Right on Time: Pre-verified Software Components for Constructuion of Real-Time Systems

Mark Lawford*and Xiayong Hu
Dept. of Computing and Software
Faculty of Engineering
McMaster University
Hamilton, Ontario Canada L8S 4L7
e-mail: lawford,huxy@mcmaster.ca

Nov. 2002

Abstract

We present a method that makes use of the theorem prover PVS to specify, develop and verify real-time software components for embedded control systems software with periodic tasks. The method is based on an intuitive discrete time "Clocks" theory by Dutertre and Stavridou that models periodic timed trajectories representing dataflows. We illustrate the method by considering a Held_For operator on dataflows that is used to specify real-time requirements. Recursive functions using the PVS TABLE construct are used to model both the system requirements and the design. A software component is designed to implement the Held_For operator and then verified in PVS. This pre-verified component is then used to guide design of more complex components and decompose their design verification into simple inductive proofs. Finally, we demonstrate how the rewriting and propositional simplification capabilities of PVS can be used to refine a component based implementation to improve the performance while still guaranteeing correctness. An industrial control subsystem design problem is used to illustrate the process.

1 Introduction

Specifying, implementing and verifying real-time requirement for embedded software systems can be a difficult and time consuming task. If a developer fails to correctly stop, start, or reset a timer under conditions that seldom occur, it can result in a design flaw detected late in the development phase, or worse, a system failure in the field.

*This work was supported by the Natural Sciences and Engineering Research Council of Canada.

The PVS verification methodology outlined in [6] allows one to perform “block comparisons” verifying the functionality of the input/output logic that often composes the majority of a system requirements. In [6], 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 (i.e. dataflows) of inputs to timed sequences of outputs and hence tend to be more difficult to design and verify. A formal method for the design and verification of timing blocks would therefore significantly aid the design and verification process.

This paper considers a discrete time setting where a supervisory controller periodically samples its inputs and updates its outputs. We use the PVS “Clocks” theory originally developed by Dutertre and Stavridou [2] as the basis for the model of the system real-time dataflows. We have added a Held_For operator that maps a predicate on data flows and a timeout value to a boolean dataflow that is TRUE when the input predicate dataflow is TRUE for the duration of the timeout. It is used at the requirements level to specify the real-time behaviour of controllers in our setting.

The definitions are implemented as reusable theories for SRI International’s automated proof assistant PVS [1]. These definitions, when combined with PVS’s support for the tabular methods [3] of Parnas *et al.* [4, 5] provide a useful environment for the specification and verification of basic real-time control properties. We illustrate the use of PVS by formally specify and verify the real-time behaviour of an industrial control subsystem. The remainder of the paper assumes some familiarity with PVS, although a reader unfamiliar with PVS should be able to understand the basic concepts with the examples we provide.

1.1 Related Work

While there has been much work in the area of formal verification of real-time system, this work has generally focused on high level requirements validation rather than lower level implementation issues. In [2] Dutertre and Stavridou develop PVS theories to formally specify and verify an avionics control systems, modelling the continuous dynamics of the plant as functions from time, modelled as positive real numbers, to appropriate types of state values and the discrete time dataflows of the digital control system as functions from discrete sample instances, or "clock values", to the appropriate type. The basis of the latter was the "Clocks" theory which we will use in the remainder of the paper. The work focused on modelling the interaction of the continuous and discrete dynamics and verifying properties of the system requirements.

The Held_For operator defined in Section 3 serves a similar purpose in our discrete time setting to the $|P|$, "since P" operator in the dense time setting of [9] where system actions can occur at any positive rational number. Saying do X when time since P was true is greater than 5 seconds can be phrased as do X when $\neg P$ has Held For 5 seconds. The $|P|$ operator is more expressive since it returns the exact time since P was last TRUE while P Held_For *timeout* returns a value of TRUE when P has been TRUE at all discrete time instance in the smallest window containing the continuous timer interval from the current sample time t to the sample at $t - timeout$. The PVS theory developed in [9] is used to verify invariants for Fischer's mutual exclusion protocol and a railroad crossing example.

Both [10] and [11] provide continuous time formalisms in PVS that attempt to insulate the user from the underlying theorem prover. A duration calculus (DC) proof assistant has been implemented on top of PVS in [10] to allow formal reason about real-time systems using the DC's interval temporal logic. It is a highly expressive, continuous time setting capable of modelling complex timing requirements. The timed automata modelling environment (TAME) [11], has been designed to provide a human style theorem proving environment for invariants and other properties of timed automata specifications. A simpler discrete time setting will suffice for our purposes.

