McMaster Centre for Software Certification

Formalizing and Verifying Function Blocks using Tabular Expressions and PVS

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

McSCert report 11

Augest 2013

McMaster Centre for Software Certification

The Need for Certification

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

The Need for Evidence

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

The Centre

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

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

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

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

Formalizing and Verifying Function Blocks using Tabular Expressions and PVS

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

McMaster Centre for Software Certification(McSCert),
Department of Computing and Software,
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 high reliability, 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 verifying the correctness of the FBs implementations in the PVS proof environment. We apply our approach to the example FBs of the IEC 61131-3 standard. Our approach identified issues in the standard, ambiguous behavioural descriptions, missing assumptions, and erroneous implementations. In addition, we are able to provide suggested resolutions to these issues.

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 [3]. Function blocks (FBs) are basic design units that implement behaviour on 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 to be compliant with an international standard, such as IEC 61131-3 [10, 11]. Standards such as DO-178C [2] (in the aviation domain) and IEEE 7-4.3.2 [1] (in the nuclear domain) list acceptance criteria of mission- or safety-critical system for practitioners to comply with. Two important criteria are that 1) the system requirements are precise and complete; and that 2) the system implementations exhibits behaviours that conform 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 [25][26] are a promising way to documenting the system requirement and have proven to be both practical and effective for use in industry [17][30]. PVS [23] 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 requirements).

As a case study, we have formalized and verified 23 of 29 FBs listed in IEC 61131-3 [10][11], which has long been one of the most important standards in the industrial automation of critical systems using PLCs[1]. 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 venders 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. From our point of view, IEC 61131-3 still lacks an adequate, formal language for describing the behaviours of FBs and for arguing 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 [10] has been in use since 2003. Most of the work reported on in this paper relates to this version. When the newer version was issued, we expected to find that the problems we had discovered in the earlier version had been corrected. Instead we found that many of the example FBs have been removed from the standard and the remaining FBs are still problematic.

Figure 1 presents our approach, and we use it to summarize our research contributions as follows:

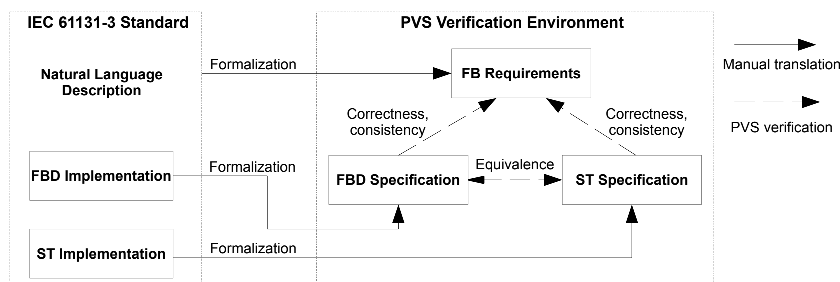


Fig. 1: Framework

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 Section 3.2. The ST and FBD

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

implementations are formalized as predicates in PVS, again making use of tables, as described in Section 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 (Section 4).

Using our approach, we have identified a number of issues in IEC 61131-3 and provide suggested resolutions (Section 5), and summarized below:

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) and (2) in further detail in Section 5. 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 of 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 meanings of programming language for PLCs, the International Electrotechnical Committee (IEC) first published the IEC 61131-3 Standard in 1993 and released two revisions in 2003 [10] and in 2013 [11]. Most of our research results were completed before the third edition was released. Nonetheless, the third edition is fully compatible with the second. In particular, those 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 (FBs) 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 of the programming languages that are covered in IEC 61131-3 for writing behavioural descriptions of FBs: structured text (ST) and function block diagrams (FBDs). The syntax of ST is block structured and resembles that of Pascal, while FBDs visualize the inter-connections or data flows between inputs and outputs of block components.

As an example of FBDs, we consider the *limits_alarm* block that we will be using as a running example for later sections. In Figure 2, the FBD of *limits_alarm* as shown below consists of two parts: 1) declaring its inputs and outputs; and 2) defining its body of computation.

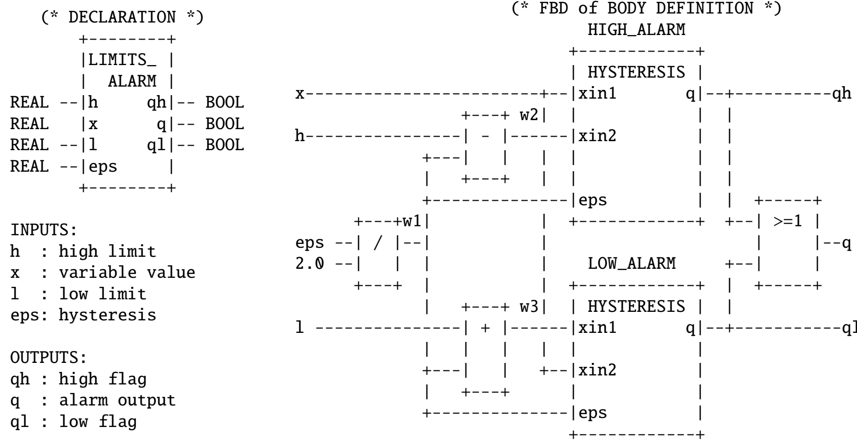


Fig. 2: *Limits_alarm* standard declaration and FBD implementation

Block *limits_alarm* implements an alarm that 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 of *limits_alarm* 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. We will discuss the precise input-output requirement, using tabular expressions, of *limits_alarm* in Section 3.2.

2.2 Tabular Expressions. Tabular expressions [12, 24–26] (a.k.a. function tables) are an effective approach to describing conditionals and relations, and they are thus ideal for documenting system requirements. They are arguably easier to comprehend and to maintain than conventional mathematical expressions. Tabular expressions have well-defined formal semantics (e.g., [13][14]). Reference [13] presents a relational semantics for tabular expressions. The model covers most of the known types of tabular expressions used in software engineering, and admits precise classification and definition of new types of tabular expressions. Recently, reference [14] presented a new semantics for tabular expressions by using indexing to decouple the appearance of a tabular expression from its semantics. They have been proven to be of great help both in inspections [30] and in testing and verification [31]. For our purpose of capturing the input-output requirements of standard functions and function blocks in IEC 61131-3, the following tabular structure in Figure 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.

