Science of Computer Programming 113 (2015) 149–190

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Formal verification of function blocks applied to IEC 61131-3

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

McMaster Centre for Software Certification, McMaster University, 1280 Main St. W, Hamilton, Ontario, L8S 4K1 Canada

ARTICLE INFO

Article history:
Received 1 May 2014
Received in revised form 3 October 2015
Accepted 5 October 2015
Available online 19 October 2015

Keywords:
Critical systems
Formal verification
Function blocks
Tabular expressions
IEC 61131-3

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: 1) to create complete and unambiguous FB requirements using tabular expressions; and 2) to verify the consistency and correctness of FB implementations in the PVS proof environment. We apply our approach to the examples in the informative Appendix F of the IEC 61131-3 standard. We examined the entire library of FBs and their supplied implementations described in structured text (ST) and function block diagrams (FBDs). Our approach identified issues in the informative examples, including: a) ambiguous behavioural descriptions; b) missing assumptions; and c) inconsistent implementations. We also proposed solutions to these issues.

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. Standards such as DO-178C [2] (in the aviation domain) and IEEE 7-4.3.2 [3] (in the nuclear domain) list acceptance criteria for mission- or safety-critical systems that practitioners need to comply with. Two important criteria are: 1) the system requirements are precise and complete; and 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. To this end, we present methods that support the use of formal notations for specifying the required behaviour of FBs, and for verifying that each FB (including composed FBs) complies with its requirements.

Tabular expressions [4,5] (a.k.a., function tables or tables) have proven to be both practical and effective in formally documenting system requirements in industry [6,7]. PVS [8] is a general purpose theorem prover that 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

* Corresponding author.
E-mail addresses: pangl@mcmaster.ca (L. Pang), wangcw@mcmaster.ca (C.-W. Wang), lawford@mcmaster.ca (M. Lawford), wassyng@mcmaster.ca (A. Wassyng).

http://dx.doi.org/10.1016/j.scico.2015.10.005
0167-6423/

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 attempted to verify the FBs[1] listed in "Informative" Annex F of the 2003 version of IEC 61131-3 [9] as well as the FBs described in the standard itself. IEC 61131-3 is an important standard with over 20 years of use on critical systems running on PLCs. We had two reasons for choosing IEC 61131-3 for our case study. First of all, this provided a number of FBs that represent useful behaviours in a number of application domains, so our methods could be applied to FBs that we knew were representative of industrial use. Secondly, although the FBs of Annex F are not technically part of the standard as indicated by the labelled "Informative", the entire document, including all annexes, has become the *de facto* standard for FBs. PLC vendors have based their libraries on the FBs from Annex F, as well as those described in the body of the standard.

The standard itself does not make any claim as to the completeness and appropriateness of the behaviour of FBs. In addition, no one has published a "validated and verified" version of the FBs in the standard. Thus, companies that develop mission-critical or safety-critical systems using PLCs had to qualify the behaviour of their libraries based on IEC 61131-3 (including Annex F), at considerable cost. If practitioners can use pre-defined and pre-verified FBs, then this will help raise the quality of FB-based implementations in industry without the overhead that would be required if each practitioner had to perform the verification separately.

Currently, some of the design specifications in the standard (expressed in source code) are incorrect, in that they are not what is commonly expected in practice. We believe that formal requirements of the FB behaviour, such as those provided by tabular expressions, help tool vendors and users of FBs have the same interpretations of the expected system behaviours. Also, formal descriptions are amenable to mechanized support such as PVS to verify the conformance of candidate implementations to a high-level, input-output requirements. For the purpose of this paper, we focus on FBs that are described in the more commonly used languages of structured texts (STs) and function block diagrams (FBDs). Note that two versions of IEC 61131-3 are cited here. The earlier version [9] has been in use since 2003. Most of the work reported in this paper relates to this version.

As we will see, a number of issues were uncovered in the FBs in the standard and in its informative Annex F. Our intent in the proceeding discussion is to illustrate how our methodology raised questions about some of these FBs. It is not a direct criticism of the standard, since the original mandate of the standard did not include the presentation of a pre-validated and pre-verified FB library. In fact, the standard does not attempt to define the required behaviour of each FB at the semantic level that we would expect from a requirements specification. Instead it uses code. These source programs are operational descriptions, making it hard to identify unexpected behaviour, and they are thus at an inappropriate level of abstraction for specifying requirements. Consequently, we had to provide the high-level requirements specifications based on our experience and on what we deduced was the intended behaviour of the FB. Readers may not agree with our version of the required behaviour, but we did make an honest attempt to define the behaviour that would be consistent with industrial norms. In any case, our motivation here is to demonstrate our methods, not to criticize the standard. We hope that readers will be interested that the methodology highlighted potential problems with FBs that have been in use for many years, and that this type of methodology can help us improve the quality of FB-based designs. In 2013, a new version of the standard was issued [10], and this version did not include Annex F. Some of the FBs in the new version do still exhibit behaviour that we believe could be improved through use of this methodology.

Our approach and contributions

We now summarize our approach and contributions with reference to Fig. 1. As shown on the left of the figure, an FB 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 created a black box tabular requirements specification in PVS for the behaviour of the FB (Section 3.2). The ST and FBD implementations are formalized as predicates in PVS, again making use of tables (Section 3.1). In the case when there are two implementations for an FB, one in an FBD and the other in ST, we attempt to prove their (functional) equivalence in PVS (Section 4.2). We also use PVS to attempt to prove the *consistency*[2] and *correctness* of each implementation with respect to its FB requirements (Section 4.1).

Using our approach, we identified a number of possible issues that warrant users' attention: ambiguous behaviour (Section 5.1); possible missing input assumptions (Section 5.2); and inconsistent implementations (Section 5.3). We compare our approach and results with other related work in Section 6, and end up with some concluding remarks in Section 7.

This paper extends [11] by including the following new contributions:

- We provide a complete example which illustrates the modelling of FB requirements in PVS (Section 3.2).
- A revised list of ST-to-PVS translation rules (Section 3.1.4) is included, sufficient to handle all the implementations included in IEC 61131-3 [9] and its Annex F. Constructs that are supported by the ST language but not used in Annex F of the standard [9] (e.g., `CASE` statement, `WHILE` and `REPEAT` loops, *etc.*) are not covered in our list of rules. Nonetheless,

[1] PVS files are available at http://www.cas.mcmaster.ca/~lawford/papers/SCP2014. All formalization and proofs are conducted using PVS 6.0.

[2] In this paper, we overload the term consistency in two contexts. Two implementations are *consistent* if they exhibit the same input-output behaviour. An implementation is *consistent* (or feasible) if for any legitimate input, it produces an expected output. However, the context should be clear when we use the term.

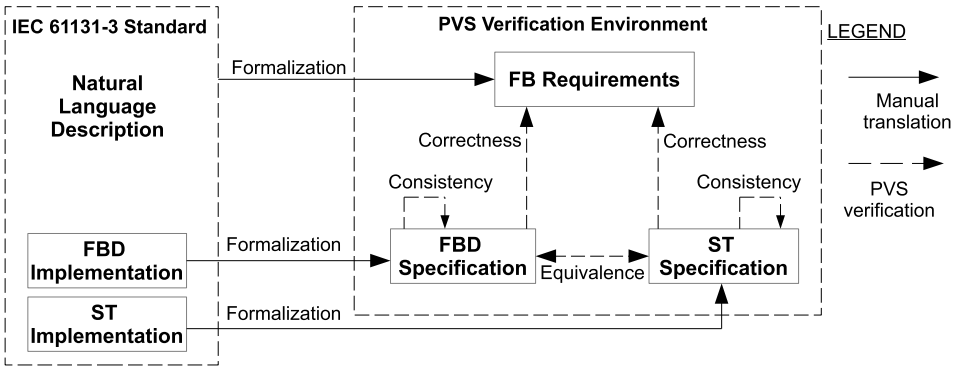


Fig. 1. Framework.

the value of our translation is justified by the fact that the Annex F example function blocks are commonly used in industry.

