Introduction to Formal Methods
(Flight Schedule Database Example)

by

Ricky W. Butler
Mail Stop 130
NASA Langley Research Center
Hampton, Virginia 23681-2199

email: R.W.Butler@nasa.gov
phone: (757) 864-6198
fax: (757) 864-4234
web: http://shemesh.larc.nasa.gov/fm/

Oct 9, 2012

Outline

- Common Techniques of Formal Methods
- Simple Database Example
- SATS Example
- Review/Intro to Emacs

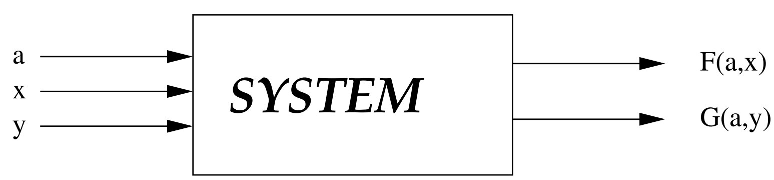
This is the only lecture that will seek to motivate the role of theorem proving in systems verification. The rest of the course will concentrate on developing skills in using the PVS Theorem Prover

Formal Specification

- Formal Specification: Use of notations derived from formal logic to describe
  - assumptions about the world in which a system will operate
  - requirements that the system is to achieve
  - the intended behavior of the system
- Styles of Specification:
  - Functions—express desired behavior or design descriptions
  - Properties—enumeration of assumptions and requirements
  - State-machines—express desired behavior or design descriptions
  - ...

Assumptions at one level become requirements at a lower level.

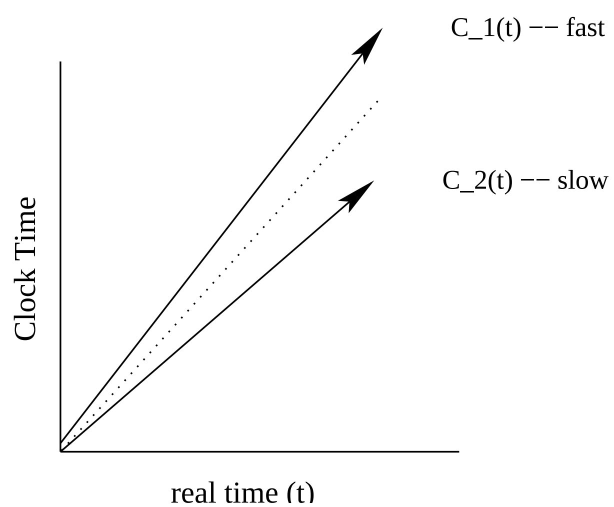
Functional Specification



$$F(a, x) = \frac{\sqrt{a^2 + x^2}}{1 - x^3}$$

$$G(a, y) = \frac{\sqrt{a^2 + y^4}}{1 + x^2}$$

Example of Property-Based Specification
(Fault-tolerant clock synchronization)



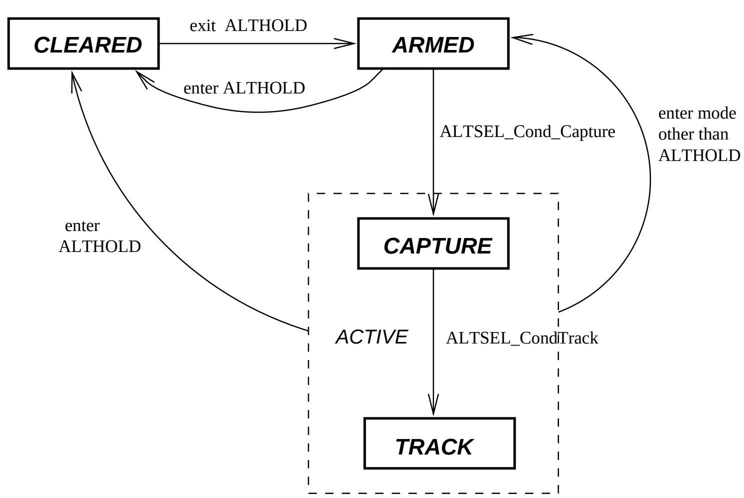
1. **There is a δ such that if clocks C_p and C_q are non-faulty at time t, then:**

