



McMaster Centre for Software Certification

Formal Verification of Real-Time Function Blocks Using PVS

Linna Pang, Chen-Wei Wang, Mark Lawford, Alan Wassying
Josh Newell, Vera Chow, and David Tremain

McMaster Centre for Software Certification

The Need for Certification

Software is essential to more and more products. In many industries — medical, automotive, aerospace, nuclear power, military equipment, for example — failure of software to meet its requirements can be disastrous. Society is increasingly demanding that software used in such critical systems must meet minimum safety, security and reliability standards. Manufacturers of these systems are in the unenviable position of not having consistent and effective guidelines as to what constitutes acceptable evidence of software quality, and how to achieve it. This drives up the cost of producing these systems without producing a commensurate improvement in dependability.

The Need for Evidence

Critical, software-intensive devices are typically certified on the basis of the process used to develop them. We believe that this is inadequate, that while a good process may be necessary for producing dependable software, it is not sufficient: certification must also be based on evidence obtained from the product. Our research is therefore into what kind of evidence is sufficient, and how different kinds of evidence may be combined into an argument for safety that is sufficient. This research is partly theoretical, but also practical: we work with industries involved in developing critical, software-intensive systems on their practical problems.

The Centre

The Centre for Software Certification was established at McMaster University in 2008. Its objective is to improve the practice of software engineering applied to critical systems involving software. To achieve this it

- performs research on how to produce software that can be certified, and on how existing software may be certified
- works with industrial partners on the development and certification of software
- works with regulatory authorities on the relevant standards and approaches to software certification
- works with universities to improve their software engineering curricula
- works with the bodies responsible for recognising professional engineers to improve their requirements

While our emphasis is on software, we recognise that the safety of products that depend on software is a problem in systems engineering: the hardware that contains the software has to be part of the engineering, and part of the certification.

To find out more, visit our web site <http://www.mcscert.ca> or contact us at mcscert@cas.mcmaster.ca.

Formal Verification of Real-Time Function Blocks Using PVS

Linna Pang¹, Chen-Wei Wang¹, Mark Lawford¹, Alan Wassyn¹
Josh Newell², Vera Chow², and David Tremaine²

¹ McMaster Centre for Software Certification, McMaster University, Canada L8S 4K1
{pangl,wangcw,lawford,wassyn}@mcmaster.ca

² Systemware Innovation Corporation, Toronto, Canada M4P 1E4
{jnewell,vchow,tremaine}@swi.com

Abstract. A critical step towards certifying safety-critical systems is to check their conformance to hard real-time requirements. A promising way to achieve this is by building the systems from pre-verified components and verifying their correctness in a compositional manner. We previously reported a formal approach to verifying function blocks (FBs), using tabular expressions and the PVS proof assistant, to facilitate the adoption of Programmable Logic Controller (PLC) based digital systems. We applied our approach to verify FB implementations defined in the IEC 61131-3 standard on PLCs. Consequently, we constructed a repository of precise specification and reusable (proven) theorems of feasibility and correctness for FBs. However, our verification of IEC 61131-3 FBs was limited to the unit level (i.e., verifying each standard FB or timer in isolation), and we did not encounter any real-time FB (i.e., one built from timers) in the standard. In this paper, based on our experience in the nuclear domain, we address these issues by conducting two realistic case studies, consisting of the software requirements and their proposed FB implementations for two subsystems of an industrial control system. Both implementations are built from IEC 61131-3 FBs, including the on-delay timer. In proving that the implementations are feasible and correct, we: 1) find issues and suggest resolutions; and 2) identify proof patterns amenable to automated support for verifying other subsystems.

Keywords: software requirements, function blocks, IEC 61131-3, timing requirements, formal verification, tabular expressions, PVS

1 Introduction

Many industrial safety-critical software control systems are based upon Programmable Logic Controllers (PLCs). Function blocks (FBs) are reusable components for implementing the behaviour of PLCs in a hierarchical way. In one of its supplements, DO-178C [1] (in the aviation domain) advocates the use of formal methods to construct, develop, and reason about the mathematical models of system behaviours. As a result, we may obtain high-quality PLCs by pre-verifying standard FBs using formal methods, building systems from pre-verified components, and verifying their correctness in a compositional manner.

We recently reported a formal methodology [20,21] for specifying requirements for FBs, and for verifying the correctness of their implementations expressed in, e.g., function block diagrams (FBDs). In our approach, we use tabular expressions [23] for specification and PVS [19] for formal verification. Tabular expressions, proven to be both practical and effective in industry [29], are

a way to document system requirements as black-box, input-output relations. PVS provides an integrated environment with mechanized support for writing specifications using tabular expressions and (higher-order) predicates, and for (interactively) proving that implementations satisfy the tabular requirements using sequent-style deductions. We successfully applied our approach to the FB library of the IEC 61131-3 [10, Annex F], an industrial standard for PLCs. Consequently, we constructed a repository of precise specification and reusable (proven) theorems of feasibility and correctness for FBs.

A critical step towards certifying PLC-based (or other safety-critical) systems is to check their conformance to hard real-time requirements. An *implementable* timing requirement must specify *tolerances* to account for various factors — e.g., sampling rates, computation time, and latency — that will delay the software controller’s response to its operating environment (i.e., the plant). A common type of functional timing requirements specifies that a monitored condition C must sustain over a time duration, say *timeout*, with tolerances $-\delta L$ and $+\delta R$, before being detected by the controller. Such sustained timing requirements may be formalized using an infix *Held_For* operator [28]. For example, we write

$$((\text{signal} \geq \text{setpoint}) \text{ Held_For } (300, -50, +50)) \Rightarrow (c_var = \text{trip}) \quad (1)$$

to specify that a sensor signal going out of its safety range should cause a “trip” if it sustains for longer than 350 ms, and should not if it lasts for less than 250 ms (to filter out the effect of a noisy signal). Such a requirement, even when implementable, allows for inconsistent implementations since it is *non-deterministic* when the triggering condition lasts in-between [250ms, 350ms].

To resolve such non-determinism, at the requirements level we adopt a deterministic operator *Held_For_I* [7, p. 86], which becomes *true* at the first sampling point after the monitored condition has been enabled for $d-\delta L$ time units. For example, by substituting the expression $((\text{signal} \geq \text{setpoint}) \text{ Held_For_I } (300 - 50))$ into Eq. 1 above, we specify that the triggering condition sustaining for longer than 250 ms should cause a “trip”. Similarly, at the implementation level we adopt the *Timer_I* operator [7, p. 98] for counting the elapsed time of some monitored condition. The relationship between these two operators is proved as a theorem *TimerGeneral_I* [7, p. 99]: $(C \text{ Held_For_I } (\text{timeout} - \delta L))$ is equivalent to $(\text{Timer_I } (C) \geq \text{timeout} - \delta L)$. More precise definitions are given in Sec. 2.

We previously did not apply our approach [20,21] to verify FBs against timing requirements, given that IEC 61131-3 only includes simple timer blocks (i.e., on-delay, off-delay, and pulse timers), but not ones built from those timers. Furthermore, our requirements model for IEC 61131-3 timers describes idealized behaviour: as the monitored condition becomes enabled, the timer instantaneously responds (i.e., starts counting the duration of enablement).

In this paper, based on our experience on the Darlington Nuclear Shutdown Systems Trip Computer Software Redesign Project [30], and motivated by anticipated FB-dependant projects, we address the above issues by conducting realistic case studies. Each case study consists of the software requirements and FBD implementation for a subsystem of an industrial control system. Both implementations are built from IEC 61131-3 FBs, including the on-delay timer.

Fig. 1 summarizes our verification process and contributions. To incorporate the notion of tolerances, we reuse the timing operators *Held_For_I* (to formalize requirements) and *Timer_I* (to formalize implementations) as described above. The verification goal is that the proposed FBD implementations, included in

the Software Design Description (SDD), are: (a) *consistent*, or feasible, meaning that an output can always be produced on valid inputs; and (b) *correct* with respect to the timing requirements specified using *Held_For_I*, included in the Software Requirements Specification (SRS). Our previous results [20,21] from verifying IEC 61131-3 FBs provide a sound semantic foundation for formalizing and verifying PLC programs expressed using FBDs.

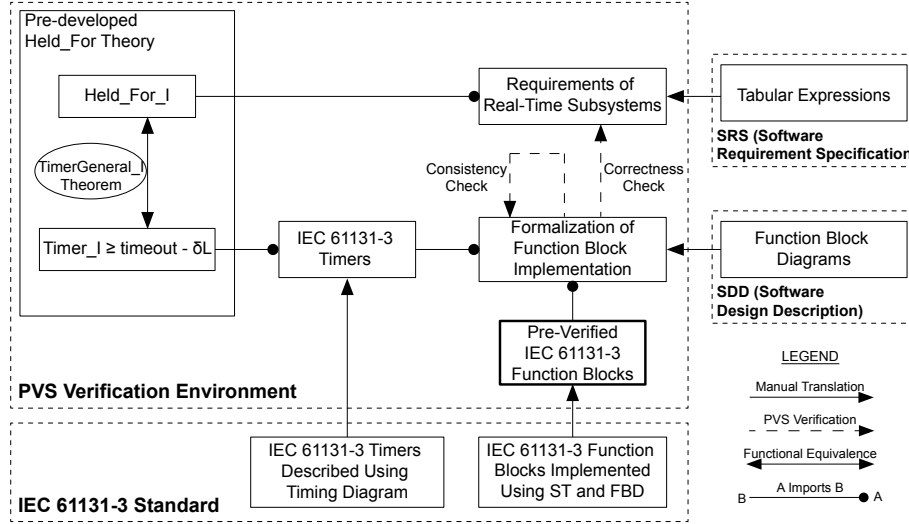


Fig. 1: Framework for Verification of FB Based Systems Timing Requirements

There are four contributions of this paper. First, to incorporate tolerances, we use the *Timer_I* operator to re-formalize all three IEC 61131-3 timer (Sec. 4). Second, for the two subsystems (Sec. 5 and 6), we use the re-formalized IEC 61131 timers for their proposed FBD implementations, and prove that they are feasible and satisfy the intended timing requirements in SRS. As the two subsystems are representative, we are assured that our approach suffices for the verification of real-time components. Third, we find issues of initialization failure and missing implementation assumptions, and resolve them by suggesting resolutions. Fourth, we identify patterns of proof commands (Sec. 8) that are amenable to strategies (or proof scripts) which will facilitate the automated verification of the feasibility and correctness of other subsystems.

Resources. Sources of the case studies (verified using PVS 6.0) are available at <http://www.cas.mcmaster.ca/~lawford/papers/FTSCS2014>.

2 Preliminaries

We first briefly introduce IEC 61131-3 standard, then review the use of tabular expressions, the relevant PVS theories of the *Held_For* timing operator [7] and of IEC 61131 FBs [20,21] that are adapted in our case studies in Secs. 5 and 6.

2.1 IEC 61131-3 Function Blocks

Programmable logic controllers (PLCs) are digital computers that are widely utilized in real-time and embedded control systems. To unify the syntax and semantics of programming languages for PLCs, the International Electrotechnical Committee (IEC) first published IEC 61131-3 in 1993 with revisions in 2003 [10] and 2013 [11]. A FB is programmed using either the textual languages Instruction List (IL) and Structured Text (ST) or the graphical languages Ladder Diagram (LD) and Function Block Diagram (FBD). IL languages is closer to machine code, whereas ST is high-level language that is block structured and syntactically resembles Pascal. Additionally, Sequential Function Chart (SFC) can also be used to describe the structure of a PLC program by displaying its sequential and parallel execution. SFC can be combined with any other IEC 61131-3 programming languages, e.g., steps are programmed in FBD. LD is based on the circuit diagrams of relay logic hardware which is suitable for binary logic operations. FBD can be used for binary and arithmetic operations in a graphical representation. As the most commonly used PLC programming language, our two case studies (Secs. 5 and 6) are implemented in FBDs.

