Formalizing and Verifying Function Blocks using Tabular Expressions and 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

Abstract. Many industrial control systems use programmable logic controllers (PLCs) since they provide a highly reliable, off-the-shelf hardware platform. On the programming side, function blocks (FBs) are reusable components provided by the PLC supplier that can be combined to implement the required system behaviour. A higher quality system may be realized if the FBs are pre-certified to be compliant with an international standard such as IEC 61131-3. We present an approach to formalizing FB requirements using tabular expressions, and to verifying the correctness of the FBs implementations in the PVS proof environment. We applied our approach to the example FBs of IEC 61131-3 and identified issues in the standard: ambiguous behavioural descriptions, missing assumptions, and erroneous implementations.

Keywords: critical systems, formal specification, formal verification, function blocks, tabular expressions, IEC 61131-3, PVS

1 Introduction

Many industrial control systems have replaced traditional analog equipment by components that are based upon programmable logic controllers (PLCs) to address increasing market demands for high quality [1]. Function blocks (FBs) are basic design units that implement the behaviour of a PLC, where each FB is a reusable component for building new, more sophisticated components or systems. The search for higher quality may be realized if the FBs are pre-certified with respect to an international standard such as IEC 61131-3 [8, 9]. Standards such as DO-178C (in the aviation domain) and IEEE 7-4.3.2 (in the nuclear domain) list acceptance criteria of mission- or safety-critical systems for practitioners to comply with. Two important criteria are that 1) the system requirements are precise and complete; and that 2) the system implementation exhibits behaviour that conforms to these requirements. In one of its supplements, DO-178C advocates the use of formal methods to construct, develop, and reason about the mathematical models of system behaviours.

Tabular expressions [20, 21] are a way to document system requirement that have proven to be both practical and effective in industry [13, 25]. PVS [18] is a non-commercial theorem prover, and 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. In this paper we report on using tabular expressions to formalize the requirements of FBs and on using PVS to verify their correctness (with respect to tabular requirements).

As a case study, we have formalized[1] 23 of 29 FBs listed in IEC 61131-3 [8, 9], an important standard with over 20 years of use on critical systems running on PLCs. There are two compelling reasons for formalizing the existing behavioural descriptions of FBs supplied by IEC 61131-3. First, formal descriptions such as tabular expressions force tool vendors and users of FBs to have the same interpretations of the expected system behaviours. Second, formal descriptions are amenable to mechanized support such as PVS to verify the conformance of candidate implementations to the high-level, input-output requirements. Currently IEC 61131-3 lacks an adequate, formal language for describing the behaviours of FBs and for arguing about their correctness. Unfortunately, IEC 61131-3 uses FB descriptions that are too close to the level of hardware implementations. For the purpose of this paper, we focus on FBs that are described in the more commonly used languages of structured text (ST) and function block diagrams (FBDs). Note that two versions of IEC 61131-3 are cited here. The earlier version [8] has been in use since 2003. Most of the work reported on in this paper relates to this version. When the new version [9] was issued, we expected to find that the problems we had discovered in the earlier version had been corrected. However, we found that many of the example FBs had been removed from the standard and the remaining FBs are still problematic.

Fig. 1: Framework

We now summarize our approach and contributions with reference to Fig. 1. As shown on the left, a function block will typically have a natural language description of the block behaviour accompanied by a detailed implementation in the ST or FBD description, or in some cases both. Based upon all of this information we create a black box tabular requirements specification in PVS for the behaviour of the FB as described in Sec. 3.2. The ST and FBD implementations are formalized as predicates in PVS, again making use of tables, as described in Sec. 3.1. In the case when there are two implementations for an FB, one in FBD and the other in ST, we attempt to prove their (functional) equivalence in PVS. For any implementation we attempt to prove the *correctness* and *consistency* with respect to the FB requirements in PVS (Sec. 4).

Using our approach, we have identified a number of issues in IEC 61131-3 and suggest resolutions (Sec. 5), which are summarized below:

[1] PVS files are available at `http://www.cas.mcmaster.ca/~lawford/papers/FTSCS2013`. All verifications are conducted using PVS 5.0.

1. The behaviour of the *pulse* timer is characterized through a timing diagram with at least two scenarios unspecified.
2. The description of the *sr* block (a set-dominant latch) lacks an explicit time delay on the intermediate values being computed and fed back. By introducing a delay FB, we verified the correctness of *sr*.
3. The description of the up-down counter *ctud* permits unintuitive behaviours. We eliminate them by incorporating a relation on its three inputs (i.e., low limit, high limit, and preset value) in the tabular requirement of *ctud*.
4. The description of the *limits_alarm* block allows the low limit and high limit alarms to be tripped simultaneously. We resolve this by explicitly constraining the two hysteresis zones to be both disjoint and ordered.
5. The ST and FBD implementations for the *stack_int* block (stack of integers) failed the equivalence proof. We identified a missing FB in the FBD implementation, and then discharged the proof.

We will discuss issues (1), (2), and (3) in further detail in Sec. 5. Details of the remaining issues that we omit are available in an extended report [19]. In the next section we discuss background materials: the IEC 61131-3 Standard, tabular expressions, and PVS.