$$|C_p(t) - C_q(t)| < \delta$$

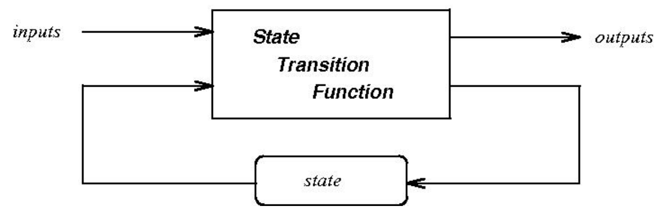
where $C_p(t)$ is clock p's time at real time t
$C_q(t)$ is clock q's time at real time t
δ is the maximum clock skew

State-machine Specification

Id	From	Events	To
1	CLEARED	Active_Vertical **EXITS** ALTHOLD	ARMED
2	$\overline{\text{CLEARED}}$	Active_Vertical **ENTERS** ALTHOLD	CLEARED
3	ARMED	ALTSEL_Cond_Capture	CAPTURE
4	ACTIVE	Active_Vertical **ENTERS** PITCH **OR** VS	ARMED
5	CAPTURE	ALTSEL_Cond_Track	TRACK



The State Transition Function



Transition Function : $inputs \times state \longrightarrow [outputs \times state]$

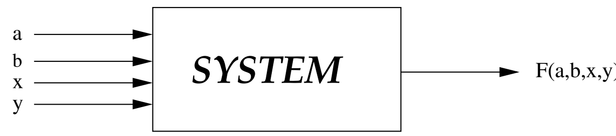
```
next_state(ev, current_state, new_mode): ALTSEL_submodes =
    COND
       new_mode = ALTHOLD         -> CLEARED,

       new_mode  = ALTSEL AND
       ARMED?(ALTSEL(current_state)) AND
       ALTSEL_Cond_Capture?(ev)   -> CAPTURE,

       new_mode  = ALTSEL AND
       ALTSEL_Cond_Track?(ev)     -> TRACK,

       new_mode /= ALTHOLD AND
       new_mode /= ALTSEL         -> ARMED

       ELSE                       -> ALTSEL(current_state)
    ENDCOND
```

Formal Proof Activities

Use of methods from formal logic to

1. **analyze specifications for certain forms of consistency, completeness**
2. **prove that specified behavior will satisfy the requirements, given the assumptions**
3. **prove that a more detailed design implements a more abstract one**

(1) Formal Analysis of a Specification



$$F(a,b,x,y) = ax + by$$

SAFETY PROP: $a^2 + b^2 = 1 \ \wedge \ x^2 + y^2 = 1 \supset F(a,b,x,y) \leq 1$

(1) Formal Analysis of a Specification (cont.) (Operational Procedure Tables)

		climb	steep_ climb	descend		dive	level	
				case 1	case 2		c1	c2
cur_mode	mode	level, climb, steep_climb	*	*	dive	*	*	descend
cur_alt < target_alt	bool	true	*	*	*	*	*	*
cur_alt < targe _alt - 1000	bool	false	true	*	false	*	*	false
cur_alt > target_alt	bool	*	*	true	false	*	*	false
cur_alt > target_alt + 1000 AND cur_alt > 5000	bool	*	*	false	false	true	*	false
target_alt - 100 <= cur_alt AND cur_alt < target_alt + 100	bool	false	*	false	false	*	true	false

- **CONSISTENCY**: no two columns operational for any values of the variables
- **COMPLETENESS**: For all values of variables one column is operational

(2) Verification of Fault-Tolerant Algorithms

Top-level: Properties that algorithm should possess

Lower-level: Abstract description of the algorithm and underlying assumptions

Prove: The algorithm satisfies desired properties given the assumptions

(3) Design Verification

Top-level: Abstract description of system (and assumptions)

Lower-level: Detailed description of system (and assumptions)

Prove: The detailed system description has the same behavior as the abstract description given the assumptions and an abstraction function relating the two systems.