2.2 Tabular Expressions

Tabular expressions [22,23] are an effective approach to describing conditionals and relations, thus ideal for documenting many system requirements. They are arguably easier to comprehend and to maintain than conventional mathematical expressions. Tabular expressions have well-defined formal semantics (e.g., [12,13]), and they are useful both in inspections and in testing and verification [29,31]. For our purpose of capturing the input-output requirements of timing function blocks, the tabular structure in Fig. 2 suffices: the input domain and the output range are partitioned into rows of, respectively, the first column (for input conditions) and the second column (for output results). The input column may be sub-divided to specify sub-conditions.

		Result
Condition	f	
C_1	$C_{1.1}$	res_1
	$C_{1.2}$	res_2

	$C_{1.m}$	res_m
...
C_n		res_n

IF C_1
IF $C_{1.1}$ **THEN** $f = res_1$
ELSEIF $C_{1.2}$ **THEN** $f = res_2$
 ...
ELSEIF $C_{1.m}$ **THEN** $f = res_m$
ELSEIF ...
ELSEIF C_n **THEN** $f = res_n$

Fig. 2: Semantics of Horizontal Condition Table (HCT)

In documenting input-output behaviours using horizontal condition tables (HCTs), we need to reason about their *completeness* and *disjointness*. Suppose there is no sub-condition, completeness ensures that at least one row is applicable to every input, i.e., $(C_1 \vee C_2 \vee \dots \vee C_n \equiv \text{True})$. Disjointness ensures that the rows do not overlap, e.g., $(i \neq j \Rightarrow \neg(C_i \wedge C_j))$. Similar constraints apply to the sub-conditions, if any. These properties can often be easily checked automatically using SMT solvers or a theorem prover such as PVS. A prototype tabular

expression toolbox for model-driven design can be used to verify the correctness of function tables by integrating PVS and the CVC3 SMT solver [5].

2.3 PVS

Prototype Verification System (PVS) [19] was developed by the Computer Science Lab of SRI International as an interactive environment for writing specifications and performing formal proofs. The PVS specification language is based on classical higher-order logic. PVS specifications are organized into theories that may include imported theorem, axioms, lemmas and goal theorem. PVS has a powerful interactive proof checker to perform sequent-style deductions. The inference mechanisms exploit the type system by generating Type Correctness Conditions (TCCs) and most of the TCCs are automatically discharged by the prover. A prelude is provided with PVS that includes over 1000 useful definitions, lemmas and theorems. The NASA PVS library is also a useful collection of formal developments contributed by the NASA Langley Formal Methods Team [17]. An example of using tabular expressions to specify and verify the Darlington Nuclear Shutdown System (SDS) in PVS can be found in [14].

Support for Tabular expressions in PVS

Two semantically equivalent table constructs are provided in the PVS specification language to support tabular expressions: **COND** and **TABLE**. The usage of table constructs causes PVS to generate disjointness and completeness proof obligations to guarantee that the tabular expression is well-defined. These can often be discharged automatically using the built in proof strategies, i.e., **COND-COVERAGE-TCC** and **COND-DISJOINT-TCC**. Useful feedback will be returned to the users if the PVS table cannot be automatically type-checked. However, for readability, it is more advisable for users to adopt the **TABLE** construct, which will be translated into the equivalent **COND** construct in PVS for typechecking and further proofs.

Proof Sequent in PVS

Sequent is the basic structure of the underlying calculus in PVS [24]. Syntactically, a PVS sequent is written as:

$$P_1, P_2, \dots, P_m \vdash Q_1, Q_2, \dots, Q_n$$

where P_i , $i = 1, 2, \dots, m$ are antecedent formulas, Q_j , $j = 1, 2, \dots, n$ are consequent formulas, and \vdash denotes entailment. There are implicit \wedge 's between the antecedent formulas and implicit \vee 's between the consequent formulas. Thus, the above sequent is equivalent to the following expression in predicate logic³:

$$P_1 \wedge P_2 \wedge \dots \wedge P_m \vdash Q_1 \vee Q_2 \vee \dots \vee Q_n$$

A PVS proof is considered as complete by determining whether at least one of its consequents is a logical consequence of its antecedents. In PVS, a sequent is displayed as follows:

³ We use \neg , \wedge , \vee , \Rightarrow , \forall , and \exists to denote, respectively, logical negation, conjunction, disjunction and implication, and universal and existential quantifiers. The corresponding notations in PVS are **NOT**, **&**, **OR**, **IMPLIES**, **FORALL**, and **EXISTS**.

$$\begin{array}{cc}
 \{-1\} & P1 \\
 \dots & \dots \\
 \{-m\} & Pm \\
 | & \text{-----} \\
 \{1\} & Q1 \\
 \dots & \dots \\
 \{n\} & Qn
 \end{array}$$

A PVS sequent can be discharged only if one of the following three cases applies: (1) **FALSE** occurs in the antecedents; (2) **TRUE** occurs in the consequents; or (3) the formula P occurs in both the antecedents and the consequents [7]. The prover maintains a graphical representation as a proof tree. Each node of the proof tree is a proof goal produces its offspring nodes by means of a proof step. The final goal is to discharge each node by invoking relevant proof commands.

A complex problem is often decomposed into smaller ones, in which each of these sub-problems is formulated and proved as a lemma. Thus, a complex PVS sequent is often discharged by splitting it into several sub-goals and by proving all of these sub-goals.

2.4 Modelling Time in the Physical Domain

As PLCs are being increasingly used in hard real-time systems, the modelling of time is a critical aspect in our formalization. In physical domain, we consider a discrete-time model, where a time series consists of equally distributed clock ticks with the period δ . More precisely:

$$\{t_0, t_1, t_2, \dots, t_n, \dots\} = \{0, \delta, 2\delta, \dots, n\delta, \dots\}$$

where $\delta \in \mathbb{R}^+$ is an arbitrarily small positive real number to represent the time interval between two consecutive clock ticks.

Constant **delta_t** is a positive real number, representing time internal δ . Type of **time** is defined by a non-empty set of non-negative real numbers. Type of **tick** is a dependent type of **time**, as the set of non-negative multiples of **delta_t**. Type of **noninit_elem** is a set of ticks without the initial one. We briefly introduce the time operations on **tick** [8]: e.g. **init** (initial tick), **pre** (previous tick) and **next** (next tick). Operator **init** is used to explicitly identify the initial values of internal or output variables of FBs in PLC-based system. Operators **pre** and **next** are used to specify the timing properties over consecutive clock ticks. Given a tick instant **t**, we use **rank(t)** to denote the ordinal of **t** in a discrete time setting. For example, tick instant 8.8 is the 4th tick given that **delta_t** = 2.2. Operator **rank** is used to define measure function guaranteeing the termination of recursive functions. An important yet simple proposition (i.e., **time_induction**) is used to prove some desired timing properties is an induction scheme over clock ticks [7]. The proposition says that predicate P is **TRUE** at arbitrary tick if (a) P is **TRUE** at the initial tick; and (b) P is **TRUE** at tick t_n (current tick) can be implied by the truth of P at tick t_{n-1} (previous tick).

```

delta_t: posreal
time: TYPE+ = nonneg_real
tick: TYPE = {t: time | EXISTS (n: nat): t = n * delta_t}
    
```



```

init(t: tick): bool = (t = 0)
noninit_elem: TYPE = {t: tick | NOT init(t)}
pre(t: noninit_elem): tick = t - delta_t
next(t: tick): tick = t + delta_t
rank(t: tick): nat = t / delta_t

time_induction: PROPOSITION
  FORALL (P: pred[tick]):
    (FORALL (t: tick): init(t) => P(t))
    & (FORALL (t: noninit_elem): P(pre(t)) => P(t))
    => (FORALL (t: tick): P(t))

```

Modelling Samples in the Software Domain

We use a variable $Sample : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ to denote the series of samples over time, such that the time of each sample (i.e., $Sample(n)$, $n \in \mathbb{N}$) maps to a valid clock tick. As shown in Fig. 3, realistically, the clock tick frequency $\frac{1}{\delta}$ in the physical domain should be significantly larger than the sampling frequency in the software domain. We bound sample intervals between $Tmin$ and $Tmax$, determined by considering the shortest time after which events must be detected.

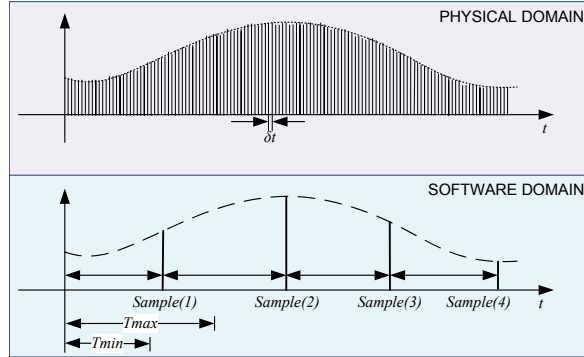


Fig. 3: Time Models in Physical and Software Domains Based on Tick Type [7]

As rates of clock ticks and sampling are distinct, a monitored signal Pf that rapidly changes between two consecutive samples (called a “spike”) can cause inconsistent results produced in the two domains. To rule out such scenarios, we define a predicate subtype **FilteredTickPred**⁴ that only allows monitored conditions which remain unchanged between consecutive samples:

```

FilteredTickPred?(P: PRED[tick]): bool =
  (FORALL t0: P(t0) /= P(next(t0)) =>
    (FORALL (t: tick[delta_t] | t0 < t AND t <= t0 + Tmax):
      P(next(t0)) = P(t)))
  AND (FORALL (t: tick[delta_t] | t <= Tmax): P(t) = P(0))

FilteredTickPred: TYPE+ = (FilteredTickPred?)
Pf: VAR FilteredTickPred

```

⁴ An example of using the subtype **FilteredTickPred** to constrain input signals can be found in the verification story of the *Pushbutton* subsystem (Sec. 6).

2.5 Operators for Specifying Timing Requirements

As discussed in Sec. 1, we define the infix operator:

$$\text{Held_For} : (\text{tick} \rightarrow \mathbb{B}) \times (\text{tick} \rightarrow \mathbb{R}_{>0}) \times (\text{tick} \rightarrow \mathbb{R}_{\geq 0}) \times (\text{tick} \rightarrow \mathbb{R}_{\geq 0}) \rightarrow (\text{tick} \rightarrow \text{bool})$$

to specify a common functional timing requirement, e.g., $P \text{ Held_For } (d, \delta L, \delta R)$, that a monitored boolean condition P should sustain over a positive time duration d , with non-negative left tolerance δL and right tolerance δR . More precisely,

$$P \text{ Held_For } (d, \delta L, \delta R)(t_{\text{now}}) \equiv (\exists t_j : t_{\text{now}} - t_j \geq d \wedge (\forall t_i : t_j \leq t_i \leq t_{\text{now}} \bullet P(t_i)))$$

where $d \in [d(t_{\text{now}}) - \delta L(t_{\text{now}}), d(t_{\text{now}}) + \delta L(t_{\text{now}})]$. In our model of time, inputs and outputs are represented as functions mapping ticks to values. For example, the left tolerance may change from $\delta L(t_1)$ to $\delta L(t_2)$. However, as discussed in Sec. 1, the behaviour of *Held_For* is nondeterministic when P has last in-between $[d - \delta L, d + \delta R]$.