The presented method is a straightforward extension of the existing successful (untimed) meth-

ods of [6]. 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. While model-checking techniques often provide the similar benefits they usually require the construction of an abstraction, either by hand, or computer aided, that restricts the type of properties and systems that can be checked. With the expressiveness of the higher order logic and associated type system used by PVS, the theorem prover model can closely resembles original requirements specification and design descriptions and allows the verification of whole classes of systems. While the lack of counter example generation capabilities limits the debugging ability of our setting, the possibility of performing refutation theorem proving, e.g., trying to prove a theorem stating that the specification and implementation are not equal for a specific input, provides some compensation.

A preliminary version of this work used a constructive, recursive definition of the Held_For operator to perform systematic design verification of a controller for specific timing values [7]. The focus was to minimize human interaction with the theorem prover but the method was severely limited due to the explosion in proof size as time bounds grew large relative to the sampling period. This preliminary work focused almost exclusively on design verification.

Below we will try to demonstrate ways in which a modern theorem proving systems can be used throughout the development of a simple discrete time digital control subsystem from formalizing the requirements to improving the performance of the implementation.

1.2 Outline of the Paper

Section 2 presents the setting for paper and the Clocks theory we borrow from [2]. Section 3 introduces the Held_For operator to capture timing requirements. We then show how the Held_For operator can be implemented in an imperative style using a timer variable. An example is used in Section 5 to illustrate the use of the Held_For operator and its implementation in all of the steps required to go from an informal system description to a pseudo code implementation that is optimized for performance.

2 Preliminaries

In the following two subsections we describe the variation of the formal setting of [6] considered in the paper and review the PVS clocks theory of [2].

2.1 Systematic Design Verification

This section provides an overview of the (functional) Systematic Design Verification (SDV) procedure used in [12, 6] that is the basis of the real-time software verification problem posed in Section 5. The method makes use of a form of Parnas' tabular representations of mathematical functions [4, 5] to specify the software's behaviour. Tables provide a mathematically precise notation in a visual format that is easily understood by domain experts, developers, testers, reviewers and verifiers alike.

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 behaviour while the SDD provides the implementation details to implement the required behaviour.

The objective of the SDV process is to verify, using mathematical techniques, that the behaviour of every output defined in the SDD, is in compliance with the requirements for the behaviour of that output as specified in the SRS. The process employed in [6] is based upon a variation of the four variable model of [13] that verifies the functional equivalence of the SRS and SDD by comparing their respective one step transition functions. The resulting proof obligation:

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

is illustrated by the solid lines in the commutative diagram of Figure 1.

Here REQ represents the SRS state transition function mapping the monitored variables $\mathbf{M}$ to the controlled variables represented by $\mathbf{C}$. The function SOF represents the SDD state transition

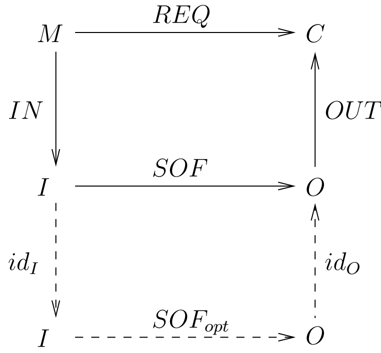

Figure 1: Commutative diagram for 4 variable model

function mapping the behaviour of the implementation input variables represented by statespace **I** to the behaviour of the software output variables represented by the statespace **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 [13], each of the 4 "variable" state spaces **M**, **I**, **O**, and **C** is a set functions of a single real valued argument t (time) that return a vector of values - one value for each of the quantities or "variables" associated with a particular dimension of the statespace at time t. Thus the relations corresponding to the arrows of the commutative diagram then relate vectors of functions of a single real valued argument.

For our purposes it is sufficient to consider a simplified discrete time 4-variable model that represents a digital control system's periodic sampling of inputs and update of outputs. In this case each of the 4 "variables" **M**, **I**, **O**, and **C** is a set of "time series vectors" or *dataflow*. For example, with a sampling period $K \in \mathbb{R}^+$, $m \in \mathbf{M}$ will be a dataflow of observations of the monitored variables at times $t = 0, K, 2K, \ldots$.