<table>
<tr><td colspan="2">Condition</td><td>Result
f</td></tr>
<tr><td rowspan="4">C_1</td><td>$C_{1.1}$</td><td>res_1</td></tr>
<tr><td>$C_{1.2}$</td><td>res_2</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>$C_{1.m}$</td><td>res_m</td></tr>
<tr><td colspan="2">...</td><td>...</td></tr>
<tr><td colspan="2">C_n</td><td>res_n</td></tr>
</table>

```
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
```

Fig. 3: Semantics of 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 $(C_1 \wedge C_{1.1} \Rightarrow f = res_{1.1})$, and so on. In documenting input-output behaviours (total functions) using horizontal condition tables (HCTs), there are two important properties to reason about: *completeness* and *disjointness*. Completeness ensures that at least one row is applicable to every input, i.e., $(C_1 \vee C_2 \vee \cdots \vee C_n)$ is always $True$. 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 $C_{1.1}, \ldots, C_{1.m}$. Checks on these properties can often be easily automated in development environments such as Matlab/Simulink [8] and PVS [23].

2.3 PVS. Prototype Verification System (PVS) [23] is developed by SRI as an interactive environment for writing formal specifications and performing formal proofs. The specification language of PVS 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 in later sections.

PVS has a powerful interactive proof checker to perform sequent-style deductions. Syntactically, a PVS sequent is showed as: $P_1, P_2, \ldots, P_m \vdash Q_1, Q_2, \ldots, Q_n$, where P_i, $i = 1, 2, \ldots, m$, and Q_j, $j = 1, 2, \ldots, n$ are formulas and $\vdash$ is entailment. In PVS, a sequent is displayed as:

```
{-1}  P1
{-2}  ...
{-3}  Pm
  |-------
{1}   Q1
{2}   ...
{3}   Qn
```

The antecedents are combined by conjunctives while consequents are connected by disjunctives. Thus, PVS sequent is logically equivalent to $P_1 \wedge P_2 \wedge \cdots \wedge P_m \vdash Q_1 \vee Q_2 \vee \cdots \vee Q_n$. In our specification, we use $\neg$, $\wedge$, $\vee$, $\Rightarrow$ to denote logical negation, conduction, disjunction and implication and $\forall$, $\exists$ for universal and existential quantifiers. It is very useful to formulate intermediate lemmas either to be reused multiple times to eliminate repeated work or to decompose complex problem into smaller ones and address one of them at a time. For example, we can formulate and verify one lemma for each output correctness verification that can be used when we verify overall correctness of this function block.

The completeness and disjointness properties are generated automatically as Type Correctness Conditions (TCCs) to be discharged. After typechecking which is required to ensure conservative extension of PVS logic, most of the generated proof obligations are discharged automatically. Complex TCCs and any other user-made lemmas and theorems need to be interactively proved by users. The proof is not considered to be complete until all TCCs and any imported and current theory have been proved. We will discuss a found issue (Section 5) where the ST implementation supplied by IEC 61131-3 is formalized as PVS table but its disjointness TCC failed to be discharged.

Three important definitions are used in our work, *equivalence*, *consistency* and *correctness*. Each of them will be represented as theorems in PVS and will be discussed in Section 4.

Equivalence Implementations of a function block in different programming languages are equivalent if their implementation predicates are logically equivalent.

Consistency Implementation of a function block is consistent (or feasible) if there exists such assignments for arguments of implementation predicate that this predicate is satisfied.

Correctness Implementation of a function block is correct against its requirement if requirement predicate can be logically implied by implementation predicate.

An example of using tabular expressions to model software specified and verified in PVS in the Darlington Nuclear Shutdown System (SDS) can be found in reference [17].

As PLCs are commonly used in real-time systems, modelling of time is a critical aspect in our formalization. We consider a discrete-time model in which a time series consists of equally divided clock ticks, denoted as

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

where $\delta \in \mathbb{R}^+$ and δ is small enough to represent time interval between two consecutive clock ticks.

