Verification of Real-Time Control Software Using PVS

Mark Lawford[1] and Hongyu Wu
Dept. of Computing and Software
Faculty of Engineering
McMaster University
Hamilton, Ontario Canada L8S 4L7
e-mail: lawford@mcmaster.ca

Abstract — **This paper provides preliminary results from an investigation of the use of PVS for the specification and verification of the real-time behavior of control systems software. Preliminary definitions are developed for specifying real-time software requirements. The definitions are used to specify a subsystem of an industrial real-time control system and then PVS is used to detect several errors in a proposed implementation and the original specification. Finally we prove that a corrected version of the implementation satisfies the updated version of the specification.**

I. Introduction

The paper considers a discrete time setting where a supervisory controller periodically samples its inputs and updates its outputs. We provide basic mathematical definitions to precisely specify some common real-time properties of controllers in this setting. The definitions are implemented as a reusable theory for SRI's automated proof assistant PVS[2]. We then use PVS to formally specify and verify the real-time behavior of an industrial control subsystem by extending the PVS "Clocks" theory originally developed by Dutertre and Stavridou for the analysis of [3]. These definitions, when combined with PVS's support for the tabular methods [4] of Parnas *et al.* [5, 6] provide a useful environment for the specification and verification of basic real-time control properties.

To illustrate the utility of the theory, we specify and verify a simple real-time subsystem of an industrial control system. As we will see, the process of formally modeling the subsystem's real-time requirements and verifying the proposed implementation against these requirements helps to uncover errors or unexpected behavior in both the specification and the implementation. As a result, both the requirements and implementation are modified and then proven to be correct. The resulting system is safer and we have a higher degree of confidence in its correctness.

Although the method has several limitations, noted below, that will hopefully be addressed by future refinements of the technique, as it now stands the method has proven useful in identifying errors in verification software involving hard real time constraints.

The PVS verification methodology outlined in [1] allows one to perform "block comparisons" verifying the functionality of the input/output logic that often composes the majority of a system requirements. In [1], the authors noted that the method is not readily applicable to the verification of subsystems with hard real-time requirements (i.e., "timing blocks"). Timing blocks are distinguished by the fact that in addition to requiring that the proper output is produced for a given input (or sequence of inputs), these blocks also require the output to be produced at the correct time. Thus, instead of relating a point in the domain (input) to a corresponding point in the range (output), timing blocks typically involve specifications relating timed sequences of inputs to timed sequences of outputs and hence tend to be more difficult to design and verify. A formal (mathematically sound) method for the verification of timing blocks would therefore significantly aid the design and verification process.

Using theorem proving techniques to verify real-time control software can be viewed as a complementary technique to testing, as theorem proving can be used to deal with the issues of domain coverage and determinism that are difficult or impossible to demonstrate with testing alone due to the well known state explosion problem.

The main advantages of the proposed method are:

- It is straightforward extension of the existing successful (untimed) methods of [1].
- Guaranteed Domain Coverage - PVS forces the verifier to demonstrate that the specification and implementation agree on *all* possible input sequences.
- Model can closely resembles original requirements specification and design descriptions due to the flexibility of PVS' type system.
- Counter examples - As described in [1], unprovable cases in PVS can help to provide counter examples that aid in the analysis of the implementation.
- Refutation - If the verifier believes that the implementation is not meeting the requirements for a particular input combination, PVS can be used to perform "refutation" by trying to prove a theorem stating that the specification and implementation are not equal for the specific input.

The main limitations of the method in its current form are its inability to address the issues of inter-sample behavior and different sampling rates, tolerances on timing values and the excessive amount of user intervention required to verify timing properties involving "large" time delays.

II. Preliminaries

This section provides an overview of the (functional) Systematic Design Verification (SDV) procedure used in [1] that is the basis of the real-time software verification problem posed in Section IV. The method makes use of a form of Parnas' tabular representations of mathematical functions [5, 6] to specify

[1]This work was partially supported by an NSERC.

the software's behavior. Tables provide a mathematically precise notation (see [7] for the formal semantics) in a visual format that is easily understood by domain experts, developers, testers, reviewers and verifiers alike [1].

We assume the underlying models of both the Software Requirements Specification (SRS) and the Software Design Description (SDD) are based upon Finite State Machines (FSM). The SDD adds to the SRS functionality the scheduling, maintainability, resource allocation, error handling, and implementation dependencies. Thus the SRS provides a high level description of the required system behavior while the SDD provides the implementation details to implement the required behavior.

Software engineering standards for safety critical software, such as [8], require that the design be formally verified against the SRS and then the code be formally verified against the SDD to ensure that the implementation meets the requirements. These formal verifications are governed by the SDV Procedure and Systematic Code Verification (SCV) Procedure. For the purposes of this paper we will concentrate on the SDV process.

