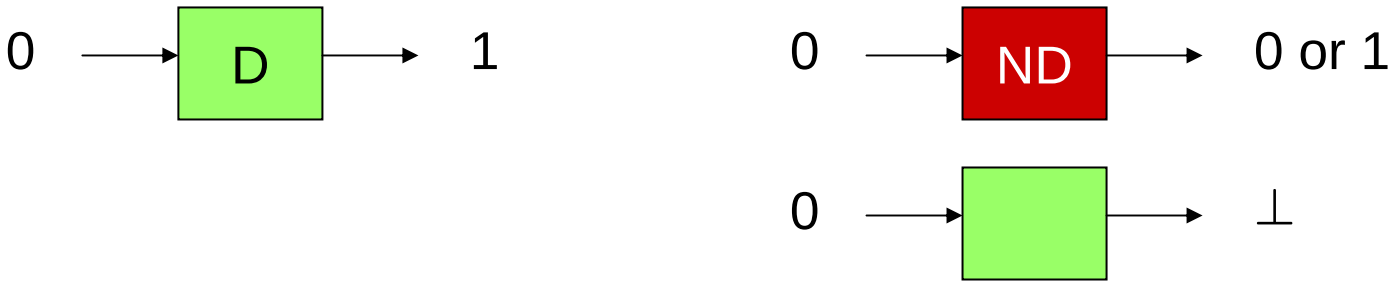
What is an Observation?

Traditional software: input and output values

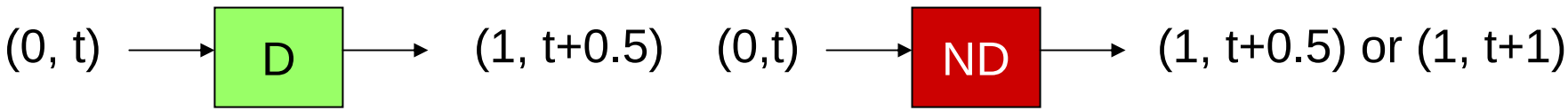


What is an Observation?