The verification of real-time properties requires us to consider REQ and SOF as mapping from input dataflows to output dataflows since there is typically no longer a direct relationship between the one step transition functions of the SRS and SDD as was the case in [12, 6]. We will use the fact that it is generally easier to verify two implementations are equivalent by comparing their one step transition functions rather than comparing trajectories generated by an implementation to those specified by more abstract requirements. Thus if a new design SOF_{opt} is created, we

may consider proving its equivalence to SOF and then infer its correctness with respect to REQ from the correctness of SOF. This is indicated by the dashed lines in Fig. 1. We make use of this result implicitly throughout the paper and explicitly when optimizing the DTS example in Section 5.4.

2.2 The Clocks Theory

The model of time employed by the proposed method builds upon a discrete time "Clocks" theory originally defined in [2]. While the model of time put forward in [2] 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"

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

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

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

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:

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

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 Dutertre and Stavridou 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 $pre(t_0)$ 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 [2] 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

```
Clocks[ K: posreal ]: THEORY
BEGIN
  non_neg: TYPE = { x: real | x>=0 }
  time: TYPE = non_neg
  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) => P(x)) AND
   (FORALL (y: noninit_elem): P(pre(y)) => P(y)) => (FORALL (x: clock): P(x))

END Clocks
```

Figure 2: PVS for Clocks Theory

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 $P(t_n)$ 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).

3 Specification of Real-time Requirements

We can now define the PVS implementation of the Held_For operator. Let *duration* denote a non-negative real number, and P represent a clock predicate (i.e. $P : clock_K \rightarrow \{TRUE, FALSE\}$). 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:

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

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 ignoring intersample behaviour of $Sensor$. The truth value of

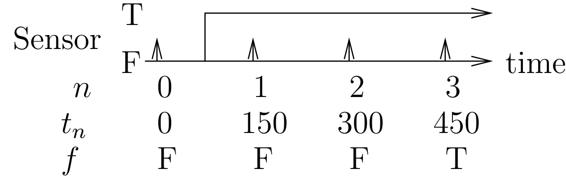

Figure 3: $f = (Sensor)\text{Held_For}(295)$ example

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:

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

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

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

END Held_For
```

Figure 4: PVS file implementing Held_For operator

The PVS function implementing the Held_For operator is Held_For, defined at the bottom of the theory.

4 Using a Timer Variable to Implement Held_For

In this section we verify that the function `TimerUpdate` in Figure 5 can be used to update a timer variable to provide a correct implementation of the Held_For operator. This `TimerUpdate`

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

  t, previous: VAR clock
  P:var pred[clock]
  timeout : VAR posreal
  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)

HeldFor_Timer: THEOREM
  IF init(t) THEN  FALSE
  ELSE P(t) AND Timer(P,timeout)(pre(t))>=timeout ENDIF
 = Held_For(P,timeout)(t)

END TimerGeneral
```

Figure 5: PVS for Timer implementation of Held_For Operator

function can then be used by the developer whenever the requirements specify a Held_For. We provide an example of this in Section 5 where `TimerUpdate`s implement a specification using nested Held_For conditions. The resulting design is then easily mapped to the imperative programming languages typically used for embedded systems development.

The `TimerUpdate` function takes a boolean input `CurrentP` representing the value of the condition that must hold for `timeout` and uses the `previous` value of the timer to compute what the current value should be. If `CurrentP` is TRUE and the timer has not yet timed out (`previous<timeout`), then we increment the timer by the sampling period K using the function `next()` function. In the case when the input is TRUE and the timer has already timed out, we leave the counter unchanged. This in effect maintains the information that the input has been TRUE for at least the required timeout while avoiding the possibility of an overflow error should the condition persist. Otherwise, if `CurrentP` is FALSE the timer is reset to 0.

The recursive function `Timer` models the dataflow of timer values that would result from using `TimerUpdate`. It takes a boolean dataflow, represented as a predicate on clock values, as well as the `timeout` argument and results in a mapping from a clock value `t` to the value of the timer at instant `t`.

We then prove in the theorem `TimerGeneral` that the condition `P(t)`, the current value of the boolean input, is TRUE, and the previous value of the timer has timed out, denoted `Timer(P,timeout)(pre(t))>=timeout`, is equivalent to P Held_For timeout (which becomes `HeldFor(P,timeout)(t)` in PVS). Thus we have converted the Held_For operator into a one step transition function. We remark that the proof of the main result was surprisingly difficult, requiring more than a day's effort, though this is in part attributable to the author's relative lack expertise in using PVS.

