Specification and Verification of Real-Time Control Software Using PVS

Mark Lawford

References

1. M. Lawford and H. Wu, "Verification of Real-Time Control Software Using PVS," In P. Ramadge and S. Verdu, eds., *Proceedings of the 2000 Conference on Information Sciences and Systems*, vol. 2, Dept. of Electrical Engineering, Princeton University, Princeton, NJ, pp. TP1-13–TP1-17, 2000.

2. H.Y. Wu, *Formal Verification of Real-Time Software*, M.Sc. Thesis, Department of Computing and Software, McMaster University, April 2001. (Also available as SERG Report 394.)

3. B. Dutertre and V. Stavridou, "Formal Requirements Analysis of an Avionics Control System", *IEEE Transactions on Software Engineering*, Vol. 23, no. 5, pp. 267–278, May, 1997.

4. N. Shankar, "Verification of Real-Time Systems Using PVS," *Computer Aided Verification, CAV '93*, LNCS 697, Springer-Verlag, pp. 280–291, 1993.

5. H. Pfeifer, A. Dold, F. W. v. Henke, and H. Rueß, *Guided Tour Through a Mechanized Semantics of Simple Imperative Programming Constructs*, Revised version of Technical Report UIB 96-11 Universität Ulm, Fakultät für Informatik, July 1997

   `http://www.informatik.uni-ulm.de/ki/PVS/semantics.html`

Outline

- Modeling Real-Time ⇒ `Clocks` Theory
- `Held_For` Theory
- Simple Example
- Sensor Lock Example
- Summary

Modeling Real-Time Properties

A *clock of period* K, is a set of "sample instances":

$$\begin{aligned} clock_K &:= \{t_0, t_1, t_2, \ldots, t_n, \ldots\} \\ &= \{0, K, 2K, \ldots, nK, \ldots\} \end{aligned}$$

E.g., for a period $K = 5$, the clock of period 5 is simply

$$clock_5 := \{0, 5, 10, 15, \ldots\}$$

Can define **pre**, **next** and **init** operators on clock values:

$$\begin{aligned} pre_K(t_n) &:= \begin{cases} t_{n-1}, & n \geq 1 \\ \text{undefined}, & \text{otherwise} \end{cases} \\ next_K(t_n) &:= t_{n+1} \\ init(t_n) &:= \begin{cases} TRUE, & n = 0 \\ FALSE, & \text{otherwise} \end{cases} \end{aligned}$$

HELD_FOR Operator

HELD_FOR : $pred(clock_K) \times \mathbb{R}^+ \rightarrow pred(clock_K)$

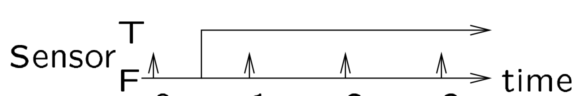
For $P : clock_K \rightarrow \{TRUE, FALSE\}$,

$$P \text{ HELD\_FOR}(duration)(t_n) = TRUE$$

iff $(\exists t_j \in clock_K)$ such that

$$(t_n - t_j \geq duration) \wedge$$
$$(\forall t_i \in clock_K)(t_j \leq t_i \leq t_n \Rightarrow P(t_i))$$

Example 1: Let $K = 150$, $duration = 295$, and $Sensor(t)$ be a clock predicate:



n	0	1	2	3
t_n	0	150	300	450
f	F	F	F	T

$f = (Sensor)$HELD_FOR(295) example

NOTE: We ignore intersample behavior of $Sensor$.

Clocks Theory

```
Clocks[ K: posreal ]: THEORY
BEGIN
non_neg: TYPE = { x: real | x>=0 }
time: TYPE = non_neg
t: VAR time

clock: TYPE = { t: time|EXISTS(n:nat): t=n*K }

x: VAR clock

init(x): bool = (x=0)
noninit_elem: TYPE ={ x | not init(x) }
y: VAR noninit_elem

pre(y): clock = y - K
next(x): noninit_elem = x + K
rank(x): nat = x/K

clock_induction: PROPOSITION
  FORALL (P: pred[clock]):
    (FORALL (x: clock): init(x)
      IMPLIES P(x)) AND
    (FORALL (y: noninit_elem): P(pre(y))
      IMPLIES P(y))
      IMPLIES (FORALL (x: clock): P(x))

END Clocks
```

Held_For Theory

```
Held_For  [K:posreal] : THEORY
  BEGIN
  IMPORTING Clocks[K]

  t, t_now: VAR clock
  duration:VAR time
  P: VAR pred[clock]

  heldfor(P, t, t_now, duration):
    RECURSIVE bool =
        IF P(t) THEN
          IF (t_now - t >= duration) THEN TRUE
          ELSIF init(t) THEN FALSE
          ELSE heldfor(P,pre(t),t_now,duration)
          ENDIF
        ELSE FALSE
        ENDIF
        MEASURE rank(t)

  Held_For(P, duration): pred[clock] =
    (LAMBDA (t:clock): heldfor(P,t,t,duration))

  END Held_For
```

