A Calculus For Evolution

Hussein Zedan, Antonio Cau and Shikun Zhou

Software Technology Research Laboratory

Introduction

- Problems
- Formal Preliminaries
- A Logic-based Formal Approach
- Introduction to the Formal Tool: ANATempura
- A Case Study
- Conclusions and Future Work

Managing System Evolution

- Continual evolution of computing systems will inevitably lead to their rapid growth in size and increased complexity.
- Software evolution is due to *changes*. Because of the complexity of evolutionary changes, the likelihood of subtle errors is much greater and some of these errors could have catastrophic consequences such as loss of life, money, time or damage to the environment.
- How to response to '*change*':
  undertaken *rapidly, efficiently* and *correctly*.

Managing Change

- Using formal methods is fundamental in managing changes of critical applications.

- Techniques, like programming slicing, both static and dynamic, and data clustering, are also useful to handle changes.

- Another important issue in managing change is to establish mechanisms to cope with its propagation.

Preliminaries: A Quick View of ITL

Expressions

$$e ::= \quad \mu \mid a \mid A \mid g(e_1, \ldots, e_n) \mid \imath a{:}\, f$$

Formulae

$$f ::= \quad p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid$$
$$\forall v \bullet f \mid \mathsf{skip} \mid f_1 \,;\, f_2 \mid f^*$$

Examples of ITL Formula

$$\Diamond[(I = 1) \wedge \Diamond(I = 2)]$$

$$\models \quad [\Box(I = 1) \wedge \bigcirc(J = 2)]$$

$$\supset \quad \bigcirc(I + J = 3)$$

$$\models \ [(K + 1 \rightarrow K)\,;(K + 2 \rightarrow K)]$$

$$\supset (K + 3 \rightarrow K)$$

Preliminaries: Abbreviations

next $\bigcirc f$	$\widehat{=}$	skip ; f
wnext $\textcircled{w} f$	$\widehat{=}$	$\neg \bigcirc \neg f$
non-empty interval *more*	$\widehat{=}$	$\bigcirc$*true*
empty interval empty	$\widehat{=}$	$\neg$*more*
infinite interval *inf*	$\widehat{=}$	*true* ; *false*
finite interval *finite*	$\widehat{=}$	$\neg$*inf*
sometimes $\Diamond f$	$\widehat{=}$	*finite* ; f
always $\Box f$	$\widehat{=}$	$\neg \Diamond \neg f$
if then else if f_0 then f_1 else f_2	$\widehat{=}$	$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$
final state *fin* f	$\widehat{=}$	$\Box$(empty $\supset$ f)
some subinterval $\textcircled{4} f$	$\widehat{=}$	*finite* ; f ; *true*
all subintervals $\boxed{4} f$	$\widehat{=}$	$\neg(\textcircled{4} \neg f)$
all unit subintervals *keep* f	$\widehat{=}$	$\boxed{4}$(skip $\supset$ f)
while loop while f_0 do f_1	$\widehat{=}$	$(f_0 \wedge f_1)^* \wedge$ *fin* $\neg f_0$
next value $\bigcirc exp$	$\widehat{=}$	$\imath a$: $\bigcirc(exp = a)$
end value *fin* exp	$\widehat{=}$	$\imath a$: *fin* $(exp = a)$
assignment $A := exp$	$\widehat{=}$	$\bigcirc A = exp$
temporal assignment $exp_1 \leftarrow exp_2$	$\widehat{=}$	*finite* $\wedge$ (*fin* exp_1) $= exp_2$
gets exp_1 *gets* exp_2	$\widehat{=}$	*keep* $(exp_1 \leftarrow exp_2)$
stability stable exp	$\widehat{=}$	exp *gets* exp