To resolve the non-determinism in *Held_For*, we define two refinement operators: *Held_For_S* and *Held_For_I*. Both operators are deterministic by fixing the duration d in the above definition of *Held_For* as $d(t_{\text{now}}) - \delta L(t_{\text{now}})$. We will only see *Held_For_I* in the case studies, but it is defined in terms of *Held_For_S*. *Held_For_S* is a partial function on *tick* that produces values only at points of sampling (i.e., it is undefined on ticks in-between samples).

```
Held_For_S(P, duration, Sample)(ne): bool =
  EXISTS (n0 | Sample(ne) - Sample(n0) >= duration):
    FORALL (n: nat | n0 <= n AND n <= ne): P(Sample(n))
```

On the other hand, *Held_For_I* is a totalized version of *Held_For_S*: its value at time t , where $\text{Sample}(n) \leq t < \text{Sample}(n + 1)$, is equivalent to that produced at time $\text{Sample}(n)$ (i.e., the closest left sample calculated by *Left_Sample*).

```
Left_Sample(Sample, t):
  {n: nat | Sample(n) <= t AND t < Sample(n + 1)} =
  sup(LAMBDA (n: nat): Sample(n) <= t)
```

```
Held_For_I(P, duration, Sample)(t): bool =
  Held_For_S(P, duration, Sample)(Left_Sample(Sample, t))
```

2.6 Implementing the Held_For_I Timing Operator

We use *Timer_I* (defined in terms of *Timer_S*) to implement the *Held_For_I* timing operator. *Timer_I* agrees on outputs from *Timer_S* at sample points and keep the same value at any clock tick until the next sample point (this is analogous to how *Held_For_I* is related to *Held_For_S*).

```
Timer_I(P, Sample, TimeOut)(t): tick =
  Timer_S(P, Sample, TimeOut)(Left_Sample(Sample, t))
```

where *Timer_S* [7, p. 97] counts, starting from the closest left sample to the clock tick in question, for how long the monitored condition P has been enabled, and stops counting when *TimeOut* is reached. The output type of *Timer_S* is *tick*, calculated from how many samples P has been held across. *Timer_S* function is updated the value through a *TimerUpdate* function by passing condition at both the current and last samples, the previous value of the timer, the timeout value and the elapsed time since the last update of the timer.

```

TimerUpdate(CurrentP, PreviousP,
             Timeout, PreviousTimerValue, step): tick =
TABLE
    %-----+-----+-----+-----+
    |[ PreviousTimerValue < Timeout| PreviousTimerValue >= Timeout]|
%-----+-----+-----+-----+
| CurrentP
  & PreviousP | PreviousTimerValue + step | PreviousTimerValue ||
%-----+-----+-----+-----+
| NOT(CurrentP
  & PreviousP)| 0 | 0 ||
%-----+-----+-----+-----+
ENDTABLE

Timer_S(P, Sample, Timeout)(ne): RECURSIVE tick =
TABLE
    %-----+-----+-----+-----+
    | ne = 0 | TimerUpdate(P(Sample(ne)), FALSE, Timeout, 0, 0) ||
%-----+-----+-----+-----+
    | ne > 0 | TimerUpdate(P(Sample(ne)), P(Sample(ne - 1)), Timeout,
      Timer_S(P, Sample, Timeout)(ne - 1),
      Sample(ne) - Sample(ne - 1)) ||
%-----+-----+-----+-----+
ENDTABLE
MEASURE ne

```

As mentioned in Sec. 1, the theorem *TimerGeneral.I* is proved to ensure that *Timer.I* is a proper implementation for *Held_For.I*.

```

TimerGeneral_I: THEOREM
  Held_For_I(P, timeout - delta_L, Sample)(t) IFF
  Timer_I(P, timeout - delta_L, Sample) >= timeout - delta_t

```

Two auxiliary lemmas are used during our verification work: *FILTER_TRUTH1* and *Sample_PROPERTY3*. Lemma *FILTER_TRUTH1* proves that if a signal (of type *FilteredTickPred?*) has the same value at sample n and $n+1$, the signal has the same value of sample n for all ticks between sample n and $n+1$. Lemma *Sample_PROPERTY3* proves that the time of sample point $n2$ is later than $n1$ at timeline, if $n2 > n1$.

```

FILTER_TRUTH1: LEMMA
  Pf(Sample(n)) = Pf(Sample(n + 1)) IMPLIES
  (FORALL (t | t > Sample(n) AND t < Sample(n + 1)):
    Pf(t) = Pf(Sample(n)))

```

```

Sample_PROPERTY3: LEMMA Sample(n2) > Sample(n1) IMPLIES n2 > n1

```

2.7 A Formal Approach to Specifying and Verifying FBs

Our reported approach [20,21] fits into the timing model as described above. For each FB, its input-output requirements and FBD implementation are formalized in PVS as two (higher-order) predicates, parameterized input and output lists. Each input or output is represented as a timed sequence (or trajectory) mapping clock ticks to values (e.g., `[tick -> real]`). The *requirements predicate* (say *FB_REQ*) returns true if its outputs are related to inputs in the expected

way (specified using tabular expressions) across all time ticks. The *implementation predicate* (say FB_IMPL) of a composite FB (e.g., see Sec. 5) is constructed by composing, using logical conjunction, the requirements predicates of its component FBs as configured in its FBD implementation. All inter-connectives in the FBD implementation are hidden using an existential quantification.

To illustrate our approach for a FBD implementation, let us consider the following composite FB and its formalizing requirement predicate in Fig. 4:

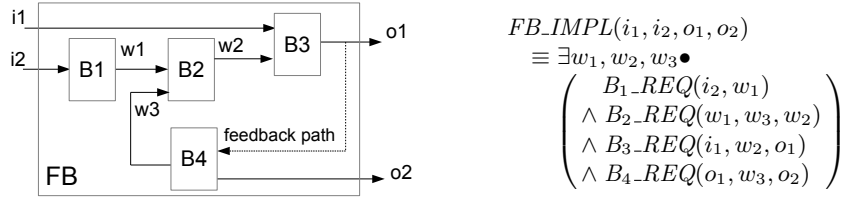


Fig. 4: Composite FB Implementation in FBD and Its Formalizing Predicate

As shown in Fig. 4, the FBD implementation (as supplied by IEC 61131-3) consists of four internal blocks, B_1 , B_2 , B_3 , and B_4 , that are already formalized (i.e., their formalizing predicates B_1_REQ, \dots, B_4_REQ exist). Note the output o_1 is fed as an input of block B_4 . It is very common for control system to feed output as another input at next time tick. The high-level black-box requirement (as opposed to the implementation supplied by IEC 61131-3) for each internal FB constrains upon its inputs and outputs, documented by tabular expressions. Similarly, the requirement of a composite FB is described in tabular expressions. To describe the overall behaviour of the above composite FB, we take advantage of our formalization mechanism being *compositional*. In other words, a composite FB is formalized by existentially quantifying over the list of its inter-connectives (i.e., w_1, w_2 and w_3), such that the conjunction of predicates that formalize the internal components hold.

With the formalized tabular requirement (say FB_REQ) and FBD implementation (say FB_IMPL), we perform two kinds of verification in PVS: consistency check and correctness check.

Proof of Consistency To ensure that the implementation is consistent or feasible, we prove that for each list of input trajectories, there exists at least one list of output trajectories such that FB_IMPL is defined:

$$\vdash \forall i_1, i_2 \bullet \exists o_1, o_2 \bullet FB_IMPL(i_1, i_2, o_1, o_2) \quad (2)$$

Proof of Correctness To ensure that the implementation is correct with respect to its intended requirement, we prove that input and output trajectories produced by the implementation satisfy the requirements:

$$\vdash \forall i_1, i_2 \bullet \forall o_1, o_2 \bullet FB_IMPL(i_1, i_2, o_1, o_2) \Rightarrow FB_REQ(i_1, i_2, o_1, o_2) \quad (3)$$

3 Architecture for System Theories

As PVS model, we have theories for requirement specification, implementation specification and proof obligations. Each imports formalizations (e.g., constants) defined in corresponding sub-theories. Fig. 5 shows the relationship between these sub-theories and each is explained as follows:

1. Common Library (**comlibrary**): Common imports used by both the SRS and SDD (e.g., time modelling);
2. SRS Library (**srslibrary**): Common imports relevant to the SRS domain (e.g., types, common functions on those types);
3. SDD Library (**sddlibrary**): Common imports relevant to the SDD domain (e.g., types, pre-verified function blocks);
4. SRS Functions (**srsfunctions**): Modelled functions from the SRS;
5. SDD Functions (**sddfunctions**): Modelled functions from the SDD;
6. Abstractions Functions (**abstractions**): Abstraction functions used to map types between the SRS and SDD domain; and
7. Obligations Functions (**obligations**): Design verification between the SRS and SDD domains.

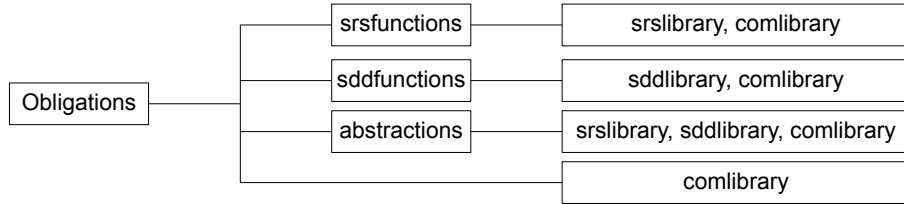


Fig. 5: PVS Theories Structure

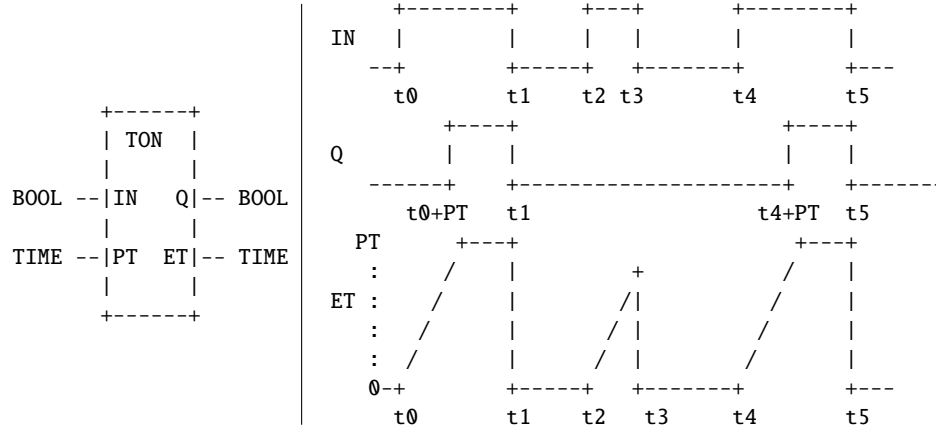
4 Formalizing IEC 61131-3 Timers with Tolerances

We present the first contribution of this paper: incorporating the notion of *timing tolerances* [28] (i.e., the controller's reaction to the environment is associated with a delay) into the formalization of the black-box, input-output requirements of IEC 61131-3 timers. Such formalization improves the accuracy of our previous work [20,21] by making the resulting requirements models *implementable*.