The method in Figure 5 is just one relatively trivial way to implement the Held_For operator of Section 3. We have also shown other implementations to be equivalent, including the recursive implementation of Held_For employed in [7]. Similar proof could be done for implementations that counting down or make RTOS API calls to start and stop a timer. While the effort to prove such a simple results may seem excessive, it is a one time cost to get a result that can be used repeatedly to simplify the verification of more complex timing properties such as the example in the following section.

5 Example: Delayed Trip System

This section introduces a variation of the Delayed Trip System (DTS), a real-time example from industry that appeared in [14]. We then describe how PVS can be used to specify and validate the system requirements, verify that an implementation meets the requirements and then help to refine the design in an effort to improve system performance.

5.1 Setting and Assumptions

The DTS is typical of many real-time systems from industry. When a certain set of circumstances arises, we want the system to provide the correct response in a timely fashion. In this case, when pressure and power measurements exceed acceptable safety limits in a particular way, we want the DTS controller to trip a relay causing the system to shut down. The result of failure to

Figure 6: Block Diagram for Delayed Trip Systems

shut down could be catastrophic. Conversely, each time the system is improperly shut down, significant financial loss could result (eg. stopping a sensitive chemical process in mid-reaction could ruin the product and possibly damage the plant, shutting down a reactor could cause a utility not to meet demand). Clearly it is important that the DTS behave in a very specific manner.

The desired input/output relationship for the DTS block diagram has the following informal description: "Any time the power exceeds power threshold PT and the pressure exceeds delayed set point DSP simultaneously for a duration of 3 seconds, open the relay for 2 seconds. Then the relay should be closed only if the power is no longer greater than PT. At system startup the relay is closed."

The DTS is to be implemented on a microprocessor system as a periodic task with a period of 100ms. That is, the system samples the inputs and passes through DTS control code every

0.1 seconds. We assume that the input signals have been properly filtered and that the sampling rate is high enough to ensure proper control.

5.2 Formalizing System Requirements

An initial attempt to formalize the system requirements might say "If (Power≥PT and Pressure≥ DSP) has HeldFor 3 seconds then open the relay and wait for 2 seconds. Then close the relay if Power<PT." This fails to consider the case Power and Pressure continuously to exceed their set points for more than 3 seconds. In order to make sure we satisfy the requirement that at *any* point in time where Power and Pressure have simultaneously exceeded their setpoints for 3 seconds or more, the relay will be open for the next two seconds, we must only allow the relay to be closed when at no point in the past 2 seconds has this Held_For condition been TRUE.

We attempt to capture this in the formal tabular Software Requirements Specification (SRS)

Condition	*Result* Delayed Trip
(Power≥PT ∧ Pressure≥DSP)Held_For timeout1	TRUE
Power<PT ∧ (¬[(Power≥PT ∧ Pressure≥DSP)Held_For timeout1]) Held_For timeout2	FALSE
¬[(Power≥PT ∧ Pressure≥DSP)Held_For timeout1] ∧ ¬[Power(t)<PT ∧ (¬[(Power≥PT ∧ Pressure≥DSP)Held_For timeout1]) Held_For timeout2]	No Change

Figure 7: Tabular Software Requirements Specification for DTS

for the DTS that appears in figure 7. Here we have replaced the values of 3 and 2 by Timeout1 and Timeout2 respectively, arbitrary positive real constants. We will also replace 100ms period by `K` in the PVS theory to make the problem more general.