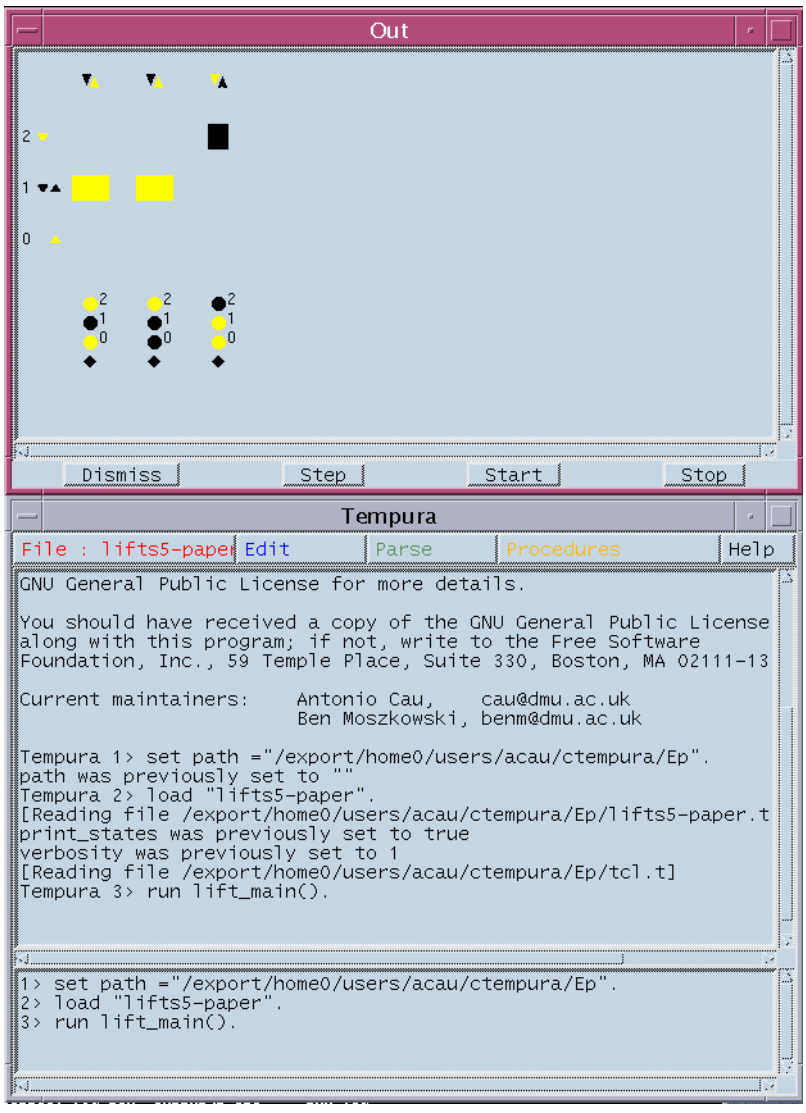
ITL and Tempura

ITL	$Tempura$
$f_1 \wedge f_2$	f_1 *and* f_2
$A := exp$	$A := exp$
more	*more*
empty	empty
$\Diamond$	*sometimes*
$\Box$	*always*
true	*true*
false	*false*
if b then f_1 else f_2	if b then f_1 else f_2
while b do f	while b do f
repeat b until f	repeat b until f
"procedures"	*define* $p(e_1, \ldots, e_n) = f$
"functions"	*define* $g(e_1, \ldots, e_n) = e$

An Example of Tempura Code

```
/* sub */ define demo1(X,Y) = {
 len(5) and
 always( if X<0 then Y=-1
                else if X=0 then Y=0
                            else Y=1)
}.

/* sub */ define demo2(X,Y) = {
 len(5) and
 always( if X=0 then Y=0
                else if X>0 then Y=1
                            else Y=-1)
}.

/* run */ define test1() = {
  exists X,Y : {
 always (input X and format("Y=%t \n",Y) )
         and demo1(X,Y) }
}.

/* run */ define test2() = {
  exists X,Y : {
 always (input X and format("Y=%t \n",Y) )
         and demo2(X,Y) }
}.
```

Tempura Tool



Computation

- A legacy *system* is a collection of *agents* , possibly executing concurrently and communicating (a)synchronously via communication links. Systems can themselves be viewed as single agents and composed into larger systems.
- Systems may have timing constraints imposed at three levels; system wide communication deadlines, agent deadlines and sub-computation deadlines (within the computation of an individual agent).
- At any instant in time a system can be thought of as having an unique *state*. The system state is defined by the state variables of the system and, for concurrent system, by the values in the communication links, the so called *frame*.

- *Computation* is defined as any process that results in a change of system state. An agent is described by a computation which may transform a private data-space and may read and write to communication links during execution.
- The computation may have both minimum and maximum execution times imposed.
- A local data-space for the agent is created when an agent starts execution with initial values which are nondeterministic. The data-space is destroyed when the agent terminates. No agent may read or write another agent's data-space.

Behaviour, Property and Satisfaction

A Behaviour: a sequence of states, i.e. an interval σ in ITL, which could be finite or infinite.

- A *full* behaviour: contains all the state variables of the system.
- A *partial* behaviour: contains a part of the state variables and can be obtained by hiding some state variables (formally it is a projected behaviour over state variables).

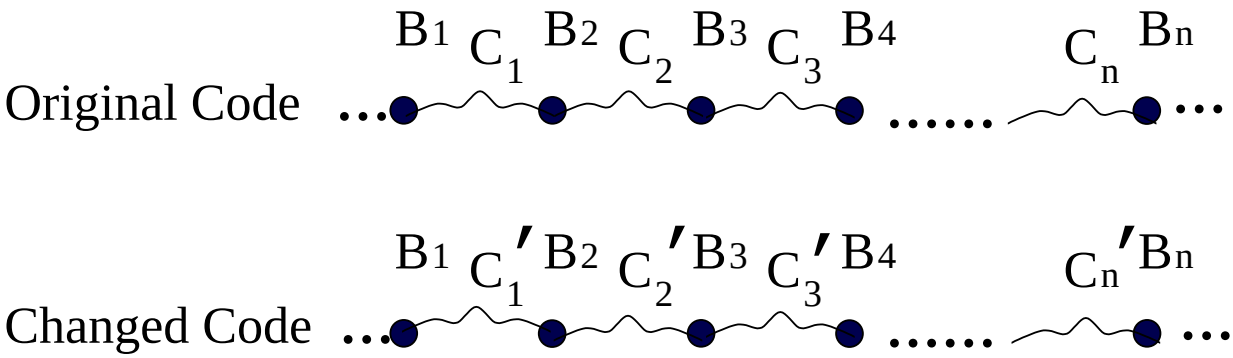
A Property P can either be a *state* or *Temporal* ITL formula.

A property describes a set of behaviours.

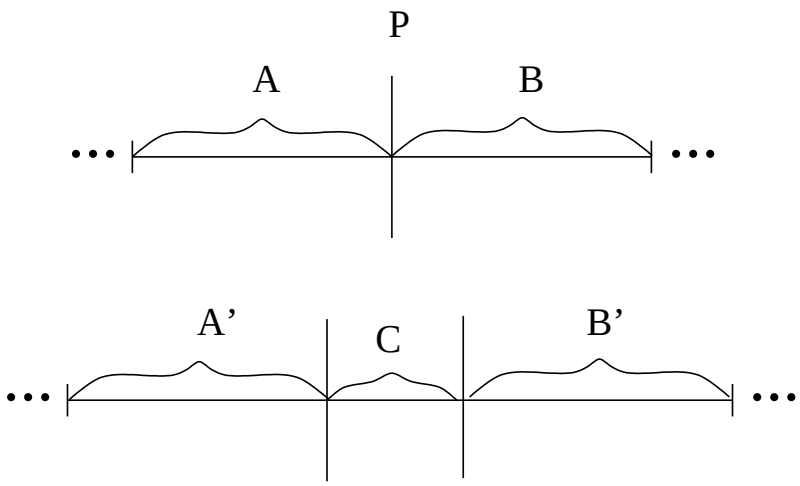
- *Safety Properties:* Something bad does not happen,
- *Liveness Properties:* Something good will eventually happen.

Satisfaction: is achieved by checking whether a captured system's behaviour is an element of a set of behaviours, which are included in a given property.