In IEC 61131-3 there are three timer FBs: *TON* (on-delay), *TOF* (off-delay), and *TP* (pulse) timers. Although the case studies presented in this paper (Secs. 5 and 6) only make use of the *TON* block, we present the re-formalizations on all these three timer FBs in this section .

4.1 On-delay Timer (TON)

The *TON* block is commonly used as components of safety-critical systems. For example, it can be used to determine the behaviour of a pushbutton (i.e., determining if the button is de-bouncing or stuck, as we will see in Sec. 6). Fig. 6


 Fig. 6: *TON* timer declaration and definition in timing diagram [10]

shows, extracted from IEC 61131-3, the input-output declaration (on the LHS) and a timing diagram⁵ (on the RHS) illustrating the expected behaviour of the *TON* block. The *TON* block is declared with two inputs (a boolean condition *IN* and a time period of length *PT*) and two outputs (a boolean value *Q* and a length *ET* of time period). Timer *TON* monitors the input condition *IN* and sets the output *Q* as true whenever *IN* remains enabled for longer than a time period of some input length *PT*. If the monitored input *IN* has been enabled for some time $t < PT$, then the timer sets the output *ET* (i.e., elapsed time) with value t ; otherwise, it sets *ET* with value *PT*.

Condition		Result
Condition		last_enabled
$\neg IN_{-1} \wedge IN$		t
$IN_{-1} \vee \neg IN$		NC

Condition		Result
Condition		Q
$IN \wedge (d \geq PT)$		TRUE
$IN \wedge (d < PT)$		FALSE
$\neg IN$		FALSE

Condition		Result
Condition		ET
$IN \wedge (d \geq PT)$		PT
$IN \wedge (d < PT)$		d
$\neg IN$		0

where d stands for duration, $d = t - \text{last_enabled}$

 Fig. 7: Tabular Requirements of Timer *TON*: Idealized Behaviour

The use of a timing diagram by IEC 61131-3 to describe the expected behaviour of the *TON* block (and the other two timers) is limited to an incomplete set of use cases. As a result, we attempted [20][21] to use function tables to formalize the black-box, input-output requirements of the three timer blocks (on-delay, off-delay, and pulse timers) listed in IEC 61131-3. Fig. 7 shows our previous attempt of the requirements specification of the *TON* block, where a time stamp *last_enabled* is used to record the exact time (with no delay) that the input condition *IN* just becomes enabled. However, the requirements model in Fig. 7 is not implementable because it describes idealized behaviour: the timer (or the controller) reacts instantaneously to changes in the environment.

As part of the contribution of this paper, we revise the function tables of all three timers in IEC 61131-3 by incorporating the notion of timing tolerances [28],

⁵ The horizontal axis is labelled with time instants t_i , $i \in 0..5$

Condition	Result	Condition	Result
	Q		ET
$d \geq PT$	TRUE	$d \geq PT$	PT
$d < PT$	FALSE	$d < PT$	d
		$\neg IN$	0

where d stands for duration, $d = (IN)Timer_I(PT, \delta L, \delta R)$

Fig. 8: Tabular Requirements of Timer *TON*: Timing Tolerances Incorporated

i.e., there is a delay associated with the controller's reaction to the environment. To achieve this, we use the pre-verified operator **Timer_I** (Sec. 2) to redefine requirements of the three timers (Fig. 8).

4.2 Off-delay Timer (TOF)

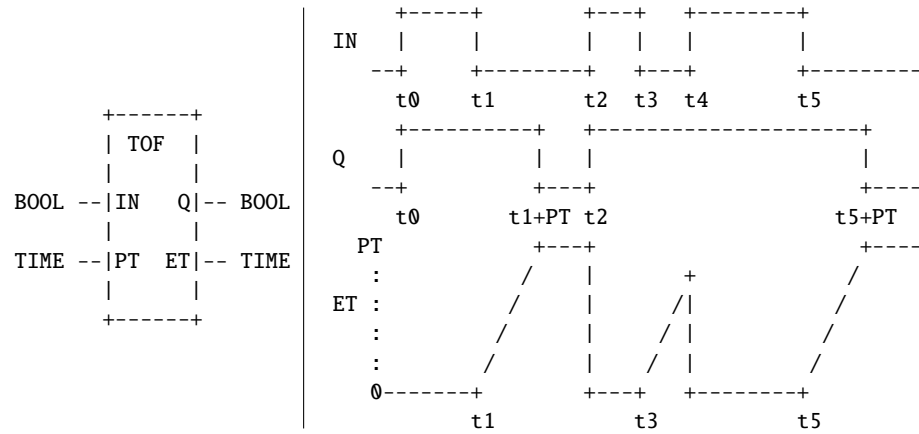


Fig. 9: *TOF* Timer Declaration and Definition in Timing Diagram [10]

The *TOF* block delays the falling edge of a boolean input by a specified duration. For example, a *TOF* block can be used to keep cooling fans on for a specific time period after the oven has been turned off. Fig. 9 shows, extracted from IEC 61131-3, the input-output declaration (on the LHS) and a timing diagram (on the RHS) illustrating the expected behaviour of the *TOF* block. The *TOF* block is declared with two inputs (a boolean condition *IN* and a time period of length *PT*) and two outputs (a boolean value *Q* and a length *ET* of time period). Timer *TOF* monitors the input condition *IN* and sets the output *Q* as false whenever *IN* remains disabled for longer than a time period of some input length *PT*. If the monitored input *IN* has been disabled for some time $t < PT$, then the timer sets the output *ET* (i.e., elapsed time) with value t ; otherwise, it sets *ET* with value *PT*.

Similar as *TON*, an auxiliary function *last_disabled* is used to document the time when *IN* was last disabled. Fig. 11 shows the behaviour of outputs *Q* and *ET* using *last_disabled* function. The re-formalized definition of *TOF* is shown in Fig. 11 using **Timer_I**.

Condition	Result	Condition	Result
$IN_{-1} \wedge \neg IN$	t	$\neg IN \wedge (t - \text{last_disabled} \geq PT)$	FALSE
$\neg IN_{-1} \vee IN$	NC	$\neg IN \wedge (t - \text{last_disabled} < PT)$	TRUE
		IN	TRUE

Condition	Result
$\neg IN \wedge (t - \text{last_disabled} \geq PT)$	PT
$\neg IN \wedge (t - \text{last_disabled} < PT)$	t - last_disabled
IN	0

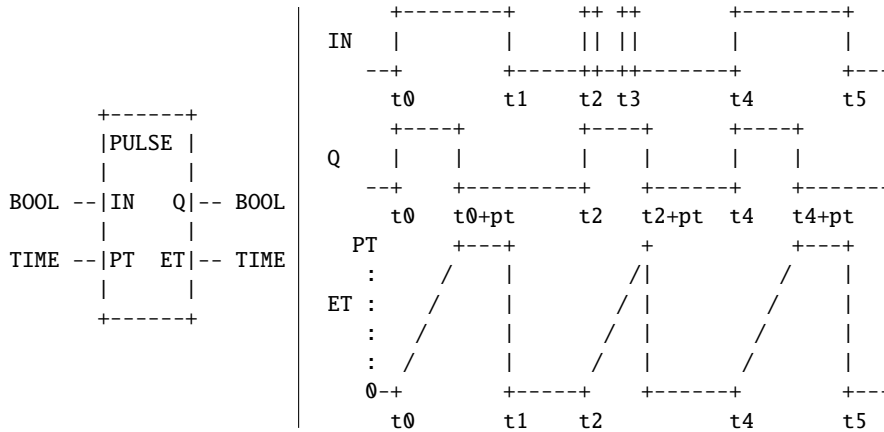
 Fig. 10: Tabular Requirements of Timer *TOF*: Idealized Behaviour

Condition	Result
$(\neg IN) \text{Timer_I}(PT, \delta L, \delta R) \geq PT$	FALSE
$(\neg IN) \text{Timer_I}(PT, \delta L, \delta R) < PT$	TRUE

Condition	Result
$(\neg IN) \text{Timer_I}(PT, \delta L, \delta R) \geq PT$	PT
$(\neg IN) \text{Timer_I}(PT, \delta L, \delta R) < PT$	$(\neg IN) \text{Timer_I}(PT, \delta L, \delta R)$
IN	0

 Fig. 11: Tabular Requirements of Timer *TOF*: Timing Tolerances Incorporated

4.3 Pulse Timer (TP)


 Fig. 12: *TP* Timer Declaration and Definition in Timing Diagram [10]

The *TP* block acts as a pulse generator: as soon as the input (*IN*) is detected to be true, it generates a pulse and remains true for a constant time period. The elapsed time that *Q* has remained true can be monitored at *ET*. Fig. 12 shows, again extracted from IEC 61131-3, the input-output declaration (on the LHS) and a timing diagram (on the RHS) illustrating the expected behaviour of the *TP* block. Function *pulse_start_time* records the time when *Q* last goes

4 Formalizing IEC 61131-3 Timers with Tolerances

to true. *Held_For_st* is a more restricted operator than *Held_For*, insisting that the starting time of *duration* is *ts*. Using these three auxiliary functions, the *TP* block is defined as shown in Fig. 13. As argued in [20], we found ambiguous behaviour with the *TP* block. In this paper, we resolve the issue by providing unambiguous formalization in tabular expressions. The formalizations of *Held_For* and *Held_For_st* in PVS are as follows:

```
Held_For(P: pred[tick],duration: posreal)(t: tick): bool =
  EXISTS(t_j: tick): (t - t_j >= duration) &
    (FORALL (t_n: tick | t_n >= t_j & t_n <= t): P(t_n))
Held_For_ts(P: pred[tick],
  duration: posreal,ts: tick)(t: tick): bool =
  (t-ts >= duration)
  & (FORALL (t_n: tick | t_n >= ts & t_n <= t): P(t_n))
```

We re-formalize the timer *TP* using **Timer_I** as shown in Fig. 14.

		Result	
Condition		Q	
$\neg Q_{-1}$	$\neg IN_{-1} \wedge IN$	1	
	$IN_{-1} \vee \neg IN$	0	
Q_{-1}	Held_For(Q, PT)	0	
	\neg Held_For(Q, PT)	1	

		Result	
Condition		pulse_start_time	
$\neg Q_{-1} \wedge Q$		t	
$Q_{-1} \vee \neg Q$		NC	

			Result	
Condition			ET	
Q			t - pulse_start_time	
$\neg Q$	\neg Held_For_ts(IN, PT, pulse_start_time)		0	
	Held_For_ts(IN, PT, pulse_start_time)	IN	PT	
		\neg IN	0	

Fig. 13: Tabular Requirements of Timer *TP*: Idealized Behaviour

		Result	
Condition		Q	
$\neg Q_{-1}$	$\neg IN_{-1} \wedge IN$	TRUE	
	$IN_{-1} \vee \neg IN$	FALSE	
Q_{-1}	(IN)Timer_I(PT, δL , δR) \geq PT	FALSE	
	(IN)Timer_I(PT, δL , δR) $<$ PT	TRUE	

			Result	
Condition			ET	
Q			(IN)Timer_I(PT, δL , δR)	
$\neg Q$	(IN)Timer_I(PT, δL , δR) $<$ PT		0	
	(IN)Timer_I(PT, δL , δR) \geq PT	IN	PT	
		\neg IN	0	

Fig. 14: Tabular Requirements of Timer *TP*: Timing Tolerances Incorporated

Remark. The essence of our first contribution presented in this section is that we incorporate the notion of timing tolerances, via the use of the pre-verified

operator **Timer_I**, into the requirements of IEC 61131 timers so that they are implementable. This allows us to conduct case studies (Secs. 5 and 6) on implementing (and verifying) subsystems implemented using the IEC 61131-3 timers.