- We extend the discussion on the *SR* block by supplying the exact definitions of: 1) what we consider should be the black-box input-output requirements table; 2) its theory of consistency; and 3) its theory of correctness (Section 5.1.2).
- We completed the verification of all FBs listed in IEC 61131-3 and found more blocks that warrant discussion: *HYSTERESIS* (Section 5.2.2), *LIMITS_ALARM* (Section 5.2.3), *DELAY* (Section 5.2.4), *AVERAGE* (Section 5.2.5), *PID* (Section 5.2.6), *DIFFEQ* (Section 5.2.7), and *STACK_INT* (Section 5.3.1). We present tabular requirements for these blocks and propose solutions[3] for the potential issues we uncovered using this methodology.

In the next section we discuss background material: the IEC 61131-3 Standard, tabular expressions, and PVS.

2. Preliminaries

2.1. IEC 61131-3 standard for function blocks

Programmable logic controllers (PLCs) are digital computers that are widely utilized in real-time and embedded control systems. In an effort to unify the syntax and semantics of programming languages for PLCs, the International Electrotechnical Committee (IEC) first published IEC 61131-3 in 1993, and later revisions in 2003 [9] and 2013 [10]. Most of our research results were completed before the third edition was released.

We applied our methodology to the standard functions and also to the FBs listed in Annex F of IEC 61131-3 (2003). 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.

Each FB is fed by input values, performs computations on them according to the behaviour specified in either ST or FBD (or both[4]), and produces output values. We focus on two programming languages that are covered in IEC 61131-3 for writing behavioural descriptions of FBs: STs and FBDs. These two languages are widely used in PLC-based control systems. The ST notation is a high level textural programming language which resembles another high-level programming language, Pascal. FBDs are a graphical programming notation. The fundamental concept behind FBDs is the inter-connections among block components, which specify the data flow dependency.

However, we found that in some cases for the same FB, its ST and FBD implementations (supplied in IEC 61131-3 2003 and its Annex F) cannot be mapped from one to the other, and thus cannot be proved to be equivalent; instead, we prove that one implementation conforms to the other (but not vice versa). For example, the FBD implementation supplied for *STACK_INT* block (discussed in Section 5.3.1) has an explicit execution order for its component FBs, and such specificity is not required in the ST implementation for the same block.

As an example of FBD implementation, we consider the *LIMITS_ALARM* block (Annex F.6.7) that will be used as a running example for later sections. The FBD of the *LIMITS_ALARM* block (Fig. 2) consists of two parts: 1) declaration of inputs and outputs; and 2) definition of the computation of its body. An alarm monitors the quantity of some input variable X, subject to a low limit L and a high limit H, with a hysteresis band of size *EPS*.

There are five component blocks of *LIMITS_ALARM*: addition ($+$), subtraction ($-$), division ($/$), logical disjunction (≥ 1), and two instances of hysteresis (i.e., *HIGH_ALARM* and *LOW_ALARM*). The internal connectives w_1, w_2 and w_3 are used to

[3] We deliberately choose the term *solution*, as opposed to *resolution*, in that we only propose possible solutions for the potential issues, and we do not intend to claim that they are the only solutions.

[4] In the main text of the standard, each basic FB is defined in one single language. In the case of composite FBs, several languages can be used as the component FBs may be described using different programming languages.

```
    (* DECLARATION *)
       +----------+
       | LIMITS_  |
       | ALARM    |
 REAL--|H       QH|--BOOL
 REAL--|X        Q|--BOOL
 REAL--|L       QL|--BOOL
 REAL--|EPS       |
       +----------+
 FUNCTION_BLOCK LIMITS_ALARM
   VAR_INPUT
     H   : REAL; (* High limit     *)
     X   : REAL; (* Variable value *)
     L   : REAL; (* Lower limit    *)
     EPS : REAL; (* Hysteresis     *)
   END_VAR
   VAR_OUTPUT
     QH : BOOL; (* High flag    *)
     Q  : BOOL; (* Alarm output *)
     QL : BOOL; (* Low flag     *)
   END_VAR
 END_FUNCTION_BLOCK
```

```
                (* Function block body in FBD language *)
                                    HIGH_ALARM
                                  +------------+
                                  | HYSTERESIS |
X-------------------------+--|XIN1       Q|--+----------QH
                  +---+ w2|  |              |  |
H-----------------| - |------|XIN2          |  |
             +---|   |   |  |              |  |
             |   +---+   |  |              |  |
             +-----------+--|EPS           |  |  +------+
       +----+w1|           |  +------------+  +--| >=1  |
EPS --| / |--|            |                     |      |--Q
2.0 --|   |  |            |    LOW_ALARM     +--|      |
       +---+  |            |  +------------+  |  +------+
              |   +---+ w3|  | HYSTERESIS |  |
L ------------| + |------|XIN1       Q|--+-----------QL
              |   |   |   |  |              |
              +---|   |   +--|XIN2          |
              |   +---+      |              |
              +--------------|EPS           |
                             +------------+
```

Fig. 2. Declaration of the block *LIMITS_ALARM* and its FBD implementation [9].

Condition		Result F
C_1	$C_{1.1}$	RES_1
	$C_{1.2}$	RES_2
	...	...
	$C_{1.m}$	RES_m
...		...
C_n		RES_n

IF C_1
  IF $C_{1.1}$ **THEN** F = RES_1
  ELSEIF $C_{1.2}$ **THEN** F = RES_2
  ...
  ELSEIF $C_{1.m}$ **THEN** F = RES_m
ELSEIF ...
ELSEIF C_n **THEN** F = RES_n

Fig. 3. Semantics of horizontal condition table (HCT).

connect these internal blocks. The body definition visualizes how the ultimate and intermediate outputs are computed using two instances of the *HYSTERESIS* block. For example, the output *QL* is computed by manipulating the two output values *Q* from the top and bottom *HYSTERESIS* block:

$$LIMITS\_ALARM(H, X, L, EPS).Q = HYSTERESIS(X, H - \tfrac{EPS}{2.0}, \tfrac{EPS}{2.0}).Q \vee HYSTERESIS(L + \tfrac{EPS}{2.0}, X, \tfrac{EPS}{2.0}).Q$$

where we write .*Q* to denote the output value resulting from the FB invocation in question.

Roadmap for the running example. We specify our interpretation of the precise input-output requirement of the *LIMITS_ALARM* block using tabular expressions (Section 3.2). To verify its FBD implementation, we first formalize it in PVS (Section 3.1.5), then we verify its consistency and correctness (Section 4.1) with respect to the tabular requirement. Furthermore, we report any potential issues uncovered regarding this block (Section 5.2.3).

2.2. Tabular expressions

Tabular expressions [12,13,4,5] are a proven and effective approach to describing conditionals and relations, and they are thus ideal for documenting many system requirements. They are arguably easier to comprehend and to maintain than conventional mathematical expressions. Reference [14] presents a relational semantics for tabular expressions which covers the most common types of tabular expressions used in software practice. Recently, reference [15] presented a new semantics for tabular expressions by using indexing to decouple the appearance of a tabular expression from its semantics. Tabular expressions have also been proven to be of great help both in inspections [7] and in testing and verification [16].