Traditional software: input and output values

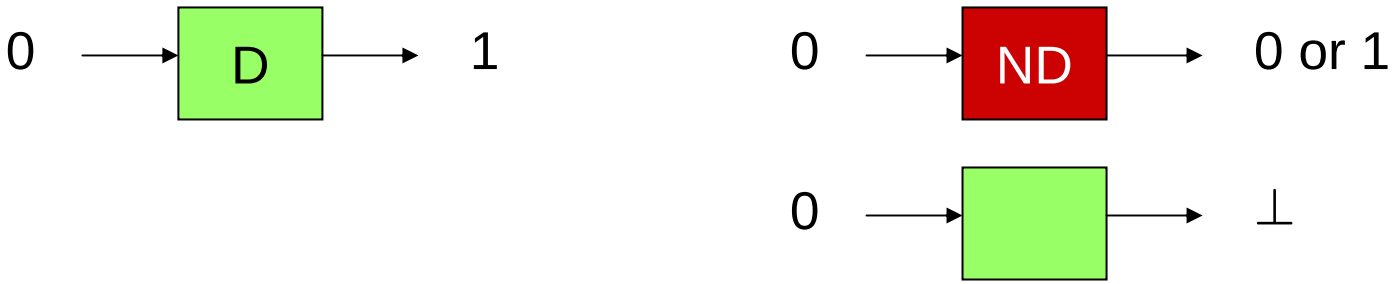


Real-time software: input and output values and times

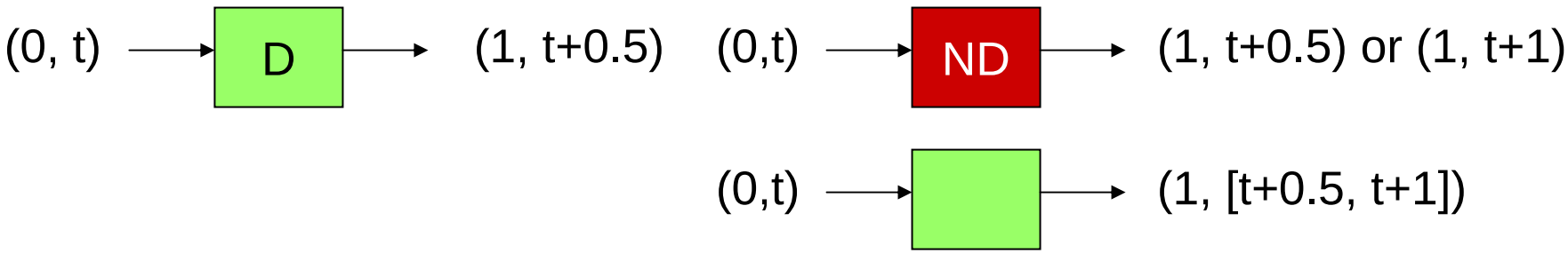


What is an Observation?

Traditional software: input and output values

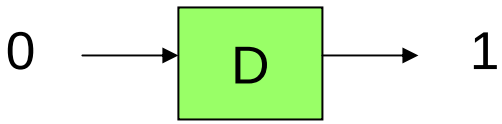


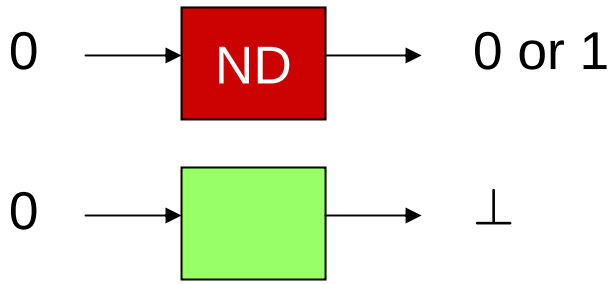
Real-time software: input and output values and times



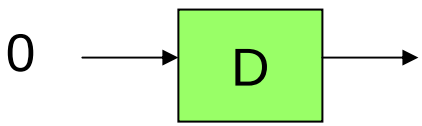
What is an Observation?

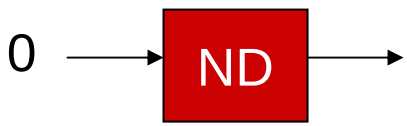
Traditional software: input and output values





Reliable software: output probabilities





Pr(0) = 0.1
Pr(1) = 0.9

or

Pr(0) = 0.2
Pr(1) = 0.8

Giotto Project: Deterministic Real Time

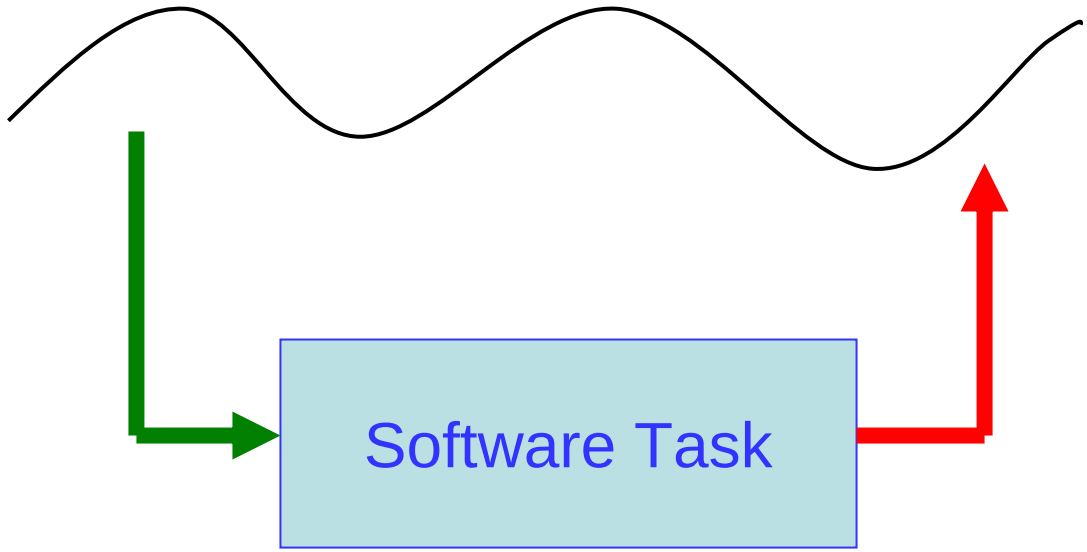
Can we build a real-time programming language that treats time in the way in which high-level languages treat memory?