The objective of the SDV process is to verify, using mathematical techniques or rigorous arguments, that the behavior of every output defined in the SDD, is in compliance with the requirements for the behavior of that output as specified in the SRS. The process employed in [1] is based upon a variation of the four variable model of [9] that verifies the functional equivalence of the SRS and SDD by comparing their respective one step transition functions. The resulting proof obligation in this special case:

$$\mathit{REQ} = \mathit{OUT} \circ \mathit{SOF} \circ \mathit{IN} \qquad (1)$$

is illustrated in the commutative diagram of Figure 1. Here

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "\mathit{REQ}"] \arrow[d, "\mathit{IN}"'] & \mathbf{C} \\
\mathbf{I} \arrow[r, "\mathit{SOF}"] & \mathbf{O} \arrow[u, "\mathit{OUT}"']
\end{tikzcd}$$

Fig. 1: Commutative diagram for 4 variable model

REQ represents the SRS state transition function mapping the monitored variables $\mathbf{M}$ (including the previous pass values of state variables) to the controlled variables and updated state represented by $\mathbf{C}$. The function SOF represents the SDD state transition function mapping the behavior of the implementation input variables represented by statespace $\mathbf{I}$ to the behavior of the software output variables represented by the statespace $\mathbf{O}$. The mapping IN relates the specification's monitored variables to the implementation's input variables while the mapping OUT relates the implementation's output variables to the specification's controlled variables.

In the 4-variable model of [9], each of the 4 "variable" state spaces $\mathbf{M}$, $\mathbf{I}$, $\mathbf{O}$, and $\mathbf{C}$ is a set functions of a single real valued argument that return a vector of values - one value for each of the quantities or "variables" associated with a particular dimension of the statespace. For instance, assuming that there are n_M monitored quantities, which we represent by the variables $m_1, m_2, \ldots, m_{n_M}$, then, the timed behavior of the variable m_i can be represented as a function $m_i^t : \mathbb{R} \rightarrow \mathit{Type}(m_i)$, where $m_i^t(x)$ is the value of the quantity m_i at time x. We can then take $\mathbf{M}$ to be the set of all functions of the form $m^t(x) = (m_1^t(x), m_2^t(x), \ldots, m_{n_M}^t(x))$. Thus the relations corresponding to the arrows of the commutative diagram then relate vectors of functions of a single real valued argument.

In order to simplify the 4-variable model and have it more closely model the dynamics of a digital control system that samples its inputs and updates its outputs at regular intervals, we restrict ourselves to the case where each of the 4 "variables" $\mathbf{M}$, $\mathbf{I}$, $\mathbf{O}$, and $\mathbf{C}$ is a set of "time series vectors". For example, $\mathbf{M}$ actually refers to all possible sets of observations ordered (and equally spaced) in time, each observation being a vector of n_M values. We will use the term *monitored variable* to refer to quantity m_i which is the ith element in the vector ($i \in \{1, \ldots, n_M\}$). Let $m \in \mathbf{M}$ be a time series vector of observations of the monitored variables. With a slight abuse of notation, we will use $m_i(z)$ to denote the zth observation of the ith element ($z \in \{0, 1, 2, \ldots\}$) of the monitored variables for the time series vector m. Similarly $m(z)$ represents the zth observation of the n_M values in the monitored variable vector for time series m.

For this model, the time increment between each of the observations is defined to be $K \in \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the positive reals. Thus observation z corresponds to time $(z * K)$. The value of m_i at any point between two observations (i.e., in the range of time $[z * K, (z+1) * K)$) is defined to be equal to $m_i(z)$.

The verification of real-time properties requires us to consider REQ and SOF as mapping from sequences of inputs to sequences of outputs since there is typically no longer a direct relationship between the one step transition functions of the SRS and SDD.

III. Specification of Real-time Requirements

The model of time employed by the proposed method builds upon a discrete time "Clocks" theory originally defined in [3]. While the model of time put forward in [3] allows for multiple clocks of different frequency and continuous time functions, we restrict ourselves to discrete time functions of a single clock frequency. The rest of this section describes the underlying real-time setting used to model systems. The section is concluded by a simple example that demonstrates the use of the HELD_FOR operator.

We will consider time to be the set of non-negative real numbers. Then for a positive real number K, we define a clock of period K, denoted clock_K, to be a set of "sample instances"

$$\mathit{clock}_K := \{t_0, t_1, t_2, \ldots, t_n, \ldots\} = \{0, K, 2K, \ldots, nK, \ldots\}$$

For a period $K = 5$, the clock of period 5 is simply

$$\mathit{clock}_5 := \{0, 5, 10, 15, \ldots\}$$

Note that clock_5, like all clocks as defined above, "starts" at time $t_0 = 0$.