More precisely, in PVS we define:

```
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. We define two type synonyms: `time` as the set of non-negative real numbers, and `tick` as the set of time instants that are equally divided. We will use operators on `tick` [9], such as `pre` and `next` that return, respectively, the previous and the next time tick. This definition of `tick` is reproduced by [9] from [18].

3 Formalizing Function Blocks using Tabular Expressions

IEC 61131-3 does not adopt a unified, formal language to define standard functions and function blocks. In many cases, both structural text (ST) and function

block diagram (FBD) are used to describe a single function block. However, both ST and FBD are informal, implementation-oriented notations, and they are thus not adequate for capturing the precise input-output relationship that is both complete and disjoint. Moreover, it is not possible to formally establish that these implementations are *correct* (i.e. consistent with the input-output requirement). We present a formal approach to defining standard functions and function blocks in IEC 61131-3 using tabular expressions. For each function block in IEC 61131-3, we: 1) translate the supplied ST or FBD implementation into predicates in PVS (Section 3.1); and 2) capture its input-output requirement using tabular expressions in PVS (Section 3.2). Consequently, we have a unified, formal framework to verify the correctness of function blocks (Section 4).

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 [6], 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:

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

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*).

A predicate representing its behaviour is expressed as "if-then-else" statement in which initial value is located in "if" branch and other cases are defined by a tabular expression embedded in "else" branch. NC is used to indicate the effect of no change. Thus, the definition of function *move* can be expressed in (1):

$$\begin{aligned} &\mathit{move}(\mathit{en{:}\ tick} \rightarrow \mathit{bool},\ \mathit{in},\ \mathit{out{:}\ tick} \rightarrow \mathit{int})(t : \mathit{tick}) : \mathit{bool} \\ &\quad \equiv \mathit{out(t)} = \\ &\quad\quad \mathit{IF}\ t = 0\ \mathit{THEN}\ 0 \\ &\quad\quad \mathit{ELSE} \end{aligned} \tag{1}$$

Condition	*Result*: out
en	in
¬en	NC

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 of 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, we may simplify the behaviour of function *move* by eliminating the enabling condition *en* and constraining that *in* and *out* are always equal. As another example, consider the standard function *add* that is used as an internal component of the FB *limits_alarm* (Section 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: $add(in_1,\, in_2) : int = in_1 + in_2$.

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* (Section 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. In the case of an ST implementation supplied by IEC 61131-3, we translate it into its equivalent expression in PVS. Figure 4 summarizes our ST-to-PVS translation strategy, illustrated using concrete examples.

<table>
<tr><th>#</th><th>ST expressions</th><th>PVS predicates</th></tr>
<tr><td>1</td><td colspan="2">basic assignments</td></tr>
<tr><td></td><td>a := a + b; c := NOT (a > 0)</td><td>a = a_{-1} + b & c = NOT (a > 0)</td></tr>
<tr><td>2</td><td colspan="2">conditional assignments</td></tr>
<tr><td></td><td>IF z THEN
b := c * 3; c := a + b;
ELSE
b := b + c; e := b - 1;
END_IF</td><td>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</td></tr>
<tr><td>3</td><td colspan="2">function block invocations, reuse</td></tr>
<tr><td></td><td>FB1(in_1 := a, in_2 := b);
FB2(in_1 := FB1.output);
out := FB2.output;</td><td>FB1_REQ(a, b, fb1_out) &
FB2_REQ(fb1_out, fb2_out) &
out = fb2_out</td></tr>
<tr><td>4</td><td colspan="2">output initialization</td></tr>
<tr><td></td><td>VAR_OUTPUT
q: int = False;
END_VAR</td><td>t: VAR tick
q: VAR [tick -> bool]
q = IF t = 0 THEN False
ELSE ... (def. of q at t > 0)
END_IF</td></tr>
<tr><td>5</td><td colspan="2">for-loops</td></tr>
<tr><td></td><td>(suppose:
q is initialized to q0)
FOR i := M TO 1 BY -1 DO;
q := q + i;
END_FOR</td><td>loop(i): RECURSIVE nat =
IF i = 0 THEN 0
ELSE loop(i-1) + i END_IF
MEASURE (LAMBDA (i: nat): i)
q = q0 + loop(M)</td></tr>
</table>

Fig. 4: Translation from ST language to PVS language

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. Pattern 2 illustrates that we reconstruct conditional statements by taking the conjunction of the assignment effect of each variable; each variable assignment is formalized via a tabular expression. The order of evaluation depends on how variables are used in the function. For ex-

ample, b is evaluated before c to compute $c = a + b$ by $b = c_{-1} + 3$. For the above example (#2 in Fig. 4), 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 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`. Pattern 4 illustrates that in a timed context, we transform the initialization of an output `q` into a predicate that constrains `q`'s value at time instant 0 accordingly. As an example, see how we formalized standard function *move* in the beginning of this section. Pattern 5 illustrates that we formalize a for-loop as a recursive function, equipped with a *measure* (or variant) function that specifies the progress towards termination.

We include an example in Section 5 formalization of the ST implementation *hysteresis*. In this example, the ST implementation is translated into its equivalent predicate expressions in PVS. Its tabular requirement is also made to be verified against with implementation. We claim that for each FB with a ST implementation supplied, there is a corresponding predicate using the above translation rules.

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 Figure 5:

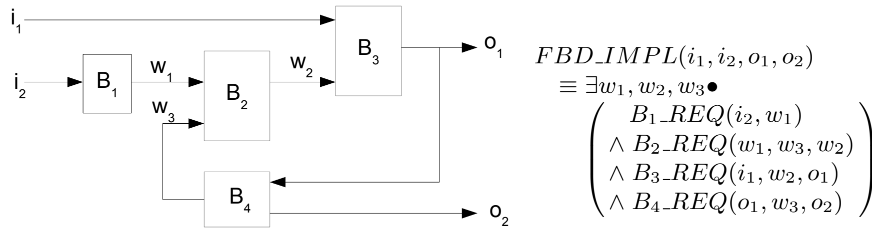


Fig. 5: Composite FB implementation in FBD and its formalizing predicate

The above FBD consists of four internal blocks, B_1, B_2, B_3, and B_4, that are previously developed for their formalization (i.e. their formalizing predicates B_1_REQ, ..., B_4_REQ are available). 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 (to be discussed in Section 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 interconnectives (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* (Section 2) as a predicate `LIMITS_ALARM_IMPL` in PVS:

```
t: VAR tick
x, h, l, w2, w3: VAR [tick -> real]
```

```
eps, w1: VAR [tick -> posreal]
qh, ql, q: VAR pred[tick]

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.

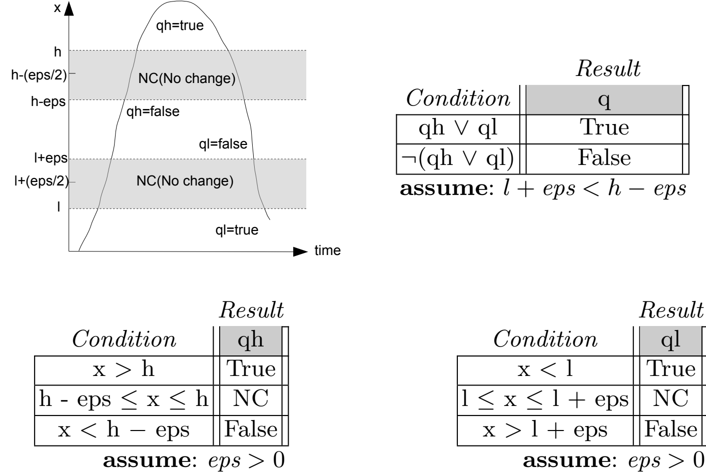
3.2 Formalizing Requirements of Function Blocks

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, or the consistency between multiple implementations, 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.

As an example, we consider the requirement for function block *limits_alarm* (Section 2). The expected input-output behaviour is depicted in the following Figure 6, and its tabular requirement (which constrains the relation between inputs x, h, l, eps and outputs q, qh, ql) is captured in the three surrounding tables. When variable value x exceeds the high limit h, the high flag qh becomes true. Symmetrically, when x goes below the low limit l, the low flag ql becomes true. Both flags qh and ql are set to *false* when x is in the exclusive range of $(l + eps, h - eps)$. There exists a hysteresis band for the high limit inside which the value of qh remains unchanged: $[h - eps, h]$. Symmetrically, there exists a hysteresis band for the low limit: $[l, l + eps]$. Finally, the alarm output q is set to *true* if and only if either of the flags is set to *true*, or set to *false* otherwise. The input-output requirement is captured in the three function tables. We will discuss (Section 5) how our formalization process revealed the need for the two missing assumptions from IEC 61131-3: $eps > 0$ and $l + eps < h - eps$.