-programmer specifies the time of outputs

-programmer assumes the platform offers sufficient performance

-compiler generates a suitable schedule or throws an exception

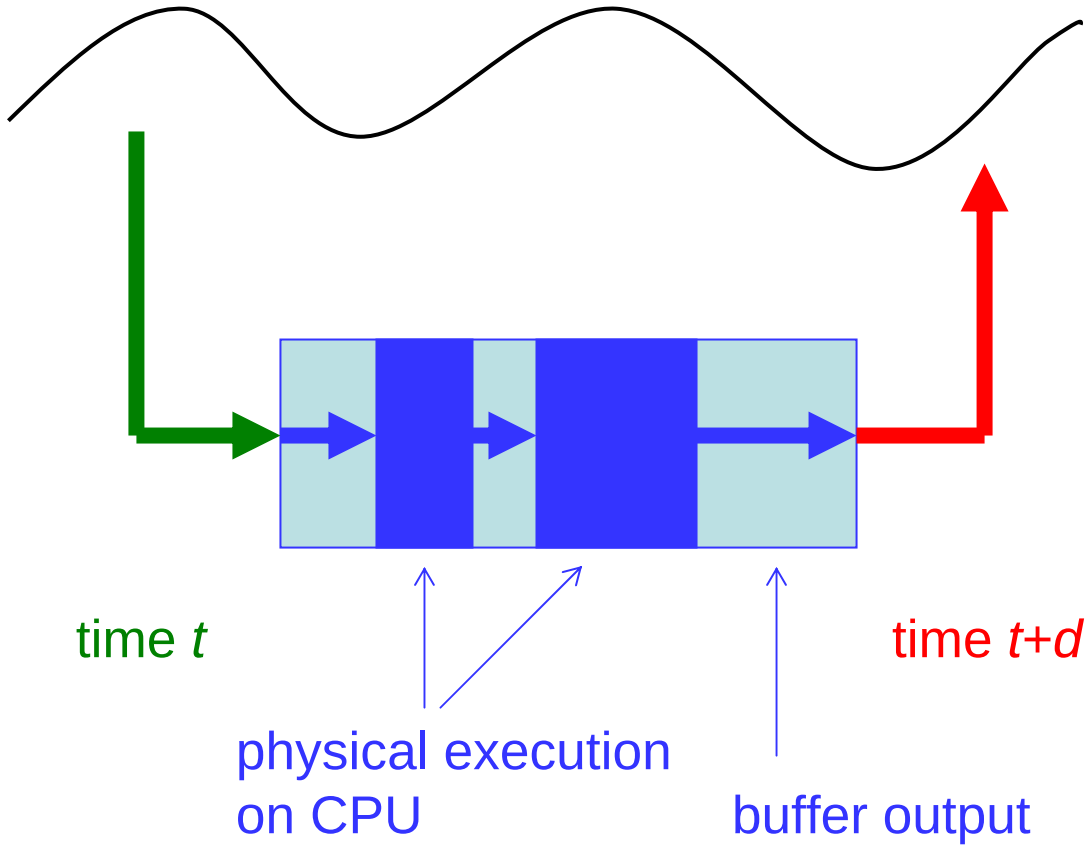
LET (Logical Execution Time) Programming Model