Assertion-Point



B: Assertion-points
C: Code Chunks



A’ and B’ are the Enviroment of C
A, A’, B, B’ and C are code chunks

A Logic-Based Formal Approach

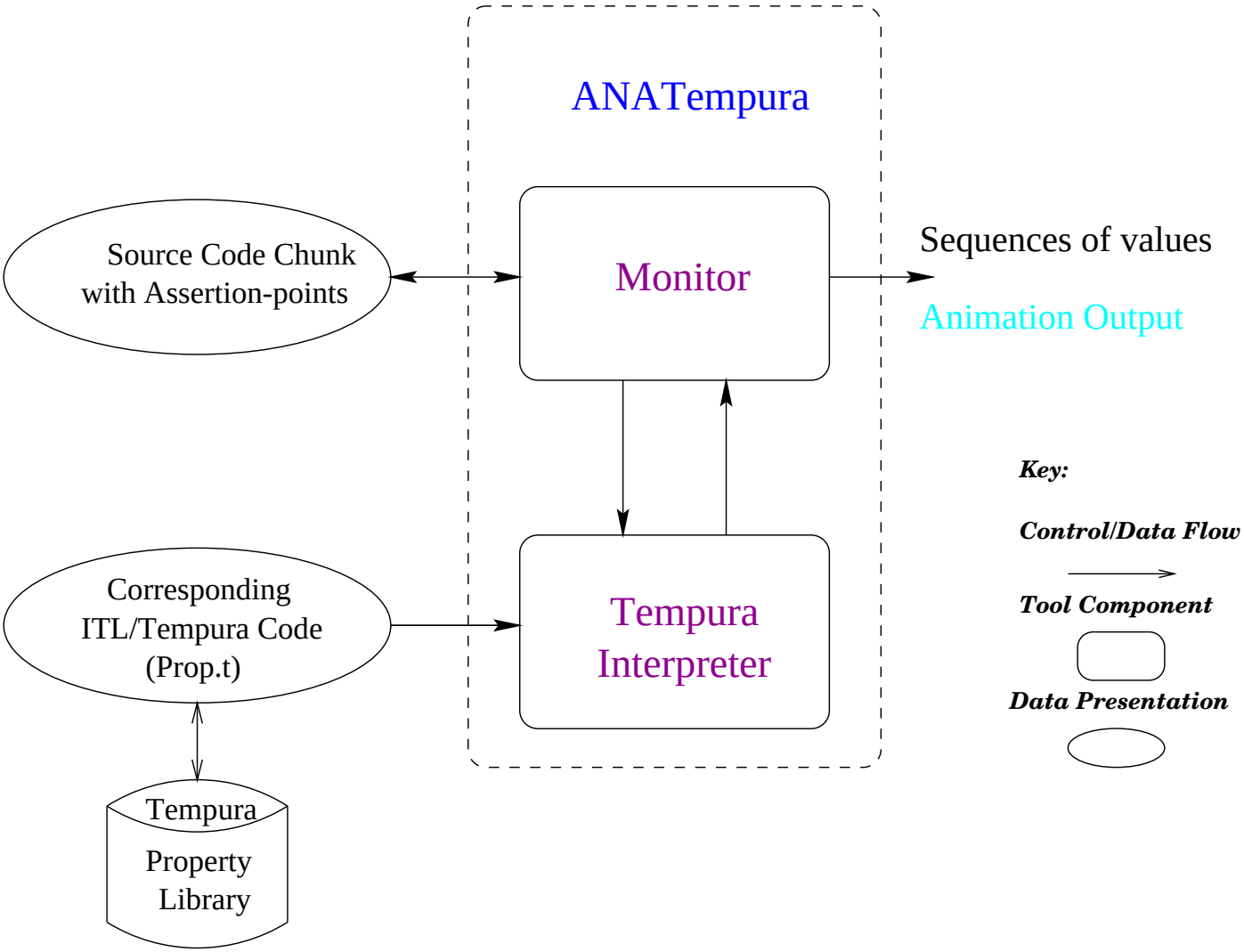
Aim: Handing *continuous* **change in system development and maintenance**

1. Establish all desirable properties of the system under consideration and expressed them in Tempura.
2. Identify suitable locations and insert assertional points.
3. Using the Tempura, check that the behaviour satisfies the desired properties.