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)
```



Condition	*Result* q
qh ∨ ql	True
¬(qh ∨ ql)	False

assume: $l + eps < h - eps$

Condition	*Result* qh
x > h	True
h - eps ≤ x ≤ h	NC
x < h − eps	False

assume: $eps > 0$

Condition	*Result* ql
x < l	True
l ≤ x ≤ l + eps	NC
x > l + eps	False

assume: $eps > 0$

Fig. 6: *Limits_alarm* requirement in tabular expression

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 consider the ST and FBD descriptions supplied by IEC 61131-3 as implementations of FBs. For each FB, under the same proof environment of PVS, we formalize (Section 3.1) its supplied implementation and capture (Section 3.2) its input-output requirement that is both complete and disjoint. We now present the two kinds of verification we perform.

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

$$\begin{aligned} \vdash \forall h, x, l, eps \ \bullet \ \forall qh, q, ql \bullet \\ limits_alrm_impl(h, x, l, eps, qh, q, ql) \\ \Rightarrow limits_alrm_req(h, x, l, eps, qh, q, ql) \end{aligned} \tag{2}$$

Furthermore, we also need to ensure that the suggested 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. It is showed in Figure 1 of the proofs of *consistency*. In the case of *limits_alarm*, we must prove the following in PVS:

$$\vdash \forall h, x, l, eps \ \bullet \ \exists qh, q, ql \ \bullet \ limits_alrm_impl(h, x, l, eps, qh, q, ql) \tag{3}$$

Verifying the Equivalence between Implementations. In IEC 61131-3, block *limits_alarm* is supplied with a ST implementation only. In theory, when both ST and FBD implementations are supplied for the same FB (e.g. block *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 observing scenarios, if any, where they are not. This is also labelled in Firgure 1 with the proofs of *equivalence*.

In Section 3.1 we discussed how to obtain, for a given FB, a predicate for its ST implementation (say *FB_st_impl*) and one for its FBD implementation (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:

$$\begin{aligned} \vdash \forall i_1, \ldots, i_m \bullet \forall o_1, \ldots, o_n \bullet \\ FB_st_impl(i_1, \ldots, i_m, o_1, \ldots, o_n) \\ \equiv FB_fbd_impl(i_1, \ldots, i_m, o_1, \ldots, o_n) \end{aligned} \tag{4}$$

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. In this case, we only need to prove that the higher level ST implementation can be logically implied from the lower level FBD implementation, i.e., replace the "≡" with "⇐" in Equation (4). Thus, we will prove the following formula instead:

$$\begin{aligned} \vdash \forall i_1, \ldots, i_m \bullet \forall o_1, \ldots, o_n \bullet \\ FB_fbd_impl(i_1, \ldots, i_m, o_1, \ldots, o_n) \\ \Rightarrow FB_st_impl(i_1, \ldots, i_m, o_1, \ldots, o_n) \end{aligned} \tag{5}$$

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 (5) unprovable.

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

To justify the value of our proposed approach (Sections 3 and 4), we have formalized and verified many[2] of the FBs listed in IEC 61131-3, as well as standard functions that are used in these function blocks. Our coverage so far reveals a number of issues that are listed in the introduction. For the purpose of this paper, we discuss the above found issues and our reactions to them.

5.1 Ambiguous Behaviour: Pulse Timer in Timing Diagrams. Block *pulse* is one of the timers defined in IEC 61131-3. Its graphical declaration is shown in on the LHS of the Figure 8 below. It takes two inputs (a boolean condition *in* and a length *pt* of time period) and produces two outputs (a boolean

[2] We have formalized and verified, in PVS, 22/29 FBs from IEC 61131-3 at the time of submitting this paper.

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 be true, 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 the Figure 8 below, 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*.

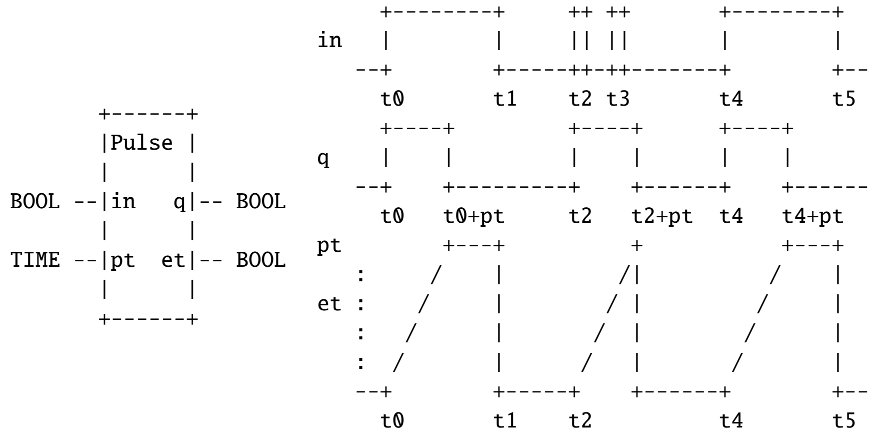


Fig. 7: *pulse* timer declaration and definition in timing diagram

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 inherently limited to a small, finite 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:

1. If a rising edge of condition in occurred at $t_2 + pt$, should there be a pulse generated to 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.
2. 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?