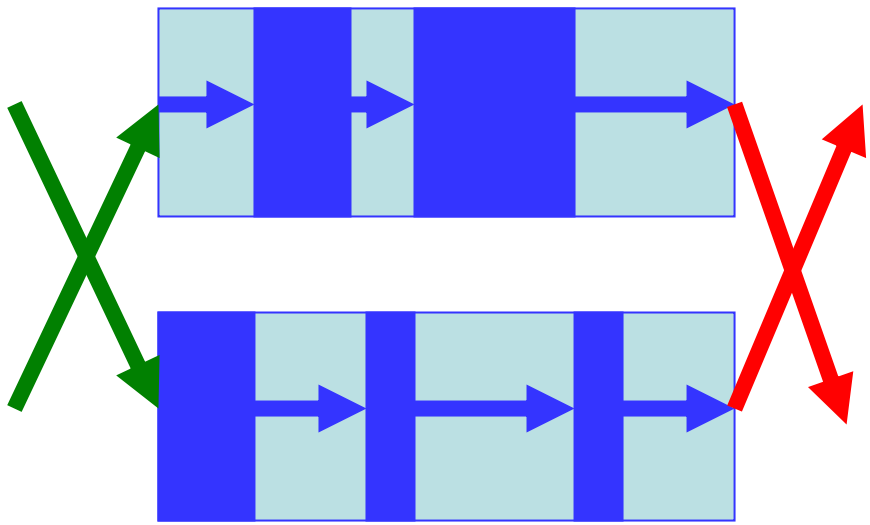
read sensor
input at time *t*

write actuator
output at time *t*+*d*,
for specified *d*

Compiler reconciles Logical and Physical Execution Times

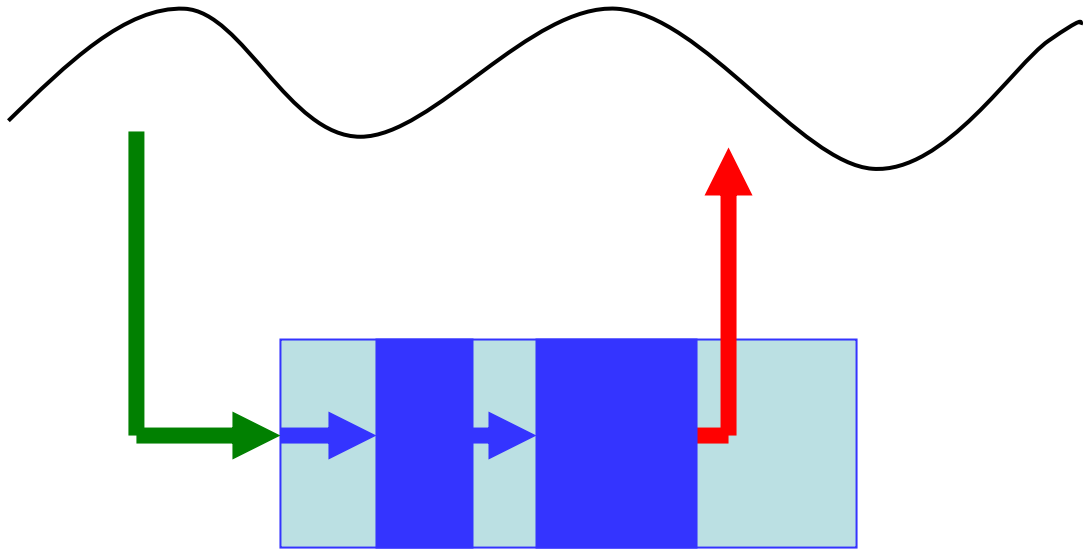


Time and Value Determinism



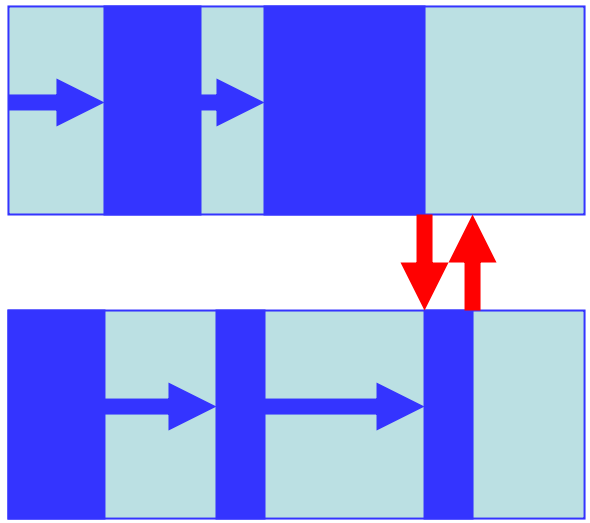
Timing predictability: minimal jitter
Value predictability: no data races

