Formal Verification of Real-Time Function Blocks Using PVS

Linna Pang, Chen-Wei Wang, Mark Lawford, Alan Wassyng

McMaster Centre for Software Certification, McMaster University, Canada L8S 4K1

{pangl,wangcw,lawford,wassyng}@mcmaster.ca

Josh Newell, Vera Chow, and David Tremaine

Systemware Innovation Corporation, Toronto, Canada M4P 1E4

{jnewell,vchow,tremaine}@swi.com

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. By applying our approach to the IEC 61131-3 standard of Programmable Logic Controllers (PLCs), we constructed a repository of precise specification and reusable (proven) theorems of feasibility and correctness for FBs. However, we previously did not apply our approach to verify FBs against timing requirements, since IEC 61131-3 does not define composite FBs built from timers. In this paper, based on our experience in the nuclear domain, we conduct two realistic case studies, consisting of the software requirements and the proposed FB implementations for two subsystems of an industrial control system. The implementations are built from IEC 61131-3 FBs, including the on-delay timer. We find issues during the verification process and suggest solutions.

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, the aviation standard DO-178C [1] advocates the use of formal methods to construct, develop, and reason about mathematical models of system behaviours. Applying the principles of DO-178C to PLC-based systems, we may obtain high-quality PLCs by: 1) pre-verifying standard FBs using formal methods; 2) building systems from pre-verified components; and 3) verifying their correctness in a compositional manner.

We recently reported a formal methodology [10] 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 (a.k.a. function tables) [12] for specification and the PVS proof assistant [9] for formal verification. Tabular expressions are a way to document system requirements as black-box, input-output relations that has proven to be practical and effective in industry [14]. 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 IEC 61131-3 [6, Annex F], an industrial standard for PLCs, resulting in a repository of: 1) precise specifications of input-output requirements; and 2) reusable theorems of feasibility and correctness for the FB library.

A critical step towards certifying 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. 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 [15]. For example, we write

$$(\,(signal \geq setpoint)\ Held\_For\,(300,\,-50,\,+50)\,) \Rightarrow (c\_var = trip) \tag{1}$$

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).

The requirement specified in Eq. 1 is *non-deterministic* since it allows any implementation that trips when *signal* $\geq$ *setpoint* sustains between [250*ms*, 350*ms*]. As we will see in our two case studies, such a simple requirement can be used as part of specifying more complex real-time behaviour. To resolve such non-determinism, at the requirements level we adopt a deterministic operator *Held_For_I* [4, 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 $((signal \geq setpoint)\ Held\_For\_I\,(300 - 50))$ into Eq. 1, 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 [4, p. 98] for counting the elapsed time of some monitored condition. The relationship between these two operators, at levels of requirements and implementation, is proved as a theorem *TimerGeneral_I* [4, p. 99]: $(C\ Held\_For\_I\,(timeout - \delta L))$ is equivalent to $(Timer\_I\,(C) \geq timeout - \delta L)$. See also Sec. 2.

We previously did not apply our approach [10] to verify FBs against this type of more complex timing requirements because IEC 61131-3 only includes simple timer blocks (i.e, on-delay, off-delay, and pulse timers), but not any more complex FBs built from those timers. Furthermore, our requirements model for IEC 61131-3 timers [10] describes idealized behaviour: as the monitored condition becomes enabled, the timer instantaneously responds (i.e., starts counting the duration of enablement), not considering sampling, computational delays, and timing tolerances.

Based on our experience on the Darlington Nuclear Shutdown Systems Trip Computer Software Re-design Project [14], and motivated by anticipated FB based 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. Implementations are built from IEC 61131-3 FBs, including the on-delay timer to implement more complex real-time behaviour.

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) from [4]. 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). This work builds on our previous results of verifying IEC 61131-3 FBs [10] that provide a sound semantic foundation for formalizing and verifying PLC programs expressed using FBDs.