2 Preliminaries

2.1 IEC 61131-3 Standard Function Blocks Programmable logic controllers (PLCs) are digital computers that are widely utilized in real-time and embedded control systems. In the light of unifying 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 [8] and 2013 [9]. The issues of ambiguous behaviours, missing assumptions, and erroneous behavioural descriptions that we found have not been resolved in the latest edition.

We applied our methodology to the standard functions and function blocks listed in Annex F of IEC 61131-3 (1993). FBs are more flexible than standard functions in that they allow internal states, feedback paths and time-dependent behaviours. We distinguish between *basic* and *composite* FBs: the former consist of standard functions only, while the latter can be constructed from standard functions and any other pre-developed basic or composite FBs. We focus on two programming languages that are covered in IEC 61131-3 for writing behavioural descriptions of FBs: structured text (ST) and function block diagrams (FBDs). ST syntax is block structured and resembles that of Pascal, while FBDs visualize inter-connections or data flows between inputs and outputs of block components.

Fig. 2 shows the FBD of the *limits_alarm* block, consisting of declarations of inputs and outputs, and the definition of computation. An alarm monitors the quantity of some variable x, subject to a low limit l and a high limit h, with the hysteresis band of size *eps*. The body definition visualizes how ultimate and intermediate outputs are computed, e.g., output *ql* is obtained by computing *HYSTERESIS(l + (eps / 2.0), x, eps)*. There are five internal component blocks

of *limits_alarm*: addition (+), subtraction(−), division (/), logical disjunction (≥ 1), and the hysteresis effect (*hysteresis*). The internal connectives are w_1, w_2 and w_3. The precise input-output tabular requirement is discussed in Sec. 3.2.

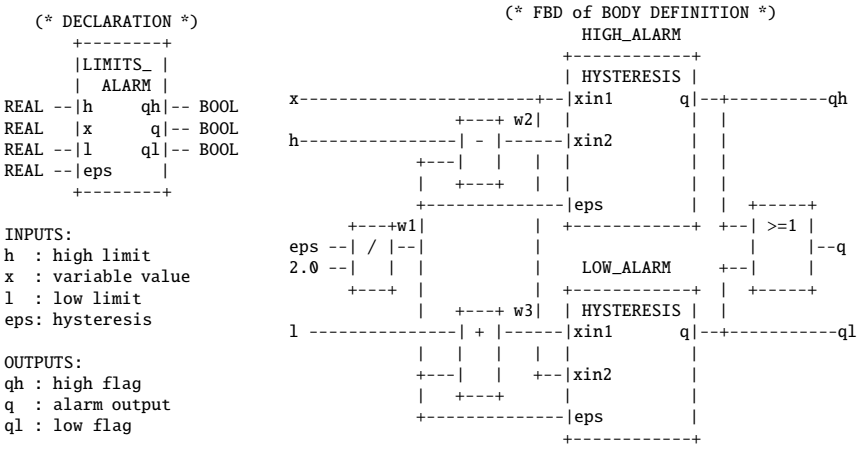

Fig. 2: *Limits_alarm* standard declaration and FBD implementation [8]

2.2 Tabular Expressions Tabular expressions [20, 21] (a.k.a. function tables) 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., [10]), and they are useful both in inspections and in testing and verification [25]. For our purpose of capturing the input-output requirements of function blocks in IEC 61131-3, the tabular structure in Fig. 3 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.

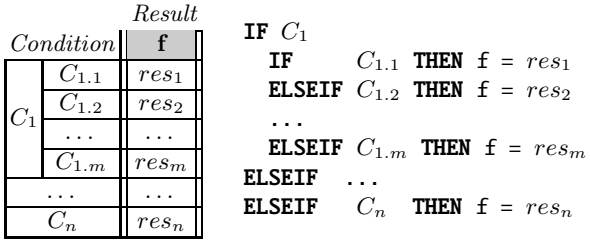

Fig. 3: Semantics of Horizontal Condition Table (HCT)

We may interpret the above tabular structure as a list of "if-then-else" predicates or logical implications. Each row defines the input circumstances under which the output f is bound to a particular result value. For example, the first row corresponds to the predicate $(C_1 \wedge C_{1.1} \Rightarrow f = res_1)$, and so on. 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., $(C_1 \vee C_2 \vee \cdots \vee C_n \equiv True)$. 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 [6].

2.3 PVS Prototype Verification System (PVS) [18] is an interactive environment for writing specifications and performing proofs. The PVS specification language is based on classical higher-order logic. The syntactic constructs that we use the most are "if-then-else" predicates and tables, which we will explain as we use them. An example of using tabular expressions to specify and verify the Darlington Nuclear Shutdown System (SDS) in PVS can be found in [13].

PVS has a powerful interactive proof checker to perform sequent-style deductions. The completeness and disjointness properties are generated automatically as Type Correctness Conditions (TCCs) to be discharged. We will discuss a found issue (Sec. 5) where the ST implementation supplied by IEC 61131-3 is formalized as a PVS table but its disjointness TCC failed to be discharged. In this paper we omit proof details that are available in an extended report [19].