5 Case Study 1: The *Trip Sealed-In* Subsystem

In this section we apply our approach (Sec. 2.7) to verify the *Trip Sealed-In* subsystem. We identify an initialization error and suggest resolution.

5.1 Input-Output Declaration and Informal Description

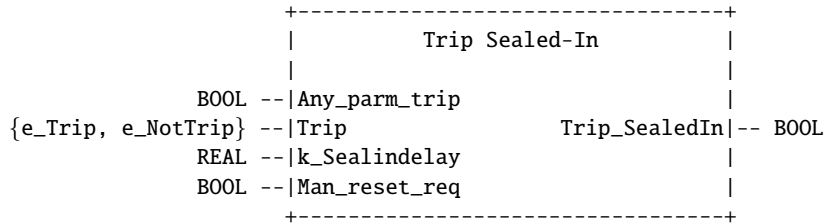


Fig. 15: Input-Output Declaration of *Trip Sealed-In* subsystem

Trip Sealed-In is a generic subsystem which monitors: 1) a set of sensor values; and 2) an alarm value produced by some other subsystem. It signals an alarm (denoted by the output *Trip_SealedIn*), which may be manipulated by other subsystems, when two conditions are met. First, any of the monitored sensor values goes out of its safety range (called a parameter trip and denoted by an input condition *Any_parm_trip*). Second, the monitored input alarm is signalled continuously for longer than some preset constant $k_Seglindelay$ ⁶ amount of time (denoted by an input value *Trip* of enumerated type $\{e_Trip, e_NotTrip\}$). Once the alarm *Trip_SealedIn* is activated, it is not deactivated until when all monitored sensor values fall back within their safety ranges, and when a manual reset is requested (denoted as an input *Man_reset_req*).

5.2 Tabular Requirements Specification with Timing Tolerances

We use a function table (Fig. 16) to perform a complete and disjoint analysis on the input domains. To incorporate timing tolerances into the requirements of *Trip Sealed-In*, we use the non-deterministic *Held_For* operator (Sec. 2.6) to specify a sustained window of duration $[k_Sealindelay - \delta L, k_Sealindelay + \delta R]$.

However, for the purpose of verification in PVS, we reformulate the non-deterministic behaviour of Fig. 16 in a recursive function⁷ using the deterministic *Held_For_I* operator to impose the constraint that only a single value (i.e., $k_Sealindelay - delta_L$ where both are declared constants) is chosen from the duration and is used consistently for detecting sustained events.

⁶ The k_- name prefix is reserved for system-wide constants.

⁷ For proving termination, its progress is measured using discrete time instants $rank(t)$.

5 Case Study 1: The *Trip Sealed-In* Subsystem

Condition		Result
		<i>Trip-SealedIn</i>
<i>Any_parm_trip</i>	$(Trip = e_Trip) \text{ Held_For } (k_Sealindelay, \delta L, \delta R)$	TRUE
	$\neg[(Trip = e_Trip) \text{ Held_For } (k_Sealindelay, \delta L, \delta R)]$	NC
$\neg Any_parm_trip$	<i>Man_reset_req</i>	FALSE
	$\neg Man_reset_req$	NC

Fig. 16: *Trip Sealed-In*: (non-deterministic) Requirements of with Tolerances

```

Trip_SealedIn_f(Any_parm_trip: pred[tick],
                Trip          : [tick->{e_Trip, e_NotTrip}],
                Man_reset_req: pred[tick])(t: tick)
: RECURSIVE bool =
  IF init(t) THEN TRUE ELSE
  LET
    TRIPPED = LAMBDA (t: tick): Trip(t) = e_Trip,
    HELD     = Held_For_I(TRIPPED, k_Sealindelay-delta_L, Sample)(t),
    PREV     = Trip_SealedIn_f(Any_parm_trip, Trip, Man_reset_req)(pre(t)) IN
    TABLE
      %-----%
      | Any_parm_trip(t)    &    HELD                | TRUE||
      %-----%
      | Any_parm_trip(t)    & NOT HELD                | PREV||
      %-----%
      | NOT Any_parm_trip(t) &    Man_reset_req(t)    | FALSE||
      %-----%
      | NOT Any_parm_trip(t) & NOT Man_reset_req(t)    | PREV||
      %-----%
    ENDTABLE
  ENDIF
  MEASURE rank(t)

```

By using the above recursive function *Trip_SealedIn_f*, over all clock ticks, we have a deterministic requirements (Fig. 17) for the *Trip Sealed-In* subsystem.

```

Trip_SealedIn_REQ(Any_parm_trip: pred[tick],
                  Trip          : [tick->{e_Trip, e_NotTrip}],
                  Man_reset_req: pred[tick],
                  TripSealedIn : pred[tick]): bool
= FORALL (t: tick):
  TripSealedIn(t) =
    Trip_SealedIn_f(Any_parm_trip, Trip, Man_reset_req)(t)

```

Fig. 17: *Trip Sealed-In*: (deterministic) Requirements of with Tolerances in PVS

Remark. Compared with Fig. 16, the use of the operator *Held_For_I* resolves the non-determinism by fixing the level of timing tolerance (i.e., as long as the alarm input *Trip* has been activated for or longer than $k_Sealindelay - \delta L$, the *Trip Sealed-In* subsystem is guaranteed to detect it and act accordingly).

5.3 Formalizing the FBD Implementation

We propose a FBD implementation (Fig. 18) which should satisfy the requirements (Fig. 17).

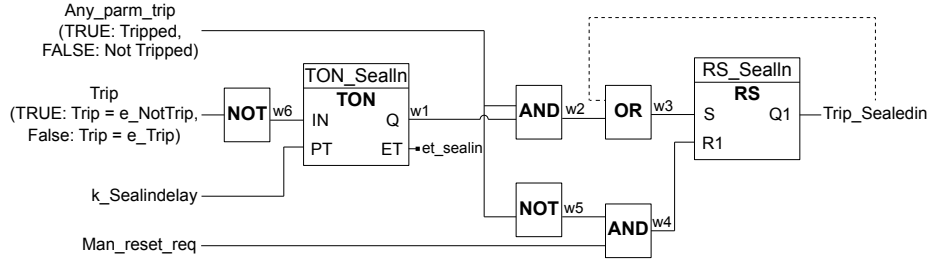


Fig. 18: *Trip Sealed-In* implementation in FBD

We use the IEC 61131 *TON* timer (see Sec. 4 for its formalization incorporated with tolerances) to implement the use of the *Held_For_I* operator (subject to a correctness proof which we will discuss below). As the recursive function used to define the requirements depends on the value of itself (at the previous time tick), we specify a feedback loop (dashed line) in the implementation.

The use of the left-most *NOT* (negation) block in Fig. 18 has to do with the mismatch between type at the requirements level (i.e., $\{e_Trip, e_NotTrip\}$) and that at the FB implementation level (i.e., boolean): somehow the engineers interpret value e_Trip as *FALSE* and $e_NotTrip$ as *TRUE*, so a conversion is necessary to make sure the *Trip Sealed-In* has the consistent interpretation. The requirements that the alarm output *Trip_Sealedin* is deactivated (or reset) when there is no parameter trips, and when a manual reset is requested, is implemented using a standard block *RS* (reset dominant flip flop).

To prove that the proposed FBD implementation of *Trip Sealed-In* (Fig. 18) is both feasible and conforms to its requirements (Fig. 17), we first follow our approach (Sec. 2.7) to formalize it by composing, using conjunction, the formalizing predicates⁸ of all component blocks (all inter-connectors are hidden using an existential quantification.):

$$\begin{aligned}
 & Trip_sealedin_IMPL(Any_parm_trip, Trip, Man_reset_req, Trip_SealedIn) \\
 & \equiv \exists w_1, w_2, w_3, w_4, w_5, w_6, et_sealin \bullet \\
 & \quad \left(\begin{array}{l}
 NOT(Trip, w_6) \\
 \wedge TON(w_6, k_Sealindelay - \delta L, w_1, et_sealin) \\
 \wedge CONJ(Any_parm_trip, w_1, w_2) \\
 \wedge DISJ(w_2, Trip_SealedIn, w_3) \\
 \wedge NOT(Any_parm_trip, w_5) \\
 \wedge CONJ(w_5, Man_reset_req, w_4) \\
 \wedge RS(w_4, w_3, Trip_SealedIn)
 \end{array} \right)
 \end{aligned}$$

⁸ Predicates *NOT* (logical negation), *CONJ* (logical conjunction), *DISJ* (logical disjunction), *TON* (on-delay timer), and *RS* (reset dominant latch).

5.4 Proofs of Consistency and Correctness

First, we prove that the FBD implementation (Fig. 18) is feasible by instantiating formula 2 in Sec. 2.7:

$$\begin{aligned} &\vdash \forall \text{Any_parm_trip}, \text{Trip}, \text{Man_reset_req} \bullet \\ &\quad \exists \text{Trip_SealedIn} \bullet \text{Trip_sealedin_IMPL}(\text{Any_parm_trip}, \text{AbstParmTrip_timed}(\text{Trip}), \\ &\quad \text{Man_reset_req}, \text{Trip_SealedIn}) \end{aligned}$$

The abstraction function *AbstParmTrip_timed* handles the mismatched types of input *Trip* at levels of requirements and implementation (e.g., *e_NotTrip* mapped to *TRUE*). We discharge the consistency proof using proper instantiations.

Second, we prove that the FBD implementation is correct with respect to Fig. 17, considering timing tolerances, by instantiating formula 3 in Sec. 2.7:

$$\begin{aligned} &\vdash \forall \text{Any_parm_trip}, \text{Trip}, \text{Man_reset_req}, \text{Trip_SealedIn} \bullet \\ &\quad \text{Trip_sealedin_REQ}(\text{Any_parm_trip}, \text{Trip}, \text{Man_reset_req}, \text{Trip_SealedIn}) \\ &\quad \Rightarrow \text{Trip_sealedin_IMPL}(\text{Any_parm_trip}, \text{AbstParmTrip_timed}(\text{Trip}), \\ &\quad \text{Man_reset_req}, \text{Trip_SealedIn}) \end{aligned}$$

As there is a feedback loop in the FBD implementation (Fig. 18), our strategy of discharging the correctness theorem is by mathematical induction (using the *time_induction* proposition in Sec. 2) over tick values. In both the base and inductive cases, we have to expand the definition of the *Timer_I* operator (Sec. 2), because it is used to formalize the requirements of the *TON* timer that contributes to the FBD implementation.

5.5 Proof Discussion

We now summarize the proof strategies applied on these verification work. For the consistency theorem, we explicitly formulate the requirements of internal components as functional form to assist instantiation steps. The consistency proof can be discharged easily with proper instantiations (using these separately defined functions) to existentially quantified inter-connectors. The completion of consistency proof proves the feasibility of the design. We need to further verify the correctness of design against the requirements, considering of timing tolerance.

For the correctness theorem, we introduce proof strategies which consist of three important steps.

1. An inductive proving approach (i.e., **time_induction**) is applied. The mathematical induction (i.e., **time_induction**) allows us to apply induction over tick values. We use this induction scheme to prove the SRS block and the corresponding SDD implementation are equivalent at all ticks within acceptable timing tolerances.
2. Based on Step 1, for the proof branch of base case (i.e., initial case of $t=0$), expending the definition of *SEL* block and applying basic PVS commands are sufficient. For the proof branch of inductive step, an important general theorem **TimerGeneral_I** is reused and instantiated properly. A large amount of proving effort can be simplified.