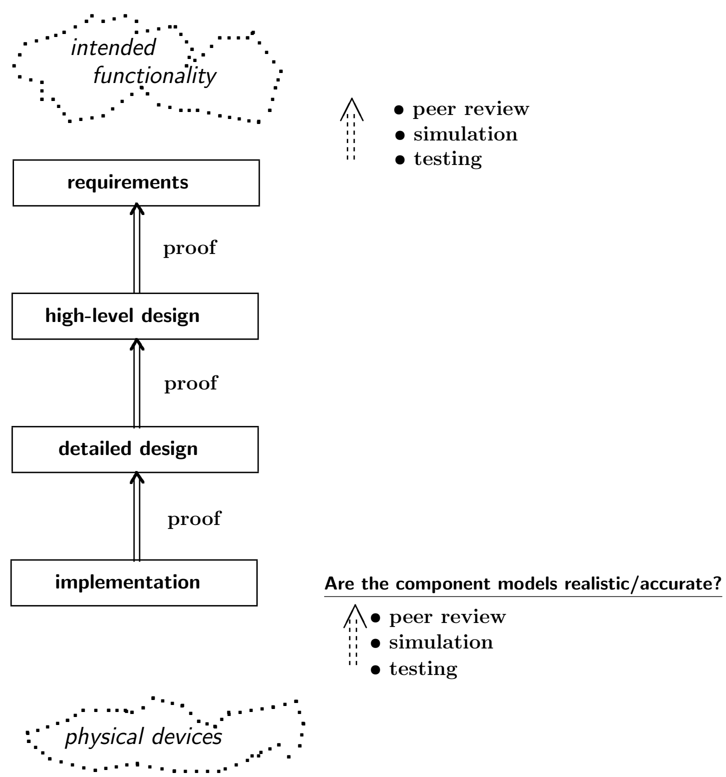
Hierarchical Verification

$$
\begin{tikzcd}
S_i(t) \arrow[r, "EXEC_i", "\text{(machine } i \text{ execution)}"'] & S_i(t+1) \\
S_{i+1}(t) \arrow[u, "Map"'] \arrow[r, "EXEC_{i+1}", "\text{(machine } i+1 \text{ execution)}"'] & S_{i+1}(t+1) \arrow[u, "Map"']
\end{tikzcd}
$$

$$\text{PROVE: } Map(EXEC_i(S_i(t))) = EXEC_{i+1}(Map(S_i(t))$$

- **Another way to do this is through theory interpretations**
  - **Prove that the axioms of the higher design specification become theorems when translated into the terms of the lower design specification**
  - **Equality requires special care**
- **Theory interpretations also provides a means to demonstrate (relative) consistency of axiomatic specifications. Became available in PVS 3.0.**

Illustration of Limitations



Recommended Reading

- Rushby, John: Formal Methods and Digital Systems Validation for Airborne Systems. NASA Contractor Report 4551, Dec. 1993. Available at http://shemesh.larc.nasa.gov/fm/fm-pubs-sri.html
- Rushby, John: Formal Methods and Their Role in Digital Systems Validation for Airborne Systems. NASA Contractor Report 4673, Aug. 1995. Available at http://shemesh.larc.nasa.gov/fm/fm-pubs-sri.html
- NASA Office of Safety and Mission Assurance, Washington, DC. Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume II: A Practitioner's Companion. Maybe available at `www.math.pku.edu.cn/teachers/zhangnx/fm/materials/NASAGB2.pdf`
- Papers at `http://pvs.csl.sri.com/documentation.shtml`
- Papers `http://shemesh.larc.nasa.gov/fm/`

Flight Schedule Example

Requirements for an Airport Flight Schedule Database

- The flight schedule database shall store the scheduling information associated with all departing and arriving flights. In particular the database shall contain:
  - departure time and gate number
  - arrival time and gate number
  - route (i.e. navigation way points)

  for each arriving and departing flight.
- There shall be a way to retrieve the scheduling information given a flight number.
- It shall be possible to add and delete flights from the database.

