AnaTempura

Antonio Cau, Shikun Zhou and Hussein Zedan

Software Technology Research Laboratory
De Montfort University

2008

STRL Overview (2)

This talk will introduce Interval Temporal Logic and its Runtime verification tool AnaTempura

Introduction (3)

Assuring the correctness of systems, is a difficult task.

- Complexity of the system.
- Size of the system.
- Requirements that need to be satisfied.

Introduction (4)

Several techniques have been proposed to assure the correctness of systems:

- testing (simulators, debuggers, etc.),
- formal methods, (theorem provers, proof checkers and model checkers),
- runtime verification (simulators + theorem checkers).

Fundamentals (5)

Notions

- System: collection of communicating processes
- State: variables and communication links
- Behaviour: sequence of states (interval)
- Property: set of behaviours

Fundamentals (6)

Interval Temporal Logic

- First order logic: variables, constants, functions and predicates.
- Temporal constructs: skip, ";" (chop) and "*" (chopstar).

Expressions

$$e ::= \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \bigcirc e \mid \text{fin } e$$

Formulae

$$f ::= p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \vee f_2 \mid \exists v \bullet f \mid \text{skip} \mid f_1 ; f_2 \mid f^*$$

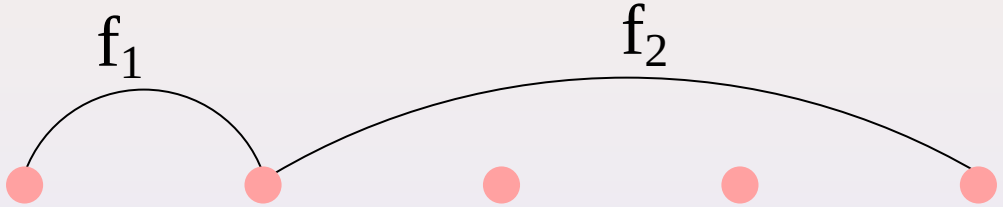
Fundamentals

Intuition:

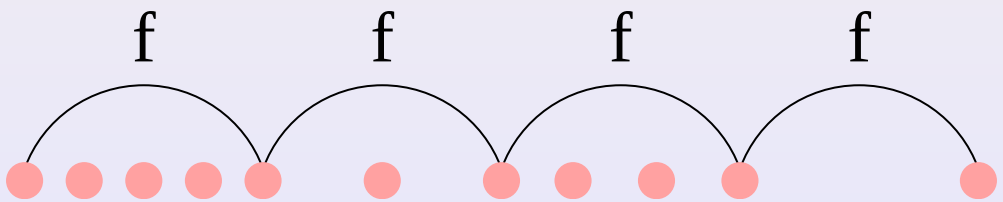
Interval is a sequence of states

- skip :

- $f_1 ; f_2$:

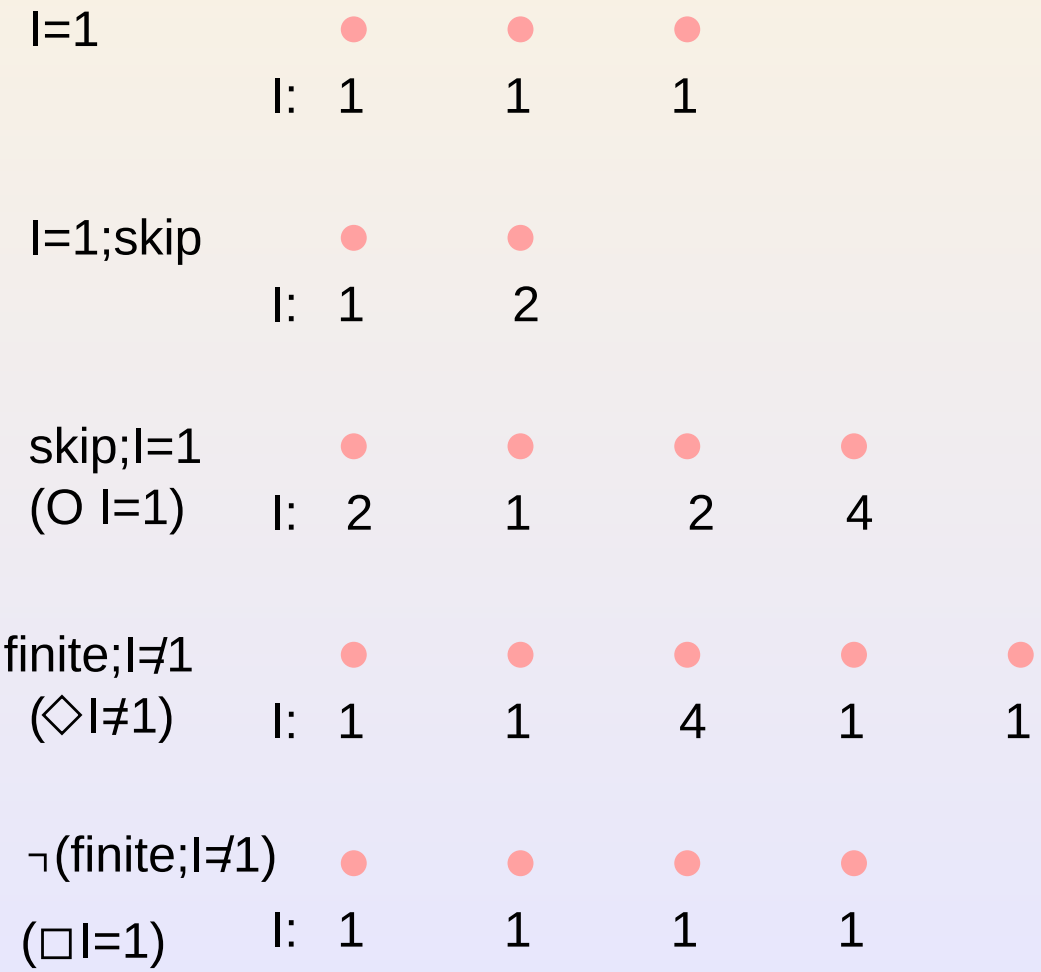


- f^* :



Fundamentals (8)

Some sample ITL formulae:



Fundamentals (9)

Tempura: executable subset of ITL.
A formula is executable if

- it is deterministic,
- the length of the corresponding interval is known.
- the values of the variables (of the formula) are known throughout the corresponding interval.

Tempura interpreter:
takes a Tempura formula and constructs the corresponding sequence of states, i.e. interval, using the following rewrite technique:

$$f \equiv \textbf{present\_state} \wedge \textcircled{w}\textbf{what\_remains}$$

Example

$$\bigcirc f \equiv \text{more} \wedge \textcircled{w} f$$
$$\Box f \equiv f \wedge \textcircled{w} \Box f$$