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. 3). Second, for the representative subsystems we study (one with a feedback loop presented in Sec. 4 and the other in Sec. 5), 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. Third, we find issues of initialization failure (Sec. 4) and missing implementation assumptions (Sec. 5), and suggest possible solutions. Fourth, we identify pat-
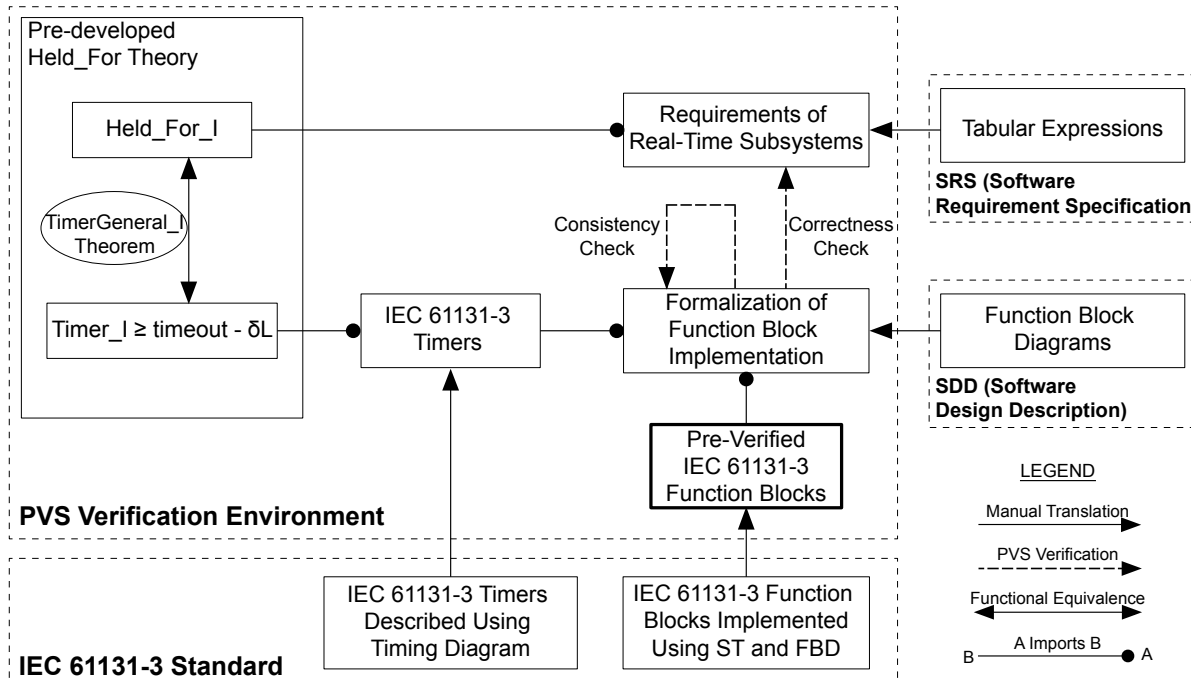
Figure 1: Framework for verification of FB based systems timing requirements

terns of proof commands (Sec. 6) that are amenable to strategies (or proof scripts) that 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/ESSS2015`. Background theories (e.g., *Held_For_I*, *Timer_I*, etc.) and complete details of case studies covered in this paper are included in an extended report [11].

2   Preliminaries

We review the use of tabular expressions, the relevant PVS theories of timing operators at levels of requirements and implementation, and the formal verification approach [10] that is adapted in our two case studies (Sec. 4 and Sec. 5).

2.1   Tabular Expressions Tabular expressions (a.k.a. function tables) [12] are an effective approach for 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., [7]). For our purpose of capturing the input-output requirements of timing function blocks, the tabular structure in Fig. 2 below suffices: rows in the first column denote input conditions, and rows in the second column denote the corresponding output results. The input column may be sub-divided to specify sub-conditions. When the output column denotes a state variable, we may write *NC* to abbreviate the case of "no change" on its value.

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

	Condition		Result *f*
C_1	$C_{1.1}$		res_1
	$C_{1.2}$		res_2
	...		...
	$C_{1.m}$		res_m
...			...
C_n			res_n

```
IF C₁
   IF      C₁.₁ THEN f = res₁
   ELSEIF  C₁.₂ THEN f = res₂
   ...
   ELSEIF  C₁.ₘ THEN f = resₘ
ELSEIF  ...
ELSEIF     Cₙ   THEN f = resₙ
```

Figure 2: Semantics of Horizontal Condition Table (HCT)

at least one row is applicable to every input, i. e., $(C_1 \lor C_2 \lor \cdots \lor C_n \equiv True)$. Disjointness ensures that rows do not overlap, e. g., $(i \neq j \Rightarrow \neg(C_i \land C_j))$. Similar constraints apply to the sub-conditions, if any.

Choice of Theorem Prover. We chose the PVS theorem prover to formalize the input-output requirements of function blocks primarily because it supports the syntax and semantics of tables. In particular, for each table that is syntactically valid, PVS automatically generates its associated healthiness conditions of completeness and disjointness as type correctness conditions (TCCs). Furthermore, we have expertise built from past experience in applying PVS to check requirements and designs in the nuclear domain [8] that gave us confidence in using the toolset. For modelling real-time behaviour, we reused parts of the PVS theories from [5, 4] (see Sec. 2.3 to 2.5).

For presentation, we show PVS listings using ASCII characters in frame boxes, whereas in the main text, we typeset names of predicates, types, theorems, *etc.*, in the math form.

2.2   Modelling Time in the Physical Domain   As PLCs are widely used in real-time systems, the modelling of time is critical. For our purpose of verification, we approximate the continuous time in the physical domain as a type *tick*, defined as a discrete series of equally-distributed clock ticks, with an arbitrarily small positive time interval δ between two consecutive clock ticks: $tick = \{t_n : \mathbb{R}_{\geq 0} \mid \delta \in \mathbb{R}_{>0} \land (\exists n : \mathbb{N} \bullet t_n = n \times \delta)\}$. We also define *not_init*, a subtype of *tick* that excludes t_0. We define operators to manipulate values at the tick level: $init(t : tick) = (t = 0)$, $pre(t : not\_init) = (t - \delta)$, $next(t : tick) = (t + \delta)$, and $rank(t : tick) = \frac{t}{\delta}$. We often apply induction to prove properties that should hold over time[1]:

```
time_induction: THEOREM
  FORALL (P: pred[tick]):
     (FORALL (t: tick): init(t) => P(t))
   & (FORALL (t: not_init): P(pre(t)) => P(t)) => (FORALL (t: tick): P(t))
```

where $pred[tick]$ is a PVS shorthand for "predicates on tick" (i.e., functions mapping *tick* to Boolean).