Formal Requirements Specification

- How do we represent the flight schedule database mathematically?

  1. a set of ordered pairs of flight numbers and schedules. Adding and deleting entries via set addition and deletion

  2. function whose domain is all possible flight numbers and range is all possible schedules. Adding and deleting entries via modification of function values.

  3. function whose domain is only flight numbers currently in database and range is the schedules. Adding and deleting entries via modification of the function domain and values.

  Note: The choice between these is strongly influenced by the verification system used.

Getting Started

Let's start with approach 2:

function whose domain is all possible flight numbers and range is all possible schedules. Adding and deleting entries via modification of function values.

In traditional mathematical notation, we would write:

$$\begin{aligned} \text{Let } N &= \text{set of flight numbers} \\ S &= \text{set of schedules} \\ D : N &\longrightarrow S \end{aligned}$$

where D represents the database and S represents all of the schedule information.

Note that the details have been abstracted away. This is one of the most important steps in producing a good formal specification.

Specifying the Flight Schedule Database

$$D : N \longrightarrow S$$

How do we indicate that we do not have a flight schedule for all possible flight numbers?

We declare a constant of type S, say "u_o", that indicates that there is no flight scheduled for this flight number.

Now can define an empty database. In traditional notation, we would write:

$$empty\_database : N \longrightarrow S$$
$$empty\_database(flt) \equiv u_o$$

$$\forall\ flt \in N$$

Accessing an Entry

Let $N =$ **set of flight numbers**
$S =$ **set of schedules**
$D =$ **set of functions :** $N \longrightarrow S$
$\forall d \in D$ **and** $flt \in N$.

$$find\_schedule : D \times N \longrightarrow S$$
$$find\_schedule(d, flt) = d(flt)$$

Note that $find\_schedule$ **is a higher-order function since its first argument is a function.**

Specifying Adding/Deleting an Entry

Let N = set of flight numbers
S = set of schedules
$D : N \longrightarrow S$
$u_o \in S$
D = set of functions : $N \longrightarrow S$
$\forall d \in D, \ \ \forall flt \in N, \ \ \forall sched \in S$

$$add\_flight : D \times N \times S \longrightarrow D$$

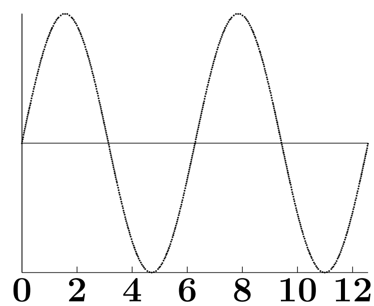
$$add\_flight(d, flt, sched)(x) = \begin{cases} d(x) & \text{if } x \neq flt \\ sched & \text{if } x = flt \end{cases}$$

$$delete\_flight : D \times N \longrightarrow D$$

$$delete\_flight(d, flt)(x) = \begin{cases} d(x) & \text{if } x \neq flt \\ u_o & \text{if } x = flt \end{cases}$$

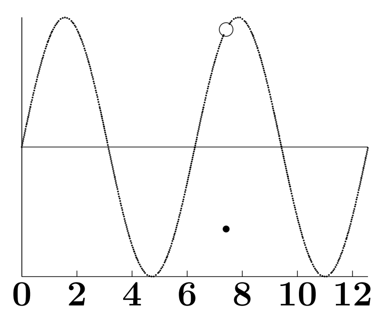
The WITH Notation

sin(x):



$$\text{sin WITH [7.4 := -0.60](x)} = \begin{cases} \text{-0.60} & \text{if } x = \text{7.4} \\ sin(x) & \text{otherwise} \end{cases}$$

sin WITH [7.4 := -0.60](x)