Contrast LET with Scheduled Programming Model



make output available
as soon as ready

Contrast LET with Scheduled Programming Model



data race

Task output values depend
on which task finishes first!

HTL Project: Deterministic Reliability

Can we build a programming language that treats reliability in the way in which high-level languages treat memory?

-programmer specifies the long-run failure rate of outputs

-programmer assumes the platform offers sufficient reliability

-compiler generates a task replication mapping or rejects the program

Program:

write actuator y every 4 ms
with failure rate $\leq$ 0.001;

LOGICAL RELIABILITY

Program:

write actuator y every 4 ms
with failure rate $\leq$ 0.001;

LOGICAL RELIABILITY

Platform:

CPU reliability 0.97;
sensor reliability 0.95;

PHYSICAL RELIABILITY

Program:

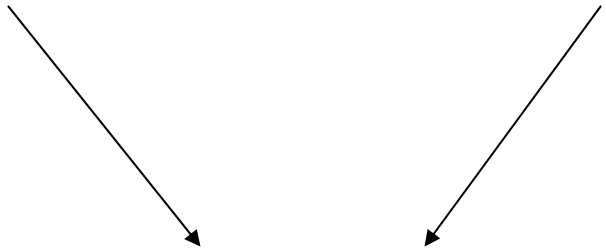
write actuator y every 4 ms
with failure rate $\leq$ 0.001;

Platform:

CPU reliability 0.97;
sensor reliability 0.95;

LOGICAL RELIABILITY

PHYSICAL RELIABILITY



Compiler replicates computation of actuator value on 2 CPUs:
$(1 - 0.97)^2 \leq 0.001$

Challenge 2:
Build Robust Systems

Challenge 2:
Build Robust Systems

Robust = Continuous

A system is *continuous* if for all real-valued quantities, small input changes cannot cause large output changes.

Challenge 2:
Build Robust Systems

Robust = Continuous

A system is *continuous* if for all real-valued quantities, small input changes cannot cause large output changes.

- $\forall \varepsilon{>}0.\ \exists \delta{>}0.$ input-change $\leq \delta \Rightarrow$ output-change $\leq \varepsilon$

- can apply only to real-valued quantities: sensor readings and actuator settings; time stamps; probabilities

In general programs are not continuous.
But they can be more continuous:

```
read sensor value x at time t;
compute “continuous” function y = f(x);
write output value y at time t+d;
```

Or less continuous:

```
read sensor value x;
if x ≤ c  then y = f1(x)
          else y = f2(x);
```

In general programs are not continuous.
But they can be more continuous:

```
read sensor value x at time t;
compute “continuous” function y = f(x);
write output value y at time t+d;
```

Or less continuous:

```
read sensor value x;
if x ≤ c  then y = f1(x)
          else y = f2(x);
```

Better:

```
if x ≤ c - ε   then y = f1(x);
if x ≥ c + ε   then y = f2(x)
               else y = (f1(x) + f2(x))/2;
```

In general programs are not continuous.
But they can be more continuous:

```
read sensor value x at time t;
compute “continuous” function y = f(x);
write output value y at time t+d;
```

Or less continuous:

```
read sensor value x;
if x ≤ c  then y = f1(x)
          else y = f2(x);
```

Better:

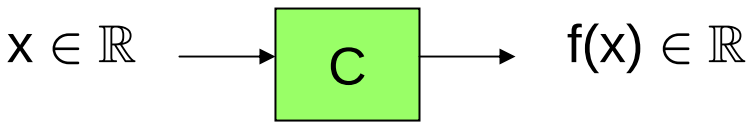
```
if x ≤ c - ε   then y = f1(x);
if x ≥ c + ε   then y = f2(x)
                    else y = (f1(x) + f2(x))/2;
```

We need system preference metrics in addition to boolean correctness criteria.