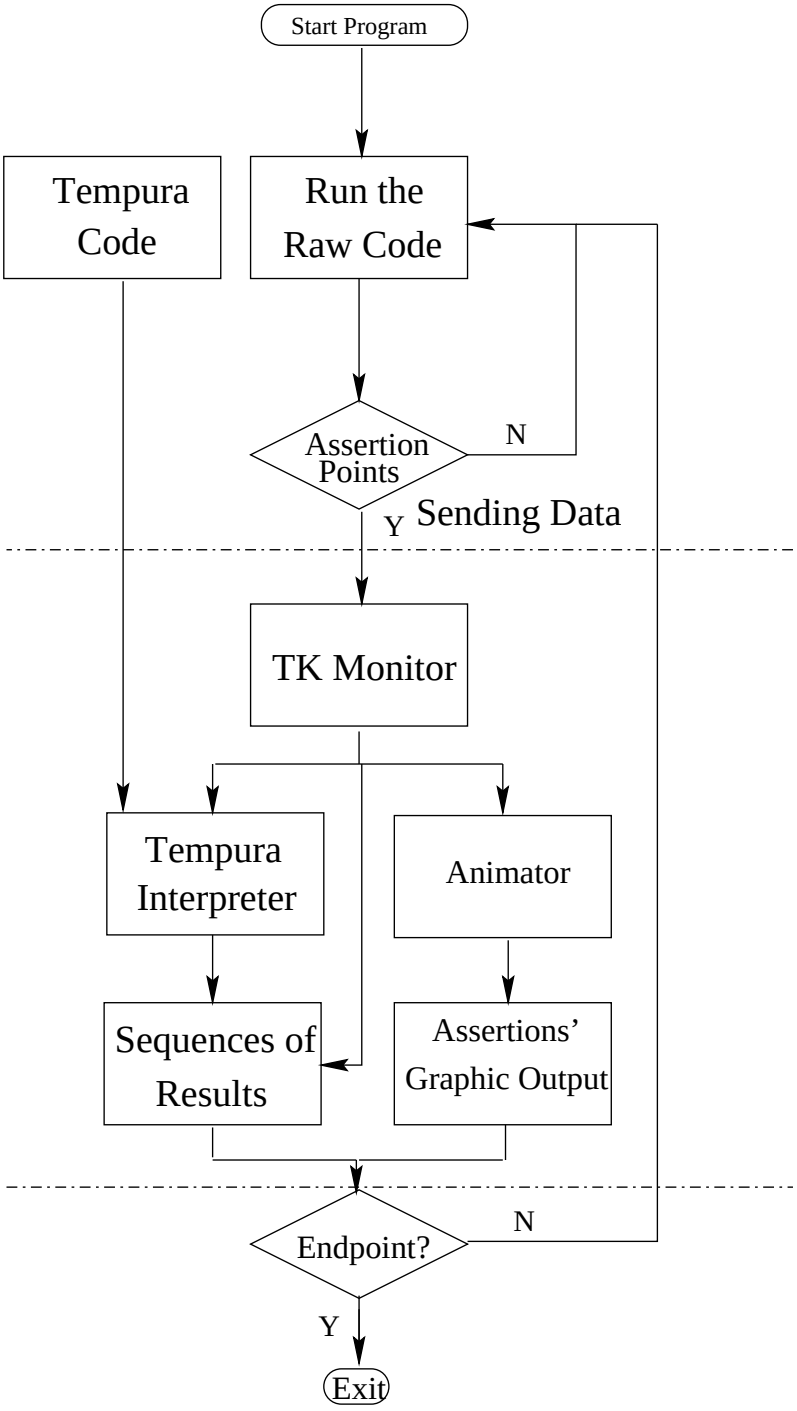
The General Use of ANATempura

- Automatically validating the assertions about the program behaviour as the time progresses during executing the program,
- Specifying timing requirements visually,
- Supplying a comprehensive test suite,
- Validating the most complex parts of a real-time system automatically,
- As aids for re-engineering and comprehension, capturing and validating the major properties of legacy systems.

The Formal Tool: ANATempura



Working Flow of the ANATempura



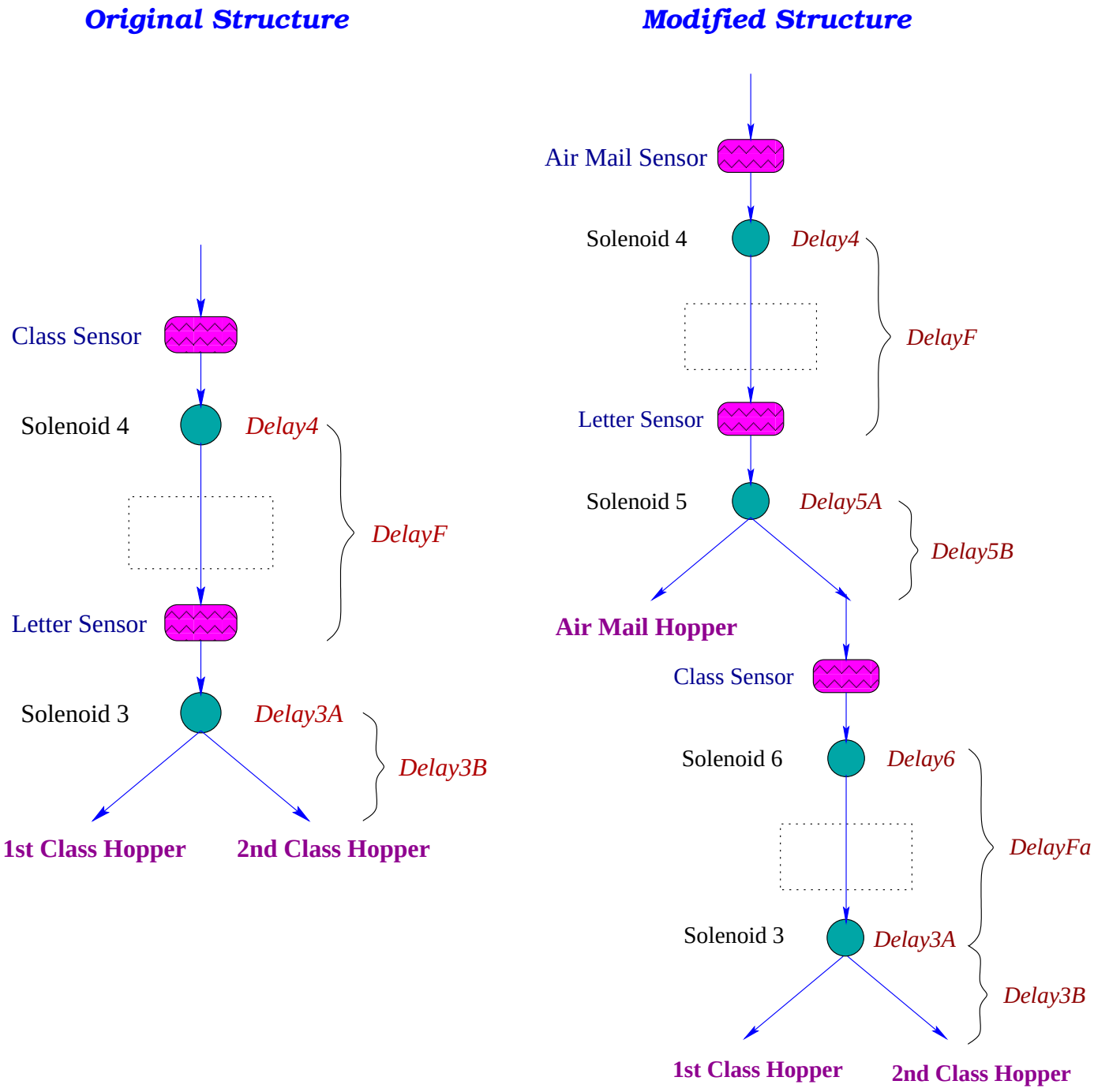
A Working Example: Factorial in C

```
#include <stdio.h>

/* Factorial tester */

main()
{
int y, fac=1;

    printf ("Enter the seed: \n");
    scanf ("%d", &y);
    while (y>0) {
        fac=1;
        while (y>1) {
          fac=fac*y;
          y=y-1;
          printf ("PROG: break
          fac:d:%d::\n", fac);       (*)
        }
        printf ("Enter the seed: ");
        scanf ("%d", &y);
    }
    printf ("PROG: end ::\n");
 }
```