As PLCs are commonly used in real-time systems, time modelling is a critical aspect in our formalization. We consider a discrete-time model in which a time series consists of equally spaced sample times or "ticks"in PVS:

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

Constant `delta_t` is a positive real number. Here `time` is the set of non-negative real numbers, and `tick` is the set of time sample times [7].

3 Formalizing Function Blocks using Tabular Expressions

Below we present a formal approach to defining standard functions and function blocks in IEC 61131-3 using tabular expressions.

3.1 Formalizing IEC 61131-3 Function Block Implementations We perform formalization at levels of standard functions, basic function blocks (FBs), and composite FBs. Similar to [4], we formulate each standard function or function block as a predicate, characterizing its input-output relation.

Standard Functions. IEC 61131-3 defines eight groups of standard functions, including: 1) data type conversion; 2) numerical; 3) arithmetic; 4) bit-string; 5) selection and comparison; 6) character string; 7) time and date types; and 8) enumerated data types. In general, we formalize the behaviour of a standard function f as a Boolean function:

$$f(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) : bool \equiv R(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n)$$

where predicate R characterizes the precise relation on the m inputs and the n outputs of standard function f. Our formalization covers both timed and untimed behaviours of standard functions. As an example of a timed function, consider function *move* that takes as inputs an enabling condition *en* and an integer *in*, and that outputs an integer *out*. The behaviour of *move* is time-dependent: at the very first clock tick, *out* is initialized to zero; otherwise, at time instant t ($t > 0$), *out* is either equal to *in* at time t, if condition *en* holds at t, or otherwise *out* is equal to *in* at time $t - \alpha * \delta$ ($\alpha = 1, 2, \ldots$) where *en* was last enabled (i.e., a case of "no change" for *out*). More precisely, we translate the input-output relation of function *move* into PVS:

```
move(en:[tick->bool],i,out:[tick->int])(t:tick): bool =
  FORALL t: out(t) = IF t = 0 THEN 0
                     ELSE TABLE +-------------------------+
                                | en(t)    |    i(t)     ||
                                +-------------------------+
                                | NOT en(t)| out(pre(t)) ||
                                +-------------------------+ ENDTABLE
                     ENDIF
```

We characterize the temporal relation between *in* and *out* as a universal quantification over discrete time instants. Functions `[tick->bool]` and `[tick->int]` capture the input and output values at different time instants. The behaviour at each time instant t is expressed as an `IF...THEN...ELSE...ENDIF` statement. Construct `TABLE...ENDTABLE` that appears in the `ELSE` branch exemplifies the use of tabular expressions as part of a predicate. The main advantage of embedding tables in predicates is that the PVS prover will generate proof obligations for completeness and disjointness accordingly.

Untimed behaviour, on the other hand, abstracts from the input-output relation at the current time instant, which makes first-order logic suffice for the formalization. For example, consider the standard function *add* that is used as an internal component of the FB *limits_alarm* (see Fig. 2), which has the obvious formalization: $add(in_1, in_2, out : int) : bool \equiv out = in_1 + in_2$. Incorporating the output value *out* as part of the function parameters makes it possible to formalize basic FBs with internal states, or composite FBs. For basic FBs with no internal states, we formalize them as function compositions of their internal blocks. As a result, we also support a version of *add* that returns an integer value: $add(in_1, in_2 : int) : int = in_1 + in_2$.

Basic Function Blocks. A basic function block (FB) is an abstraction component that consists of standard functions. When all internal components of a basic FB are functions, and there are no intermediate values to be stored, we formalize the output as the result of a functional composition of the internal functions. For example, given FB *weigh*, which takes as inputs a gross weight gw and a tare weight tw and returns the net weight nw, we formalize weigh by defining the output nw as $nw = int2bcd(subtract(bcd2int(gross), tare))$, where *int2bcd* and *bcd2int* are standard conversion functions between binary-coded decimals and integers. On the other hand, to formalize a basic FB that has internal states to be stored, we take the conjunction of the predicates that formalize its internal functions. We formalize composite FBs in a similar manner.

Composite Function Block. Each composite FB contains as components standard functions, basic FBs, or other pre-developed composite FBs. For example, *limits_alarm* (Sec. 2) is a composite FB consisting of standard functions and two instances of the pre-developed composite FB *hysteresis*. Our formalization of each component as a predicate results in *compositionality*: a predicate that formalizes a composite FB is obtained by taking the conjunction of those that formalize its components. IEC 61131-3 uses structured texts (ST) and function block diagrams (FBD) to describe composite FBs.

Remark. Predicates that formalize basic or composite FBs represent their black-box input-output relations. Since we use function tables in PVS to specify these predicates, their behaviours are deterministic. This allows us to easily compose their behaviours using logical conjunction. The conjunction of deterministic components is functionally deterministic.

Formalizing Composite FB Implementation: ST. We translate an ST implementation supplied by IEC 61131-3 into its equivalent expression in PVS. We illustrate (parts of[2]) our ST-to-PVS translation using concrete examples.

Pattern 1 illustrates that we transform sequential compositions (`;`) into logical conjunctions (`&`). We write a_{-1} to denote the value of variable a at the previous time tick (i.e., before the current function block is executed). In general, we constrain the relationship between each variable v and v_{-1} to formalize the effect of its containing function block.