The dataflow specified by this table can be represented as a recursive function in PVS that is very similar in appearance to the original table (see Figure 8). As a result of PVS's built in support for tabular specifications, coverage and disjointness Type Correctness Conditions (TCCs) are generated forcing us to prove the table defines a total function thereby guaranteeing completeness and determinism of the specification. For the `DelayedTrip_SRS` table the Type checking this specification coverage TCC is proved automatically while the disjointness TCC is

```
IMPORTING Held_For[K]
timeout1, timeout2: posreal
t: VAR clock
Power,Pressure: VAR [clock->real]

DelayedTrip_SRS(Power,Pressure)(t): RECURSIVE bool=
IF init(t) THEN FALSE
ELSE LET PP(t) = Power(t)>=PT & Pressure(t)>=DSP,
     NoChange = DelayedTrip_SRS(Power,Pressure)(pre(t)) IN
TABLE
%----------------------------------------------------------------+---------||
|Held_For(PP,timeout1)(t)                                         | TRUE    ||
%----------------------------------------------------------------+---------||
|Power(t)<PT & Held_For(NOT Held_For(PP,timeout1),timeout2)(t)   | FALSE   ||
%----------------------------------------------------------------+---------||
|NOT Held_For(PP,timeout1)(t) &                                 %         ||
 NOT (Power(t)<PT &                                             %         ||
 Held_For(NOT Held_For(PP,timeout1),timeout2)(t))               | NoChange||
%----------------------------------------------------------------+---------||
ENDTABLE
ENDIF
MEASURE rank(t)
```

Figure 8: PVS Specification for Delayed Reactor Trip Example

proved with minimal user intervention.

The theorem proving capabilities of PVS allow us to perform additional checks of our requirements to insure that we have properly captured the informal system requirements. Figure 9 contains two such examples. The first lemma validates the requirements by checking that whenever Held_For(P,timeout1)(t), then for the next timeout2 time units the Delayed Trip output will be true. The second lemma, which checks that the relay only changes from the open to closed state when Power<PT, is proved automatically with the `(GRIND)` command in PVS. The first lemma required some user interaction.

5.3 A First Implementation of the DTS

The `HeldFor_Timer` theorem of Fig. 5 provides guidance in creating a first implementation of the DTS shown in Fig. 10. We use two timer variables, `Timer1` and `Timer2`, for the first and second `Held_For`s respectively in the requirements specification.

```
Validation1:LEMMA
FORALL (timeout:noninit_elem|pre(timeout)<timeout2 AND timeout2<=timeout):
Held_For(lambda t:Power(t)>=PT & Pressure(t)>=DSP,timeout1)(t) =>
Held_For(DelayedTrip_SRS(Power,Pressure),timeout2)(t+ timeout)
END DelayedTrip3

Validation2:LEMMA
FORALL (t:noninit_elem):
DelayedTrip_SRS(Power,Pressure)(pre(t)) &
NOT DelayedTrip_SRS(Power,Pressure)(t)     => Power(t)< PT
```

Figure 9: PVS Theorems to Validate Requirements

We then replace all occurrences of `Held_For(PP,timeout1)(t)` with `PP&(Timer1(S)>=timeout1)` and `Held_For(NOTHeld_For(PP,timeout1),timeout2)(t)` with `NOT(PP&Timer1(S)>=timeout1)&Timer2(S)>=timeout2` to obtain the function `RelayUpdate`. It is used to update the implementation's relay state from the previous clock value to the current clock value. Updates for `Timer1`and `Timer2` make use of the `TimerUpdate` function from the `TimerGeneral` theory. The recursive function `SDD` models the implementations dataflow. Given sequences of input values for power and pressure at clock values it produces the corresponding sequence of implementation states, providing a mapping from clock value to values for the relay and timers.

We break down the verification into two stages. In the first stage we use lemmas `Timer1_Timer` and `Timer2_Timer` to prove that each of the timers is being updated in the same manner as the `Timer` in the `Timer_General` theory. Then in proving the main block comparison theorem `DelayedTrip_Block` we use these lemmas to rewrite the proof obligation so that the `HeldFor_Timer` theorem of Fig. 5 can be used to replace the timer conditions in the `RelayUpdate` table, effectively reversing the process that we used to create `RelayUpdate` and finishing the proof.