2.3   Modelling Samples in the Software Domain   We use a variable $Sample : \mathbb{N} \to \mathbb{R}_{\geq 0}$ 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.

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.

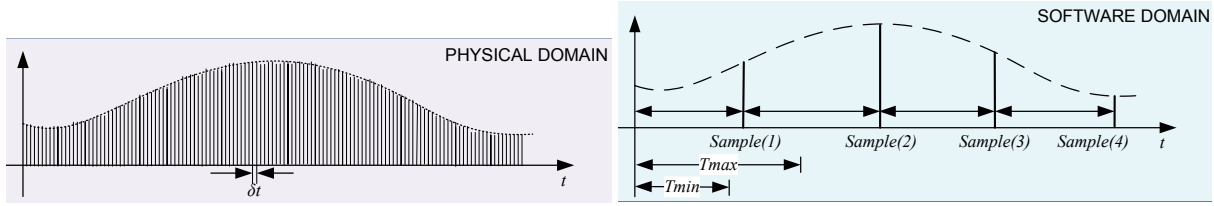[1] `pred[tick]` is synonymous to the function type `[tick -> bool]`

Figure 3: Clock Ticks in Physical Domain vs. Samples in Software Domain [4, p81]

To rule out such scenarios, we define a predicate subtype *FilteredTickPred*[2] 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):
            t0 < t AND t <= t0 + Tmax => P(next(t0)) = P(t)) )
  AND ( FORALL (t: tick):
            t <= Tmax => P(t) = P(0) )

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

2.4 Operators for Specifying Timing Requirements
As discussed in Sec. 1, we define the infix operator:

$$Held\_For : (tick \to \mathbb{B}) \times (tick \to \mathbb{R}_{>0}) \times (tick \to \mathbb{R}_{\geq 0}) \times (tick \to \mathbb{R}_{\geq 0}) \to (tick \to bool)$$

to specify a common functional timing requirement, e.g., P **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,

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

where $d \in [d(t_{now}) - \delta L(t_{now}), d(t_{now}) + \delta L(t_{now})]$. 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 held *TRUE* for a period that is bounded by $[d - \delta L, d + \delta R]$.

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 $d(t_{now}) - \delta L(t_{now})$. 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 : nat):
        Sample(ne) - Sample(n0) >= duration
    AND FORALL (n: nat): n0 <= n AND n <= ne => P(Sample(n))
```

---

[2]An example of using the subtype *FilteredTickPred* to constrain input signals can be found in the verification story of the *Pushbutton* subsystem (Sec. 5).

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

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

2.5   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*[4, 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. As mentioned in Sec. 1, the theorem *TimerGeneral_I* is proved [4, p. 99] to ensure that *Timer_I* is a proper implementation for *Held_For_I*.

2.6   A Formal Approach to Specifying and Verifying Function Blocks Our reported approach [10] 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 by input and output lists. Each input or output is represented as a timed sequence (or trajectory) mapping clock ticks to values (e.g., $[tick \rightarrow real]$). Without loss of generality we write *i* and *o* to denote, respectively, the lists of input and output trajectories.

   Consider a composite function block *FB* (e.g., see Fig. 9 in Sec. 4). The *requirements predicate* of *FB* (denoted as *FB_REQ*, e.g., *Trip_sealedin_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* of *FB* (denoted as *FB_IMPL*, e.g., *Trip_sealedin_IMPL*) is constructed by composing, using logical conjunction, the requirements predicates of its component FBs (e.g., *TON*, *CONJU*, *etc.*) as configured in its FBD implementation. All inter-connectives (e.g., *w1*, *w2*, *etc.*) in the FBD implementation are hidden using an existential quantification.

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:

$$\vdash \forall i \bullet \exists o \bullet FB\_IMPL(i, o) \tag{2}$$

Proof of Correctness To ensure that the implementation is correct with respect to the intended requirement, we prove that the observable inputs and outputs conform to those of the requirements:

$$\vdash \forall i \bullet \forall o \bullet FB\_IMPL(i, o) \Rightarrow FB\_REQ(i, o) \tag{3}$$

3   Formalizing IEC 61131-3 Timers with Tolerances

We present the first contribution of this paper: incorporating the notion of *timing tolerances* [15] (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 [10] 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. As case studies presented in this paper (Sec. 4 and Sec.5) only make use of the *TON* block, in this section we present its re-formalization only and report details of the other two timer blocks in [11].