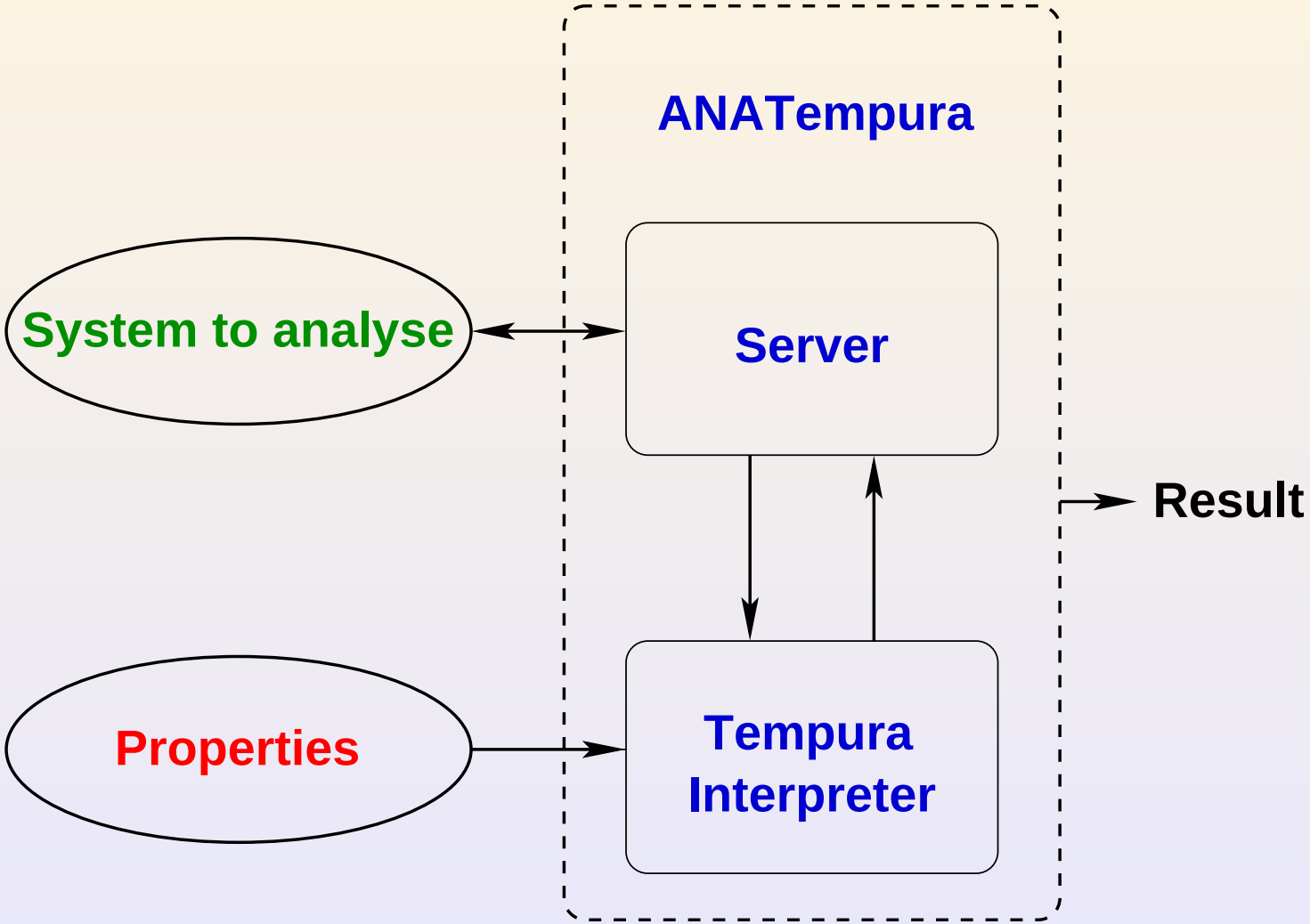
Some examples of executable formulae:

$$\text{skip} \wedge \boldsymbol{I = 0} \wedge \bigcirc(\boldsymbol{I = 1})$$
$$\text{len}\,\mathbf{(3)} \wedge \Box(\boldsymbol{I = 2})$$
$$\text{len}\,\mathbf{(4)} \wedge \boldsymbol{I = 0} \wedge \boldsymbol{I} \text{ gets } \boldsymbol{I + 1}$$

Some examples of non-executable formulae:

$$\text{empty} \wedge (\boldsymbol{I = 0} \vee \boldsymbol{I = 1})$$
$$\boldsymbol{I = 0 \,;\, I = 1}$$
$$\text{len}\,\mathbf{(4)} \wedge \boldsymbol{I} \text{ gets } \boldsymbol{I + 1}$$

Runtime Verification (11)



$$\Box(\mathit{input}\ X_{prog} \wedge \mathit{Check}(X_{prog}, X_{prop})) \wedge \mathit{Prop}(X_{prop})$$

Runtime Verification (12)

Runtime Verification:

1. Establish all desirable system properties (functional, timing, resource, ...)
2. Insert at suitable places in the source code of the system assertion points
3. Use AnaTempura to check that the generated behaviour satisfies the desired properties

Runtime Verification (13)

Assertion points ⟶ behaviour.
Example: Computation of factorial.

```
main()
{  long y, fac=1;
   assertion("fac", fac);
    text_out("Enter the seed:");
    scanf("%d",&y); assertion("y", y);
     while (y>1) {
       fac=fac*y;
        assertion("fac", fac);
       y=y-1;
        assertion("y", y);  } }
```

Runtime Verification (14)

Will generate the following sequence of changes (seed is 4):

```
!PROG: assert fac:1:!
!PROG: assert y:4:!
!PROG: assert fac:4:!
!PROG: assert y:3:!
!PROG: assert fac:12:!
!PROG: assert y:2:!
!PROG: assert fac:24:!
!PROG: assert y:1:!
```

The corresponding behaviour (interval):

Runtime Verification (15)

Where to put assertion points?