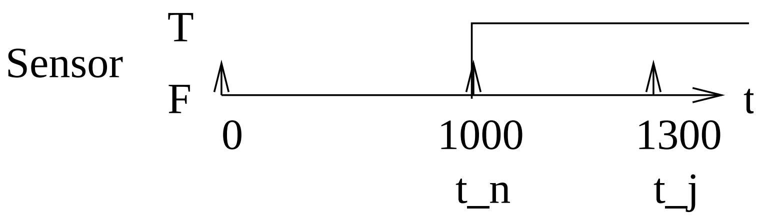
Alternative Held_For Theory

```
Held_For  [K:posreal] : THEORY
  BEGIN
  IMPORTING Clocks[K]

  t, t_now,t_n,t_j: VAR clock
  duration:VAR time
  P: VAR pred[clock]


  Held_For(P, duration): pred[clock]=
    (LAMBDA (t_n):
      EXISTS(t_j):(t_n-t_j>=duration) and
        FORALL(t:clock|t>=t_j&t<=t_n):P(t))


END Held_For
```

It is possible to prove that this version is equivalent to the recursive version.

Sometimes one form is more convenient than the other.

A Simple Example




```
simple  : THEORY
BEGIN

K: posreal  = 50
IMPORTING Held_For[K]

t: VAR clock

Sensor(t):bool = IF (t<1000) THEN FALSE
                 ELSE TRUE ENDIF

duration:time = 295

good: THEOREM (t>=1000+duration) IMPLIES
   Held_For(Sensor,duration)(t)

bad: THEOREM (t>=1000+duration-K)
   IMPLIES Held_For(Sensor,duration)(t)

END simple
```

A Simple Example (cont.)

Theorem `good` is easily proved in PVS since 1st clock value greater than 1000+duration=1295 is 1300.

Attempting `bad` results in unprovable sequent:

```
[-1]    n!1 >= 0
[-2]    50 * n!1 >= 0
[-3]    t!1 = 50 * n!1
[-4]    (50 * n!1 >= 1245)
  |-------
{1}    Sensor(50 * n!1 - 300)
```

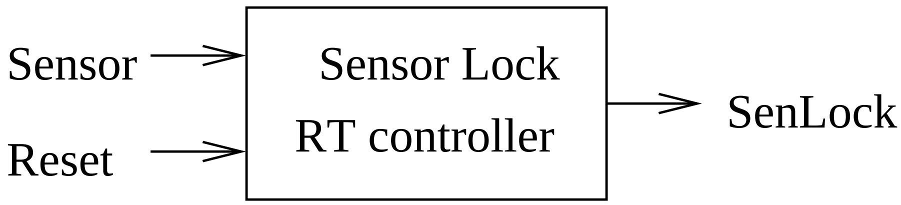
This sequent corresponds to the equation:

$$(\forall t_n \in clock_{50})t_n \geq 1245 \Rightarrow Sensor(t_n - 300)$$

Notes: 1245 = 1000 + 295 - 50 = 1000 + duration - K.

But for $t_{25} = 1250 \geq 1245$, all formulas are true except 1 since Sensor(950)=FALSE.

Software Verification Example



Sensor Lock real-time controller:

- inputs $Sensor$ and $Reset$ and output SenLock are booleans
- Sample inputs and update output $K = 100$ms.

Behavior:

- When $Sensor$ is continuously $TRUE$ for 150ms or longer, then the sensor is "locked" and $SenLock$ is set to $TRUE$.
- Once sensor is "locked" (i.e. $SenLock = TRUE$), it stays locked until manually reset indicated by making $Reset = TRUE$.

Software Requirements

The required behaviour of the update function is summarized by the following table:

Condition		*Result* SenLock
(Sensor) Held for (ldelay)		TRUE
NOT [(Sensor) Held for (ldelay)]	Reset	FALSE
	¬Reset	No Change

Here $ldelay = 150$ms.

When the conjunction of atomic proposition in a given row of the *Condition* columns is $TRUE$, then $SenLock$ is set to the *Result* value for that row. E.g., when

$$NOT[(Sensor)Held\_For(ldelay)] \wedge Reset$$

then $SenLock = False$.

Software Design

The SDD or "implementation" of this specification is given by the following table:

Condition			*Results* Elock	LTime
NOT Sensor	Elock =Lock	Reset	Good	0
		¬Reset	Lock	0
	Elock≠Lock		Good	0
Sensor	LTime=0		Bad	next(LTime)
	0 <LTime<Idelay		NC	next(LTime)
	LTime≥Idelay		Lock	0

Here $ELOCK$ has type $\{GOOD, BAD, LOCK\}$. The designer wants to use the additional information elsewhere in the system.

$$ELOCK = Lock \equiv SenLock = TRUE$$

"NC" denotes "No Change".