To identify the initial clock value and thereby specify initial system states, we define the *init* predicate which is TRUE only at t_0:

$$\mathit{init}(t_n) := \begin{cases} \mathit{TRUE}, & n = 0 \\ \mathit{FALSE}, & \text{otherwise} \end{cases}$$

Identifying the initial clock value allows one to define recursive functions that use t_0 as the base case and then define the system state at any clock value in terms of the system state at the previous clock value. To formalize the notion of "previous clock value" and aid in proving termination properties of recursive functions defined over $clock_K$, we define the *rank of* t_n to be n. Formally: $rank_K : clock \to \mathbb{N}$ where $t_n \mapsto n$.

When defining recursive functions that have a clock of period K, for a particular instance of time (clock value) it is often convenient to be able to refer to the next sample time or previous sample time. To this end we can define $next_K$ and pre_K operators on the elements of $clock_K$ as follows:

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

When the value of K is unambiguous from the current context, we will omit the operator subscripts and simply write $rank()$, $next()$ and $pre()$.

Note that $pre(t0)$ is undefined. PVS requires that all functions are total (i.e. defined at every value in their domain). In the case of the $pre()$ operator, this is easily accomplished through the use of the subtype:

$$noninit\_elem_K := \{t_n \in clock_K | \neg init(t_n)\}$$

as the $pre()$ operator's domain. PVS allows the application of a function to any element belonging to a supertype of the function's domain and then generates a proof obligation or Type Correctness Condition (TCC). The TCC requires the user to prove the element the function is applied to is of the same type as the function's domain. For example, any time the $pre()$ operator is applied to an arbitrary clock value t_n, a TCC is generated requiring the user to prove that t_n is never equal to 0, and hence has a previous value.

We now state a preliminary definition that will aid us in defining the timing operators in the remain subsections. For the $clock_K$, the set of clock predicates, denoted $pred(clock_K)$, is the set of all boolean functions of $clock_K$:

$$pred(clock_K) := \{f | f : clock_K \to \{TRUE, FALSE\}$$

Figure 2 contains a simplified version of Dutertre and Stavridou's [3] PVS specification file that implements the parametrized theory Clocks defining the type clock that corresponds to $clock_K$ above.

The clock_induction proposition is a simple statement of proof by induction over clock values. It says that for a clock predicate P, if (i) $P(t_0)$ is $TRUE$, and (ii) for any $n > 0$, $P(t_{n-1})$ is $TRUE$ implies that $P(tn)$ is $TRUE$, then $P(t_n)$ is $TRUE$ for all t_n in $clock_K$. We will use this proposition to prove that an SRS function and SDD function are equivalent at all sample instance (clock values).

We can now define the PVS implementation of the HELD_FOR operator that we will use in the example of section IV. Let *duration* denote a non-negative real number, and P represent a clock predicate (i.e. $P : clock_K \to \{TRUE, FALSE\}$). HELD_FOR is an infix operator that takes a clock predicate as its first argument, a non-negative real number as its second argument and returns a clock predicate:

$$\text{HELD\_FOR} : pred(clock_K) \times \mathbb{R}^+ \to pred(clock_K)$$

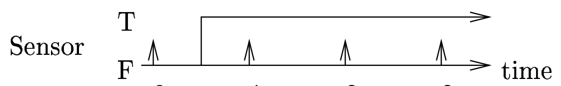
```
Clocks[ K: posreal ]: THEORY
BEGIN
non_neg: TYPE = { x: real | x>=0 }
time: TYPE = non_neg
t: VAR time

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
```

Fig. 2: PVS for Clocks Theory

such that $(P)\text{HELD\_FOR}(duration)(t_n) = TRUE$ 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 as shown in Figure 3: Note that we are



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

Fig. 3: $f = (Sensor)\text{HELD\_FOR}(295)$ example

ignoring intersample behavior of *Sensor*. The truth value of HELD_FOR is only dependent upon the value of *Sensor* at the sampling instances corresponding to the clock values.

The PVS theory defining the Held_For operator is shown in figure 4: The PVS function implementing the HELD_FOR operator is Held_For, defined at the bottom of the theory. It implements the HELD_FOR operator by evaluating the recursive function heldfor, which, as long as $P(t)$ is $TRUE$, back tracks to the previous value of t until t_now - t is greater than or equal to duration. If at any point before the recursion terminates $P(t)$ is $FALSE$or the initial state is reached, heldfor returns $FALSE$.

```
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
```

Fig. 4: PVS file implementing HELD_FOR operator

The theory `simple` in Figure 5 illustrates the use of the PVS Held_For theory.

The theorem `good` is easily proved by the (GRIND) command. This is expected since the first clock value greater than 1000+duration is 1300 and in this case 1300-1000¿295 while Sensor is true at 1000, 1300 and all clock values in between.