- Location of assertion points is determined by the variables used in our property of interest.
- Simple search through the source code will locate all places where the variables are changing.
- We will place the assertion points directly after those “places of change”.
- This will ensure that behaviour generated during the runtime of the system is indeed the “correct” behaviour.

Runtime Verification (16)

The property we want to check:
(Invariant of the while loop)

```
define seed = 1 + (random mod 10).
define fac_rel(Y) = {
     if Y=seed  then Y
               else Y * fac_rel(Y+1)    }.
```

The check:

```
define test() = {
  exists Fac,Y : {
    prog_send(seed);
    check(1,Fac,Y);
     if seed>1  then {
      while Y>1  do
          check(fac_rel(Y),Fac,Y)
    }  else  empty   }   }.
```

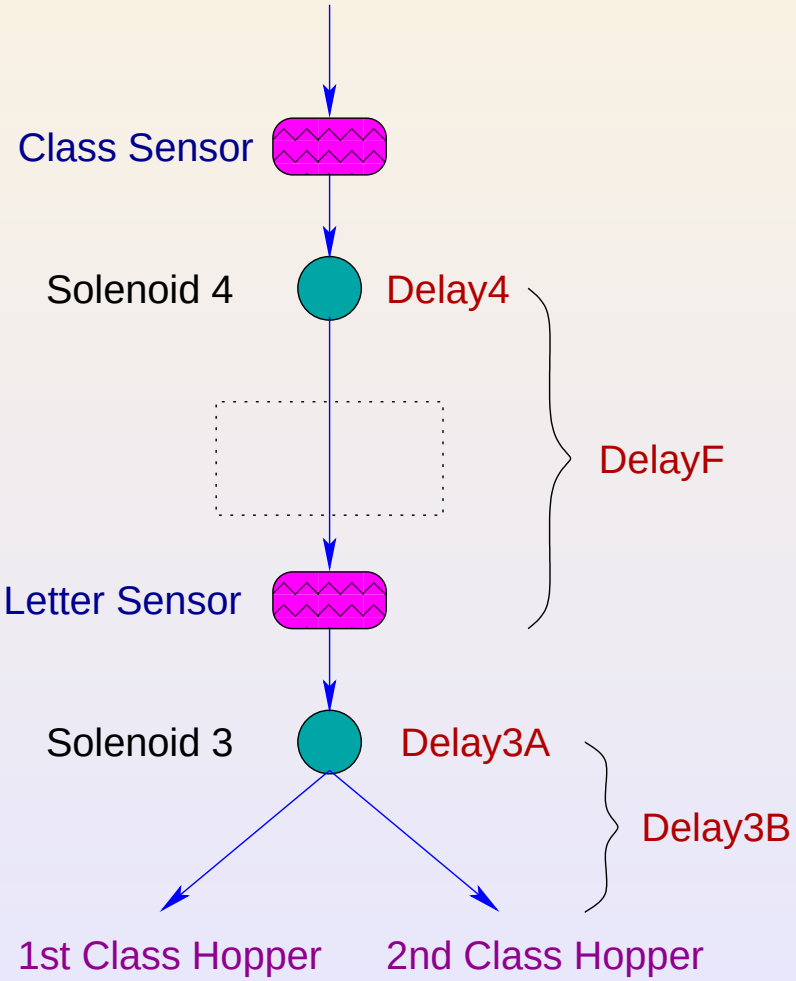
Runtime Verification (17)

Sample test run (seed is 10):

```
Tempura 1> run test().
!:: Pass Prog      10 Prop      10
!:: Pass Prog      90 Prop      90
!:: Pass Prog     720 Prop     720
!:: Pass Prog    5040 Prop    5040
!:: Pass Prog   30240 Prop   30240
!:: Pass Prog  151200 Prop  151200
!:: Pass Prog  604800 Prop  604800
!:: Pass Prog 1814400 Prop 1814400
!:: Pass Prog 3628800 Prop 3628800
Done!  Computation length:  30.
Total Passes:  31.
Total reductions: 4164(4164 successful).
Maximum reduction depth:  31.
```

Applications

Letter sorter: 2 sensors and 2 solenoids.