For our purpose of capturing the input-output requirements of function blocks in IEC 61131-3, tabular expressions of the form shown in Fig. 3 are appropriate. These tabular expressions are called *horizontal condition tables* (HCTs). The input domain is partitioned into condition rows in the left column(s), while rows in the right column(s), inside double borders, denote the corresponding output results. Rows in the input columns may be divided to specify sub-conditions. We may interpret the tabular structure in Fig. 3 as a list of "if–then–else" statements, without the sequence implications of the "if–then–else" construct. This is shown in the right part of the figure. 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 HCTs as illustrated in Fig. 3, we need to reason about their *completeness* and *disjointness*. Completeness ensures that there is an output specified for every combination of inputs – the rows cover

all input combinations, i.e., if we suppose that there are no sub-conditions, ($C_1 \vee C_2 \vee \cdots \vee C_n \equiv TRUE$). Disjointness ensures that the rows do not overlap, e.g., ($i \neq j \Rightarrow \neg(C_i \wedge C_j)$, $i, j \in \{1, 2, \ldots, n\}$). Similar constraints apply to the sub-conditions, if any.

2.3. PVS language and prover

The Prototype Verification System (PVS) [8] was developed by the Computer Science Lab at SRI International as an interactive environment for writing specifications and conducting proofs. PVS consists of a specification language, predefined theories, a parser, a type checker, a theorem prover which supports several decision procedures, a symbolic model checker, pre-developed libraries, and utilities and documentation with examples in different application areas.

The PVS specification language is based on classical, typed higher-order logic. The base types include uninterpreted types and built-in types such as the Booleans. The type-constructors include functions, sets, tuples, records, enumerations, and inductively-defined (or coinductively-defined) abstract data types. In addition, users can adopt predicate subtypes and dependent types to introduce constraints to greatly increase the expressiveness and naturalness of specifications. But the expense is that these constrained types may generate proof obligations called *Type Correctness Conditions* (TCCs) during type-checking. In many cases, these generated TCCs can be discharged automatically by the theorem prover. PVS specifications are organized into theories that may include imported theorems, assumptions, definitions, axioms, lemmas, and goal theorems. Furthermore, the theories can be parameterized with constants, types, and theory instances. Definitions are conservative, e.g., subtype TCC generated with dependent types and termination TCC generated with recursive function definitions. PVS expressions support the arithmetic and logical operators, function application, lambda abstraction, and quantifiers, within a natural syntax. Tabular expressions are also provided with automated checks for disjointness and completeness. A prelude is included in PVS to provide over 1000 useful definitions and lemmas. The NASA PVS Library is also a collection of formal developments contributed and maintained by the NASA Langley Formal Methods Team [17].

The built-in theorem prover provides a collection of powerful proof commands to conduct propositional and quantifier rules, equality, and arithmetic formal reasoning under user guidance. Proof commands can be combined to form higher-level proof strategies. The PVS specification language is designed to work with the prover so that the inference mechanisms exploit the type information of a defined term and most of the generated TCCs are automatically discharged by the prover. To facilitate debugging of proofs, the PVS proof checker allows any proof step to be undone. It also permits modification of specification over the course of a proof. Proof scripts can be edited and rerun to support proof maintenance, allowing many similar theorems to be proved efficiently and adjusted economically.

We chose the PVS theorem prover to formalize the input-output requirements of function blocks primarily because it supports the syntax and semantics of tables (Section 2.3.1). In particular, for each table that is syntactically valid, PVS automatically generates its associated healthiness conditions of completeness and disjointness as TCCs. We have expertise built from past experience in applying PVS to check requirements and designs in the nuclear domain [6] that gave us confidence in using the toolset, and for modelling real-time behaviour we reused parts of the PVS theories from [18,19]. Our ongoing work on proving properties of real-time function blocks that consider timing tolerances also relies on the same set of theories. Nonetheless, the techniques presented in this paper are transferable to other theorem provers that support reasoning in higher-order logic, although checks of completeness and disjointness may then have to be manually encoded or a generator for the properties would have to be developed.

The function blocks in Annex F of IEC 61131-3 [9] involve only simple expressions using linear integer or real arithmetic and in our experience, when these constraints are provable, the table-related TCCs generated were typically automatically discharged by PVS's built-in default strategies. Alternatively, these table correctness conditions can be automatically discharged by an SMT solver, using the solver's theories for linear integer and real arithmetic. However, such verification may not be as convenient as in PVS, since one will need to manually encode these constraints for each table in the SMT solver, unless an existing tool that supports automated generation of correctness conditions (e.g., [20]) is chosen to create the tables. Further, when handling additional user-defined or library blocks that involve nonlinear arithmetic, the table's correctness conditions may be undecidable by an SMT solver. In this case, the PVS environment allows us to interactively prove these conditions.

2.3.1. Support for function tables in PVS

The PVS specification language provides two alternative built-in constructs for specifying function tables: *COND* and *TABLE*. They are semantically equivalent to a series of *IF–THEN–ELSE–ENDIF* statements. The use of *COND* and *TABLE* causes PVS to generate the proof obligations on disjointness and completeness to guarantee that the function table is well-defined. These can often be discharged automatically using the built-in proof strategies in PVS, i.e., (*COND–COVERAGE–TCC*) and (*COND–DISJOINT–TCC*). When the table cannot be automatically proved as well-defined, some useful feedback is returned. However, for readability, it is more advisable for users to adopt the *TABLE* construct, which will be translated into the equivalent *COND* construct in PVS for typechecking and proofs. Later in this paper (Section 5.2.2), we will discuss an issue in which the ST implementation supplied by IEC 61131-3 is formalized as a PVS table but the table fails the proof on the TCC of disjointness. The syntactic constructs that we use the most are *IF–THEN–ELSE–ENDIF* predicates and tables. An example of using tabular expressions to specify and verify the Darlington Nuclear Shutdown System (SDS) in PVS can be found in [6].

```
x: VAR int

f_cond(x) : bool =
 COND x >= 0 -> TRUE,
      x < 0  -> FALSE
 ENDCOND

f_table(x) : bool =
 TABLE | x >= 0 | TRUE ||
       | x < 0  | FALSE ||
 ENDTABLE
```

```
% Disjointness TCC generated (at line 15, column 2) for
   % COND x >= 0 -> TRUE, x < 0 -> FALSE ENDCOND
   % proved - complete
f_cond_TCC1: OBLIGATION
 FORALL (x: int): NOT (x >= 0 AND x < 0);

% Coverage TCC generated (at line 15, column 2) for
   % COND x >= 0 -> TRUE, x < 0 -> FALSE ENDCOND
   % proved - complete
f_cond_TCC2: OBLIGATION
 FORALL (x: int): x >= 0 OR x < 0;
```

Fig. 4. Function tables and their TCCs in PVS.

```
x: VAR real

g(x) : real = 1/x
```

```
% Subtype TCC generated (at line 108, column 17) for x
   % expected type nznum
   % unfinished
g_TCC1: OBLIGATION FORALL (x: real): x/ = 0;
```

Fig. 5. Expressions and well-definedness TCCs in PVS.

2.3.2. Type correctness conditions

We briefly review failed TCCs that we encountered in our verification process. PVS automatically generates TCCs as proof obligations, which often can be automatically discharged, if they are provable, using the default proof strategies. However, in cases where they are too complicated to be discharged automatically, human interaction is required to guide the prover. Unproven TCCs often help users reveal issues (e.g., incompleteness, non-disjointness, ill-definedness, etc.) that can be traced back to the original specifications. One may choose to continue other proofs for the same specification while bypassing unproven TCCs, but until all TCCs have been discharged, a specification is not considered as type-correct, and lemmas and theorems that depend on theses unproven TCCs are considered provisional.

PVS checks the completeness and disjointness properties for a function table (Section 2.2) by automatically generating two types of TCCs: (*COND–COVERAGE–TCC*) for coverage (i.e., completeness) and (*COND–DISJOINT–TCC*) for disjointness.