#	ST expressions	PVS predicates
1	*basic assignments*	
	`a := a + b; c := NOT (a > 0)`	`a = a_-1 + b & c = NOT (a > 0)`

Pattern 2 illustrates that we reconstruct conditional statement by taking the conjunction of the assignment effect of each variable; each variable assignment is formalized via a tabular expression. How variables are used in the tables is used to derive the order of evaluation. For example, b is evaluated before c to compute $c = a + b$.

[2] Other translation patterns can be found in [19].

2	*conditional assignments*	

	IF z **THEN**
	  b := c * 3; c := a + b;
	ELSE
	  b := b + c; e := b - 1;
	END_IF
		b = **TABLE** | z | c_{-1} * 3 ||
		| **NOT** z | b_{-1} + c ||
		ENDTABLE &
		c = **TABLE** | z | a + b ||
		| **NOT** z | c_{-1} ||
		ENDTABLE &
		e = **TABLE** | **NOT** z | b - 1 ||
		| z | e_{-1} ||
		ENDTABLE

For the above example, an "if-then-else" conditional that returns the conjunction of the variable update predicates more closely correspond to the original ST implementation may instead be used. In general though when assignment conditions become more complicated, we feel it is clearer to isolate the update of each variable.

Pattern 3 illustrates that we translate each invocation of a function block `FB` into an instantiation of its formalizing predicate `FB_REQ`, where the return value of `FB` (i.e., `FB.output`) is incorporated as an argument of `FB_REQ`.

3	*function block invocations, reuse*	
	FB1(in_1 := a, in_2 := b); FB2(in_1 := FB1.**output**); out := FB2.**output**;	FB1_REQ(a, b, fb1_out) & FB2_REQ(fb1_out, fb2_out) & out = fb2_out

Formalizing Composite FB Implementation: FBD. To illustrate the case of formalizing a FBD implementation supplied by IEC 61131-3, let us consider the following FBD of a composite FB and its formalizing predicate in Fig. 4:

$$
\begin{aligned}
&FBD\_IMPL(i_1, i_2, o_1, o_2)\\
&\quad \equiv \exists w_1, w_2, w_3 \bullet\\
&\qquad \begin{pmatrix}
\quad B_1\_REQ(i_2, w_1)\\
\wedge\ B_2\_REQ(w_1, w_3, w_2)\\
\wedge\ B_3\_REQ(i_1, w_2, o_1)\\
\wedge\ B_4\_REQ(o_1, w_3, o_2)
\end{pmatrix}
\end{aligned}
$$

Fig. 4: Composite FB implementation in FBD and its formalizing predicate

Fig. 4 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, ..., B_4_REQ exist). The high-level 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 (see Sec. 3.2). To describe the overall behaviour of the above composite FB, we take advantage of our formalization being *compositional*. In other words, we formalize a composite FB 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.

For example, we formalize the FBD implementation of block *limits_alarm* (Sec. 2) as a predicate `LIMITS_ALARM_IMPL` in PVS:

```
LIMITS_ALARM_IMPL(h,x,l,eps,qh,q,ql)(t): bool =
 FORALL t:
  EXISTS (w1,w2,w3):
    div(eps(t),2.0,w1(t)) & sub(h(t),w1(t),w2(t)) &
    add(l(t),w1(t),w3(t)) & disj(qh(t),ql(t),q(t)) &
    HYSTERESIS_req_tab(x,w2,w1,qh)(t) & HYSTERESIS_req_tab(w3,x,w1,ql)(t)
```

We observe that predicate `LIMITS_ALARM_IMPL`, as well as those for the internal components, all take a time instant $t \in tick$ as a parameter. This is to account for the time-dependent behaviour, similar to how we formalized the standard function *move* in the beginning of this section.

The above predicates that formalize the internal components, e.g., predicate `HYSTERESIS_req_tab`, do not denote those translated from the ST implementation of IEC 61131-3. Instead, as one of our contributions, we provide high-level, input-output requirements that are missing from IEC 61131-3 (to be discussed in the next section). Such formal, compositional requirement are developed for the purpose of formalizing and verifying sophisticated, composite FBs.

3.2 Formalizing Requirements of Function Blocks As stated, IEC 61131-3 supplies low-level, implementation-oriented ST or FBD descriptions for function blocks. For the purpose of verifying the correctness of the supplied implementation, it is necessary to obtain requirements for FBs that are both complete (on the input domain) and disjoint (on producing the output). Tabular expressions (in PVS) are an excellent notation for describing such requirements. Our method for deriving the tabular, input-output requirement for each FB is to partition its input domain into equivalence classes, and for each such input condition, we consider what the corresponding output from the FB should be.