A Working Example: Factorial in Tempura

```
load "tcl".
/* prog fac */
set print_states = false.

define fc = 7.

/* sub */ define check(X,Y) = {
  if strint(suf(X,6))= Y then {
    format("Pass test\n")
  } else {
     format("Fac: Prog %d Real %d\n",
       strint(suf(X,6)),Y)
    }
}.

/* sub */ define fac(K) = {
   if K=0 then fc
          else (fc - K) * fac(K-1)
}.

/* run */ define test() = {
  exists X,Z : {
    if fc>1 then {
     prog_send(fc); skip;
     {len(fc-2) and  Z=0 and
      always (input X and check(X,fac(Z))) and
      Z gets Z+1
     }
    } else { prog_send(fc) }
  }
}.
```

A Case Study: Mail Sorter



An Example Formula

$$Cs\_controller \mathrel{\widehat{=}} [75]\, Cs\_check\_send$$

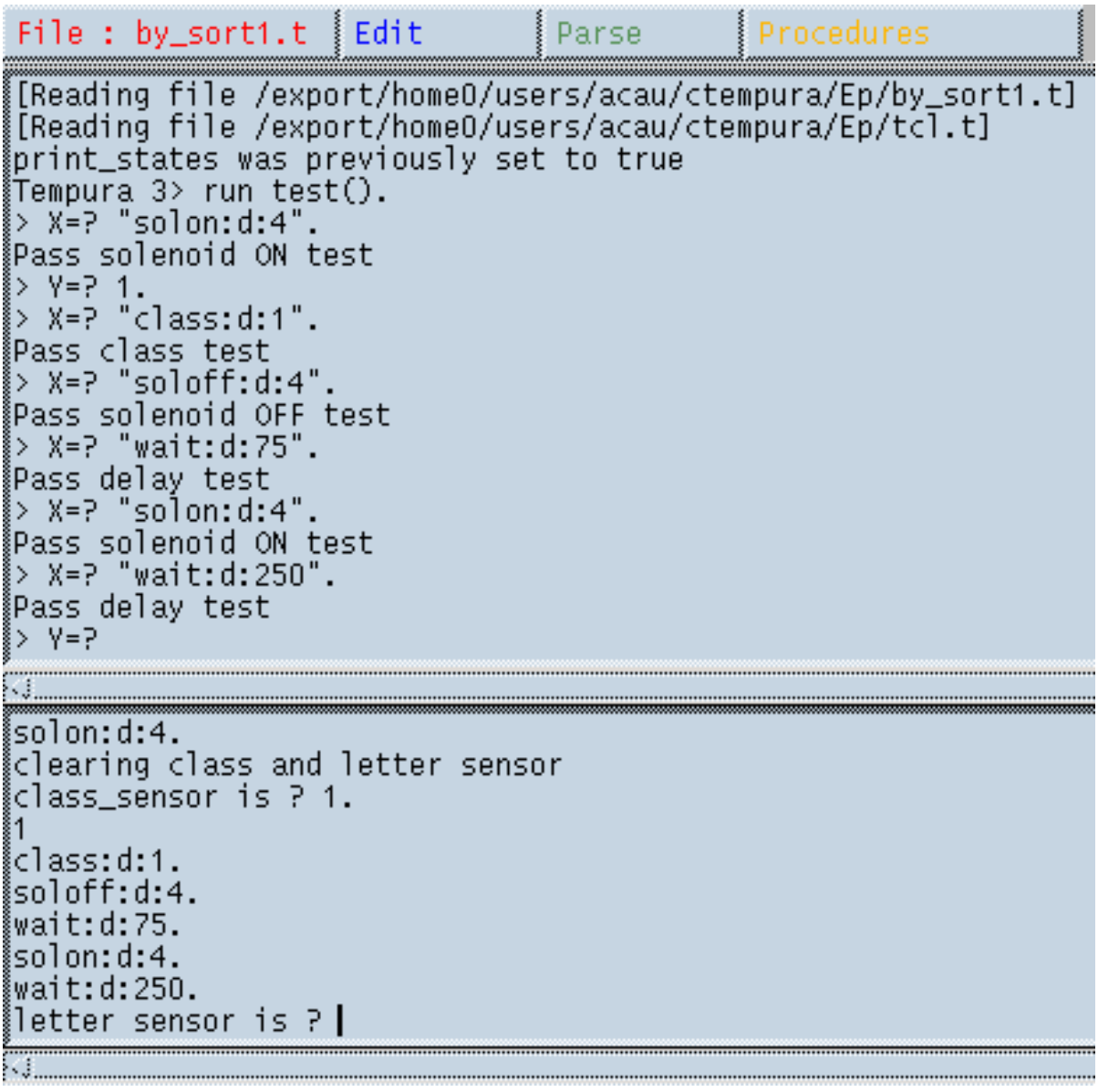
$$[t]\, S \mathrel{\widehat{=}} \mathit{len} = t \wedge (S\ ;\ \mathit{true}) \wedge (S \supset \mathit{len} \leq t)$$

$$\begin{aligned}
Sorter \mathrel{\widehat{=}} \quad & (\\
& (\mathit{len} = 235 \wedge \\
& (Sensor = Air \supset \\
& \Diamond(Solenoid4 = ON)) \\
&) \vee \\
& (Sensor \neq Air \supset \\
& \Diamond(Solenoid4 = OFF) \wedge \\
& (\mathit{len} = 235 \wedge \\
& (Sensor = 0 \supset \\
& \Diamond(Solenoid3 = OFF)) \vee \\
& (Sensor = 1 \supset \\
& \Diamond(Solenoid3 = ON)) \vee \\
& (Sensor = 2 \supset \\
& \Diamond(Solenoid3 = OFF)) \\
&) \\
&) \\
&)
\end{aligned}$$