As an example, consider a simple Boolean function $f(x)$ with an integer parameter x:

$$f(x) = \begin{cases} TRUE & \text{if } x \geq 0 \\ FALSE & \text{if } x < 0 \end{cases}$$

In PVS, function f can be specified as a function table using either the *COND* construct or the *TABLE* construct as shown on the Left Hand Side (LHS) of Fig. 4. The associated TCCs[5] of (*COND–COVERAGE–TCC*) and (*COND–DISJOINT–TCC*) are automatically generated by PVS – see the Right Hand Side (RHS) of Fig. 4.

Since constraints can be imposed on the types in a PVS specification, subtype TCCs are generated for expressions whose types are defined using the predicate subtype notation (e.g., positive real numbers *posreal*). It makes very explicit and intuitive statements about the domains and ranges of functions, thereby contributing to the clarity of the PVS specification. The price paid is that it requires theorem proving to prove that expressions satisfy the constraints attached to types. Consider a general PVS function F, which is defined as:

$$F : \{ x : T \mid P(x) \} \rightarrow \{ y : T' \mid Q(y) \}$$

with the domain type constrained by predicate P and range type constrained by predicate Q. Whenever a function f is invoked, subtype TCCs are generated to ensure that, the output has to satisfy predicate constraint Q and the input has to satisfy predicate constraint P. Division is a particular instance of this problem.

As an example, consider a function $g(x)$ with a real parameter x:

$$g(x) = 1/x$$

To model g in PVS, we use the built-in division operator (LHS in Fig. 5). For g to be well-defined, all expressions involved in its definition must be well-defined, i.e., the denominator x must be non-zero. Such a well-definedness constraint is formulated automatically by PVS as a TCC (RHS in Fig. 5).

There are other categories of TCCs that are automatically generated in PVS: existence TCCs and termination TCCs. Existence TCCs are generated for expressions whose types are declared as non-empty. Termination TCCs are generated to ensure that recursive functions always terminate for all possible inputs by requiring a well-founded *measure* to strictly decrease on each recursive calls. More precisely, recursive functions must be specified with a *measure*, that is a function whose signature

[5] We show only the generated TCCs for function `f_COND`, as the same TCCs are generated for `f_TABLE`.

matches that of the recursive function, but with range type the domain of the order relation, which defaults to $<$ on *nat* or *ordinal*.

2.3.3. Proofs in PVS

PVS has a powerful interactive proof checker to perform sequent-style deductions. The basic structure of the underlying calculus in PVS is a sequent [21]. 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, $i = 1, 2, \ldots, m$ are antecedent formulas, Q_j, $j = 1, 2, \ldots, n$ are consequent formulas, and $\vdash$ denotes entailment. Where the context is empty (i.e., no antecedent), $\vdash$ may be dropped. The antecedents are combined by conjunctives while consequents are connected by disjunctives. Thus, the above PVS sequent is equivalent to the following expression in predicate logic[6]:

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

The final goal of a PVS sequent is to determine whether at least one of its consequents is a logical consequence of its antecedents. In an editor panel of the PVS prover, a sequent is displayed as follows:

```
{-1} P1
 ... ...
{-m} Pm
  |-------
 {1} Q1
 ... ...
 {n} Qn
```

A sequent can be discharged only if one of the following three cases applies: 1) *FALSE* occurs in the antecedents; 2) *TRUE* occurs in the consequents; or 3) the formula P occurs in both the antecedents and the consequents [19]. A PVS sequent may be discharged by splitting it into sub-goals and by proving all of these sub-goals. The prover maintains a proof tree, and the final goal is to discharge each of its leaves by invoking relevant proof commands.

In practice, it is useful to decompose a complex problem into smaller ones, and to formulate and prove each of these sub-problems as a lemma. For example, to verify the overall correctness of the *LIMITS_ALARM* block (Section 4.1), we formulate the correctness conditions of its three output variables (i.e., *QL*, *Q*, and *QH*) as separate lemmas and conduct independent proofs.

Our justification for decomposing requirements by their outputs is two-fold. First, output variables in our proposed requirements tables do not depend upon each other. As discussed in Section 2.4, our requirements model describes idealized behaviour with an arbitrarily small clock-tick. At each discrete time instant, outputs are produced simultaneously as the inputs are updated. This means that the value of each output of a function block depends solely on those of the inputs. Second, all input-output requirements tables that we propose are completely functional. This claim is supported by the fact that all our proposed function tables are provably complete and disjoint, meaning that at any time instant, exactly one value can be produced for each output. Consequently, it is always possible to separate the definition of an output by projecting onto its relevant range of values. For example, consider any input types I_1 and I_2 and output types O_1 and O_2, then we declare

$$\begin{aligned} REQ &: I_1 \times I_2 \to O_1 \times O_2 \\ req_1 &: I_1 \times I_2 \to O_1 \\ req_2 &: I_1 \times I_2 \to O_2 \end{aligned}$$

where *REQ* represents the overall requirements function, and req_1 and req_2 represent, respectively, the requirements for the first output and second output, then for input values $i_1 \in I_1$ and $i_2 \in I_2$, we have

$$\begin{aligned} req_1(i_1, i_2) &= \pi_1 \circ REQ(i_1, i_2) \\ req_1(i_1, i_2) &= \pi_2 \circ REQ(i_1, i_2) \end{aligned}$$

where π_1 and π_2 are operators for, respectively, the first projection and second projection. This example can be generalized to arbitrary numbers of inputs and outputs.

Consequently, for each output, we are able to: 1) specify a separate function table that characterizes its relationship with the inputs; and 2) prove its correctness separately.

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

2.4. Modelling time in PVS

As PLCs are widely used in real-time systems, the modelling of time is a critical aspect in our formalization. We consider a discrete-time model, where a time series consists of equally distributed time samplings, or "ticks". More precisely:

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

where $\delta \in \mathbb{R}^+$ is small enough to represent the time interval between two consecutive clock ticks. This kind of definition of *tick* is reproduced by [18] from [22]. It represents the type TIME in IEC 61131-3. In the real world, the sampling frequency is usually different from the clock tick frequency, i.e., the clock tick frequency should be significantly larger than the sampling frequency. In the software domain, all the actions occurring at the sampling times can be captured at the corresponding clock ticks. To approximate the continuous time model, the value of δ may be arbitrarily small.

As a result, we define a *Time* theory in PVS:

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

The 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 non-negative multiples of *delta_t*. We will perform operations on *tick* [18]: e.g., *init* (the very first tick) and *pre*(*t*) (the tick preceding *t*, given that *init*(*t*) does not hold).

We define a characteristic predicate *init* which is *TRUE* only at the initial tick t_0:

```
init(t : tick) : bool = (t = 0)
```

It is important to explicitly identify the initial values of internal or output variables of FBs in PLC-based control system.

Given a time instant *t*, we use *rank*(*t*) to denote the ordinal of *t* in a discrete time setting.

```
rank(t : tick) : nat = t / delta_t
```

For example, time instant 8.8 is the 4th tick given that $delta\_t = 2.2$.

However, we choose to adopt the notion of real-valued ticks, rather than their corresponding integer ranks, for specifying function blocks (and their properties) as they more closely correspond to the sampling times in reality. In other words, the notion of ticks is more meaningful for the user to manipulate: e.g., for timer blocks, an output that denotes the elapsed time should be measured in real-valued units rather than integer ranks. However, given some fixed *delta_t*, the set of real-valued ticks is isomorphic to its set of integer ranks. Consequently, proving lemmas or theorems in both domains is equally complex.