```
                                    +--------+     +---+      +--------+
                               IN   |        |     |   |   |  |        |        |
                                  --+        +-----+   +---+--+        +---
                                    t0           t1    t2 t3        t4         t5
            +------+                      +----+                         +----+
            | TON  |                 Q    |    |    |                    |    |    |
            |      |                    ------+    +--------------------+    +-------
  BOOL --|IN    Q|-- BOOL                   t0+PT    t1                   t4+PT  t5
            |      |               PT          +---+                        +---+
  TIME --|PT  ET|-- TIME          :         /    |           +            /    |
            |      |               ET  :       /     |          /|          /     |
            +------+                :      /      |         /  |        /      |
                                    :     /       |        /   |       /       |
                                    0-+           +-----+    +-------+     +---
                                    t0           t1    t2    t3        t4         t5
```
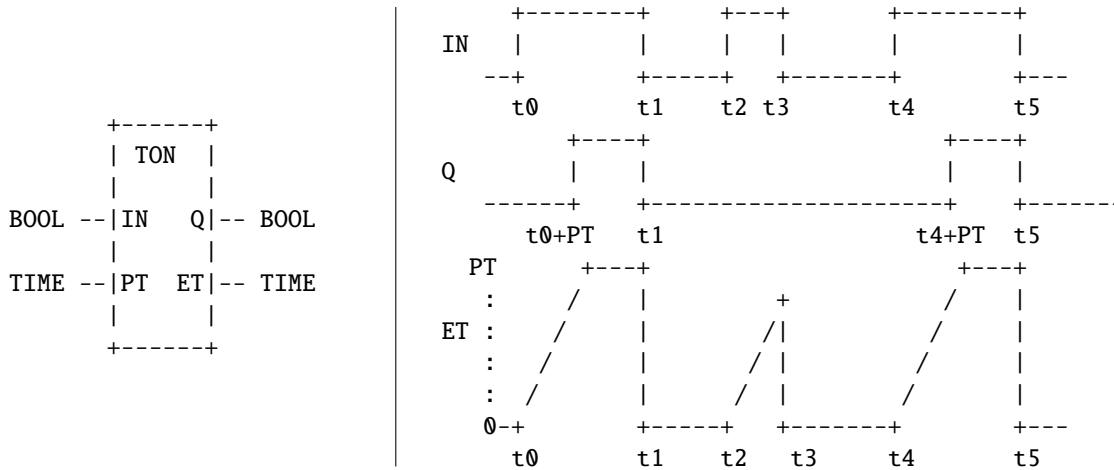
Figure 4: *TON* timer declaration and definition in timing diagram [6]

The *TON* block is commonly used as a component of safety-critical systems. For example, it can be used to determine if a sensor signal has gone out of its safety range for too long, as we will see in Sec. 4 and Sec. 5. Fig. 4 shows, extracted from IEC 61131-3, the input-output declaration (on the LHS) and a timing diagram[3] (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*.

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 in [10] 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. 5 shows our previous attempt of the requirements specification of the *TON* block, where t denotes the current clock tick, and 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. 5 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 [15]. To achieve this, we use the pre-verified operator *Timer_I* (Sec. 2) to redefine requirements of the three timers (e.g., Fig. 6 for the *TON* timer).

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 such as the one in Sec. 4 on implementing and verifying subsystems using the IEC 61131-3 timers.

---

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

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

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

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

where d stands for duration, d = t - last_enabled

Figure 5: Tabular Requirements of Timer *TON*: Idealized Behaviour

Condition	Result Q
$d \geq PT$	TRUE
$d < PT$	FALSE

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

where d stands for duration, d = (IN) Timer_I (PT, δL, δR)

Figure 6: Tabular Requirements of Timer *TON*: Timing Tolerances Incorporated

4   Case Study 1: the *Trip Sealed-In* Subsystem

In this section we apply our approach (Sec. 2.6) to verify a candidate FBD implementation for the *Trip Sealed-In* subsystem. We identify an initialization error and suggest a fix.

4.1   Input-Output Declaration and Informal Description   The figure below declares the inputs and outputs of the *Trip Sealed-In* subsystem:

```
                  +---------------------------------+
                  |            Trip Sealed-In        |
                  |                                  |
         BOOL --|Any_parm_trip                     |
{e_Trip, e_NotTrip} --|Trip              Trip_SealedIn|-- BOOL
         REAL --|k_Sealindelay                     |
         BOOL --|Man_reset_req                     |
                  +---------------------------------+
```

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_Sealindelay*[4] 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 all monitored sensor values fall back within their safety ranges, and then a manual reset is requested (denoted as an input *Man_reset_req*).

4.2   Tabular Requirements Specification with Timing Tolerances   We use a function table (Fig. 7) 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) to specify a sustained window of duration $[k\_Sealindelay - \delta L, k\_Sealindelay + \delta R]$.

---

[4]The $k\_$ name prefix is reserved for system-wide constants.

	Condition	Result
		Trip_SealedIn
Any_parm_trip	(*Trip*=*e_Trip*) **Held_For** (*k_Sealindelay*, δL, δR)	TRUE
	¬[(*Trip*=*e_Trip*) **Held_For** (*k_Sealindelay*, δL, δR)]	NC
¬*Any_parm_trip*	*Man_reset_req*	FALSE
	¬*Man_reset_req*	NC

Figure 7: *Trip Sealed-In*: (non-deterministic) Requirements of with Tolerances

However, for the purpose of verification in PVS, we reformulate the non-deterministic behaviour of Fig. 7 in a recursive function[5] 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.

Below we define a recursive function *Trip_SealedIn_f* over all clock ticks:

```
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)
```

Using *Trip_SealedIn_f*, we have deterministic requirements (Fig. 8) for the *Trip Sealed-In* subsystem:

Remark. Compared with Fig. 7, the use of the operator *Held_For_I* in Fig. 8 resolves the non-determinism by fixing the level of tolerance (i.e., 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).

4.3 Formalizing the FBD Implementation We propose a FBD implementation (Fig. 9) which should satisfy the requirements (Fig. 8).
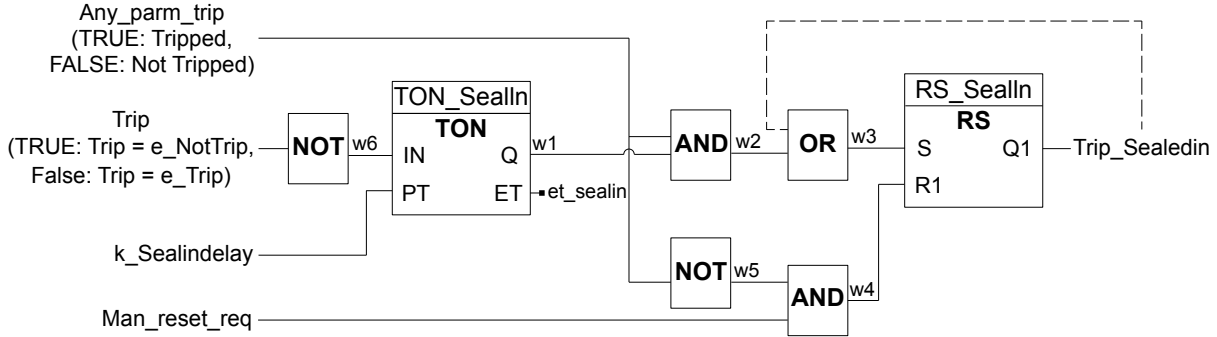
We use the IEC 61131 *TON* timer (see Sec. 3 for its formalization incorporated with tolerances) to implement the use of the *Held_For_I* operator (subject to a correctness proof which we will discuss below). As the recursive function used to define the requirements depends on the value of itself (at the previous time tick), we specify a feedback loop (dashed line) in the implementation.

---

[5] For proving termination, its progress is measured using discrete time instants *rank(t)*.