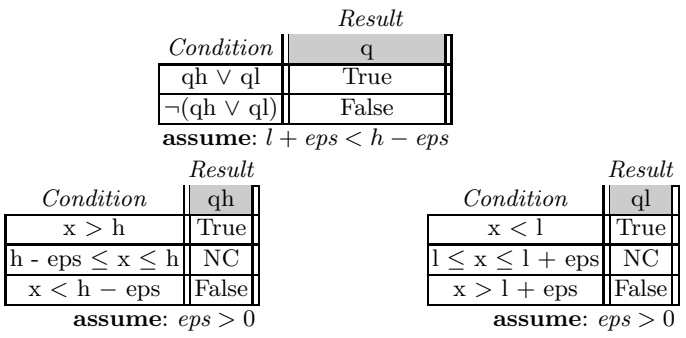

Fig. 5: *Limits_alarm* requirement in tabular expression

As an example, we consider the requirement for function block *limits_alarm* (Sec. 2). The expected input-output behaviour and its tabular requirement (which constrains the relation between inputs x, h, l, eps and outputs q, qh, ql) is depicted in Fig. 5. Our formalization process revealed the need for two missing assumptions from IEC 61131-3: $eps > 0$ and $l + eps < h - eps$. They allow us to ensure that the two hysteresis zones $[l, l + eps]$ and $[h - eps, h]$ are non-empty, disjoint and ordered [19].

Let predicates `f_qh`, `f_ql`, and `f_q` be those that formalize, respectively, the table for qh, ql and q, we then translate the above requirement into PVS as:

```
LIMITS_ALARM_REQ(h,x,l,eps,qh,q,ql)(t): bool =
  f_qh(x,h,eps,qh)(t) & f_ql(x,l,eps,ql)(t) & f_q(qh,ql,q)(t)
```

By using the function definitions of q, qh and ql, we can verify the correctness of the FBD implementation of *limits_alarm*, formalized as the predicate above. This process can be generalized to verify other FBDs in IEC 61131-3.

4 Verifying Function Blocks in PVS

We now present the two kinds of verification we perform.

4.1 Verifying the Correctness of an Implementation Given an implementation predicate I, our correctness theorem states that, if I holds for all possible inputs and outputs, then the corresponding requirement predicate R also holds. This corresponds to the proofs of *correctness* shown in Fig. 1. For example, to prove that the FBD implementation of block *limits_alarm* in Sec. 3.1 is *correct* with respect to its requirement in Sec. 3.2, we must prove the following in PVS:

$$\begin{aligned} \vdash \forall h, x, l, eps \bullet \forall qh, q, ql \bullet limits\_alrm\_impl(h, x, l, eps, qh, q, ql) \Rightarrow \\ limits\_alrm\_req(h, x, l, eps, qh, q, ql) \end{aligned} \tag{1}$$

Furthermore, we also need to ensure that the implementation is consistent or feasible, i.e., for each input list, there exists at least one corresponding list of outputs, such that I holds. Otherwise, the implementation trivially satisfies any requirements. This is shown in Fig. 1 as proofs of *consistency*. In the case of *limits_alarm*, we must prove the following in PVS:

$$\vdash \forall h, x, l, eps \bullet \exists qh, q, ql \bullet limits\_alrm\_impl(h, x, l, eps, qh, q, ql) \tag{2}$$

4.2 Verifying the Equivalence of Implementations In IEC 61131-3, block *limits_alarm* is supplied with ST only. In theory, when both ST and FBD implementations are supplied for the same FB (e.g., *stack_int*), it may suffice to verify that each of the implementations is *correct* with respect to the requirement. However, as the behaviour of FBs is intended to be deterministic in most cases, it would be worth proving that the implementations (if they are given at the same level of abstraction) are equivalent, and generate scenarios, if any, where they are not. This is also labelled in Fig. 1 as proofs of *equivalence*.

In Sec. 3.1 we discussed how to obtain, for a given FB, a predicate for its ST description (say *FB_st_impl*) and one for its FBD description (say *FB_fbd_impl*). Both predicates share the same input list $i_1, \ldots, i_m$ and output list $o_1, \ldots, o_n$. Consequently, to verify that the two supplied implementations are equivalent, we must prove the following in PVS:

$$\begin{array}{l} \vdash \forall i_1, \ldots, i_m \bullet \forall o_1, \ldots, o_n \bullet \\ \quad FB\_st\_impl(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv FB\_fbd\_impl(i_1, \ldots, i_m, o_1, \ldots, o_n) \end{array} \tag{3}$$

However, the verification of block *stack_int* is an exception. Its ST and FBD implementations are at different levels of abstraction: the FBD description is closer to the hardware level as it declares additional, auxiliary variables to indicate system errors (Appendix E of IEC 61131-3) and thus cause interrupts. Consequently, we are only able to prove a refinement (i.e., implication) relationship instead (i.e., the FBD implementation implies the ST implementation).

Although IEC 61131-3 (2003) had been in use for almost 10 years, while performing this verification on *stack_int*, we found an error (of a missing FB in the FBD implementation) that made the above implication unprovable [19].

5 Case Study: Issues Found in Standard IEC 61131-3

To justify the value of our approach (Secs. 3 and 4), we have formalized and verified 23 of 29 FBs from IEC 61131-3. Our coverage so far has revealed a number of issues that are listed in the introduction. We briefly discuss the first three and our reactions to them. The complete discussion is available in [19].