- Based on Step 1 and 2, for each proof branch after instantiating theorem **TimerGeneral_I**, we encounter a situation where the form of the lambda expression of inter-connector w_6 mismatched the form appears as the first argument of **Held_For_I** operator. The occurrence of λ -expressions can be daunting during the proof in PVS. We introduce an auxiliary lemma (i.e., **PROPERTY0**) to prove their logical equivalence. Lemma **PROPERTY0** can be proved by *extensinality* axiom for the required types (i.e., the types of two λ -expressions). The PVS specification of **PROPERTY0** is as follows:

PROPERTY0: LEMMA

FORALL c_Trip:

```
(LAMBDA (t: tick[delta_t]): % lambda expression with w6
  NOT COND (c_Trip(t) = e_Trip) -> FALSE, ELSE -> TRUE ENDCOND) =
  (LAMBDA (t: tick[delta_t]): % lambda expression occurs in Held_For_I
    c_Trip(t) = e_Trip)
```

However, by trying to prove the base case in Step 2 (when $t = 0$), we found that the initial value of output $Q1$ of the *RS_Sealin* block and the initial value of the subsystem output *Trip_SealedIn* — these two values are directly connected in the initial FBD implementation (Fig. 18) — are inconsistent. According to the SRS, value of *Trip_SealedIn* is initialized to *TRUE*, whereas that of $Q1$ is *FALSE*. We resolve this issue of inconsistency by suggesting a revised FBD implementation (Fig. 19) and prove that it is correct with respect to Fig. 17.

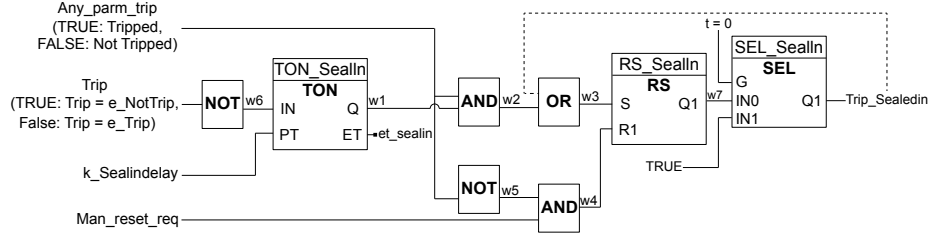


Fig. 19: Revised *Trip Sealed-In* implementation in FBD

In this revised implementation, we add an IEC 61131-3 selection block *SEL_Sealin*, acting as a multiplexer to discriminate the value of $Q1$ (at the initial tick and at the non-initial tick) that is output as *Trip_SealedIn*. If input G is *TRUE* (i.e., initial case of $t=0$), output *Trip_SealedIn* is set to *TRUE*; otherwise, it is set to the value from input INO (i.e., from output $Q1$ of block *RS_Sealin*). By imposing block *SEL_Sealin* and applying the above three proof steps, the correctness theorem of the *Trip Sealed-In* subsystem can be completed.

5.6 Proof Structure

As shown in Figs. 20 and 21, we summarize the proof structure for the consistency and correctness checks of the *Trip Sealed-In* subsystem:

- Consistency check:

5 Case Study 1: The *Trip Sealed-In* Subsystem

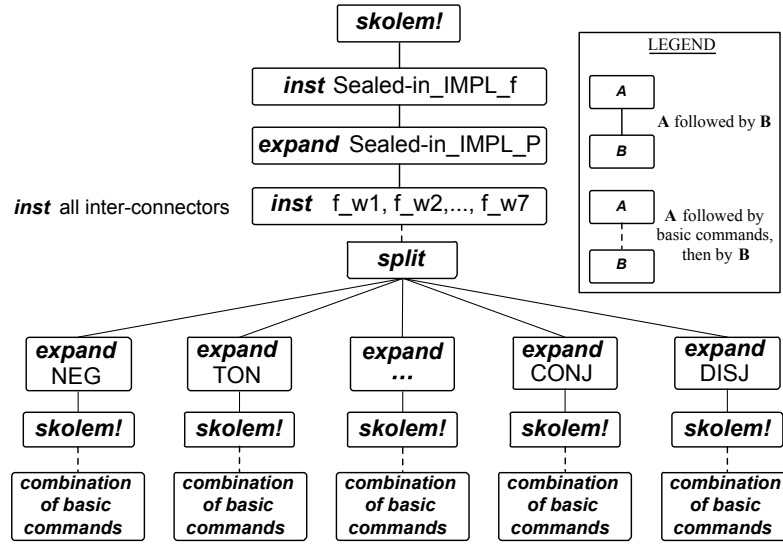


Fig. 20: Consistency Proof Structure for the *Trip Sealed-In* Subsystem

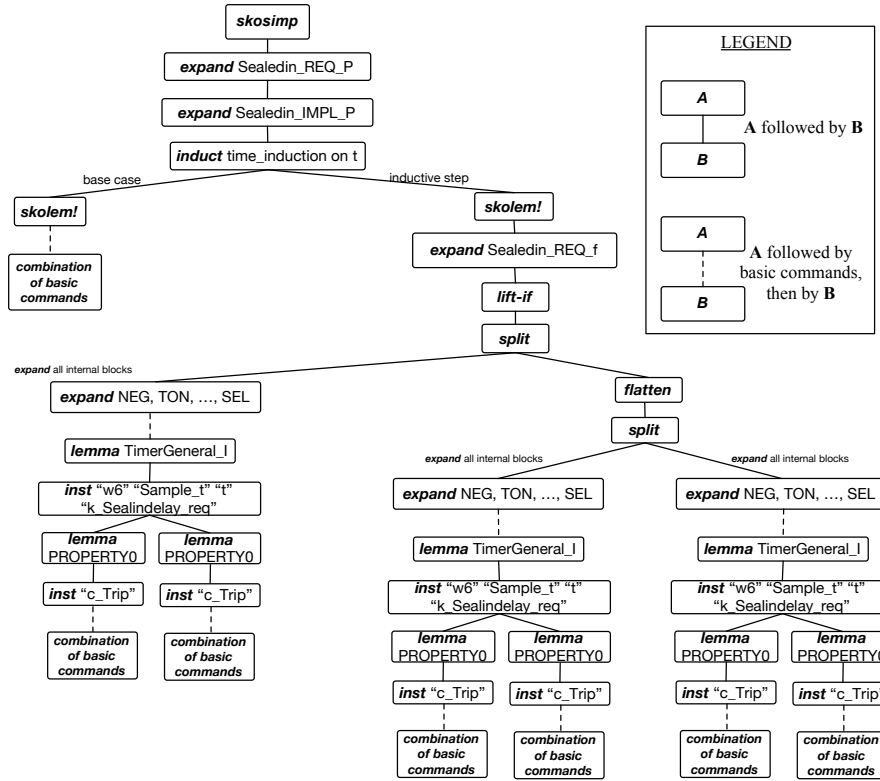


Fig. 21: Correctness Proof Structure for the *Trip Sealed-In* Subsystem

1. Applying **skolem!** to introduce Skolem constant on the universally quantified input variables in the consequent;
 2. Using **inst** to instantiate existentially quantified output variable in the consequent with a pre-defined function;
 3. Using **expand** to expand implementation predicate;
 4. Using **inst**'s to instantiate existentially quantified inter-connector variables in the consequent with pre-defined function for each;
 5. Applying **split** to select and split the conjunctive predicate for each internal component;
 6. Applying **expand**'s to expand the requirement predicate for each internal component, and then introducing Skolem constant to existentially quantified t in the consequent via **skolem!**; and
 7. Using a combination of basic commands to complete the proof.
- Correctness check:
1. Applying **skosimp**, which is a compound of **skolem!** and **flatten**, to introduce Skolem constant on the universally quantified input and output variables and then flatten the implication from implementation predicate to requirement predicate;
 2. Using **expand**'s to expand requirement predicate and implementation predicate;
 3. Using **skolem!** to give Skolem constant for each inter-connectors in the antecedent;
 4. Applying induction scheme **time_induction** on time tick t , generating two branches: base case and inductive step;
 5. Using **skolem!** and several basic commands to complete the base case;
 6. For inductive branch, using **skolem!** and **expand** to introduce Skolem constant for t and then expand the requirement function;
 7. Combining **lift-if** and **split** to explicitly generate each sub-goals;
 8. Repeatedly performing the following proof patterns: expanding definition predicate for each internal component by **expand**'s; using lemma **TimerGeneral_I** with proper instantiations to link the requirement and implementation domains with *Held_For*; using lemma **PROPERTY0** to replace lambda expression in the *Held_For* with a more explicit expression; and using a combination of basic commands to complete the proof.

6 Case Study 2: the *Pushbutton* Subsystem

In this section we apply our approach (Sec. 2.7) to verify the *Pushbutton* subsystem. We identify a missing assumption of implementation and suggest resolution.

6.1 Input-Output Declaration and Informal Description

Fig. 22 below declares the inputs and outputs of the *Pushbutton* subsystem. *Pushbutton* is a generic subsystem which monitors the status of a pushbutton (denoted by an input $m \in \{e_Pressed, e_NotPressed\}$), which may be pressed to manually, e.g., enable or disable a sensor trip⁹, and determines its behaviour (denoted by an output $f_Pushbutton \in \{e_pbNotDebounced, e_pbDebounced, e_pbStuck\}$).

⁹ A sensor trip occurs if the sensor signal in question goes above its set point.

6 Case Study 2: the *Pushbutton* Subsystem

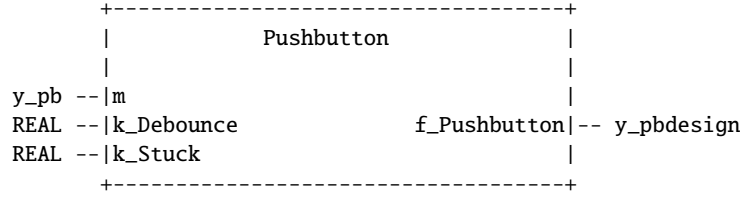


Fig. 22: Input-output Declaration for the *Pushbutton* subsystem

Pushbutton determines if either: (a) the button is not pressed, or pressed but not for a sufficient period of time (denoted by some pre-set value $k_Debounce$) to register as a press; (b) the button is pressed long enough to qualify as a press; or (c) the button is pressed for longer than some pre-set period of time (denoted by k_Stuck) without bouncing back and thus is considered stuck.

6.2 Tabular Requirements Specification with Timing Tolerances

For the purpose of verification in PVS, we use the function table (Fig. 23) to perform a complete and disjoint analysis on the domain of button status. To incorporate timing tolerances, similar to the requirements specification for the *Trip Sealed-In* subsystem (Fig. 17, p.17), we use the deterministic *Held_For_I* operator (Sec. 2.6), where values $k_Debounce - \delta L$ and $k_Stuck - \delta L$ are chosen and used consistently for detecting the sustained events.