```
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)
```

Figure 8: *Trip Sealed-In*: (deterministic) Requirements of with Tolerances in PVS

Figure 9: *Trip Sealed-In* implementation in FBD

The use of the left-most *NOT* (negation) block in Fig. 9 has to do with the mismatch between types 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 a 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. 9) is both feasible and conforms to its requirements (Fig. 8), we follow our approach (Sec. 2.6) to formalize it by composing, using conjunction, the formalizing predicates[6] of all component blocks (all inter-connectors are hidden using an existential quantification.):

$$
\begin{aligned}
&Trip\_sealedin\_IMPL(Any\_parm\_trip, Trip, Man\_reset\_req, Trip\_SealedIn) \\
&\equiv \exists\, w_1, w_2, w_3, w_4, w_5, w_6, et\_sealin \bullet \\
&\qquad \left(
\begin{array}{l}
NOT(Trip, w_6) \\
\land\, TON(w_6, k\_Sealindelay - \delta L, w_1, et\_sealin) \\
\land\, CONJ(Any\_parm\_trip, w_1, w_2) \\
\land\, DISJ(w_2, Trip\_SealedIn, w_3) \\
\land\, NOT(Any\_parm\_trip, w_5) \\
\land\, CONJ(w_5, Man\_reset\_req, w_4) \\
\land\, RS(w_4, w_3, Trip\_SealedIn)
\end{array}
\right)
\end{aligned}
$$

---

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

4.4 Proofs of Consistency and Correctness First, we prove that the FBD implementation (Fig. 9) is feasible by instantiating formula (2) in Sec. 2.6:

$$\vdash \forall \textit{Any\_parm\_trip}, \textit{Trip}, \textit{Man\_reset\_req} \bullet$$
$$\exists \textit{Trip\_SealedIn} \bullet \textit{Trip\_sealedin\_IMPL}($$
$$\textit{Any\_parm\_trip}, \textit{AbstParmTrip\_timed(Trip)}, \textit{Man\_reset\_req}, \textit{Trip\_SealedIn})$$

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. 8, considering timing tolerances, by instantiating formula (3) in Sec. 2.6:

$$\vdash \forall \textit{Any\_parm\_trip}, \textit{Trip}, \textit{Man\_reset\_req}, \textit{Trip\_SealedIn} \bullet$$
$$\textit{Trip\_sealedin\_IMPL}(\textit{Any\_parm\_trip}, \textit{AbstParmTrip\_timed(Trip)}, \textit{Man\_reset\_req}, \textit{Trip\_SealedIn})$$
$$\Rightarrow \textit{Trip\_sealedin\_REQ}(\textit{Any\_parm\_trip}, \textit{Trip}, \textit{Man\_reset\_req}, \textit{Trip\_SealedIn})$$

As there is a feedback loop in the FBD implementation (Fig. 9), our strategy of discharging the correctness theorem is by mathematical induction (using the *time_induction* proposition in Sec. 2) over tick values. Since the *Timer_I* operator (Sec. 2) is used to formalize the requirements of the *TON* timer that contributes to the FBD implementation, its definition is expanded in both the base and inductive cases.

However, when proving the base case (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. 9) — are inconsistent. According to the SRS (Software Requirements Specification), the 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. 10) and prove that it is correct with respect to Fig. 8. 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*.

Remark. We just illustrated that, by adopting our approach, we are able to justify the appropriateness of a candidate FBD implementation, and to fix it accordingly if necessary.
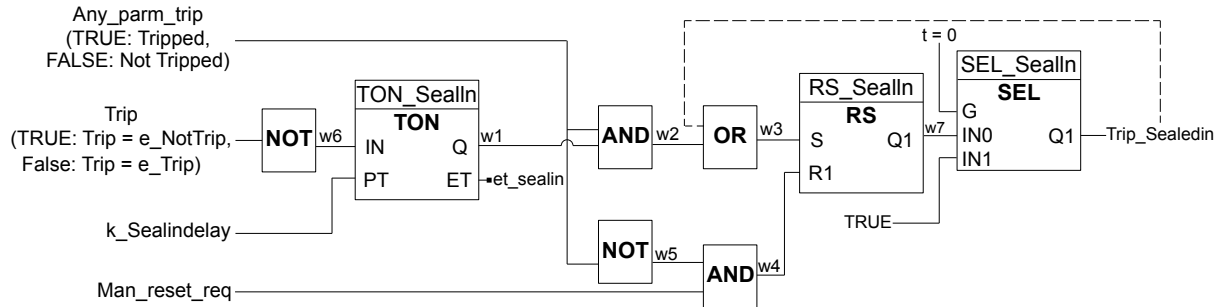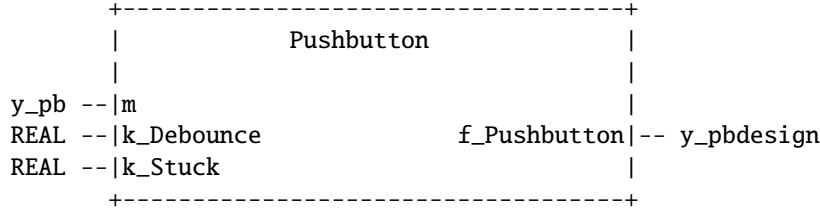

Figure 10: Revised *Trip Sealed-In* implementation in FBD

5   Case Study 2: the *Pushbutton* Subsystem

In this section we apply our approach (Sec. 2.6) to verify a candidate FBD implementation for the *Pushbutton* subsystem. We identify a missing assumption of implementation and suggest a solution.

5.1   Input-Output Declaration and Informal Description
The figure below declares the inputs and outputs of the *Pushbutton* subsystem.