What is an Observation?

Traditional software:

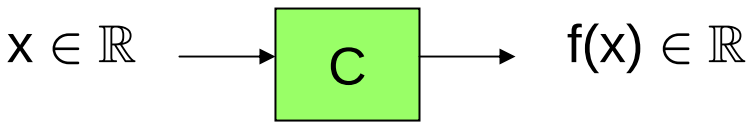
Sensor software:



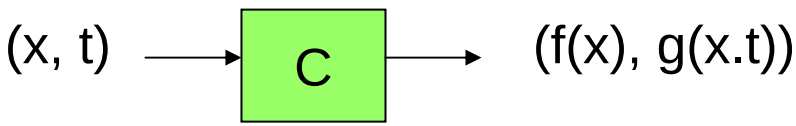
What is an Observation?

Traditional software:

Sensor software:

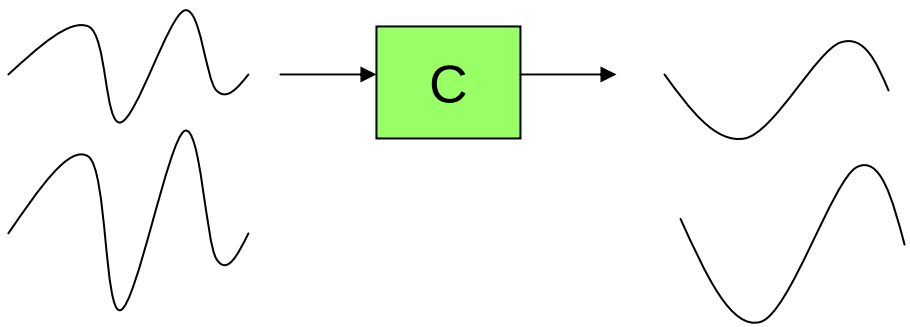


Real-time software:



Topology on Observations

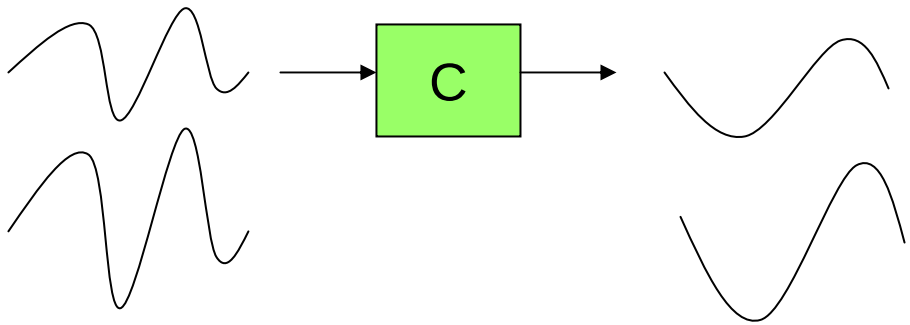
Real time:



e.g. Skorohod
metric on signals

Topology on Observations

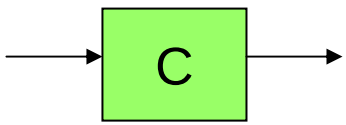
Real time:



e.g. Skorohod
metric on signals

Reliability:

sensor value
validity 0.99



actuator value
validity 0.95

Conclusion

Topology of observations *T* should be an essential part of every system specification:

1. Determinism of a model depends on *T*.
2. Continuity of a model depends on *T*.

Different choices of *T* will lead to different designs.

Conclusion

Topology of observations *T* should be an essential part of every system specification:

1. Determinism of a model depends on *T*.
2. Continuity of a model depends on *T*.

Different choices of *T* will lead to different designs.

The choice of modeling language should depend on the choice of *T*.

Research Challenge: For given *T*, we need languages that guarantee determinism and continuity.