

# AnaTempura

Antonio Cau, Shikun Zhou and Hussein Zedan

Software Technology Research Laboratory  
De Montfort University

2008

The logo for the Software Technology Research Laboratory (STRL) is displayed in a stylized, blue, italicized serif font. The letters are bold and have a slight shadow effect, giving them a three-dimensional appearance. The logo is centered on the page.

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

- 1 Introduction
- 2 Fundamentals
- 3 Runtime Verification
- 4 Applications

Assuring the correctness of systems, is a **difficult** task.

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

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).

## Notions

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

## Interval Temporal Logic

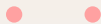
- First order logic: **variables**, **constants**, **functions** and **predicates**.
- Temporal constructs: **skip**, **“;”** (chop) and **“\*”** (chopstar).

*Expressions*
$$e ::= \mu \mid a \mid A \mid g(e_1, \dots, e_n) \mid \circ e \mid \text{fin } e$$
*Formulae*
$$f ::= p(e_1, \dots, e_n) \mid \neg f \mid f_1 \vee f_2 \mid \exists v \cdot f \mid \\ \text{skip} \mid f_1 ; f_2 \mid f^*$$

Intuition:

**Interval** is a sequence of states

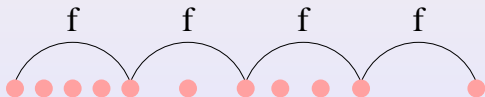
- skip :



- $f_1 ; f_2$  :



- $f^*$  :



Some sample ITL formulae:

$l=1$                     ●            ●            ●  
 l: 1            1            1

$l=1; \text{skip}$             ●            ●  
 l: 1            2

$\text{skip}; l=1$             ●            ●            ●            ●  
 ( $\bigcirc l=1$ )    l: 2            1            2            4

$\text{finite}; l \neq 1$             ●            ●            ●            ●            ●  
 ( $\diamond l \neq 1$ )    l: 1            1            4            1            1

$\neg(\text{finite}; l \neq 1)$             ●            ●            ●            ●  
 ( $\square l=1$ )    l: 1            1            1            1

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:

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

## Example

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

$$\square f \equiv f \wedge \textcircled{w} \square f$$

Some examples of **executable** formulae:

$\text{skip} \wedge I = 0 \wedge \bigcirc(I = 1)$

$\text{len}(3) \wedge \square(I = 2)$

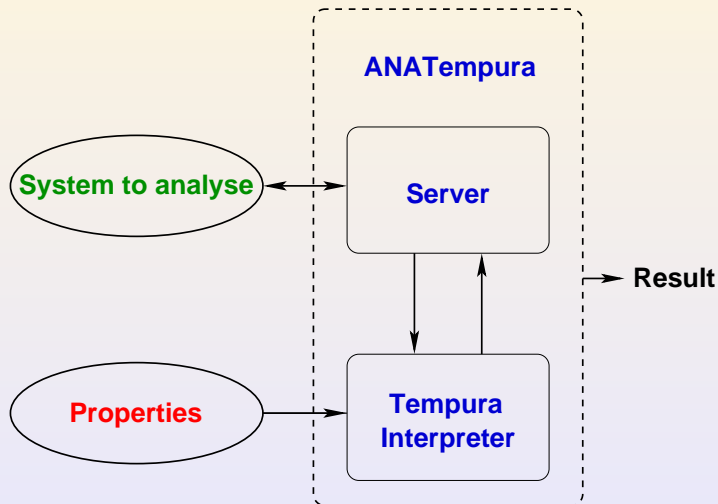
$\text{len}(4) \wedge I = 0 \wedge I \text{ gets } I + 1$

Some examples of **non-executable** formulae:

$\text{empty} \wedge (I = 0 \vee I = 1)$

$I = 0 ; I = 1$

$\text{len}(4) \wedge I \text{ gets } I + 1$



$$\square(\text{input } X_{prog} \wedge \text{Check}(X_{prog}, X_{prop})) \wedge \text{Prop}(X_{prop})$$

## 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**

Assertion points  $\longrightarrow$  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);  } }
```

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):

	●	●	●	●	●	●	●
y: ?	4	4	3	3	2	2	1
fac: 1	1	4	4	12	12	24	24

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.

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)    } .
```

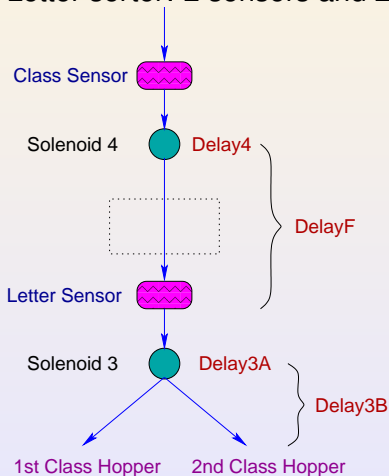
The check:

```
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    }    } .
```

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.
```

Letter sorter: 2 sensors and 2 solenoids.










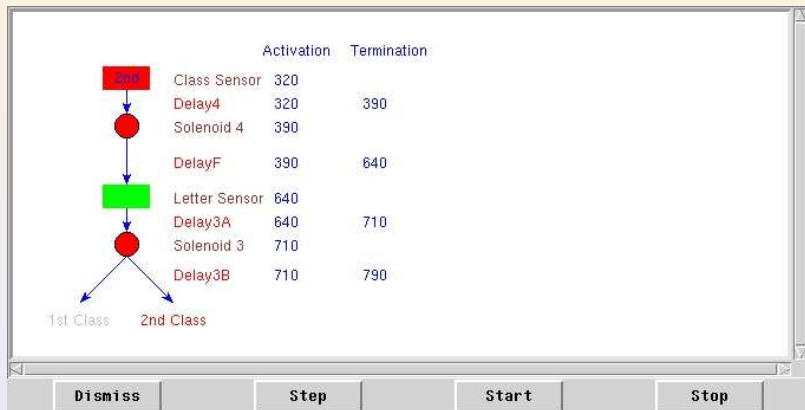
```
scan_csensor (&class_sensor);
  assertion("class", class_sensor);
if (class_sensor < 2)
  { SolOff(4); Delay(delay4,1);
    SolOn(4); Delay(delayF,4);
    scan_lsensor (&letter_sensor);
    assertion("lsens",letter_sensor);
    if ( !YellowSet )
      { Delay(delay3A,2); SolOff(3);
        Delay(delay3B,3);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

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



## 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

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.

- 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.