Complete Spec (Omitting Function Signatures)

Let $N =$ set of flight numbers
$S =$ set of schedules
$D =$ set of functions : $N \longrightarrow S$
$\forall d \in D, \;\; \forall flt \in N, \;\; \forall sched \in S$

$$find\_schedule(d, flt) = d(flt)$$

$$add\_flight(d, flt, sched) = d \textbf{ WITH } [flt := sched]$$

$$delete\_flight(d, flt) = d \textbf{ WITH } [flt := u_o]$$

Can test spec with some putative theorems:

LEMMA **putative 1** : $find\_schedule(add\_flight(d, flt, sched), flt) = sched$

LEMMA **putative 2** : $delete\_flight(add\_flight(d, flt, sched), flt) = d$

Attempted Verification Of Putative 2 Reveals a Problem

Putative 2: $delete\_flight(add\_flight(d, flt, sched), flt) = d$
Proof:

$$delete\_flight(add\_flight(d, flt, sched), flt) =$$

$$delete\_flight(d \textbf{ WITH } [flt := sched], flt) =$$

$$d \textbf{ WITH } [flt := sched] \textbf{ WITH } [flt := u_o] =$$

$$d \textbf{ WITH } [flt := u_o] = \;\; ??$$

But there is no way to reach d, because

$$d \textbf{ WITH } [flt := u_o] \neq d$$

unless $d(flt) = u_o$.

This is only true if the flt is currently not scheduled in the flight database.

Verification Reveals Oversight

- We realize that we only want to add a flight with flight number flt, if one is not already in the database.
- If flt is already in the database, we probably need the capability to change it.

Thus, we modify add_flight and create a new function $change\_flight$:

Verification Reveals Oversight (Cont.)

Let $N =$ set of flight numbers

$S =$ set of schedules

$D =$ set of functions : $N \longrightarrow S$

$\forall d \in D, \ \ \forall flt \in N, \ \ \forall sched \in S$

$$scheduled?(d, flt) : boolean = d(flt) \neq u_o$$

$add\_flight(d, flt, sched) =$

IF $scheduled?(d, flt)$ THEN d

ELSE d WITH $[flt := sched]$ ENDIF

$change\_flight(d, flt, sched) =$

IF $scheduled?(d, flt)$ THEN d WITH $[flt := sched]$

ELSE d ENDIF

Putative 2 Proof After Correction

Putative 2: NOT $scheduled?(d, flt) \supset$
$$delete\_flight(add\_flight(d, flt, sched), flt) = d$$

Proof:

$$delete\_flight(add\_flight(d, flt, sched), flt)$$

$$= delete\_flight(\textbf{ IF } scheduled?(d, flt) \textbf{ THEN } d \textbf{ ELSE } d \textbf{ WITH } [flt := sched] \textbf{ ENDIF }, flt)$$

$$= delete\_flight(d \textbf{ WITH } [flt := sched], flt)$$

$$= d \textbf{ WITH } [flt := sched] \textbf{ WITH } [flt := u_o]$$

$$= d \textbf{ WITH } [flt := u_o]$$

$$= d \qquad \text{(because NOT } scheduled?(d, flt) \supset d(flt) = u_o \text{)}$$

A Minor Problem

To check our new function `schedule?` we postulate the following putative theorem:

SchedAdd: LEMMA $scheduled?(add\_flight(d, flt, sched), flt)$

Proof:

$$scheduled?(add\_flight(d, flt, sched)) =$$

$$scheduled?(\textbf{ IF } scheduled?(d, flt) \textbf{ THEN } d \textbf{ ELSE } d \textbf{ WITH } [flt := sched] \textbf{ ENDIF }) =$$

$$\textbf{IF } d(flt) \neq u_o \textbf{ THEN } d(flt) \neq u_o \textbf{ ELSE } d \textbf{ WITH } [flt := sched](flt) \neq u_o \textbf{ ENDIF } =$$

$$d \textbf{ WITH } [flt := sched](flt) \neq u_o$$

$$sched \neq u_o$$

which is not provable because nothing prevents $sched = u_o$.