As PVS requires that all functions are total, to define the *pre* operator, we need a subtype *noninit_elem* that denotes the set of ticks starting from t_1 (i.e., excluding t_0):

```
noninit_elem : TYPE = { t : tick | NOT init(t) }
```

Using *noninit_elem*, the *pre* operator is defined as follows:

```
pre(t : noninit_elem) : tick = t − delta_t
```

An important yet simple proposition we use in our model to prove some desired properties is an induction scheme over time ticks [19]. It states that a predicate *P* holds at all ticks if (a) *P* holds at the initial tick t_0; and (b) for any $t > 0$, the fact that *P* holds at tick t_{n-1} implies that *P* holds at tick t_n. The formalization of this induction scheme is as follows:

```
time_induction : PROPOSITION
 FORALL (P : pred[tick]) :
   (FORALL (t : tick) : init(t) ⇒ P(t)) ∧ (FORALL (t : noninit_elem) : P(pre(t)) ⇒ P(t))
    ⇒ (FORALL (t : tick) : P(t))
```

We consider most FBs listed in IEC 61131-3 as time-dependent. Each FB is formalized as a theory in PVS, parameterized by the constant time interval *delta_t* and by importing our timing theory presented in this section.

3. Formalizing standard functions and function blocks using tabular expressions

We need to tailor our approach depending on the language(s) used to describe the required behaviour and the implemented behaviour of the FBs. In this case we have tailored our approach to deal with the languages used in IEC 61131-3. In many cases, IEC 61131-3 uses both ST and FBD to describe a single function block. However, both ST and FBD are informal, implementation-oriented notations, and they are thus not suitable for capturing a 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), since the required behaviour of the FBs in the standard and in Annex F is defined using natural language or not defined at all. We present a formal approach to define IEC 61131-3 standard functions and function blocks using tabular expressions and PVS. For each function block, 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 the level of standard functions, basic function blocks (FBs), and composite FBs. Similar to [23], we formulate each standard function or function block as a predicate, characterizing its input-output relation.

3.1.1. 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 relation (i.e., Boolean function or predicate):

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

where the symbol $\triangleq$ denotes that function f is formalized using relation (or predicate) R. Predicate R represents the specification of function f with input vector i and output vector o, by characterizing the precise relation on the m inputs and the n outputs of function f. Our formalization covers both timed and untimed behaviours of standard functions.

As an example, consider function *WEIGH* (Annex F.1), which takes as inputs a gross weight *gross_weight* (a word encoding in Binary-Coded Decimal (BCD)) and a tare weight *tare_weight* (an integer), and returns the net weight *net_weight* (a BCD-encoded word). The standard supplies a one-line ST code program for the implementation of *WEIGH*: `WEIGHT := INT_TO_BCD(BCD_TO_INT(gross_weight) - tare_weight)`, where `INT_TO_BCD` and `BCD_TO_INT` are standard conversion functions [9, p. 55]. We formalize the ST description of *WEIGH* in PVS by defining the output *net_weight* as

$$net\_weight = INT\_TO\_BCD(\ SUB(\ BCD\_TO\_INT(gross\_weight),\ tare\_weight\ )\ ),$$

where *INT_TO_BCD* and *BCD_TO_INT* are PVS functions, whose names are deliberately chosen to match those in the standard, that formalize the corresponding conversions and *SUB* is the standard subtraction function. We use bit vectors supported by PVS to model words, and follow the standard rules of performing conversions between BCD-encoded words and integers. However, as our modelling is performed at the level of requirements, we do not consider implementation issues such as arithmetic overflows. Therefore, unless the input or output values are explicitly bounded like in the case of *WEIGH*, we use mathematical, unbounded integers or reals to model input and output values.

Nonetheless, as we stated earlier in the paper, since the focus of the standard is primarily on the notations used to describe the FB implementations, the standard does not include precise descriptions of the required behaviour of each FB. So as a demonstration of our approach, based on our experience and whatever we can deduce from the standard itself, we propose a formal requirements specification for the FB. More precisely, in the example above, we make the requirements of *WEIGH* and explicitly define the inputs domain. Of course, readers may disagree with our requirements specification and may have another in mind. This is quite usual in practice. The essential point here is that the requirements behaviour needs to be precise, and we did not make up these requirements behaviours simply to generate discrepancies between the requirements and implementations.

To complete this section we also discuss another standard function *ADD* (i.e., "+"). This function is stateless, and it may be used as an internal component of other FBs, such as *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 predicate parameters makes it possible to formalize basic FBs with internal states, or composite FBs. The predicate formalizing *ADD* can be reused to produce more complex 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: $ADD(IN_1, IN_2 : int) : int = IN_1 + IN_2$. The functional formalization of *ADD* is used to discharge a consistency proof using instantiation, if an *ADD* block is one of the internal components.

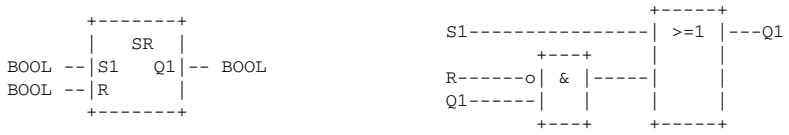


Fig. 6. Declaration of the block *SR* and its FBD implementation [9].

3.1.2. 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.

As an example in Fig. 6., consider the *SR* block, which implements a set-dominant latch (a.k.a., flip-flop). Block *SR* 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* itself. The value of Q_1 remains *TRUE* as long as the set flag S_1 is enabled. Q_1 is reset to *FALSE* not only when the reset flag is enabled, but also when the set flag is disabled (so it cannot dominate the output result). Otherwise, Q_1 stays unchanged. There should be a delay between the value of Q_1 which is computed and passed to the next execution cycle. We formalize this by adding the explicit unit delay block z^{-1} and conjoining predicates for the internal blocks. The formalization of the delay block will be introduced in Section 5.1.2. IEC 61131-3 uses a circle (e.g., the upper input to conjunction block in Fig. 6) to negate the value of Boolean input signal. We explicitly replace such circle with a negation block wherever it occurs.

We formalize composite FBs in a similar manner.

3.1.3. Composite function blocks

Each composite FB contains as components standard functions, basic FBs, or other pre-developed composite FBs. For example, *LIMITS_ALARM* (Fig. 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 ST or FBD, or both in the case that component FBs are described using different languages, to describe the behaviour of composite FBs. At this point we should note that 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.

3.1.4. Formalizing composite FB implementations: ST

As discussed in Section 2.1, in general it is not possible to translate an arbitrary ST implementation into its equivalent FBD implementation. Instead, for the purpose of our verification in PVS, we develop a limited set of translation rules that suffices to translate the ST implementations that are supplied by Annex F of IEC 61131-3 [9] into their equivalent expressions in PVS. This step of formalization in PVS allows us to verify the correctness of ST implementations against their input-output requirements (Section 3.2).

In this section, we discuss our ST-to-PVS translation in four phases: 1) state the challenge, scope of translation, and input assumptions; 2) provide an overview of translation; 3) provide a list of formal rules of translations; and 4) illustrate our translation rules via a number of examples.

ST-to-PVS: challenge, scope of translation, and input assumptions

The main challenge of using PVS to formalize ST is that these two languages belong to two distinct paradigms. The ST programming language is an imperative notation, whereas the PVS specification language is a functional notation. For example, an `IF-THEN-ELSE` statement in ST is meant to perform conditional updates on the state (i.e., output or local variables), whereas an *IF–THEN–ELSE* expression in PVS is side-effect-free and returns a value (corresponding to the satisfying branch condition).

Nonetheless, our ultimate goal is to use *only* function blocks that are listed in the Annex of the standard [9] to illustrate our proposed approach. Consequently, our intention is not to formalize any arbitrary ST code whose syntax conforms with the standard. Instead, for the purpose of our verification, our rules of *ST-to-PVS* translation are designed to only handle the syntactic constructs of the ST language that are exploited in Annex F. That is, constructs that are supported by the ST language but not used in the Annex of the standard [9] (e.g., `CASE` statement, `WHILE` and `REPEAT` loops, etc.) Nonetheless, the value of our translation should be justified by the fact that the Annex F example function blocks are commonly used in industry. In other words, our translation rules should be able to handle many other similar function blocks outside the scope of Annex F [9].