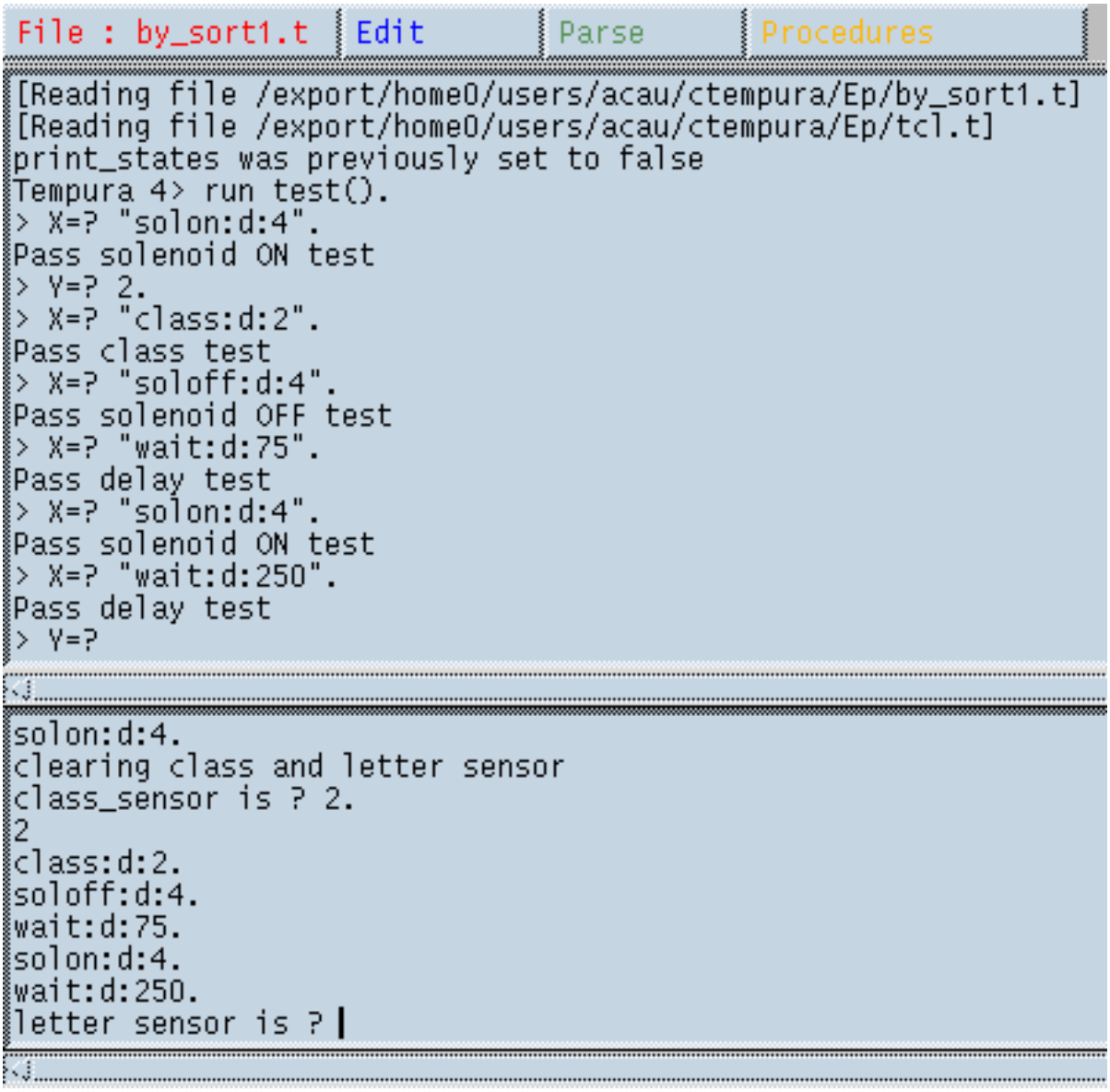
Corresponding C code for Sorting 1st Class Letters

```
if (class_sensor == 1)
          { /* its 1st Class Letter -
           activate solinoid 3 */
          printf("PROG: assert class:d:1::\n");
          /* THE ASSERTION-POINT */
          SolOff(4);
          Delay(Delay4);
          SolOn(4);
          Delay(DelayF);
          printf("letter sensor is ? ");
          scanf ("%d", &letter_sensor);
          if( ! YellowSet)
             {
             Delay(Delay3A); SolOff(3);
             Delay(Delay3B); YellowSet = 1;
             }
           }
```

Validating Sorting 1st Class Letters



Validating Sorting 2nd Class Letters



Validating Sorting Air Mail Letters

File : by_sort1a.t | Edit | Parse | Procedures

```
print_states was previously set to false
Tempura 4> run test().
> X=? "solon:d:4".
Pass solenoid ON test
> X=? "solon:d:6".
Pass solenoid ON test
> Y=? 1.
> X=? "air:d:1".
Pass class test
> X=? "soloff:d:4".
Pass solenoid OFF test
> X=? "wait:d:75".
Pass delay test
> X=? "solon:d:4".
Pass solenoid ON test
> X=? "wait:d:250".
Pass delay test
> Y=?
```

```
solon:d:6.
clearing sensors
air_sensor is ? 1.
1
air:d:1.
soloff:d:4.
wait:d:75.
solon:d:4.
wait:d:250.
letter sensor is ?
```

Timing Parameters of the Original Sorter

$SensorName$	$ExecutionTime$	$Deadline$
$ClassSensor$	$Delay4$	$75ms$
$LetterSensor$	$Delay3A$	$75ms$
$ActuatorName$	$ExecutionTime$	$Deadline$
$Solenoid4$	$DelayF$	$70ms$
$Solenoid3$	$Delay3B$	$250ms$

The Original Sorter in ITL

$$
\begin{aligned}
Sorter \quad \widehat{=} \quad & (\mathit{len} = 470 \wedge \\
& ((Sensor = 1 \supset \Diamond(Solenoid3 = ON)) \vee \\
& (Sensor = 2 \supset \Diamond(Solenoid3 = OFF)) \\
&) \\
&)
\end{aligned}
$$

$[\mathit{len} = Delay4 \wedge LetterState = at\_class\_sensor \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = DelayF \wedge LetterState = at\_Solenoid\_4 \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = Delay3A \wedge LetterState = at\_letter\_sensor \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = Delay3B \wedge LetterState = at\_Solenoid\_3 \wedge \text{stable}\,(LetterState)]$