We use the three tables in Figure 8 to formalize the behaviour of the *pulse* timer, where outputs q and et, as well as an internal variable *pulse_start_time*, are initialized to, respectively, *false*, 0, and 0.

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

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

<table>
<tr><th colspan="4">Condition</th><th>Result
et</th></tr>
<tr><td colspan="4">q</td><td>t - pulse_start_time</td></tr>
<tr><td rowspan="4">¬q</td><td colspan="3">¬Held_For_ts(in,pt,pulse_start_time)</td><td>0</td></tr>
<tr><td rowspan="3">Held_For_ts (in,pt, pulse_start_time)</td><td rowspan="2">in</td><td>t - pulse_start_time ≥ pt</td><td>pt</td></tr>
<tr><td>t - pulse_start_time < pt</td><td>t - pulse_start_time</td></tr>
<tr><td colspan="2">¬in</td><td>0</td></tr>
</table>

Fig. 8: *pulse* timer specification in tabular expression

The above tables have their obvious equivalents in PVS. To make the timing behaviour of the *pulse* timer 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)* is true when the input predicate P holds true at least a period of time of *duration*. Predicate *Held_For_ts(P, duration, ts)* is a more restricted version of *Held_For*, insisting that the starting time of *duration* is *ts*.

The value of our approach lies in that we make our assumptions explicit 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 and still stays true, so the value of et remains as pt without further increasing.

As there is no implementation supplied in IEC 61131-3 for the *pulse* timer, there are no proofs of correctness or equivalence to be discharged. Nonetheless, obtaining a precise, complete, and disjoint requirement above for the *pulse* timer is itself valuable for any future concrete implementations.

5.2 Ambiguous Behaviour: Implicit Delay Unit. PLC applications often specify feedback loops: the output of a FB is connected as the input either of another FB, or of itself. IEC 61131-3 specifies paths of feedback loops either through a connecting line, or through adopting the same input and output name. However, in real systems computations of feedback values (or of intermediate output values) cannot occur instantaneously.

We address this issue by introducing a delay block Z_{-1} and its formalization showed in Figure 9:

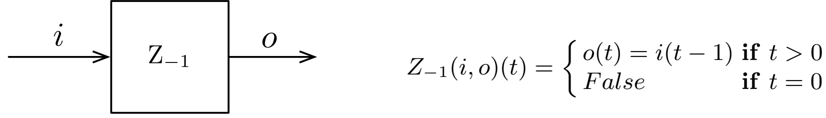


Fig. 9: Definition on delay unit

There is an explicit delay (i.e. one tick of clock) between the input and output of block Z_{-1}. Consequently, we can use block Z_{-1} to explicitly denote feedback values as output values that are produced in the previous execution cycle. The type of i and o can be any defined type, e.g. boolean type in the following example, but have to be the same type.

To illustrate the use of block Z_{-1}, we consider the block *sr* that creates a set-dominant latch (a.k.a. flip-flop) in Figure 10. Block *sr* takes as inputs a boolean set flag s_1 and a boolean reset flag r, and it returns a boolean output q_1. In fact, 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. Obviously, 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 by taking the conjunction of the predicates for the internal blocks: Blocks B_1 (formalized by predicate *neg*), B_2 (*conj*), B_3

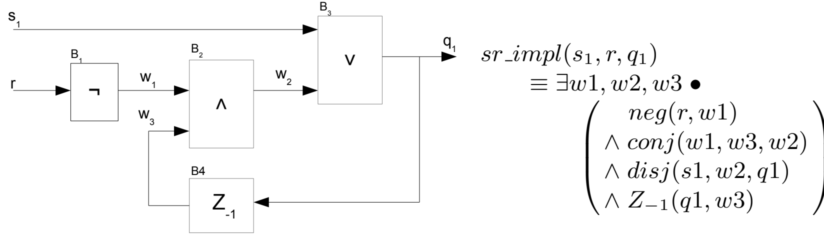


Fig. 10: Block *sr* implementation in FBD and its formalizing predicate

(*disj*), and B_4 (Z_{-1}) in the above FBD denote the FB of, respectively, logical negation, conjunction, disjunction, and delay. Arrows w_1, w_2, w_3 are internal connectives.

Adding an explicit block Z_{-1} of delay to formalize feedback loops led us to discharge the correctness theorem (Section 4) of the above FBD implementation. More precisely, the following theorems are discharged in PVS, in which *sr_fbd_impl*, *sr_req* denote FBD implementation and tabular requirement of block *sr*:

$$\vdash \forall s_1, r \bullet \forall q_1 \bullet sr_fbd_impl(s_1, r, q_1) \Rightarrow sr_req(s_1, r, q_1) \tag{6}$$

$$\vdash \forall s_1, r \bullet \exists q_1 \bullet sr_fbd_impl(s_1, r, q_1) \tag{7}$$

Our tabular requirement for *sr* block is specified in Figure 11:

Condition		Result: q
s1		True
¬s1	r	False
	¬r	NC

Fig. 11: Tabular requirement for block *sr*

5.3 Missing Assumption: Limit on Counter FBs. IEC 61131-3 describes three types of counters. An up-down counter (*ctud*) 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 $qd \equiv (cv <= 0)$.

The ST implementation and graphical declaration are provided in the standard in Figure 12:

```
       +--------+
       |  CUTD  |
       |        |
BOOL --|cu    qu|-- BOOL
BOOL --|cd    qd|-- BOOL
BOOL --|r       |
BOOL --|ld      |
 INT --|pv    cv|-- INT
       +--------+
```

```
IF R THEN CV := 0 ;
ELSIF LD THEN CV := PV ;
ELSE
  IF NOT (CU AND CD) THEN
    IF CU AND (CV < PVmax)
    THEN CV := CV + 1;
    ELSIF CD AND (CV > PVmin)
    THEN CV := CV - 1;
    END IF;
  END IF;
END IF;
QU := (CV >= PV) ;
QD := (CV <= 0) ;
```

Fig. 12: Block *ctud* declaration and ST implementation

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 $cv > 10$, and is 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]: $\boxed{PVmin < PVmax}$.

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

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

true, as the counter decrements when $cv > PVmin$. As a result, we introduce an assumption: $\boxed{PVmin < \ pv < PVmax}$.

<table>
<tr><th colspan="4">Condition</th><th>Result
cv</th></tr>
<tr><td colspan="4">r</td><td>0</td></tr>
<tr><td rowspan="7">$\neg$r</td><td colspan="3">ld</td><td>pv</td></tr>
<tr><td rowspan="6">$\neg$ld</td><td colspan="2">cu $\wedge$ cd</td><td>NC</td></tr>
<tr><td rowspan="2">cu$\wedge\neg$cd</td><td>$cv_{-1} <$ PVmax</td><td>cv_{-1}+1</td></tr>
<tr><td>$cv_{-1} \geq$ PVmax</td><td>NC</td></tr>
<tr><td rowspan="2">$\neg$cu$\wedge$cd</td><td>$cv_{-1} >$ PVmin</td><td>cv_{-1}-1</td></tr>
<tr><td>$cv_{-1} \leq$ PVmin</td><td>NC</td></tr>
<tr><td colspan="2">$\neg$cu $\wedge$ $\neg$cd</td><td>NC</td></tr>
</table>

assume: $PVmin < pv < PVmax$

Fig. 13: Tabular requirement for block *CTUD*

Our tabular requirement for the up-down counter that incorporates the missing assumption is shown above in Figure 13. Similarly, we added $pv < PVmax$ and $PVmin < \ pv$ as assumptions for, respectively, the up and down counters.

5.4 Missing Assumption: Hysteresis Value. We revisit the tabular requirement of block *limits_alarm* (Section 3.2) that implements a high/low limit alarm with the hysteresis effect. Two internal blocks of *limits_alarm* are *high_alarm* and *low_alarm*. We introduced an assumption $eps > 0$ (i.e. positive hysteresis epsilon value) to ensure that the two hysteresis zones $[l, l + eps]$ and $[h - eps, h]$ are positive and are computed at the right directions. Moreover, we reckon that the intention of having both high and low limits is to have two disjoint hysteresis zones. Otherwise, the high and low alarms may be tripped on simultaneously, and this is reflected by PVS as a failure to discharge the type correctness constraint of disjointness of the tabular requirement. As a result, we introduced another assumption: $h - eps > l + eps$, or equivalently $h - l > 2eps$.

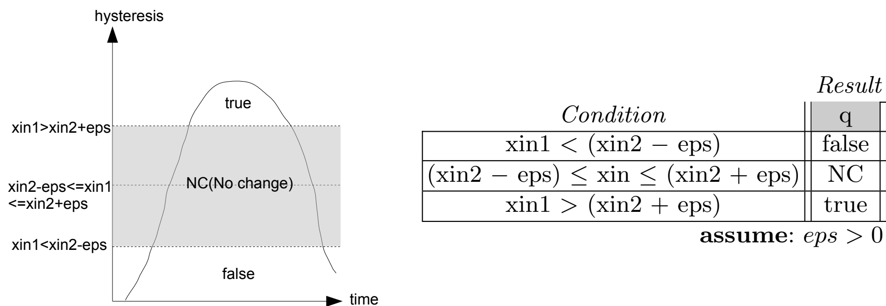
With our tabular requirement that incorporates these two assumptions, we proved that the ST implementation supplied by IEC 61131-3 is both correct and feasible (Section 4). The verification process involved the predicates for 5 predefined functions and FBs, 3 lemmas for implementation correctness, 1 theorem for implementation feasibility, 1 theorem for implementation correctness, and about 140 PVS proof commands. The verified *limits_alarm* block can thus be safely reused to construct more complex FBs.

Similarly, another example of function block *hysteresis* showed in Figure 14 needs the same assumption on *eps*, i.e. $eps > 0$. This block implements boolean hysteresis on the difference of two real inputs. Declaration and ST implementation are showed above. It takes as three inputs, two reals *xin1*, *xin2* and hysteresis epsilon value *eps* and one output *q* which indicates boolean hysteresis effect of the difference of *xin1*, *xin2*. The behaviour of block *hysteresis* is illustrated as following together with our tabular expression which has incorporated with the missing assumption in Figure 15. The shaded area is used to present hysteresis effect. When $xin1 < xin2 - eps$, output q becomes false and $xin1 > xin2 + eps$ then q becomes *true*. While *xin1* is in between $xin2 - eps$ and $xin2 + eps$, the value of q is left unchanged by setting it to its previous value.

```
      (* DECLARATION *)
       +------------+
       | HYSTERESIS |
       |            |                 FUNCTION BLOCK HYSTERESIS
REAL --|xin1       q|-- BOOL          VAR INPUT XIN1, XIN2, EPS : REAL;
REAL   |xin2        |                 END VAR
REAL --|eps         |                 VAR OUTPUT Q : BOOL := 0;
       |            |                 END VAR
       +------------+                 IF Q THEN IF XIN1 < (XIN2 - EPS) THEN Q := 0;
                                      END IF ;
INPUTS:                               ELSIF XIN1 > (XIN2 + EPS) THEN Q := 1;
xin1  : real input1                   END IF ;
xin2  : real input2                   END FUNCTION BLOCK
eps: hysteresis

OUTPUTS:
q : hysteresis indicator
```

Fig. 14: Block *hysteresis* declaration and ST implementation



Condition	*Result* q
xin1 $<$ (xin2 $-$ eps)	false
(xin2 $-$ eps) $\leq$ xin $\leq$ (xin2 $+$ eps)	NC
xin1 $>$ (xin2 $+$ eps)	true

assume: $eps > 0$

Fig. 15: Block *hysteresis* behaviour in tabular expression

Tabular expression without the last line of assumption is the one translated directly from the standard. It can not be correctly typechecked in PVS. An unprovable disjointness TCC is generated by PVS prover as follows:

```
% Disjointness TCC generated (at line 54, column 7) for
% TABLE
%-------------------------------------------------------------------+-------++
% | xin1(t) < (xin2(t) - eps_no(t))                                 | FALSE ||
%-------------------------------------------------------------------+-------++
% |(xin2(t) - eps_no(t)) <= xin1(t) & xin1(t) <= (xin2(t) + eps_no(t))| prev  ||
%-------------------------------------------------------------------+-------++
% | (xin2(t) + eps_no(t)) < xin1(t)                                 | TRUE  ||
%-------------------------------------------------------------------+-------++
% ENDTABLE
% unfinished