5.1 Ambiguous Behaviour: Pulse Timer in Timing Diagrams Block *pulse* is a timer defined in IEC 61131-3, whose graphical declaration is shown on the LHS of Fig. 6. It takes two inputs (a boolean condition *in* and a length *pt* of time period) and produces two outputs (a boolean value *q* and a length *et* of time period). It acts as a pulse generator: as soon as the input condition *in* is detected to hold, it generates a pulse to let output *q* remain *true* for a constant *pt* of time units. The elapsed time that *q* has remained *true* can also be monitored via output *et*. IEC 61131-3 presents a timing diagram[3] as depicted on the RHS of Fig. 6, where the horizontal time axis is labelled with time instants t_i ($i \in 0..5$), to specify (an incomplete set of) the behaviour of block *pulse*.
The above timing diagram suggests that when a rising edge of the input condition *in* is detected at time t, another rising edge that occurs before time $t+pt$ may not be detected, e.g., the rising edge occurring at t_3 might be missed as $t_3 < t_2 + pt$.

The use of timing diagrams to specify behaviour is limited to a small number of use cases; subtle or critical boundary cases are likely to be missing. We formalize the *pulse* timer using tabular expressions that ensure both completeness and disjointness. We found that there are at least two scenarios that are not covered by the above timing diagram supplied by IEC 61131-3. *First*, if a rising edge of condition *in* occurred at $t_2 + pt$, should there be a pulse generated to

[3] For presenting our found issues, it suffices to show just the parts of *in* and *q*.

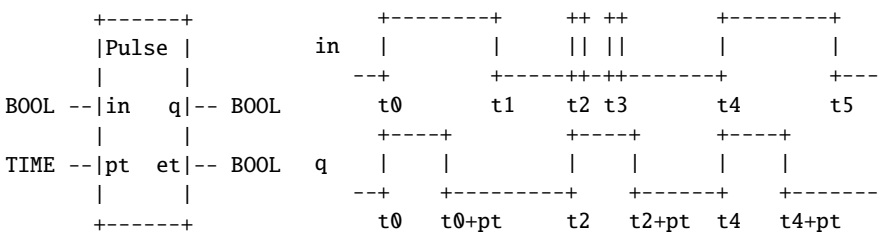

Fig. 6: *pulse* timer declaration and definition in timing diagram

let output q remain *true* for another pt time units? If so, there would be two connected pulses: from t_2 to $t_2 + pt$ and from $t_2 + pt$ to $t_2 + 2pt$. *Second*, if the rising edge that occurred at t_3 stays high until some time t_k, $(t_2 + pt \leq t_k \leq t_4)$, should the output *et* be default to 0 at time $t_2 + pt$ or at time t_k?

	Condition	*Result* q
$\neg q_{-1}$	$\neg in_{-1} \wedge in$	true
	$in_{-1} \vee \neg in$	false
$\mathbf{q_{-1}}$	**Held_For(q,pt)**	**false**
	¬Held_For(q,pt)	true

Condition	*Result* pulse_start_time
$\neg q_{-1} \wedge q$	t
$q_{-1} \vee \neg q$	NC

Condition			*Result* et
q			t − pulse_start_time
¬q	¬Held_For_ts(in,pt,pulse_start_time)		0
	Held_For_ts (in,pt,pulse_start_time)	**in** **t − pulse_start_time ≥ pt**	**pt**
		t − pulse_start_time < pt	t − pulse_start_time
		¬in	0

Fig. 7: Requirement of *pulse* timer using tabular expressions

We use the three tables in Fig. 7 to formalize the behaviour of the *pulse* timer, where outputs q and *et* and the internal variable *pulse_start_time* are initialized to, respectively, *false*, 0, and 0. The tables have their obvious equivalents in PVS. To make the timing behaviour precise, we define two auxiliary predicates:

```
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))
```

Predicate *Held_For(P, duration)* holds when the input predicate P holds for at least *duration* units of time. Predicate *Held_For_ts(P, duration, ts)* is more restricted, insisting that the starting time of *duration* is *ts*. As a result, we make

explicit assumptions to disambiguate the above two scenarios. Scenario 1 would match the condition row (in bold) in the upper-left table for output q, where q at the previous time tick holds (i.e., q_{-1}) and q has already held for pt time units, so the problematic rising edge that occurred at $t_2 + pt$ would be missed. Due to our resolution to Scenario 1, at time $t2 + pt$, Scenario 2 would match the condition row (in bold) in the lower table for output et, where q at the current time tick does not hold (i.e., $\neg q$), condition in has held for more than pt time units, so the value of et remains as pt without further increasing.

As *pulse* timer is not supplied with implementation, there are no correctness and consistency proofs to be conducted. Nonetheless, obtaining a precise, complete, and disjoint requirement is valuable for future concrete implementations.