Timing Parameters of the New Sorter

$SensorName$	$ExecutionTime$	$Deadline$
$AirMailSensor$	$Delay4$	$50ms$
$ClassSensor$	$Delay6$	$60ms$
$LetterSensor$	$Delay5A$	$50ms$
$ActuatorName$	$ExecutionTime$	$Deadline$
$Solenoid6$	$DelayFa$	$60ms$
$Solenoid5$	$Delay5B$	$85ms$
$Solenoid4$	$DelayF$	$50ms$
$Solenoid3$	$Delay3A$	$115ms$

The New Sorter in ITL

$$
\begin{aligned}
Sorter \quad \widehat{=} \quad & (\\
& (\mathit{len} = 235 \wedge \\
& (Sensor = Air \supset \Diamond(Solenoid4 = ON)) \\
&) \vee \\
& (Sensor \neq Air \supset \Diamond(Solenoid4 = OFF) \wedge \\
& (\mathit{len} = 235 \wedge \\
& (Sensor = 0 \supset \Diamond(Solenoid3 = OFF)) \vee \\
& (Sensor = 1 \supset \Diamond(Solenoid3 = ON)) \vee \\
& (Sensor = 2 \supset \Diamond(Solenoid3 = OFF)) \\
&) \\
&) \\
&)
\end{aligned}
$$

Sub-process 1:

$[\mathit{len} = Delay4 \wedge LetterState = at\_air\_sensor \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = DelayF \wedge LetterState = at\_Solenoid\_4 \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = Delay5A \wedge LetterState = at\_letter\_sensor \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = Delay5B \wedge LetterState = at\_Solenoid\_5 \wedge \text{stable}\,(LetterState)]$

Sub-process 2:

$[\mathit{len} = Delay6 \wedge LetterState = at\_class\_sensor \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = DelayFa \wedge LetterState = at\_Solenoid\_6 \wedge \text{stable}\,(LetterState)]$; skip;

$[\mathit{len} = Delay3A \wedge LetterState = at\_Solenoid\_3 \wedge \text{stable}\,(LetterState)]$

Conclusions and Future Work

- Approach: use formal methods and simulation to validate properties of the code
- A Formal Tool (ANATempura to support and illustrate the Approach
- Implementation of the Formal Tool
- Case Studies have been carried to illustrate both the approach and the Formal Tool

Assumption/Committement and Specification

$$Ass \wedge Prog \quad \supset \quad Com$$

$$\omega : \phi \sqsubseteq Prog$$

$$\omega : Ass \quad \supset \quad Com \sqsubseteq Prog$$

For a system Sys the *assumption/commitment* style can be expressed in ITL as follows:

$$\vdash \; w \wedge As \wedge Sys \;\supset\; Co \wedge \mathit{fin}\; w'.$$

Within this framework, *compositional* validation of properties can be acheived.

For example the following rule for sequential composition is sound

$$\frac{\begin{array}{l}\vdash \; w \wedge As \wedge Sys \;\supset\; Co \wedge \mathit{fin}\; w' \\ \vdash \; w' \wedge As \wedge Sys' \;\supset\; Co \wedge \mathit{fin}\; w''\end{array}}{\vdash \; w \wedge As \wedge (Sys; Sys') \;\supset\; Co \wedge \mathit{fin}\; w''\;.} \quad (1)$$

Given:

- $\omega : Ass_0 \quad \supset \quad Com_0 \sqsubseteq Prog_0$
- $Prog_1 \equiv \tilde{C}(Prog_0, Q)$

  Where $\tilde{C}$ is program operator such as ;, Conditional, ...
- $\omega : Ass \quad \supset \quad Com_Q \sqsubseteq Q$

What is the relationship between Ass_0, Ass_Q, and Com_0, Com_Q?

If $\tilde{C}$ is ; then

$$\begin{aligned} Ass_0 &\supset Com_0 \wedge Ass_0 \\ &\supset Com_0 \sqsubseteq Prog_0;\ Q \end{aligned}$$

if

$$Com_0 \supset Ass_Q$$

and

$$Com_Q \supset Com_0$$

Informal Semantics of ITL

- $\imath a$: f: choose a value of a such that f holds. If there is no such an a then $\imath a$: f takes an arbitrary value from a's range.
- skip: unit interval (length 1).
- f_1 ; f_2: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

Formal Semantics of ITL

The formal semantics is listed in the table below: Let χ be a choice function which maps any nonempty set to some element in the set. We write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v.

$$\mathcal{E}_\sigma[\![v]\!] = \sigma_0(v)$$

$$\mathcal{E}_\sigma[\![g(exp_1, \ldots, exp_n)]\!] = \hat{g}(\mathcal{E}_\sigma[\![exp_1]\!], \ldots, \mathcal{E}_\sigma[\![exp_n]\!])$$

$$\mathcal{E}_\sigma[\![\mu a\colon f]\!] = \begin{cases} \chi(u) & \text{if } u \neq \{\} \\ \chi(\mathrm{Val}_a) & \text{otherwise} \end{cases}$$

where $u = \{\sigma'(a) \mid \sigma \sim_a \sigma' \wedge \mathcal{M}_{\sigma'}[\![f]\!] = \mathrm{tt}\}$

$\mathcal{M}_\sigma[\![p(exp_1, \ldots, exp_n)]\!] = \mathrm{tt}$ iff $\hat{p}(\mathcal{E}_\sigma[\![exp_1]\!], \ldots, \mathcal{E}_\sigma[\![exp_n]\!])$

$\mathcal{M}_\sigma[\![\neg f]\!] = \mathrm{tt}$ iff $\mathcal{M}_\sigma[\![f]\!] = \mathrm{ff}$

$\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] = \mathrm{tt}$ iff $\mathcal{M}_\sigma[\![f_1]\!] = \mathrm{tt}$ and $\mathcal{M}_\sigma[\![f_2]\!] = \mathrm{tt}$

$\mathcal{M}_\sigma[\![\forall v \cdot f]\!] = \mathrm{tt}$ iff for all σ' s.t. $\sigma \sim_v \sigma', \mathcal{M}_{\sigma'}[\![f]\!] = \mathrm{tt}$