For our *ST-to-PVS* translation, there are two primary assumptions about the input ST code. Both of the following assumptions are satisfied by all function blocks listed in Annex F [9].

```
FUNCTION_BLOCK F
  VAR_INPUT
    v1 : T1
  END_VAR
  VAR_OUTPUT
    v2 : T2
  END_VAR
  VAR
    v3 : T2
  END_VAR
  v3 := f(p1 := e1, p2 := e2);
  IF e5 THEN
    v2 := v3;
  ELSEIF e6 THEN
    G(p3 := e3, p4 := e4);
    v2 := G.Q;
  END_IF;
END_FUNCTION_BLOCK
```

```
F [(IMPORTING Time) delta_t: posreal] : THEORY
BEGIN IMPORTING ClockTick[delta_t]
  v1 : VAR [tick -> [[T1]]]
  v2 : VAR [tick -> [[T2]]]
  F_st_impl (v1, v2): bool =
    EXISTS (v3: [tick -> [[T2]]], Q : [tick -> [[T2]]]):
      FORALL (t: tick):
        ( NOT [[e5]] AND [[e6]] => G([[e3]], [[e4]], Q) ) IMPLIES
          IF init(t) THEN TRUE
          ELSE
            v3(t) = f([[e1]], [[e2]], v3) AND
            v2(t) = TABLE
                    | [[e5]]                    | v3(t)       ||
                    | NOT [[e5]] AND [[e6]]     | Q(t)        ||
                    | NOT [[e5]] AND NOT [[e6]] | v2(pre(t)) ||
                    ENDTABLE
          ENDIF
END F
```

Fig. 7. *ST-to-PVS* translation: a contrived example.

- *Type correctness*. Each ST code is assumed to be type-correct: e.g., no references to unknown function blocks in variable declarations, no references to undeclared variables, no references to unknown formal parameters of a function in its invocation, etc. The PVS type system may be exploited to type-check the ST code, because if the source ST code is not type-correct, then neither will its corresponding formalized PVS theory. However, for the purpose of tracing type errors in the original code, if any, adopting a third-party ST programming tool is more appropriate.
- *Single assignment*. Each output or local variable in the body of the ST code gets assigned at most once. This will allow us to formalize each sequential composition operator (;) in ST as a logical conjunction ($\wedge$) in PVS. As far as the formalization of function blocks in Annex F [9] is concerned, this assumption is always met. However, to relax this assumption, we will need to introduce a mechanism of building the dependency graph of variable assignments and, when it is acyclic, introduce auxiliary variables on the PVS side to impose the topological order.

Given the above assumptions, and the richness of the specification language and supported libraries of PVS, our *ST-to-PVS* translation is reasonably straightforward. Our translation rules shown below, although presented in a formal way, are still meant as guidance for users who want to translate the ST code manually into PVS. To adapt them for automation, some further context-sensitive analysis needs to be performed beforehand. Extension to the full coverage of ST syntax, or to the automation of these rules, is outside the scope of this paper.

ST-to-PVS: an overview

Our strategy of translation is to map each complete ST program (i.e., with variable declarations and function block body) into a PVS theory. More precisely, we map (unconditional, conditional, or iterative) variable assignments into PVS predicates (Boolean functions) that encode the intended state effect as variable constraints. Let $[\![\_]\!] : ST \rightharpoonup PVS$ denote our translation function that maps ST code to PVS expressions. Since we do not intend to handle the full ST syntax, the translation function is declared as partial.

An example translation Fig. 7 presents an overview of our *ST-to-PVS* translation. On the LHS of Fig. 7 we have the complete definition of a function block named F, declared with an input v_1 (of type T_1), an output v_2 (of type T_2). There is also a local variable v_3 whose type is declared to match that of the output v_2. We assume that: 1) a standard function f is declared with parameters p_1 and p_2 and a return value of type T_2; and 2) a function block G is declared with parameters p_3 and p_4 and an output value of type T_2; 3) types of expressions e_1, e_2, e_3, and e_4 match those of, respectively, p_1, p_2, p_3, and p_4; and 4) e_5 and e_6 are Boolean expressions.

The body of function block F is defined as a sequential composition (denoted by a semicolon ;) of three programming statements: 1) assign to v_3 the return value of invoking the standard function f; 2) invoke the function block G; and 3) assign to v_2, depending on the values of e_5 and e_6. In both cases of invoking a standard function and a function block, the order in which argument values are passed is flexible: names of the formal parameters (e.g., p_1, p_2, etc.) are specify explicitly to bind those argument values. Moreover, there is a distinction between invocations of a standard function and of a function block: the former is an expression (a R-value), whereas the latter is a statement whose output must be retrieved in a later statement (e.g., $G.Q$).

On the RHS of Fig. 7 we have a PVS theory[7] F that formalizes the function block F defined on the LHS. As our translation is recursive, we write $[\![T_1]\!]$, $[\![T_2]\!]$, $[\![e_1]\!]$, $[\![e_2]\!]$, etc. to denote the corresponding, equivalent PVS expressions. In the following, we summarize (part of) our translation strategy as exemplified in Fig. 7:

[7] Note that negation (*NOT*) binds the tightest. Conjunction (*AND*) binds tighter than implication (*IMPLIES* or $\Rightarrow$).

- For readability, we retain names of the function block and all its declared variables.
- The theory is always parameterized by an arbitrary clock tick interval *delta_t*, which is used to instantiate the imported timing theory (Section 2.4).
- We formalize all ST (input, output, and local) variables as time-dependent logical variables in PVS (i.e., functions with the *tick* domain). However, we treat the parameter types of standard functions and function blocks differently in PVS. We formalize ST function blocks as input-output relations whose parameters are time-dependent (i.e., function blocks constrain inputs and outputs over time). On the other hand, parameters of standard functions are "untimed" (i.e., they are simple values instead of functions). All ST input and output variables are translated into global variables in PVS, so that they are implicitly universally quantified. On the other hand, local variables and return values from function invocations are translated into dummy variables of an existential quantification, so that they are hidden inside the function block.
- The function block body is formalized as a relation (i.e., Boolean function) which constrains the list of input and output variables over all discrete time ticks. The name of the relation has the `_st_impl` suffix to indicate that it is translated from some ST code.
- We define the input-output relation using a logical implication.
  - The antecedent constrains output values of function block invocations, so that their output values can be referenced in the consequence. For each invocation that occurs in the context of some (nested) conditional branch, we guard it using an implication (e.g., the invocation of function block G is guarded by $\neg[\![e_5]\!] \wedge [\![e_6]\!]$). The guard (or the antecedent) may be used to prove that the input assumptions of the function block are satisfied upon its invocation. For example, in ST we may invoke the *HYSTERESIS* block under the condition $EPS > 0$, and we formalize it as the constraint $EPS > 0 \Rightarrow HYSTERESIS(\dots)$ in PVS.
  - In the consequence, as output and local variables may be initialized, we use a universal quantification (over discrete *tick* values) to distinguish cases of the initial tick and non-initial ticks. At the initial tick, we constrain the values of those output and local variables that are explicitly initialized in the ST code; if no variables are explicitly initialized, the constraint is *TRUE*. At non-initial ticks, we constrain the value of each output variable according to how it is updated in the ST code. For example, the value of v_2 at time t, where $\neg\, init(t)$, is equal to either: 1) the value of v_3 at time t if $[\![e_5]\!]$ holds; 2) the value of Q at time t if $\neg[\![e_5]\!] \wedge [\![e_6]\!]$ holds[8]; or 3) itself at the previous time tick if $\neg[\![e_5]\!] \wedge \neg[\![e_6]\!]$.