5.2 Ambiguous Behaviour: Implicit Delay Unit PLC applications often use feedback loops: outputs of a FB are connected as inputs of either another FB or the FB itself. IEC 61131-3 specifies feedback loops through either a connecting line or shared names of inputs and outputs. However, feedback values (or of intermediate output values) cannot be computed instantaneously in reality. We address this issue by introducing a delay block Z_{-1} and its formalization below:

$$Z_{-1}(i,o)(t) = \begin{cases} o(t) = i(t-1) & \textbf{if } t > 0 \\ \mathit{False} & \textbf{if } t = 0 \end{cases}$$

$$\begin{aligned} &\mathit{sr\_impl}(s_1, r, q_1) \\ &\quad \equiv \exists w_1, w_2, w_3 \bullet \\ &\qquad \begin{pmatrix} \mathit{neg}(r, w_1) \\ \wedge\ \mathit{conj}(w_1, w_3, w_2) \\ \wedge\ \mathit{disj}(s_1, w_2, q_1) \\ \wedge\ Z_{-1}(q_1, w_3) \end{pmatrix} \end{aligned}$$

Fig. 8: Block *sr* implementation in FBD and its formalizing predicate

There is an explicit, one-tick delay between the input and output of block Z_{-1}, making it suitable for denoting feedback values as output values produced in the previous execution cycle. The types of i and o must match. For example, block *sr* creates a set-dominant latch (a.k.a. flip-flop), takes as inputs a boolean set flag s_1 and a boolean reset flag r, and returns a boolean output q_1. The value of q_1 is fed back as another input of block *sr*. Value of q_1 remains *true* as long as the set flag s_1 is enabled, and q_1 is reset to *false* only when both flags are disabled. There should be a delay between the value of q_1 is computed and passed to the next execution cycle. We formalize this by adding the explicit delay block Z_{-1} and conjoining predicates for the internal blocks (as shown in Fig. 8). Blocks B_1 (formalized by predicate *neg*), B_2 (*conj*), B_3 (*disj*), and B_4 (Z_{-1}) in Fig. 8 denote the FB of, respectively, logical negation, conjunction, disjunction, and delay. Arrows w_1, w_2, and w_3 are internal connectives. Adding an explicit

delay block Z_{-1} to formalize feedback loops led us to discharge the correctness and consistency theorems (Sec. 4) of the FBD implementation in Fig. 8.

5.3 Missing Assumption: Limit on Counters FBs An up-down counter (*ctud*) in IEC 61131-3 is composed of an up counter (*ctu*) and a down counter (*ctd*). The output counter value *cv* is incremented (using the up counter) if a rising edge is detected on an input condition *cu*, or *cv* is decremented (using the down counter) if a rising edge is detected on the input *cd*. Actions of increment and decrement are subject to, respectively, a high limit $PVmax$ and a low limit $PVmin$. The value of *cv* is loaded to a preset value *pv* if a load flag *ld* is *true*; and it is default to 0 if a reset condition *r* is enabled. Two Boolean outputs are produced to reflect the change on *cv*: $qu \equiv (cv > pv)$ and $qd \equiv (cv <= 0)$.

As we attempted to formalize and verify the correctness of the ST implementation of block *ctud* supplied by IEC 61131-3, we found two missing assumptions.

First, the relationship between the high and low limits is not stated. Let $PVmin$ be 10 and $PVmax$ be 1, then the counter can only increment when $cv < 1$, decrement when $cv > 10$ (disabled when $1 \leq cv \leq 10$). This contradicts with our intuition about how low and high limits are used to constrain the behaviour of a counter. Consequently, we introduce a new assumption[4]: $PVmin < PVmax$.

				Result
Condition				cv
r				0
¬r	ld			pv
	¬ld	cu ∧ cd		NC
		cu∧¬cd	cv_{-1}< PVmax	cv_{-1}+1
			cv_{-1}≥ PVmax	NC
		¬cu∧cd	cv_{-1}> PVmin	cv_{-1}-1
			cv_{-1}≤ PVmin	NC
		¬cu ∧ ¬cd		NC

assume: $PVmin < pv < PVmax$

Fig. 9: Tabular requirement of *ctud*

Second, the range of the preset value *pv*, with respect to the limits $PVmin$ and $PVmax$, is not clear. If *cv* is loaded by the value of *pv*, such that $pv > PVmax$, the output *qu* can never be *true*, as the counter increments when $cv < PVmax$. Similarly, if *pv* is such that $pv < PVmin$ and $pv = 1$, the output *qd* can never be *true*, as the counter decrements when $cv > PVmin$. As a result, we introduce another assumption: $PVmin < pv < PVmax$. Our tabular requirement for the up-down counter that incorporates the missing assumption is shown in Fig. 9. Similarly, we added $pv < PVmax$ and $PVmin < pv$ as assumptions for, respectively, the up and down counters.

6 Related Work

There are many works on formalizing and verifying PLC programs specified by programming languages covered in IEC 61131-3, such as sequential function charts (SFCs). Some approaches choose the environment of model checking: e.g., to formalize a subset of the language of instruction lists (ILs) using timed

[4] If the less intuitive interpretation is intended, we fix the assumption accordingly.

automata, and to verify real-time properties in Uppaal [15]; to automatically transform SFC programs into the synchronous data flow language of Lustre, amenable to mechanized support for checking properties [12]; to transform FBD specifications to Uppaal formal models to verify safety applications in the industrial automation domain [23]; to provide the formal operational semantics of ILs which is encoded into the symbolic model checker Cadence SMV, and to verify rich behavioural properties written in linear temporal logic (LTL) [5]; and to provide the formal verification of a safety procedure in a nuclear power plant (NPP) in which a verified Coloured Petri Net (CPN) model is derived by reinterpretation from the FBD description [17]. There is also an integration of SMV and Uppaal to handle, respectively, untimed and timed SFC programs [2].