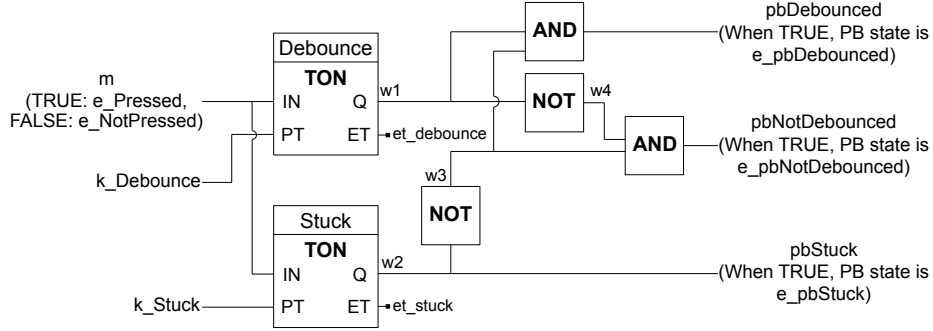
Condition	Result
$m = e_NotPressed$	$e_pbNotDebounced$
$(m = e_Pressed) \wedge \neg debounced$	$e_pbNotDebounced$
$debounced \wedge \neg stuck$	$e_pbDebounced$
$stuck$	$e_pbStuck$

where $debounced = (m = e_Pressed) \text{ Held_For_I } (k_Debounce - \delta L)$
 $stuck = (m = e_Pressed) \text{ Held_For_I } (k_Stuck - \delta L)$

Fig. 23: *Pushbutton*: (non-deterministic) Requirements of with Tolerances

6.3 Formalizing the FBD Implementation

We propose a FBD implementation which should satisfy the requirements:



We use two IEC 61131 *TON* timers (see Sec. 4 for its formalization) to implement the predicates *debounced* and *stuck* in the above requirements table that involve the use of the *Held_For_I* operator. Since only the button status is monitored, there is no need to specify a feedback loop in the implementation. To prove that this FBD implementation is consistent and correct, similar to what we do for that for the *Trip Sealed-In* subsystem (Fig. 18, p.18), formalize it by composing the formalizing predicates of all its component blocks using conjunction, and by hiding inter-connectors using an existential quantification:

$$\begin{aligned}
 & \text{Pushbutton_IMPL}(m, k_Debounce, k_Stuck, \\
 & \quad pbNotDebounced, pbDebounced, pbStuck) \\
 & \equiv \exists w_1, w_2, w_3, w_4, et_debounce, et_stuck \bullet \\
 & \quad \left(\begin{array}{l}
 TON(m, k_Debounce - \delta L, w_1, et_debounce) \\
 \wedge NEG(w_2, w_3) \\
 \wedge NEG(w_1, w_4) \\
 \wedge CONJ(w_1, w_3, pbDebounced) \\
 \wedge TON(m, k_Stuck - \delta L, w_2, et_stuck) \\
 \wedge CONJ(w_3, w_4, pbNotDebounced) \\
 \wedge (w_2 = pbStuck)
 \end{array} \right)
 \end{aligned}$$

6.4 Proof Obligations: Consistency and Correctness

The consistency and correctness theorems for the *Pushbutton* subsystem are stated in a similar manner as those for the *Trip Sealed-In* subsystem. The consistency (as formula 2 in Sec. 2.7) and correctness (as formula 3 in Sec. 2.7) theorems are formulated as follows:

$$\begin{aligned}
 & \vdash \forall m, k_Debounce, k_Stuck, \bullet \\
 & \quad \exists pbNotDebounced, pbDebounced, pbStuck \bullet \\
 & \quad \text{Pushbutton_IMPL}(m, k_Debounce, k_Stuck, \\
 & \quad \quad pbNotDebounced, pbDebounced, pbStuck) \\
 \\
 & \vdash \forall m, k_Debounce, k_Stuck, \bullet \\
 & \quad \forall pbNotDebounced, pbDebounced, pbStuck \bullet \\
 & \quad \text{Pushbutton_REQ}(m, k_Debounce, k_Stuck, \\
 & \quad \quad pbNotDebounced, pbDebounced, pbStuck) \Rightarrow \\
 & \quad \text{Pushbutton_IMPL}(m, k_Debounce, k_Stuck, \\
 & \quad \quad pbNotDebounced, pbDebounced, pbStuck)
 \end{aligned}$$

6.5 Proof Discussion

We had difficulties when first attempting to prove that the above requirements table possess the disjointness property. To resolve this, we tried to simplify the requirements table by collapsing the first two rows into a single one with the input condition $\neg pressed \wedge \neg stuck$ ¹⁰ (Fig. 24).

Condition	Result
	<i>f_Pushbutton</i>
$\neg debounced \wedge \neg stuck$	<i>e_pbNotDebounced</i>
<i>debounced</i> \wedge $\neg stuck$	<i>e_pbDebounced</i>
<i>stuck</i>	<i>e_pbStuck</i>
where <i>debounced</i> = (<i>m</i> = <i>e_Pressed</i>) Held_For_I (<i>k_Debounce</i> − δL) <i>stuck</i> = (<i>m</i> = <i>e_Pressed</i>) Held_For_I (<i>k_Stuck</i> − δL)	

Fig. 24: *Pushbutton*: Revised Requirements of with Tolerances

In attempting to prove that the revised requirements table is equivalent to the original one, we found a problematic scenario where the value of output *f_Pushbutton* is produced inconsistently at the requirements and implementation levels: when the input condition *m* varies rapidly and generates a “spike”, whose duration is shorter than the timing resolution. To rule out the “spike” scenarios for input *m*, we added an assumption, at the FBD implementation level, using the predicate subtype **FilteredTickPred** (Sec. 2).

Finally, the revised requirements table can be proved for its completeness, disjointness, consistency, and correctness by following a similar pattern of proofs as for the *Trip Sealed-In* subsystem. For proving the correctness theorem, as there is not a feedback loop in the above FBD implementation, we do not need to discharge the correctness theorem using mathematical induction. Furthermore, as the *TON* components in the FBD implementation are formalized using the *Timer_I* operator (Sec. 4), we need to reuse the theorem *TimerGeneral_I* with proper instantiations to show their equivalence to the *Held_For_I* expressions in the revised requirements table.

We now summarize three proof obligations in details:

1. Similar to the *Trip Sealed-In* subsystem, the consistency check can be discharged by instantiating the formulated functions for each inter-connector.
2. For the correctness check, it can also be proved by following the proof patterns as for the *Trip Sealed-In* subsystem.
3. We need to prove the functional equivalence between the original (Fig. 23) and revised requirement tables (Fig. 24), which is formulated as follows:

$$\vdash \forall m, k_Debounce, k_Stuck \bullet \\ f_Pushbutton_Original_REQ(m, k_Debounce, k_Stuck) = \\ f_Pushbutton_Revised_REQ(m, k_Debounce, k_Stuck)$$

During the course of verifying this equivalence check, we encounter two proof obligations as listed below:

¹⁰ This is done based on the observations that both row 1 and row 2 map to the same output value *e_pbNotDebounced*, and that $m = e_Pressed \vee m = e_NotPressed \equiv true$.

- (1). If the *Held_For* is *TRUE* for a time period of k_1 , *Held_For* is also *TRUE* for another time period of k_2 , where $k_2 \leq k_1$. The lemmas of **HFS_TimedDuration_PROPERTY** and **HFI_TimedDuration_PROPERTY** are proved for two versions of *Held_For* operator, i.e., **Held_For_S** and **Held_For_I**. These can be proved by instantiating the sample point when condition argument of the *Held_For* first becomes *TRUE*.
- (2). If the *Held_For* is *TRUE* for a time period of k , the condition argument of the *Held_For* is *TRUE* at the current time tick (as lemma **Held_For_IMPLIES_P_assump**). Lemmas **Held_For_IMPLIES_P_Debounce** and **Held_For_IMPLIES_P_Stuck** are formulated for the debounced and stuck case respectively. However, we obtain an unprovable proof obligation which reflects the fact that the behaviour of input can cause inconsistency between the SRS and the SDD. In order to implement *Pushbutton* correctly, we need the following implementation assumption:

Consider P is the condition input to the Pushbutton subsystem, it should not produce “spike” behaviour between any two consecutive sample points.

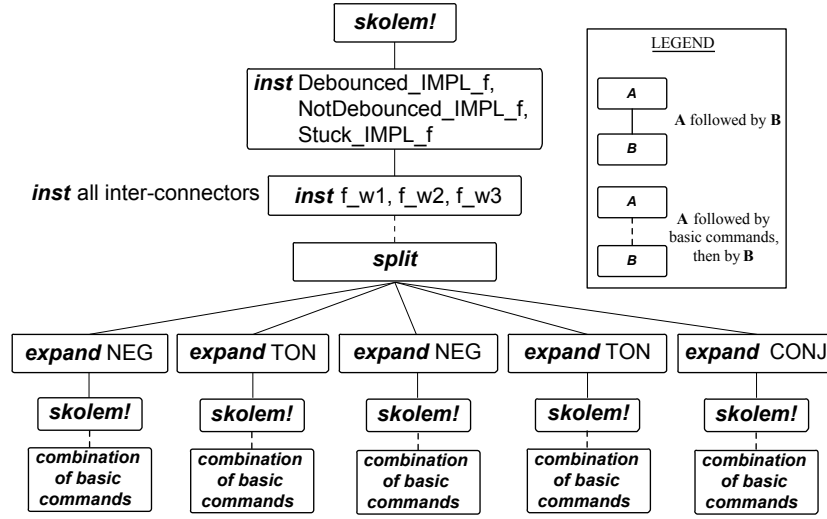
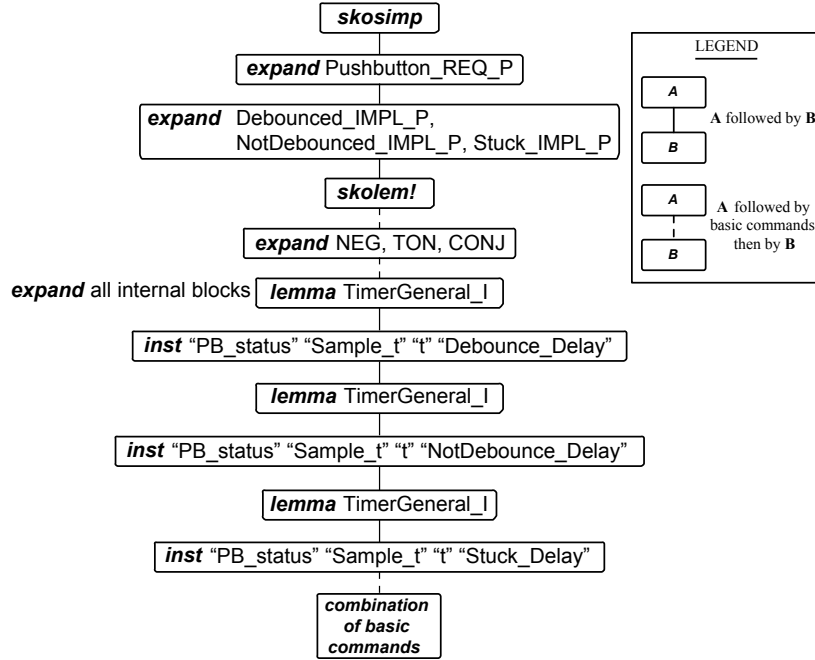
Between two consecutive samples, it is possible that the input signal P varies rapidly and generates “spikes”. The time duration of a “spike” is shorter than the timing resolution. Based on the relationship between physical and software domains for *Held_For*, it is possible that P is *FALSE* while **Held_For_I** is *TRUE*. Because **Held_For_I** keeps the value of last sample point and it won’t change until the next sample point. Condition P can change its value at any time tick in physical domain. We need to guarantee the truth of P at the next sample point to link these two in different domains. We assume that condition input P : 1) is of type **FilteredTickPred**; and 2) is *TRUE* at the next sample point. With these two assumptions, lemma **Held_For_IMPLIES_P_assump** is proved using two pre-verified lemmas (i.e., timing properties of **FILTER_TRUTH1** and **Sample_PROPERTY3**). It is used to complete the verification proofs of **Held_For_IMPLIES_P_Debounce** and **Held_For_IMPLIES_P_Stuck**.