A Minor Problem Repaired

We then realize that our specification does not rule out the possibility of assigning a "u_o" schedule to a real flight

$$
\begin{aligned}
&\text{Let } N = \text{set of flight numbers}\\
&\quad S = \text{set of schedules}\\
&\quad S^* = \text{set of schedules not including } u_o\\
&\quad D = \text{set of functions} : N \longrightarrow S\\
&\quad \forall d \in D,\ \ \forall flt \in N,\ \ \forall sched \in S^*\\
&find\_schedule : D \times N \longrightarrow S\\
&add\_flight : D \times N \times S^* \longrightarrow D\\
&change\_flight : D \times N \times S^* \longrightarrow D\\
&delete\_flight : D \times N \longrightarrow D
\end{aligned}
$$

This type of problem is often not manifested until when one attempts a mechanical verification.

Another Example of a Putative Theorem

$$
\begin{aligned}
&\quad (\forall i : flt_i \neq flt) \wedge\\
&\\
&\quad find\_schedule(d_0, flt) = sched \wedge\\
&\quad d_1 = add\_flight(d_0, flt_1, sched_1) \wedge\\
&\quad d_2 = add\_flight(d_1, flt_2, sched_2) \wedge\\
&\quad \ \vdots \qquad\qquad\qquad \vdots\\
&\quad d_n = add\_flight(d_{n-1}, flt_n, sched_n)\\
&\supset\\
&\quad find\_schedule(d_n, flt) = sched
\end{aligned}
$$

- Formal methods can establish that even in the presence of an arbitrary number of operations a property holds.
- Testing can never establish this.

Some Observations

- Our specification is abstract. The functions are defined over infinite domains.
- As one translates the requirements into mathematics, many things that are usually left out of English specifications are explicitly enumerated.
- The formal process exposes ambiguities and deficiencies in the requirements.
- Putative theorem proving and scrutiny reveals deficiencies in the formal specification.

PVS Spec

```
flight_sched3: THEORY
BEGIN

  N : TYPE+                    % flight numbers
  S : TYPE+                    % schedules
  D : TYPE = [N -> S]          % flight database

  u0: S                        % unscheduled

  S_good: TYPE = {sched: S | sched /= u0}

  flt   : VAR N
  d     : VAR D
  sched : VAR S_good

  emptydb(flt): S = u0

  find_schedule(d, flt): S = d(flt)

  scheduled?(d,flt): boolean = d(flt) /= u0
```

```
  add_flight(d, flt, sched): D =
       IF scheduled?(d,flt) THEN d
       ELSE d WITH [flt := sched] ENDIF

  change_flight(d, flt, sched): D =
       IF scheduled?(d,flt) THEN d WITH [flt := sched]
       ELSE d ENDIF

  delete_flight(d, flt): D = d WITH [flt := u0]


  putative2: LEMMA NOT scheduled?(d,flt) IMPLIES
                     delete_flight(add_flight(d,flt,sched),flt) = d

  SchedAdd : LEMMA scheduled?(add_flight(d,flt,sched),flt)


END flight_sched3
```

Sequent Proof Style

The formula

$$P_1 \wedge P_2 \wedge P_3 \supset Q_1 \vee Q_2$$

can be presented as follows:

```
[-1] P1
[-2] P2
[-3] P3
|------
[1]  Q1
[2]  Q2
```

which is convenient because you can directly reference the individual terms.