HYSTERESIS_tab_without_assumption_TCC1: OBLIGATION
  FORALL (xin1: [tick[delta_t] -> real], xin2: [tick[delta_t] -> real],
          eps_no: [tick[delta_t] -> real], q: pred[tick[delta_t]],
          t: tick[delta_t], t1):
    NOT init(t1) IMPLIES
     (FORALL (prev: bool):
        prev = q(pre(t1)) IMPLIES
         NOT (xin1(t1) < (xin2(t1) - eps_no(t1)) AND
               (xin2(t1) - eps_no(t1)) <= xin1(t1) &
                xin1(t1) <= (xin2(t1) + eps_no(t1)))
          AND
          NOT (xin1(t1) < (xin2(t1) - eps_no(t1)) AND
                (xin2(t1) + eps_no(t1)) < xin1(t1))
           AND
           NOT (((xin2(t1) - eps_no(t1)) <= xin1(t1) &
                  xin1(t1) <= (xin2(t1) + eps_no(t1)))
                 AND (xin2(t1) + eps_no(t1)) < xin1(t1)));
```

Thus, this function table is not well-defined because first and third row can be satisfied at the same time but return different values. To address this problem, we explicitly constraint the type of *eps* from any real numbers to positive real numbers, i.e. from `[tick -> real]` to `[tick -> posreal]` to discharge the above TCC. Such missing assumption can be potential danger propagating from requirement to detailed implementation.

5.5 Error: Failed Implication Proof of Implementations. IEC 61131-3 supplies both ST and FBD implementations for block *stack_int* (a stack of integers). It takes as inputs *push*, *pop*, *r1*, *in*, *n* and three outputs, *out*, *empty* and *oflo*. The explanation for each is as follows:

It may perform operations of push(set by *push*), pop(set by *pop*), or reset(set by *r1*), subject to some limit on its depth(set by *n*). It outputs an integer value(*out*), depending upon which operation was performing, and Boolean values reflecting if the current stack is empty(denoted by *empty*) or has an overflow(denoted by *oflo*). The data pushed into stack is from *in*. The maximum stack depth is determined at the time of resetting.

The ST and FBD implementations are not given at the same abstraction level, i.e. ST implementation is an abstraction of FBD implementation. Hence, we should be able to prove the implication from FBD to ST implementation.

```
       +-----------+
       | STACK_INT |         INPUTS:                  OUTPUTS:
       |           |         push  : push operation  empty  : stack
BOOL --|push  empty|-- BOOL  pop   : pop operation            is empty
BOOL --|pop    oflo|-- BOOL  r1    : reset           oflo   : stack
BOOL --|r1      out|-- INT   in    : input data               is overflow
 INT --|in         |         n     : depth of stack  out    : top of stack
 INT --|n          |
       |           |
       +-----------+
```

Fig. 16: Block *stack_int* declaration

$$\vdash \forall\ push, pop, r1, in, n\ \bullet\ \forall\ out, empty, oflo\ \bullet$$
$$stack_int_fbd_impl(push, pop, r1, in, n, out, empty, oflo) \quad (8)$$
$$\Rightarrow stack_int_st_impl(push, pop, r1, in, n, out, empty, oflo)$$

We provide the *push_stk* part of its implementations in ST in Figure 17 and FBD in Figure 18 below which is relevant to this undischarged proof sequent. There are four kinds internal FBs, move(:=), addition(+), substraction(−) and boolean selection(*sel*). Block move updates output by input, block addition returns the result of additions of two inputs, block subtraction returns the result of subtraction of two inputs and block selection outputs one of two inputs depending on a selection input. Note both enable input (*en*) and output (*eno*) are used in each FB. The FB is enabled if en is hold and functions as specified, otherwise, the output remains unchanged and eno is set to *false*.

```
 ...(* omit the others parts of this implementation*)
ELSIF  PUSH  &  NOT   OFLO  THEN
      EMPTY := 0; PTR := PTR + 1; OFLO := (PTR = NI);
     IF NOT OFLO THEN  OUT := IN;  STK[PTR] := IN;
      ELSE OUT := 0;
      END_IF;
  END_IF;
```

Fig. 17: *push_stk* part of implementation of block *stack_int* in ST

However, we failed to prove this functional implication specified above in (8). The following unprovable sequent in PVS makes the proof of (8) incomplete. As introduced in Section 2, this proof sequent can be completed if the disjunction of consequents can be implies by the conjunction of antecedents. The variables ending in "!1" are skolem constants (i.e. arbitrary constant of corresponding type) that are used to eliminate quantifiers. By the definition of time (Section 2), tick `t!2` is the number `n!2` tick on the time axis. Internal stack pointer is `ptr!1`. `inp!1` is the value pushed into stack which is set by input `push!1`. Line `{-1,-2,-3}` is the definition of `t!2`. Line `{-4}` indicates input `push!1` is hold. The stack is overflow at tick `t!2` which is showed in line `{1}`. Line `{2}`

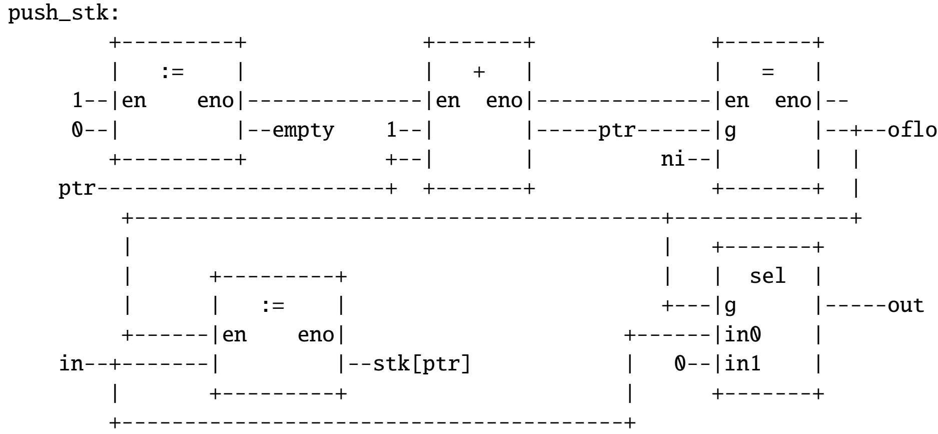


Fig. 18: *push_stk* part of implementation of block *stack_int* in FBD

shows that the value pointed by `ptr!1` is equal to the value pushed in by `inp!1` at tick `t2!1`. The following sequent states that, at tick `t!2`, if a value `inp!1` is pushed into stack by setting `push!1`, either the stack is overflow denoted by `oflo!1` or the top of stack is updated by `inp!1`. This is not the case expressed in ST implementation, i.e. the top of stack is updated by `inp!1` happens when the stack is not overflow.