Using the results from (1) and (2), we complete the proof of functional equivalence between the original and updated requirements tables (as proof obligation 3). Thus, all the above three proof obligations are complete.

6.6 Proof Structure

As shown in Figs. 25 and 26, we summarize the proof structures for the consistency and correctness checks of the *Pushbutton* subsystem. We only show the key commands and lemmas to illustrate our proof pattern. Most of the “a combination of basic commands” can be discharged by the **grind**, i.e., a catch-all strategy in PVS.

- Consistency check:
 1. Applying **skolem!** to introduce Skolem constant on the universally quantified input variables in the consequent;
 2. Using **inst**’s to instantiate existentially quantified output variables in the consequent with pre-defined function for each;
 3. Using **inst**’s to instantiate existentially quantified inter-connecter variables in the consequent with pre-defined function for each;


 Fig. 25: Consistency Proof Structure for the *Pushbutton* Subsystem

 Fig. 26: Correctness Proof Structure for the *Pushbutton* Subsystem

4. Applying **split** to select and split the conjunctive predicate for each internal component;
5. Applying **expand**'s to expand the requirement predicate for each internal component, and then introducing Skolem constant to existentially quantified t in the consequent via **skolem!**; and
6. Using a combination of basic commands to complete the proof.

- Correctness check:
 1. Applying **skosimp** to introduce Skolem constant on the universally quantified input and output variables and then flatten the implication from implementation predicate to requirement predicate;
 2. Using **expand**'s to expand requirement predicate, and each output predicate;
 3. Using **skolem!** to give Skolem constant for each inter-connectors in the antecedent;
 4. Expanding definition predicate for each internal component by **expand**'s;
 5. Repeatedly using lemma **TimerGeneral_I** with proper instantiations to link the requirement and implementation domains with *Held_For*; and
 6. Using a combination of basic commands to complete the proof.

7 Related Work

There are many works on formalizing and verifying PLC programs defined by programming languages covered in IEC 61131-3. [25] transforms FBDs to Uppaal models to verify safety applications in the industrial automation domain. [3] provides the formal operational semantics of ILs which is encoded into the symbolic model checker Cadence SMV, and verifies rich behavioural properties written in linear temporal logic (LTL). [2] checks the correctness of SFC programs, automatically generated from a graphical front-end. [2] verifies the correctness of Sequential Function Charts in Coq. [16] formalizes Instruction lists using timed automata, and verifies real-time properties in Uppaal. [26] formalizes PLC programs using higher-order logic and to discharge safety properties in HOL. [18] verifies a safety procedure in a nuclear power plant (NPP) in which a verified Coloured Petri Net (CPN) model is derived by reinterpretation from FBDs. These work are similar to our methodology on function block verification that the PLC programs are formalized and verified with a mechanized tool support.

For real time system verification, many achievements have been reported in [27], in which the majority of the cited work are dedicated to the specification and validation of real-time requirements. Relatively little work address the issues of the timing tolerances verifying implementation against its requirement. [4] models timing tolerances as timed automata with ASAP (as soon as possible) semantics which can be used to verify implementation of its requirements. [32] presents a PSPACE-complete decision procedure to test implementability. [33] provides a semi-decision procedure to compute the maximal reaction delay with the implementation which preserves the correctness of the design. Some concepts in [4] are equivalent to our used work of [8], e.g., implementable condition. A global constraint as the constant upper bound of the delay during the stage of implementation is introduced in [6,9]. Although it is clear to have single global tolerance, it does not make much sense at the requirements level that requires different timing requirements. [8] verified practical timing operator *Held_For* that we used in this paper. All of the formalizations and proofs of [8] are completed in PVS which is easy for us to import.

In this paper, our approach for verifying timing function blocks are practical in that: 1) we further re-formalize the IEC 61131-3 timers using pre-verified timing operator [8] with the consideration of timing tolerances; 2) we combine our methodology on function block verification with the re-formalized timing operator that allows us to be able to verify timing properties; and 3) we suggest proof strategies to the verification work for real time application.

8 Lessons Learned

As we summarized in Secs. 5.6 and 6.6, we present the proof patterns with key steps, lemma and theorems to discharge consistency and correctness checks. We also list the lessons learned during the verification work as follows:

1. Instantiations: It is very important to instantiate the correct witness to the formula. It is necessary to separately define the function or predicate being instantiated if it is complex.
2. Hidden Formulae: To obtain a better view of sequent, we always hide irrelevant formulae during the proofs. However, we also reveal the hidden formulae to discharge sub-type TCCs.
3. Lemmas: Key lemmas that have been proved can significantly save our effort (i. e., `TimerGeneral_I`).

9 Conclusion and Future Work

In this paper we report our application of a formal approach on using FBs from IEC 61131-3 to verify a subsystem of an industrial software control system from the nuclear domain. We re-formalize all three IEC 61131-3 timers to incorporate the notion of tolerances. We use a re-formalized IEC 61131 timer for its proposed FBD implementation, and prove that it is feasible and satisfies the intended timing requirements. We find an issue of initialization failure, and suggest resolution. Another issue of missing implementation assumption is reported in an extended report. We identify patterns of proof commands that are amenable to strategies that will facilitate the automated verification of the feasibility and correctness of other subsystems. As ongoing and future work, we aim to verify subsystems with more sophisticated timing requirements, e.g., nested *Held_For* expressions. We also aim to prove safety properties from the composition of real-time subsystems.

References

1. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Special Committee 205 of RTCA (2011)
2. Blech, J.O., Biha, S.O.: On formal reasoning on the semantics of PLC using Coq. CoRR abs/1301.3047 (2013)
3. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnoebelen, P.: Towards the automatic verification of PLC programs written in instruction list. In: IEEE International Conference on Systems, Man and Cybernetics. pp. 2449–2454 (2000)
4. De Wulf, M., Doyen, L., Raskin, J.F.: Systematic implementation of real-time models. In: FM 2005: Formal Methods: International Symposium of Formal Methods Europe Proceedings. LNCS, vol. 3582, pp. 139 – 156. Springer-Verlag, Newcastle, UK (Jul 2005)
5. Eles, C., Lawford, M.: A tabular expression toolbox for Matlab/Simulink. In: NASA Formal Methods. pp. 494–499 (2011)
6. Florescu, O., Voeten, J., Huang, J., corporaal, H.: Error estimation in model-driven development for real-time software. In: Forum on specification and Design Languages. pp. 228–239 (2004)
7. Hu, X.: Proving implementability of timing properties with tolerance. Ph.D. thesis, McMaster University, Department of Computing and Software (August 2008)

References

8. Hu, X., Lawford, M., Wassyng, A.: Formal verification of the implementability of timing requirements. In: FMICS. LNCS, vol. 5596, pp. 119–134. Springer (2009)
9. Huang, J., Voeten, J., Florescu, O., van der Putten, P., Corporaal, H.: Predictability in real-time system development. In: Advances in Design and Specification Languages for SoCs. pp. 123–139. Kluwer Academic Publishers (2005)
10. IEC: 61131-3 Ed. 2.0 en:2003: Programmable Controllers — Part 3: Programming Languages. International Electrotechnical Commission (2003)
11. IEC: 61131-3 Ed. 3.0 en:2013: Programmable Controllers — Part 3: Programming Languages. International Electrotechnical Commission (2013)
12. Janicki, R., Wassyng, A.: Tabular expressions and their relational semantics. *Fundam. Inf.* 67(4), 343–370 (2005)
13. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. *Science of Computer Programming* 75(11), 980 – 1000 (2010)
14. Lawford, M., McDougall, J., Froebel, P., Moum, G.: Practical application of functional and relational methods for the specification and verification of safety critical software. In: Proc. of AMAST 2000. LNCS, vol. 1816, pp. 73–88. Springer (2000)
15. Lawford, M., Wu, H.: Verification of real-time control software using pvs. In: Ramadge, P., Verdu, S. (eds.) *Proceedings of the 2000 Conference on Information Sciences and Systems*. vol. 2, pp. TP1–13–TP1–17. Dept. of Electrical Engineering, Princeton University, Princeton, NJ (Mar 2000)
16. Mader, A., Wupper, H.: Timed automaton models for simple programmable logic controllers. In: ECRTS. pp. 114–122. IEEE (1999)
17. NASA Langley PVS Libraries Official Website: <http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html> (2014)
18. Németh, E., Bartha, T.: Formal verification of safety functions by reinterpretation of functional block based specifications. In: FMICS, pp. 199–214. Springer (2009)
19. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: CADE. LNCS, vol. 607, pp. 748–752 (1992)
20. Pang, L., Wang, C.W., Lawford, M., Wassyng, A.: Formalizing and Verifying Function Blocks using Tabular Expressions and PVS. In: FTSCS. Communications in Computer and Information Science, vol. 419, pp. 163–178. Springer (2013)
21. Pang, L., Wang, C.W., Lawford, M., Wassyng, A.: Formal Verification of IEC 61131-3 Function Blocks using Tabular Expressions. *Science of Computer Programming* (2014), invited to submission for a special issue (ID: SCICO-D-14-00102).
22. Parnas, D.L., Madey, J.: Functional documents for computer systems. *Science of Computer Programming* 25(1), 41–61 (1995)
23. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering* 20, 948–976 (1994)
24. Shankar, N., Owre, S., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Prover Guide. Computer Science Laboratory, SRI International, Menlo Park, CA (Sep 1999)
25. Soliman, D., Thramboulidis, K., Frey, G.: Transformation of function block diagrams to Uppaal timed automata for the verification of safety applications. *Annual Reviews in Control* (2012)
26. Völker, N., Krämer, B.J.: Automated verification of function block-based industrial control systems. *Science of Computer Programming* 42(1), 101 – 113 (2002)
27. Wang, F.: Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE* 92(8), 1283 – 1307 (August 2004)
28. Wassyng, A., Lawford, M., Hu, X.: Timing tolerances in safety-critical software. In: FM 2005: Formal Methods: International Symposium of Formal Methods Europe Proceedings. LNCS, vol. 3582, pp. 157 – 172. Springer-Verlag, Newcastle, UK (Jul 2005)
29. Wassyng, A., Janicki, R.: Tabular expressions in software engineering. In: Proceedings of ICSSEA’03. vol. 4, pp. 1–46. Paris, France (2003)
30. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: FME 2003: International Symposium

- of Formal Methods Europe Proceedings. Lecture Notes in Computer Science, vol. 2805, pp. 133–153. Springer-Verlag (Aug 2003)
31. Wassyng, A., Lawford, M., Maibaum, T.S.E.: Software Certification Experience in The Canadian Nuclear Industry: Lessons for The Future. In: EMSOFT. pp. 219–226 (2011)
 32. Wulf, M.D., Doyen, L., Markey, N., Raskin, J.F.: Robustness and implementability of timed automata. In: Proc. of FORMATS04,. Lecture Notes in Computer Science, vol. 3253, pp. 152–166. Grenoble (2004)
 33. Wulf, M.D., Doyen, L., Raskin, J.F.: Almost asap semantics: From timed models to timed implementations. In: Proc. of HSCC04. Lecture Notes in Computer Science, vol. 2993, pp. 296 – 310 (2004)