ALL of the following are equivalent

$$P_1 \wedge P_2 \wedge P_3 \supset Q_1 \vee Q_2$$
$$P_1 \wedge P_2 \wedge P_3 \wedge \textbf{ NOT } Q_1 \supset Q_2$$
$$P_1 \wedge P_2 \supset Q_1 \vee Q_2 \vee \textbf{ NOT } P_3$$

because

$$P \supset Q \equiv \neg P \vee Q$$

PVS Does Not Like Leading NOTs To Hang Around

$$\neg y < x \ \wedge \ \neg z < y \ \supset \ \ x <= z$$

```
  |-------
{1} FORALL (x,y,z: real): NOT y < x AND NOT z < y IMPLIES  x <= z

Rule? (SKOSIMP*)

  |-------
{1}   y!1 < x!1
{2}   z!1 < y!1
{3}   x!1 <= z!1

Rule? (ASSERT)
Q.E.D.
```

In your mind you translate {1} and {2} to a premise

$$[-1] \ \ x \leq y \leq z$$

Introduction to a PVS Proof

- **Illustrative proof**

```
putative2 :

  |-------
{1} (FORALL (d: D, flt: N, sched: S_good):
       NOT scheduled?(d, flt)
         IMPLIES del_flight(add_flight(d, flt, sched), flt) = d)

Rule? (SKOSIMP*)

  |-------
{1}   scheduled?(d!1, flt!1)
{2}   del_flight(add_flight(d!1, flt!1, sched!1), flt!1) = d!1

Rule? (EXPAND "del_flight")

  |-------
[1]   scheduled?(d!1, flt!1)
{2}   add_flight(d!1, flt!1, sched!1) WITH [flt!1 := u0] = d!1
```

```
Rule? (EXPAND "add_flight")

  |-------
[1] scheduled?(d!1, flt!1)
{2} IF scheduled?(d!1, flt!1) THEN d!1
      ELSE d!1 WITH [flt!1 := sched!1] ENDIF
          WITH [flt!1 := u0] = d!1

Rule? (ASSERT)

  |-------
[1]   scheduled?(d!1, flt!1)
{2}   d!1 WITH [flt!1 := sched!1] WITH [flt!1 := u0] = d!1

Rule? (EXPAND "scheduled?")

  |-------
{1}   d!1(flt!1) /= u0
[2]   d!1 WITH [flt!1 := sched!1] WITH [flt!1 := u0] = d!1




Rule? (APPLY-EXTENSIONALITY 2 :HIDE? T)

 |-------
{1} d!1 WITH [flt!1 := sched!1] WITH [flt!1 := u0](x!1) = d!1(x!1)
[2] d!1(flt!1) /= u0

Rule? (LIFT-IF)

  |-------
{1}   IF flt!1 = x!1 THEN u0 = d!1(x!1)
      ELSE IF flt!1 = x!1 THEN u0 = d!1(x!1)
        ELSE d!1(x!1) = d!1(x!1)
        ENDIF
      ENDIF
[2]   d!1(flt!1) /= u0

Rule? (GROUND)

Q.E.D.

Run time  = 2.25 secs.
Real time = 4.29 secs.
```

Observations

- **With formal methods a clear, unambiguous, abstract specification can be constructed.**
- **Mechanized formal methods allows you can CALCULATE (prove) whether the specification has certain properties.**
- **These calculations can be done early in the lifecycle on abstract descriptions.**
- **And they can cover ALL the cases**

Emacs Essentials

`C-g`	**clear/reset the Emacs input buffer**
`C-x C-f`	**load file into buffer (i.e. a window)**
`C-x C-s`	**save contents of buffer into file**
`C-x b`	**switch to another buffer**
`C-x C-b`	**list all of your buffers**
`C-x 1`	**remove split screen: show only 1 buffer**
`C-k`	**cut (kill) line**
`C-x k`	**kill the buffer**
`C-y`	**paste (yank) line**
`C-x u`	**undo**
`C-d`	**delete character**
`C-a`	**move cursor to beginning of line**
`C-e`	**move cursor to end of line**
`M-f`	**move forward a word at a time**
`M-b`	**move backword a word at a time**
`C-<space>`	**set mark**
`C-w`	**cut region between mark and cursor**