STRL Applications (19)

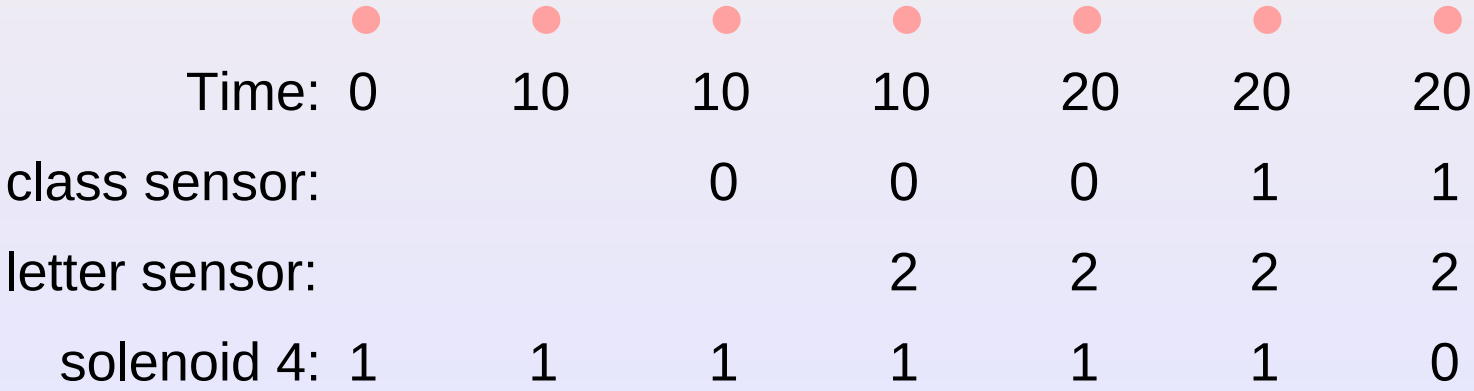
```
scan_csensor (&class_sensor);
 assertion("class", class_sensor);
if (class_sensor < 2)
   { SolOff(4); Delay(delay4,1);
     SolOn(4);  Delay(delayF,4);
     scan_lsensor (&letter_sensor);
      assertion("lsens",letter_sensor);
     if ( !YellowSet )
       { Delay(delay3A,2); SolOff(3);
         Delay(delay3B,3); YellowSet = 1; }
} else {
     SolOff(4); Delay(delay4,1);
     SolOn(4); Delay(delayF,4);
     scan_lsensor (&letter_sensor);
      assertion("lsens",letter_sensor);
     if ( YellowSet )
       { Delay(delay3A,2); SolOn(3);
         Delay(delay3B,2); YellowSet = 0; }  }
```