Some other approaches adopt the verification environment of a theorem prover: e.g., to check the correctness of SFC programs, automatically generated from a graphical front-end, in Coq [3]; and to formalize PLC programs using higher-order logic and to discharge safety properties in HOL [24]. These works are similar to ours in that PLC programs are formalized and supported for mechanized verifications of implementations. An algebra approach for PLC programs verification is presented in [22]. In [14], a trace function method (TFM) based approach is presented to solve the same problem as ours.

Our work is inspired by [16] in that the overall system behaviour is defined by taking the conjunction of those of internal components (circuits in [16] or FBs in our case). Our resolutions to the timing issues of the *pulse* timer are consistent with [11]. However, our approach is novel in that 1) we also obtain tabular requirements to be checked against, instead of writing properties directly for the chosen theorem prover or model checker; and 2) our formalization makes it easier to comprehend and to reason about properties of disjointness and completeness.

7 Conclusion and Future Work

We present an approach to formalizing and verifying function blocks (FBs) using tabular expressions and PVS. We identified issues concerning ambiguity, missing assumptions, and erroneous implementations in the IEC 61131-3 standard of FBs. As future work, we will apply the same approach to the remaining FBs in IEC 61131, and possibly to IEC 61499 that fits well with distributed systems.

References

1. Bakhmach, E., O.Siora, Tokarev, V., Reshetytskyi, S., Kharchenko, V., Bezsalyi, V.: FPGA - based technology and systems for I&C of existing and advanced reactors. International Atomic Energy Agency p. 173 (2009), IAEA-CN-164-7S04
2. Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., Stursberg, O.: Verification of PLC programs given as sequential function charts. In: Integration of Software Specification Techniques for Applications in Engineering, LNCS, vol. 3147, pp. 517–540. Springer Berlin Heidelberg (2004)
3. Blech, J.O., Biha, S.O.: On formal reasoning on the semantics of PLC using Coq. CoRR abs/1301.3047 (2013)

4. Camilleri, A., Gordon, M., Melham, T.: Hardware verification using higher-order logic. Tech. Rep. UCAM-CL-TR-91, Cambridge Univ. Computer Lab (1986)
5. 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)
6. Eles, C., Lawford, M.: A tabular expression toolbox for Matlab/Simulink. In: NASA Formal Methods. pp. 494–499 (2011)
7. Hu, X., Lawford, M., Wassyng, A.: Formal verification of the implementability of timing requirements. In: FMICS. LNCS, vol. 5596, pp. 119–134. Springer (2009)
8. IEC: 61131-3 Ed. 2.0 en:2003: Programmable Controllers — Part 3: Programming Languages. International Electrotechnical Commission (2003)
9. IEC: 61131-3 Ed. 3.0 en:2013: Programmable Controllers — Part 3: Programming Languages. International Electrotechnical Commission (2013)
10. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Science of Computer Programming 75(11), 980 – 1000 (2010)
11. John, K.H., Tiegelkamp, M.: IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids. Springer, 2nd edn. (2010)
12. Kabra, A., Bhattacharjee, A., Karmakar, G., Wakankar, A.: Formalization of sequential function chart as synchronous model in Lustre. In: NCETACS. pp. 115–120 (2012)
13. 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)
14. Liu, Z., Parnas, D., Widemann, B.: Documenting and verifying systems assembled from components. Frontiers of Computer Science in China 4(2), 151–161 (2010)
15. Mader, A., Wupper, H.: Timed automaton models for simple programmable logic controllers. In: ECRTS. pp. 114–122. IEEE (1999)
16. Melham, T.: Abstraction mechanisms for hardware verification. In: VLSI Specification, Verification and Synthesis. pp. 129–157. Kluwer Academic Publishers (1987)
17. Németh, E., Bartha, T.: Formal verification of safety functions by reinterpretation of functional block based specifications. In: FMICS, pp. 199–214. Springer (2009)
18. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: CADE. LNCS, vol. 607, pp. 748–752 (1992)
19. Pang, L., Wang, C.W., Lawford, M., Wassyng, A.: Formalizing and verifying function blocks using tabular expressions and PVS. Tech. Rep. 11, McSCert (Aug 2013)
20. Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming 25(1), 41–61 (1995)
21. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Transactions on Software Engineering 20, 948–976 (1994)
22. Roussel, J.M., Faure, J.: An algebraic approach for PLC programs verification. In: 6^{th} International Workshop on Discrete Event Systems. pp. 303–308 (2002)
23. 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)
24. Völker, N., Krämer, B.J.: Automated verification of function block-based industrial control systems. Science of Computer Programming 42(1), 101 – 113 (2002)
25. Wassyng, A., Janicki, R.: Tabular expressions in software engineering. In: Proceedings of ICSSEA'03. vol. 4, pp. 1–46. Paris, France (2003)