ST-to-PVS: formal rules of translation

In this section, we provide the list of translation rules that is sufficient for translating ST code supplied by Annex F [9] into PVS.

Notational convention For clarity, we typeset ST constructs in the code style (e.g., `a + b`), and PVS constructs in the math style (e.g., $a + b$). As our translation is recursive, when the translation of an ST construct (e.g., `If-THEN-ELSE` statement) involves the translation of its components (e.g., branching conditions, body statements, etc.), say e, then we write $[\![e]\!]$ to denote the translated PVS expression for e. Moreover, as partly illustrated in Fig. 7, we adopt the following conventions: 1) e, e_1, e_2, etc., denote ST expressions; 2) v, v_1, v_2, etc., denote ST variables; 3) f, g, h, etc. denote standard functions; 4) F, G, H, etc. denote function block names; 5) T, T_1, T_2, etc. denote ST types; 6) S, S_1, S_2, etc., denote ST statements; and 7) i denotes a loop counter.

Translation context Our translation function $[\![\_]\!]$ often needs to carry around context information from the translation of one component to another. First, since all ST variables are mapped into time-dependent variables in PVS, when generating a reference to a variable v, we need to determine either to refer to: 1) its entirety v as a timed sequence; 2) its value $v(0)$ at the initial tick; or 3) its value $v(t)$ at some non-initial tick t. Second, since for each output variable we need to infer its intended update as constraints, the current translation may need to know the target variable in order to make the corresponding inference. As a result, given that v is the target variable, and that $t \in \{\ init,\ ninit,\ seq\ \}$ is the context for variable references, we write $[\![\_]\!]^t_v$ to denote the corresponding translation. We drop the context when it is not necessary for the translation in question to proceed. As an example, we write $[\![\_]\!]^{seq}$ for translating the invocation of a function block, where its arguments are expected to be time-dependent (i.e., timed sequences). In this example, the target variable is irrelevant and is thus dropped.

Context-sensitive analysis To assist our translation, we often need to extract information from the ST code fragment under consideration. For example, given a statement (e.g., the function block body as a sequential composition), we may extract the list of function block invocations that it makes. Furthermore, for those invocations, we need to extract the exact conditions where they occur and guard them accordingly (e.g., see Fig. 7 where the invocation of function block G is properly guarded). As another example, we may calculate the *write* set of a given statement (i.e, the set of variables that appear at the RHS

[8] This branching condition is guaranteed by the fact that the ST `IF-THEN-ELSE` statement evaluates those conditions in order.

Table 1
ST-to-PVS: function block definition.

ST function block definition	PVS theory	Delegates

FUNCTION_BLOCK F
VAR_INPUT
v_1 : T_1
END_VAR
VAR_OUTPUT
v_2 : T_2
END_VAR
VAR
v_3 : T_3 := e
END_VAR
S_1
END_FUNCTION_BLOCK
	F [(**IMPORTING** *Time*) *delta_t*: *posreal*] : **THEORY**
	BEGIN IMPORTING *ClockTick*[*delta_t*]
	$[\![v_1 : T_1]\!]$
	$[\![v_2 : T_2]\!]$
	F_st_impl (v_1, v_2): bool =
	EXISTS ($[\![v_3 : T_3]\!]$, $[\![Q : T_Q]\!]$):
	$F([\![e_1]\!]^{seq}, \ldots, [\![e_n]\!]^{seq}, Q)$ **IMPLIES**
	FORALL (t: tick):
	IF init(t) **THEN** $v_3(0) = [\![e]\!]^{init}$
	ELSE
	$v_2(t) = [\![S_1]\!]_{v_2}$ **AND**
	$v_3(t) = [\![S_1]\!]_{v_3}$
	ENDIF
	END F
		Table 2
		Tables 7–6
		Tables 8–10

of assignments), so as to determine if a variable has already been written. Tasks of such kind are standard and we do not address them in detail.

Translation rule: function block definition Table 1 presents the translation rules for function block definitions. The definition of each function block consists of two parts: variable declarations and body definition (denoted as S_1). Without loss of generality, we consider the case where the function block declares one variable from each of the categories (i.e., input, output, and local).

As illustrated in Fig. 7, each function block defined in ST is mapped into a PVS theory that has a matching name, and instantiates our timing theory (Section 2.4) with an arbitrarily small, positive clock tick interval *delta_t*. We delegate the translation of each input or output declaration in ST to Table 2.

The ST function block body S_1 is mapped into the PVS relation *F_st_impl* that constrains values of its parameters: the list of inputs and outputs. Inside the definition of this relation, we use an existential quantification to hide: 1) the list of local variables (i.e., v_3); and 2) return values from function invocations. For 2), we use Q (of type T_Q) to denote the list of return values that are referenced in S_1, if any.

In the case of function block invocations, as discussed, we model each function block F as a relation (a Boolean function) on the lists of inputs ($i_1, i_2, \ldots, i_n$) and outputs (Q), and each input or output is time-dependent and thus modelled as a timed sequence. In the case where the computation of output values depends on those of local variables, as mentioned above, we translate the relevant local variables into dummy variables of the corresponding existential quantification (see Fig. 7). As a result, the translated argument values are expected to be timed sequences (i.e., $[\![e_1]\!]^{seq}, \ldots, [\![e_n]\!]^{seq}$). As illustrated in Fig. 7, we use matching names[9] to capture values of outputs, so that in the same scope of context, these output variables may be referenced to define the constraints at both initial and non-initial time ticks. In the case where an invocation occurs within some (nested) conditional branch, we need to add an antecedent accordingly to guard the invocation. Inferring the exact antecedent to guard each invocation is an example of the context-sensitive analysis mentioned above, and we omit its details here.

In the case of the initial time tick, we constrain values of variables according to their specified initial values, if specified.[10] For example, v_3 is initialized with the value $[\![e]\!]^{init}$. In the case of non-initial ticks, each declared local or output variable will trigger the generation of a constraint that encodes its intended update. When there are multiple output variables, we combine all these constraints using logical conjunctions. For example, for output variable v_2, we generate its constraint of intended update via $[\![S_1]\!]_{v_2}$ (see Tables 7–6). So when given an ST statement S_1, our translation function effectively "projects" S_1 onto the target variable (e.g., v_2). The result of the projection is a list of "guarded values", where guards correspond to the branching conditions of the `IF-THEN-ELSE` statements in the source ST code. The resulting list of guarded values can then be straightforwardly encoded as a *TABLE* expression in PVS. For example, as already seen in Fig. 7, the projection onto output variable v_2 results in three guarded values.

Translation rule: variable declarations Table 2 presents the translation rules for variable declarations, where we reuse all variable names in PVS. Our treatment of the declarations of input (declared under `VAR_INPUT ... END_VAR`), output (declared under `VAR_OUTPUT ... END_VAR`), and local variables (declared under `VAR ... END_VAR`) are the same.

[9] Where multiple function blocks (e.g., *FB*1 and *FB*2) have outputs with the same name (e.g., Q), we resolve the ambiguity by adding their names as prefixes (i.e., *FB1_Q* and *FB1_Q*).

[10] We may choose to specify an initial value for some uninitialized variable, but this is beyond the scope of the translation.

Table 2
ST-to-PVS: variable declarations.

ST variable declaration	PVS variable declaration	Delegates
Case without initialization		Table 3
v : T	v : **VAR** [*tick* -> $[\![T]\!]$]	
Case with initialization		
v : T := e	v : **VAR** [*tick* -> $[\![T]\!]$]	

Table 3
ST-to-PVS: types.