$\mathcal{M}_\sigma[\![\mathsf{skip}]\!] = \mathrm{tt}$ iff $|\sigma| = 1$

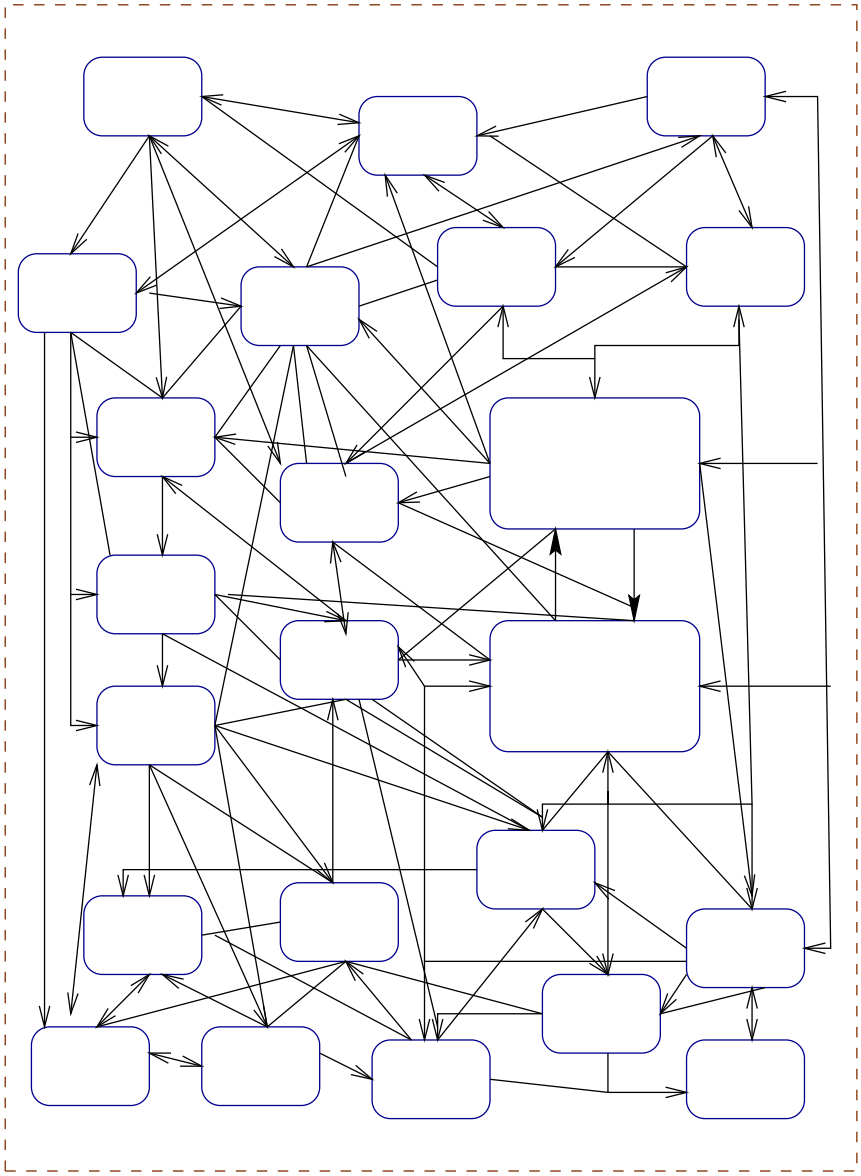
$\mathcal{M}_\sigma[\![f_1 ; f_2]\!] = \mathrm{tt}$ iff

(exists a k, s.t. $\mathcal{M}_{\sigma_0 \ldots \sigma_k}[\![f_1]\!] = \mathrm{tt}$ and
(((σ is infinite and $\mathcal{M}_{\sigma_k \ldots}[\![f_2]\!] = \mathrm{tt}$) or
(σ is finite and $k \leq |\sigma|$ and $\mathcal{M}_{\sigma_k \ldots \sigma_{|\sigma|}}[\![f_2]\!] = \mathrm{tt}$)))
or (σ is infinite and $\mathcal{M}_\sigma[\![f_1]\!]$)

$\mathcal{M}_\sigma[\![f^*]\!] = \mathrm{tt}$ iff

if σ is infinite then
(exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and
$\mathcal{M}_{\sigma_{l_n} \ldots}[\![f]\!] = \mathrm{tt}$ and
for all $0 \leq i < n$, $l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$)
or
(exist an infinite number of l_i s.t. $l_0 = 0$ and
for all $0 \leq i, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$)
else
(exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and
for all $0 \leq i < n, l_i < l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \mathrm{tt}$)

Current System



New Addition

Rules Of Evolution

Within our Assumption/commitment framework, refinement may be expressed as

$$(Ass \ \supset \ Com) \sqsubseteq Sys$$

For convenience we have abbreviated $(As \wedge w)$ to Ass and $(Co \wedge \mathit{fin}\ w')$ to Com.

The main rule for guided evolution is as follows:
Let Sys be the system that has to evolve, X be the 'addition' and $C(Sys, X)$ be the evolved system.

$$\frac{(Ass \ \supset \ Com) \sqsubseteq Sys, \quad (Ass_X \ \supset \ Com_X) \sqsubseteq X, \quad R_C(Ass_C, Com_C, Ass, Com, Ass_X, Com_X)}{(Ass_C \ \supset \ Com_C) \sqsubseteq C(Sys, X)}$$

Where C is a system composition operator such as sequential composition (;), conditional, iteration, parallel composition, etc.
The condition

$R_C(Ass_C, Com_C, Ass, Com, Ass_X, Com_X)$ is the condition under which the 'adding' of X to Sys is sound.

For example, if C is the sequential composition, then the rule is as follows:

$$\frac{\begin{array}{l}(Ass \supset Com) \sqsubseteq Sys,\\ (Ass_X \supset Com_X) \sqsubseteq X,\\ Ass_C \supset Ass,\\ Com \supset Ass_X,\\ Com_X \supset Com_C\end{array}}{(Ass_C \supset Com_C) \sqsubseteq (Sys\,;X)}$$