Attempting to prove bad results in the 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 unprovable sequent corresponds to the equation:

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

The number 1245 = 1000 + 295 - 50 = 1000 + duration - K. The first clock value greater than or equal to this number is 1250, but when t!1=1250 all formulas are true except 1 since Sensor(950)=FALSE resulting in an unprovable sequent.

In addition to the HELD_FOR operator, we have defined PVS version of DELAYED_BY and PERIODIC operators that allow one to formally specify the timing of an output event relative to an input event and periodic behavior respectively.

IV. Verification of Real-time Requirements

In this section we describe how PVS has been used to verify as simple real-time controller with inputs and output as shown in Figure 6. The Sensor Lock real-time controller takes two boolean valued inputs, *Sensor* and *Reset*, and produces a single boolean valued output *SenLock* every $K = 100$ms. When

```
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
```

Fig. 5: PVS file utilizing Held_For

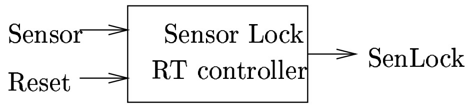


Fig. 6: Block diagram for real-time Sensor lock controller

the value of *Sensor* is continuously *TRUE* for *ldelay* = 150ms or longer, then the sensor is "locked" and *SenLock* is set to *TRUE*. Once the sensor is "locked", it stays locked until the system is manually reset indicated by making *Reset* = *TRUE*. This behaviour is summarized by the following table:

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

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*.

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	Lock
	0 <LTime<ldelay		NC	next(LTime)
	LTime≥ldelay		Lock	0

Here *Elock* is a three valued function corresponding to *SenLock* and *LTime* is a timer used to implement the Held_For. The designer used a three valued function for *Elock* so that this output could also provide some information about *Sensor* to the rest of the system. The idea is that $Elock = Lock$ when the sensor is "locked", it is *Bad* when the sensor is unlocked and $Sensor = TRUE$ and it is *Good* when the sensor is unlocked $Sensor = FALSE$. The value NC in the *Elock* column is short for "No Change".

Due to time and space constraints we do not include the PVS verification file here. It is available from the first authour upon request. 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.

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.

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

			Result
Condition			SenLock
(Sensor) Held for (ldelay)			True
NOT [(Sensor) Held for (ldelay)]	Reset	¬Sensor	False
		Sensor	No Change
	¬Reset		No Change

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

It is possible to use PVS to prove that the new version of the SDD implements the SRS.

V. Conclusion

PVS has been used to verify simple timing properties. The main benefit of the PVS Real-Time method is that it delivers a guarantee of domain coverage. When properly applied this method for the verification of timing blocks provides an increased level of confidence in the verification process and aids in detecting subtle timing errors.

The main advantages of the proposed method is that it is a relatively straight-forward extension of an existing methods, it checks all possible input sequences, and in the case when the SRS and SDD are not equivalent it provides some insight into the reasons for any discrepancies. Moreover, when a verifier suspects discrepancy, he can refutation theorem proving to confirm that the implementation does not satisfy the specification. The work currently has several limitations. Most significantly, the implementation ignores intersample behavior and timing tolerances. Also, more effective proof techniques are required for real-time properties spanning "large" durations.

References

[1] M. Lawford and P. Froebel, "Application of tabular methods to the specification and verification of a nuclear reactor shutdown system." Submitted to Formal Methods in System Design.

[2] N. Shankar, S. Owre, and J. M. Rushby, *PVS Tutorial.* Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[3] B. Dutertre and V. Stavridou, "Formal requirements analysis of an avionics control system," *IEEE Transactions on Software Engineering*, vol. 23, pp. 267–278, May 1997.

[4] S. Owre, J. Rushby, and N. Shankar, "Integration in PVS: Tables, types, and model checking," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)* (E. Brinksma, ed.), vol. 1217 of *Lecture Notes in Computer Science*, (Enschede, The Netherlands), pp. 366–383, Springer-Verlag, Apr. 1997.

[5] R. Janicki, D. L. Parnas, and J. Zucker, "Tabular representations in relational documents," in *Relational Methods in Computer Science* (C. Brink, W. Kahl, and G. Schmidt, eds.), Advances in Computing Science, ch. 12, pp. 184–196, Springer Wien NewYork, 1997.

[6] D. Parnas, "Tabular representation of relations," Tech. Rep. 260, Communications Research Laboratory, McMaster University, Oct. 1992.

[7] R. Janicki and R. Khédri, "On a formal semantics of tabular expressions," *Science of Computer Programming*, 2000. To appear.

[8] P. Joannou *et al.*, "Standard for Software Engineering of Safety Critical Software." CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 1, Jan. 1995.

[9] D. Parnas and J. Madey, "Functional documentation for computer systems engineering," Tech. Rep. CRL No. 273, Telecommunications Research Institute of Ontario, McMaster University, Sept. 1991.