```
           +----------------------------------+
           |              Pushbutton          |
           |                                  |
y_pb    --|m                                 |
REAL    --|k_Debounce          f_Pushbutton|-- y_pbdesign
REAL    --|k_Stuck                           |
           +----------------------------------+
```

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 trip[7]. Its behaviour is denoted by an output $f\_Pushbutton \in \{e\_pbNotDebounced, e\_pbDebounced, e\_pbStuck\}$. *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 $k\_Debounce$[8]) to register as a press; (b) the button is pressed long enough to quality 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.
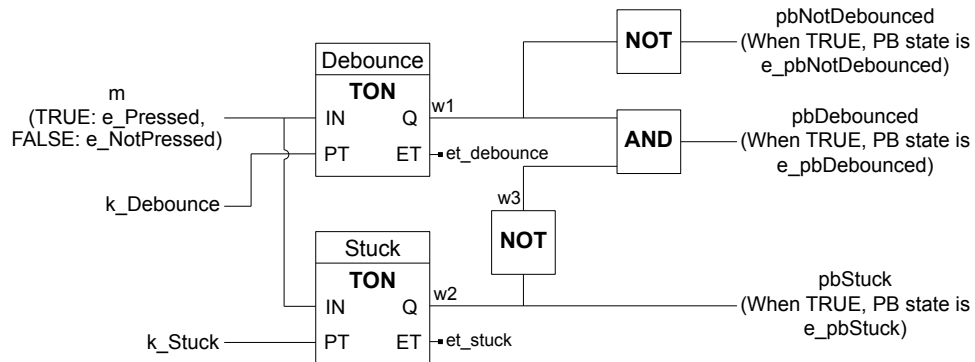
5.2   Tabular Requirements Specification with Timing Tolerances
For the purpose of verification in PVS, we use the function table below[9] to perform a complete and disjoint analysis on the domain of the button status. To incorporate timing tolerances, similar to the requirements specification for the *Trip Sealed-In* subsystem (Fig. 8, p.74), we use the deterministic *Held_For_I* operator (Sec. 2), where values $k\_Debounce - \delta L$ and $k\_Stuck - \delta L$ are chosen and used consistently for detecting the sustained events.

	Result
Condition	*f_Pushbutton*
$m = e\_NotPressed$	$e\_pbNotDebounced$
$(m = e\_Pressed) \wedge \neg debounced$	$e\_pbNotDebounced$
$debounced \wedge \neg stuck$	$e\_pbDebounced$
$stuck$	$e\_pbStuck$

$$\textbf{where} \quad debounced \quad = (m = e\_Pressed) \textbf{ Held\_For\_I } (k\_Debounce - \delta L)$$
$$stuck \qquad\quad = (m = e\_Pressed) \textbf{ Held\_For\_I } (k\_Stuck - \delta L)$$

5.3   Formalizing the FBD Implementation
We propose a FBD implementation which should satisfy the requirements:

---

[7]A sensor trip occurs if the sensor signal in question goes above its set point.

[8]The $k\_$ name prefix is reserved for system-wide constants.

[9]The PVS encoding of this table is not shown in this paper.

We use two IEC 61131 *TON* timers (see Sec. 3 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 (see Fig. 9, p.74), we formalize it by composing the formalizing predicates of all its component blocks using conjunction, and by hiding inter-connectors using an existential quantification.

5.4 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 by properly instantiating, respectively, formulas 2 and 3 in Sec. 2.6. However, we had difficulties when first attempting to prove that the above requirements table for *f_Pushbutton* possesses 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 $\neg pressed \wedge \neg stuck$. This is done based on the observations that both row 1 and row 2 map to the same output value *e_pbNotDebounced*, and that $m = e\_Pressed \vee m = e\_NotPressed \equiv true$.

When proving 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. 3), 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.

6 Proof Structure

In the industrial software control system that we consider for this paper, the *Trip Sealed-In* subsystem implemented using a feedback loop (Sec. 4) and the *Pushbutton* subsystem (Sec. 5) are representative[10] of functionality in which FBD implementations make use of IEC 61131 timer blocks. Structures of their consistency and correctness proofs shall guide the proofs for many other subsystems of a similar nature.

For illustration, we consider the correctness proof structure for the *Trip Sealed-In* subsystem. In principle, there are eight key steps to discharge the correctness theorem for a real-time subsystem implemented with a feedback loop (e.g., *Trip Sealed-In*). Except for the fourth step, where the *time_induction* theorem is used to handle the feedback loop, others are standard commands.

1) Apply `skosimp` to eliminate the universal quantification over input and output variables, and then apply `flatten` to simplify theorem structure *impl* $\Rightarrow$ *req* by moving *impl* to the antecedent and *req* to the consequent. 2) Apply multiple `expand` commands to unfold definitions of the requirements and implementation predicates. 3) In the antecedent, apply `skolem!` to eliminate the existential quantification over inter-connectors. 4) To handle the recursive feedback loop, use the theorem *time_induction* on $t \in tick$.

---

[10]This judgement is based on the use of a generic timing function, the *Held_For* operator, in the tabular expressions that describe the required behaviour.

5) Apply a series of basic commands to complete the proof for the base case. 6) To prove the inductive case, first apply `skolem!` and then `expand` to unfold the recursive function that is used to define the requirements predicate (e.g., see Fig. 8, p.74). 7) Apply `split` and `lift-if` to generate sub-goals. 8) Repeatedly apply: `expand` commands to unfold definitions of the predicates for internal components, theorem *TimerGeneral_I* with proper instantiations to link between *Held_For_I* in the requirements and *Timer_I* in the implementation, and basic commands to complete the proof for the inductive step.

7   Related Work

The focus of this paper is the practical verification of real-time behaviour against timing requirements with tolerances. Our approach to specifying and verifying FBs [10], compared with others on verifying PLC programs in contexts of model checking and theorem proving, is novel in three aspects: (1) extent of the case study; (2) practical application in the safety-critical industry; and (3) mature tool support of theorem proving.