Applications

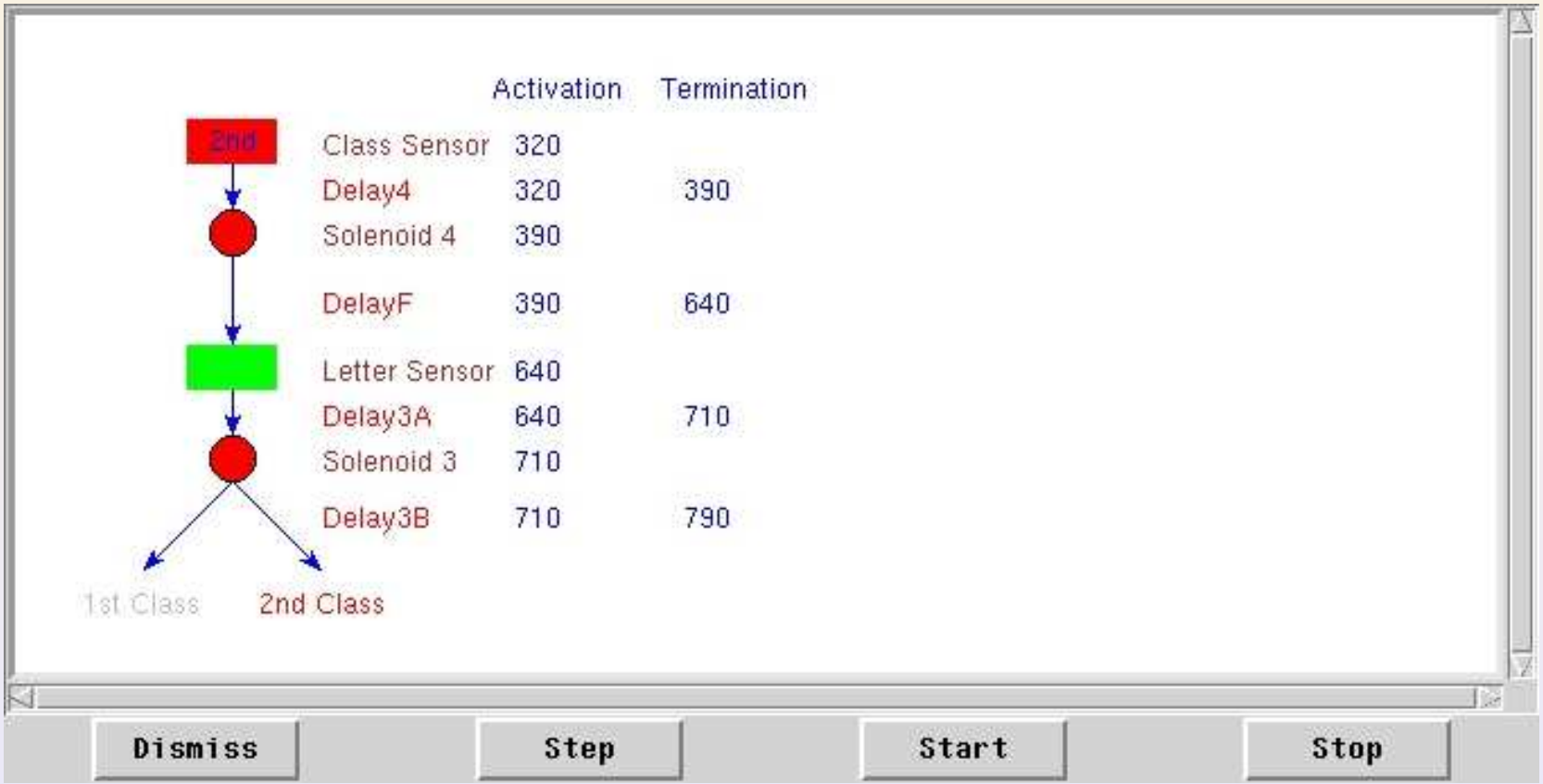
Time is modelled as a variable:

```
!  0:: Solenoid 4 ON: Pass
! 10:: Class sensor 0: Pass
! 10:: Letter sensor 2: Pass
! 20:: Class sensor 1: Pass
! 20:: Solenoid 4 OFF: Pass
. . .
```

corresponds to following interval:



Time:	0	10	10	10	20	20	20
class sensor:			0	0	0	1	1
letter sensor:				2	2	2	2
solenoid 4:	1	1	1	1	1	1	0


Activation
Termination
2nd
Class Sensor 320
Delay4 320 390
Solenoid 4 390
DelayF 390 640
Letter Sensor 640
Delay3A 640 710
Solenoid 3 710
Delay3B 710 790
1st Class
2nd Class
Dismiss
Step
Start
Stop

Applications

Bridging the Abstraction Gaps in Verilog

1. Highest level:
   Cycle-Accurate behavioral description
2. Intermediate level:
   Finite State Machine and Datapath
3. Lowest level:
   Combinational module description

Goal: move between the levels in a formal way
Necessary: ITL semantics for each level

Applications

Use AnaTempura to determine the semantics

The idea is as follows:

1. Insert assertion points in the Verilog specification.
2. Describe the semantics of the Verilog specification in Tempura code.
3. Use AnaTempura to check whether they match.

Conclusion

- Runtime verification is an integration of testing and formal verification.
- Our runtime verification approach doesn't suffer from the state explosion problem but it doesn't perform complete tests.