$LTime$ is timer variable used to implement the $Held\_For$.

Systematic Design Verification

```
SenLock_ELOCK: THEOREM
   SenLock(t) = lock?(Elock(ELOCK(t)))
```

To apply PVS to this Verification Problem we use the strategy `(INDUCT "t" 1 "clock_induction")`. This breaks proof into two parts: (i) Base Case when t=0, and (ii) inductive case. In the course of proving these cases, we find the following errors:

1. Wrong initial condition for Elock.

2. Elock becomes unlocked without a manual reset.

3. Cases exist where manual reset unlocks the SenLock but not Elock.

Systematic Design Verification (cont)

The complete specification and design require fail-safe operation so the value of $SenLock$ was initially set to $TRUE$. In the original design $Elock$ was initialized to Bad.

The SDD becomes unlocked because the $LTime$ counter is reset to 0 when Elock is set to Lock. As a result the system loses the "history" of $Sensor$. Although $Elock$ does not correctly implement this requirement as specified by $SenLock$, it also illustrates how $SenLock$ could be made "safer". When $Sensor = TRUE$, $Elock$ will not allow a manual reset, while $SenLock$ will permit such a reset if $Sensor$ was $FALSE$ in the recent past.

Systematic Design Verification (cont)

Taking these issues into consideration, we provide "fixed" versions of the specification and implementation below:

<table>
<tr><th colspan="3">Condition</th><th>Result
SenLock</th></tr>
<tr><td colspan="3">(Sensor) Held for (ldelay)</td><td>True</td></tr>
<tr><td rowspan="3">NOT [(Sensor) Heldfor (ldelay)]</td><td rowspan="2">Reset</td><td>$\neg$Sensor</td><td>False</td></tr>
<tr><td>Sensor</td><td>No Change</td></tr>
<tr><td colspan="2">$\neg$Reset</td><td>No Change</td></tr>
</table>

<table>
<tr><th colspan="3" rowspan="2">Condition</th><th colspan="2">Results</th></tr>
<tr><th>Elock</th><th>LTime</th></tr>
<tr><td rowspan="3">NOT Sensor</td><td rowspan="2">Elock =Lock</td><td>Reset</td><td>Good</td><td>0</td></tr>
<tr><td>$\neg$Reset</td><td>Lock</td><td>0</td></tr>
<tr><td colspan="2">Elock$\neq$Lock</td><td>Good</td><td>0</td></tr>
<tr><td rowspan="3">Sensor</td><td rowspan="2">LTime$<$ ldelay</td><td>Elock$\neq$Lock</td><td>Bad</td><td>next(LTime)</td></tr>
<tr><td>Elock=Lock</td><td>Lock</td><td>next(LTime)</td></tr>
<tr><td colspan="2">LTime$\geq$ ldelay</td><td>Lock</td><td>NC</td></tr>
</table>

A Systematic Approach

Problem: Getting complicated timing properties right in the implementation can be difficult when designer has to start and stop timers to implement timing constructs.

Solution: Used preverified blocks of code to implement recurring types of timing requirements.

E.g., In the previous example we actually implement the $(Sensor)Heldfor(ldelay)$ as:

$$Sensor \wedge LTime \geq ldelay$$

Why not reuse this timer implementation for all $Heldfor$s?

```
TimerGeneral  [K:posreal] : THEORY

  BEGIN
  IMPORTING Held_For[K]
  t, previous:var clock
  u:VAR noninit_elem
  timeout : var posreal
  P:var pred[clock]
  CurrentP:var bool

TimerUpdate(CurrentP,timeout,previous):clock= TABLE
                %-------------------------------------%
                |[previous<timeout|previous>=timeout]|
%----------------------------------------------------%
|CurrentP       |next(previous)   |previous         ||
%----------------------------------------------------%
|NOT CurrentP   |   0             |   0             ||
%----------------------------------------------------%
ENDTABLE

Timer(P,timeout)(t):RECURSIVE clock=
  IF init(t) THEN TimerUpdate(P(t),timeout,0)
  ELSE TimerUpdate(P(t),timeout,Timer(P,timeout)(pre(t)))
  ENDIF
  MEASURE rank(t)

Timer_Held_For: THEOREM
 (P(u) AND Timer(P,timeout)(pre(u))>=timeout)
= Held_For(P,timeout)(u)

END TimerGeneral
```

Summary

- PVS has been used to verify simple timing properties
- Unprovable sequents help to provide counter examples
- No "domain reasoning" required - PVS checks ALL cases
- Current implementation ignores intersample behavior and timing tolerances and has troubles with "large" time periods
- PVS can do much more for timing verification!