In our formal setting, proving that an FBD implementation is correct (with respect to its intended input-output timing requirements) is essentially proving that it is a valid refinement. However, our purpose of verification is on the observable input-output behaviour, as opposed other properties such as boundedness, liveness, and robustness (e.g., [3, 16, 13, 2]). Of more relevance is the use of timed automata to model timing tolerances with ASAP (as soon as possible) semantics to verify the correctness of implementation [17], but with no suggestion for either tool support or its adoption in practice.

8   Conclusion

In this paper we report our application of a formal approach on using FBs (including timers) from IEC 61131-3 to verify two subsystems 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. Specifically, we use the re-formalized IEC 61131 on-delay timer for the proposed FBD implementations, and prove that they are feasible and correct (i.e., satisfies the intended timing requirements). While attempting to verify the two subsystems, we find an issue of initialization failure, and an issue of missing implementation assumption. In both cases, we suggest possible solutions. 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 first aim to verify subsystems with more sophisticated timing requirements, e.g., nested *Held_For* expressions. Second, we aim to prove safety properties from the composition of real-time subsystems. Third, we aim to automate the process of proofs that share a common structure.

References

[1] (2011): *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. Special Committee 205 of RTCA.

[2] Ed Brinksma, Angelika Mader & Ansgar Fehnker (2002): *Verification and optimization of a PLC control schedule*. International Journal on Software Tools for Technology Transfer (STTT) 4(1), pp. 21–33. Available at `http://dx.doi.org/10.1007/s10009-002-0079-0`.

[3] Zhijun Ding, Changjun Jiang & Mengchu Zhou (2013): *Design, Analysis and Verification of Real-Time Systems Based on Time Petri Net Refinement*. *ACM Trans. Embed. Comput. Syst.* 12(1), pp. 4:1–4:18. Available at `http://dx.doi.org/10.1145/2406336.2406340`.

[4] Xiayong Hu (2008): *Proving implementability of timing properties with tolerance*. Ph.D. thesis, McMaster University, Department of Computing and Software.

[5] Xiayong Hu, Mark Lawford & Alan Wassyng (2009): *Formal Verification of the Implementability of Timing Requirements*. In: *FMICS*, *LNCS* 5596, Springer, pp. 119–134. Available at `http://dx.doi.org/10.1007/978-3-642-03240-0_12`.

[6] IEC (2003): *61131-3 Ed. 2.0 en:2003: Programmable Controllers — Part 3: Programming Languages*. International Electrotechnical Commission.

[7] Ying Jin & David Lorge Parnas (2010): *Defining The Meaning of Tabular Mathematical Expressions*. *Science of Computer Programming* 75(11), pp. 980 – 1000. Available at `http://dx.doi.org/10.1016/j.scico.2009.12.009`.

[8] Mark Lawford, Jeff McDougall, Peter Froebel & Greg Moum (2000): *Practical application of functional and relational methods for the specification and verification of safety critical software*. In: *Proc. of AMAST 2000*, *LNCS* 1816, Springer, pp. 73–88. Available at `http://dx.doi.org/10.1007/3-540-45499-3_8`.

[9] Sam Owre, John M. Rushby & Natarajan Shankar (1992): *PVS: A Prototype Verification System*. In: *CADE*, *LNCS* 607, pp. 748–752. Available at `http://dx.doi.org/10.1007/3-540-55602-8_217`.

[10] Linna Pang, Chen-Wei Wang, Mark Lawford & Alan Wassyng (2013): *Formalizing and Verifying Function Blocks using Tabular Expressions and PVS*. In: *FTSCS*, *Communications in Computer and Information Science* 419, Spring, pp. 163–178. Available at `http://dx.doi.org/10.1007/978-3-319-05416-2_9`.

[11] Linna Pang, Chen-Wei Wang, Mark Lawford, Alan Wassyng, Josh Newell, Vera Chow & David Tremaine (2014): *Formal Verification of Real-Time Function Blocks using PVS*. Technical Report 16, McSCert. `https://www.mcscert.ca/index.php/documents/mcscert-reports?view=publication&task=show&id=16`.

[12] David Lorge Parnas, Jan Madey & Michal Iglewski (1994): *Precise Documentation of Well-Structured Programs*. *IEEE Transactions on Software Engineering* 20, pp. 948–976. Available at `http://dx.doi.org/10.1109/32.368133`.

[13] Ocan Sankur (2013): *Shrinktech: A Tool for the Robustness Analysis of Timed Automata*. In: *Computer Aided Verification*, *LNCS* 8044, Springer, pp. 1006–1012. Available at `http://dx.doi.org/10.1007/978-3-642-39799-8_72`.

[14] Alan Wassyng & Mark Lawford (2003): *Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project*. In: *FME 2003*, *LNCS* 2805, Springer, pp. 133–153. Available at `http://dx.doi.org/10.1007/978-3-540-45236-2_9`.

[15] Alan Wassyng, Mark Lawford & Xiaoyong Hu (2005): *Timing Tolerances in Safety-Critical Software*. In: *FM 2005*, *LNCS* 3582, Springer, pp. 157 – 172. Available at `http://dx.doi.org/10.1007/11526841_12`.

[16] Anton Wijs & Luc Engelen (2013): *Efficient Property Preservation Checking of Model Refinements*. In: *TACAS*, *LNCS* 7795, Springer, pp. 565–579. Available at `http://dx.doi.org/10.1007/978-3-642-36742-7_41`.

[17] Martin De Wulf, Laurent Doyen & Jean-Franois Raskin (2005): *Almost ASAP semantics: from timed models to timed implementations*. *FAC* 17(3), pp. 319–341. Available at `http://dx.doi.org/10.1007/978-3-540-24743-2_20`.