```
STACK_INT_fbd_implies_st.3.14 :
{-1}  n!2 >= 0
{-2}  n!2 * delta_t >= 0
{-3}  t!2 = n!2 * delta_t
{-4}  push!1(n!2 * delta_t)
  |-------
{1}   oflo!1(n!2 * delta_t)
{2}   stk!1(ptr!1(n!2 * delta_t)) = inp!1(n!2 * delta_t)
```

By inspecting the FBD implementation, we found that there was a missing negation FB to specify that normal actions proceed only the current stack is *not* in an overflow state (i.e. opposite to the formula in Line `{1}`). By correcting this found error, we proved that ST implementation can be logically implied from FBD implementation. In order to have the same notations as used in standard, we add a small circle to indicate where to insert this negation FB. This circle (negation) is inserted between *oflo* and the enable input of bottom block move. After correcting, the logical implication from FBD to ST implementation can be proved, i.e. the above proof sequent can be discharged.

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

automata, and to verify real-time properties in Uppaal [20]; to automatically transform SFC programs into the synchronous data flow language of LUSTRE, amenable to mechanized support for checking properties [16]; to transform FBD specifications to Uppaal formal models to verify safety applications in the industrial automation domain [28]; 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) [7]; 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 [22]. There is also an interesting integration of both model checker Cadence SMV and the real-time model checker Uppaal to handle, respectively, untimed and timed SFC programs [4]. 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 [5]; and to formalize PLC programs using higher-order logic and to discharge safety properties in HOL [29]. 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 [27]. In [19], a trace function method (TFM) based approach is presented to solve the same problem as ours where tabular expression are used to document internal variables.

Our work is inspired by [21] developed in the context of hardware verification. The similarity is that the overall system behaviour is defined by taking the conjunction of those of internal components (circuits [21] or FBs in our case). Also, our resolutions to the timing issues of the *pulse* timer are consistent with those provided in [15]. The novelty of our approach lies in that we also obtain high-level, input-output tabular requirements to be checked against, instead of writing properties directly in the language of the chosen theorem prover or model checker. Our formalism is precise yet easy to understand compared with other approaches and more importantly, disjointness and completeness properties can be guaranteed.

7 Conclusion and Future Work

We presented an approach to formalizing and verifying function blocks using both tabular expressions and PVS. Using our approach, we identified issues concerning ambiguity, missing assumptions, and erroneous implementations in the IEC 61131-3 standard of function blocks. 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. IEEE 7-4.3.2: Standard for Digital Computers in Safety Systems of Nuclear Power Generating Stations (Revision of IEEE Std 7-4.3.2-2003), The Institute of Electrical and Electronics Engineers (IEEE) (2010)
2. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Special Committee 205 of RTCA (2011)
3. 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

4. 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)
5. Blech, J.O., Biha, S.O.: On Formal Reasoning on the Semantics of PLC using Coq. CoRR abs/1301.3047 (2013)
6. Camilleri, A., Gordon, M., Melham, T.: Hardware verification using higher-order logic. Tech. Rep. UCAM-CL-TR-91, Cambridge Univ. Computer Lab (1986)
7. 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)
8. Eles, C., Lawford, M.: A tabular expression toolbox for Matlab/Simulink. In: NASA Formal Methods. pp. 494–499 (2011)
9. Hu, X., Lawford, M., Wassyng, A.: Formal verification of the implementability of timing requirements. In: FMICS. LNCS, vol. 5596, pp. 119–134. Springer (2009)
10. IEC: 61131-3 Ed. 2.0 en:2003: Programmable Controllers — Part 3: Programming Languages. International Electrotechnical Commission (2003)
11. IEC: 61131-3 Ed. 3.0 en:2013: Programmable Controllers — Part 3: Programming Languages. International Electrotechnical Commission (2013)
12. Janicki, R., Parnas, D.L., Zucker, J.: Tabular representations in relational documents. In: in Relational Methods in Computer Science. pp. 184–196. Springer Verlag (1997)
13. Janicki, R., Wassyng, A.: Tabular expressions and their relational semantics. Fundam. Inf. 67(4), 343–370 (2005)
14. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Science of Computer Programming 75(11), 980 – 1000 (2010)
15. 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)
16. Kabra, A., Bhattacharjee, A., Karmakar, G., Wakankar, A.: Formalization of sequential function chart as synchronous model in LUSTRE. In: NCETACS. pp. 115–120 (2012)
17. Lawford, M., Froebel, P., Moum, G.: Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. FMSD (2004)
18. Lawford, M., Wu, H.: Verification of real-time control software using PVS. In: Ramadge, P., Verdu, S. (eds.) Proceedings of the 2000 Conference on Information Sciences and Systems. vol. 2, pp. TP1–13–TP1–17. Dept. of Electrical Engineering, Princeton University, Princeton, NJ (Mar 2000)
19. Liu, Z., Parnas, D., Widemann, B.: Documenting and verifying systems assembled from components. Frontiers of Computer Science in China 4(2), 151–161 (2010), `http://dx.doi.org/10.1007/s11704-010-0026-2`
20. Mader, A., Wupper, H.: Timed automaton models for simple programmable logic controllers. In: ECRTS. pp. 114–122. IEEE (1999)
21. Melham, T.: Abstraction mechanisms for hardware verification. In: VLSI Specification, Verification and Synthesis. pp. 129–157. Kluwer Academic Publishers (1987)
22. Németh, E., Bartha, T.: Formal verification of safety functions by reinterpretation of functional block based specifications. In: FMICS, pp. 199–214. Springer (2009)
23. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: CADE. LNCS, vol. 607, pp. 748–752 (1992)
24. Parnas, D.L.: A generalized control structure and its formal definition. Commun. ACM 26, 572–581 (August 1983), `http://doi.acm.org/10.1145/358161.358168`
25. Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming 25(1), 41–61 (1995)
26. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Transactions on Software Engineering 20, 948–976 (1994)

27. Roussel, J.M., Faure, J.: An algebraic approach for PLC programs verification. In: 6th International Workshop on Discrete Event Systems. pp. 303–308 (2002)
28. 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)
29. Völker, N., Krämer, B.J.: Automated verification of function block-based industrial control systems. Science of Computer Programming 42(1), 101 – 113 (2002)
30. Wassyng, A., Janicki, R.: Tabular expressions in software engineering. In: Proceedings of ICSSEA'03. vol. 4, pp. 1–46. Paris, France (2003)
31. Wassyng, A., Lawford, M., Maibaum, T.S.E.: Software certification experience in the canadian nuclear industry: Lessons for the future. In: EMSOFT. pp. 219–226 (2011)