We use the PVS strategy `(INDUCT "t" 1 "clock_induction")` to solve the in proving the main theorem. This breaks proof into two parts: (i) Base Case when t=0, and (ii) inductive case. In the course of proving these cases, we detected improper initialization of Timer2. At time $t = 0$ the `TimerUpdate` function was initially called with the first argument `FALSE` instead of `TRUE`. This was detected when the base case of the inductive proof failed. Once this mistake

```
Relay_State: TYPE = {OPEN, CLOSED}
SDD_State: TYPE = [# Relay: Relay_State, Timer1: clock,Timer2:clock #]

RelayUpdate(pwr,pres:real,S:SDD_State):Relay_State =
LET PP = pwr>= PT & pres>= DSP,
    NoChange = Relay(S) IN
TABLE
%-------------------------------------+--------||
|PP & (Timer1(S) >= timeout1)         |OPEN    ||
%-------------------------------------+--------||
|NOT (PP & Timer1(S)>=timeout1)       %        ||
 & pwr<PT & Timer2(S)>=timeout2       |CLOSED  ||
%-------------------------------------+--------||
|  NOT(PP& Timer1(S)>=timeout1)       %        ||
 & NOT(pwr<PT & Timer2(S)>=timeout2)  |NoChange||
%-------------------------------------+--------||
ENDTABLE

SDD(Power,Pressure)(t):RECURSIVE SDD_State =
LET pp = Power(t)>=PT & Pressure(t)>=DSP IN
  IF init(t) THEN (#    Relay := CLOSED,
                        Timer1:=TimerUpdate(pp,timeout1,0),
                        Timer2:=TimerUpdate(TRUE,timeout2,0)    #)
  ELSE
    (#  Relay:=RelayUpdate(Power(t),Pressure(t),SDD(Power,Pressure)(pre(t))),
        Timer1:=TimerUpdate(pp,timeout1,Timer1(SDD(Power,Pressure)(pre(t)))),
        Timer2:=TimerUpdate(
                  NOT (pp & Timer1(SDD(Power,Pressure)(pre(t)))>=timeout1),
                  timeout2, Timer2(SDD(Power,Pressure)(pre(t)))) #)
  ENDIF
  MEASURE rank(t)

Timer1_Timer: LEMMA
  Timer(lambda t:Power(t)>=PT & Pressure(t)>=DSP,timeout1)(t)
= Timer1(SDD(Power,Pressure)(t))

Timer2_Timer: LEMMA
  Timer(NOT Held_For(lambda t:Power(t)>=PT & Pressure(t)>=DSP,timeout1),
        timeout2)(t)
= Timer2(SDD(Power,Pressure)(t))

DelayedTrip_Block :THEOREM
  DelayedTrip_SRS(Power,Pressure,timeout1,timeout2)(t)
= OPEN?(Relay(SDD(Power,Pressure,timeout1,timeout2)(t)))
```

Figure 10: PVS for Initial DTS Implementation and Verification

was correct the

The entire process required less than an hour of interactive proof effort. As we refine the theories, become more familiar with their use, and have more proven examples, we expect the time to perform such verifications will decrease further still.

5.4 Speeding up the DTS Implementation

Often in real-time systems a designer is faced with the challenge of improving the performance of part of the system that may force the developer into a designing for speed vs. verifiability trade-off A good modular design that reuses existing, verified code will make the verification task much easier. While the extra overhead associated with the additional function calls of a modular software design can be partially mitigated through the use of macros, inline functions and compiler optimizations, the resulting code may still have extra conditional evaluations that result from the modularization of the code.

A theorem prover's rewriting and propositional simplification can help a developer to eliminate redundant conditional statements and simplify complicated conditions. To do this we define an uninterpreted function symbol, `fastSDD` and state the proposition `optimize` as follows:

```
fastSDD(Power,Pressure)(t):SDD_State
Optimize: PROPOSITION fastSDD(Power,Pressure)(t)=SDD(Power,Pressure)(t)
```

The `fastSDD` function has the same type as the dataflow for the original implementation dataflow `SDD`. We then try to prove theorem `optimize` by expanding the definition of the original implementation and then repeatedly using the prover rewrite rules `(LIFT-IF)`and `(SIMPLIFY)` until no changes result. At this point we use the `(BDDSIMP)` command to do propositional simplification using BDDs. This results in one or more sequents being generated for each of the distinct assignments made by original implementation. These sequents then become cases in the new implementation that appears in Fig. 11.

For example, the first pair of assignments in the pseudocode corresponds the case when Power≥PT and Pressure≥DSP and the previous value of Timer1≥timeout1. In this case we

```
IF Power>=PT & Pressure>=DSP THEN
  IF Timer1>=timeout1 THEN
            Relay := OPEN,
            Timer2 := 0
  ELSIF Timer2>=timeout2 THEN
            Timer1 := Timer1 + K
  ELSE
            Timer1 := Timer1 + K,
            Timer2 := Timer2 + K
  ENDIF
ELSIF Timer2>=timeout2 THEN
    IF Power<PT THEN
            Relay := CLOSED,
            Timer1 := 0
    ELSE
            Timer1 := 0
    ENDIF
ELSE
            Timer1 := 0,
            Timer2 := Timer2 + K
ENDIF
```

Figure 11: Pseudo code for "Optimized" Delayed Reactor Trip Example

open the relay, leave Timer1 unchanged and reset Timer2 to 0. This case was obtained from the proof sequent Optimize.3:

```
 Optimize.3 :

{-1}  Power!1(t!1) >= PT
{-2}  Pressure!1(t!1) >= DSP
{-3}  (Timer1(SDD(Power!1, Pressure!1)(t!1 - K)) >= timeout1)
  |-------
{1}   (t!1 = 0)
{2}   fastSDD(Power!1, Pressure!1)(t!1) =
       (# Relay := OPEN,
          Timer1 := Timer1(SDD(Power!1, Pressure!1)(t!1 - K)),
          Timer2 := 0 #)
```

Comparing the number of conditional evaluations resulting in branches in the code for the original and optimized versions of the code we get the following:

	Original		Optimized	
Measure	Min	Max	Min	Max
Branches	3	6	2	3

Once specified in PVS it is easily verified that the two implementations result in identical dataflows.

6 Conclusion

PVS can be used throughout the real-time development process, from formalizing and validating initial system requirements to verifying and optimizing an implementation. The main benefits of the presented PVS Real-Time method is that it delivers a guarantee of domain coverage, can be used throughout the development/verification process, and can verify whole classes of systems by proving timing properties for arbitrary sampling periods and timeout values. The expressiveness of the PVS specification language permits a model of the system that more closely resembles the actual specification and code. The theorem proving capabilities allow the exploration of implementations and indirectly provides insight into how a design could be optimized for performance considerations. 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.

References

[1] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, pp. 107–125, Feb. 1995.

[2] 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.

[3] 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.

[4] 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.

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

[6] M. Lawford, P. Froebel, and G. Moum, "Application of tabular methods to the specification and verification of a nuclear reactor shutdown system." Accepted for publication. Also available at `http://www.cas.mcmaster.ca/~lawford/papers/`.

[7] M. Lawford and H. Wu, "Verification of real-time control software using pvs," in *Proceedings of the 2000 Conference on Information Sciences and Systems* (P. Ramadge and S. Verdu, eds.), vol. 2, (Princeton, NJ), pp. TP1–13–TP1–17, Dept. of Electrical Engineering, Princeton University, Mar. 2000.

[8] G. Holzmann and M. Smith, "An automated verification method for distributed systems software based on model extraction," *IEEE Transactions on Software Engineering*, vol. 28, pp. 651–660, Apr. 2002.

[9] N. Shankar, "Verification of real-time systems using PVS," in *Computer-Aided Verification, CAV '93* (C. Courcoubetis, ed.), vol. 697 of *Lecture Notes in Computer Science*, (Elounda, Greece), pp. 280–291, Springer-Verlag, June/July 1993.

[10] J. U. Skakkebæk and N. Shankar, "Towards a Duration Calculus proof assistant in PVS," in *Formal Techniques in Real-Time and Fault-Tolerant Systems* (H. Langmaack, W.-P. de Roever, and J. Vytopil, eds.), vol. 863 of *Lecture Notes in Computer Science*, (Lübeck, Germany), pp. 660–679, Springer-Verlag, Sept. 1994.

[11] M. Archer, C. Heitmeyer, and S. Sims, "TAME: A PVS interface to simplify proofs for automata models," in *User Interfaces for Theorem Provers*, (Eindhoven, The Netherlands), July 1998. Informal proceedings available at `http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.

[12] M. Lawford, J. McDougall, P. Froebel, and G. Moum, "Practical application of functional and relational methods for the specification and verification of safety critical software," in *Algebraic Methodology and Software Technology, AMAST 2000* (T. Rus, ed.), vol. 1816 of *Lecture Notes in Computer Science*, (Iowa City, IA), pp. 73–88, Springer-Verlag, May 2000.

[13] D. L. Parnas and J. Madey, "Functional documents for computer systems," *Science of Computer Programming*, vol. 25, pp. 41–61, Oct. 1995.

[14] M. Lawford and W. Wonham, "Equivalence preserving transformations of timed transition models," *IEEE Trans. Autom. Control*, vol. 40, pp. 1167–1179, July 1995.