ST type	PVS type	Delegates
INT	*int*	Tables 8–10
REAL	*real*	
BOOL	*bool*	
WORD	*bvec*	
TIME	*tick*	
F	F	
ARRAY[e_1 .. e_2] OF T	**ARRAY**[**subrange**($[\![e_1]\!]^{init}$, $[\![e_2]\!]^{init}$) -> $[\![T]\!]$]	

Table 4
ST-to-PVS: basic statements.

ST statement	PVS expression	Delegates
Assignments	With context variable v	Tables 8–10
v := e	$[\![e]\!]^{ninit}$	
x := e	$\perp$	
v[e_1] := e_2	$v(pre(t))$ WITH [($[\![e_1]\!]^{ninit}$) := $[\![e_2]\!]^{ninit}$]	
x[e_1] := e_2	$\perp$	

Since each ST variable is time-dependent in our execution context of function blocks, we parameterize the PVS type $[\![T]\!]$ (translated from the ST type t) by discrete time ticks (Section 2.4). At the level of variable declarations, the translation does not consider whether or not an initial value is specified in the source ST code. Instead, such information is considered at the higher level of function block definitions (Table 1), where the context *init* is passed for translating the specified initial value (i.e., $[\![e]\!]^{init}$).

Translation rule: types Table 3 presents the translation rules for types supported by ST. We categorize these types into four kinds: 1) primitive types (integers, reals, Booleans); 2) built-in types (e.g., words, time, etc.); 3) user-defined function blocks (e.g., F); and 4) arrays.

For primitive types, we can easily find the direct corresponding types in PVS. For built-in types, we import relevant theories to support their operations (e.g., bit vectors *bvec* from the *bv* prelude library, *tick* in Section 2.4, etc.). For a function block F that is user-defined, we simply reuse its name, assuming that its full definition is translated into a PVS theory.

The only structured type that we need for the purpose of Annex F [9] is that of arrays, which is also directly supported in PVS. The *ARRAY* type in PVS is essentially a function with a contiguous subset of integers for the domain and a proper range type, but the associated TCCs, e.g., validity of indices, are automatically generated by the prover. The operator *subrange* is supported by PVS to denote an integer range with specified lower and upper bounds. Presumably, e_1 and e_2 should be integer expressions, which is guaranteed by our assumption of input type-correctness. As the size of an array does not change at runtime, values of e_1 and e_2 must be available initially. As a result, we write $[\![e_1]\!]^{init}$ and $[\![e_2]\!]^{init}$ to denote the translated values in PVS.

Translation rule: statements Tables 4–7 present the translation rules for statements in ST (with no returned values). We assume that the context variable is v, meaning that $[\![\_]\!]_v$ is applied to infer the guarded values for v. We partition ST statements into two categories: 1) simple statements, including variable assignments and function block invocations; and 2) program combinators, including sequential compositions, IF-THEN-ELSE conditionals, and loops. Since all statements (including assignments) appear in the context of non-initial ticks, all involved expressions are translated via the invocation of $[\![\_]\!]^{ninit}$ (e.g., $[\![e_1]\!]^{ninit}$).

Table 4 presents the translation for variable assignments in ST. In both cases of assignments, we return a special value $\perp$ when the assignment target does not match the context variable v. A match in the case of a simple variable assignment is then straightforward: just return the translated value of the assignment source (i.e., $[\![e]\!]^{ninit}$). A match in the case of an array variable assignment returns an array that is identical to the original (i.e., $v(pre(t))$), except that the item at the specified index is updated. To specify this, we pass as arguments the translated values of array index (i.e., $[\![e_1]\!]^{ninit}$) and assignment source (i.e., $[\![e_2]\!]^{ninit}$) to the PVS function override operator *WITH*.

Table 5 presents the translation for the IF-THEN-ELSE conditional statement in ST. Indeed, the rule generalizes the case presented in Fig. 7, with a (possibly empty) list of *ELSIF* statements. If the context variable v is not written at all

Table 5
ST-to-PVS: conditional statements.

ST statement	PVS expression	Side condition
`IF` e_0 `THEN` S_0 `ELSIF` e_1 `THEN` S_1 ... `ELSIF` e_{n-1} `THEN` S_{n-1} `ELSE` S_n `END_IF`	**TABLE** | $[\![e_0]\!]^{ninit}$ | $[\![S_0]\!]_v^{ninit}$ || | **NOT** ($[\![e_0]\!]^{ninit}$) **AND** $[\![e_1]\!]^{ninit}$ | $[\![S_1]\!]_v^{ninit}$ || ... | **NOT** ($\bigwedge_{j=0}^{n-2}[\![e_j]\!]^{ninit}$) **AND** $[\![e_{n-1}]\!]^{ninit}$ | $[\![S_{n-1}]\!]_v^{ninit}$ || | **NOT** ($\bigwedge_{j=0}^{n-1}[\![e_j]\!]^{ninit}$) | $[\![S_n]\!]_v^{ninit}$ || **ENDTABLE**	$written\ (v)$
	$\bot$	$\neg written\ (v)$

where $written\ (v) \triangleq (\exists i \bullet v \in write\ (S_i))$

Table 6
ST-to-PVS: loop statements.

ST statement	PVS expression	Side condition

`FOR` i `:=` e_1 `TO` e_2
`DO`
S
`END_FOR`
	for[$[\![T]\!]$]($[\![e_1]\!]^{ninit}$, $[\![e_2]\!]^{ninit}$, $v\,(pre\,(t))$,
	LAMBDA (i: **subrange**($[\![e_1]\!]^{ninit}$, $[\![e_2]\!]^{ninit}$), v: $[\![T]\!]$):
	v = $[\![S]\!]_v$
	)
		$v \in write\ (S)$

	$\bot$	$v \notin write\ (S)$

Table 7
ST-to-PVS: sequential composition.

ST statement	PVS expression	Side condition
Sequential composition		
s_1 ; s_2	$[\![S_1]\!]_v$	$[\![S_2]\!]_v = \bot$
s_1 ; s_2	$[\![S_2]\!]_v$	$[\![S_1]\!]_v = \bot$

by any of the body statements S_i $(0 \le i \le n)$, then we return $\bot$. Otherwise, to correspond to the execution semantics of the ST `IF-THEN-ELSE` statement, each guard in PVS is defined as the conjunction of: 1) the translated value of the corresponding branching condition (e.g., $[\![e_1]\!]^{ninit}$); and 2) translated values of all branching conditions that are checked before it (e.g., $[\![e_0]\!]^{ninit}$). We use $\bigwedge$ as a meta-operator to denote the conjunction of a sequence of expressions occurring in PVS.

The resulting PVS table in Table 5 is a list of guarded values. If any of the body statements (S_0, S_1, ..., S_n) contain further nested `IF-THEN-ELSE` statements, then we will have nested table expressions in the resulting PVS, which are allowed. If the `ELSE` part is missing from the source ST code, then there is no change on the value of v. Accordingly, we specify $v(pre(t))$ as the return value in the PVS table: it is as if v were assigned to its value at the previous tick.

Table 6 presents the translation for the loop statement in ST. Similar to the case of translating the `IF-THEN-ELSE` statement, if the context variable v is not written by the loop body statement S, then we return $\bot$. Otherwise, with the use of the *for* higher-order function from the *structures* library provided by NASA [24, p. 114], encoding of an ST loop statement, with respect to the context variable v (of type T), is then straightforward.

We instantiate the *for* function by passing: 1) the translated type of v (i.e., $[\![T]\!]$); 2) the translated lower and upper bounds (i.e., $[\![e_1]\!]^{ninit}$ and $[\![e_2]\!]^{ninit}$); 3) the initial value of v, which is its value at the previous time tick (i.e., $v(pre(t))$); and 4) an anonymous lambda function which encodes the loop body. The lambda function encodes the loop body by taking the loop counter i (within the specified bounds) and the accumulated value of v, and by specifying that the value of v is constrained according to the list of guarded values inferred from S_1 (i.e., $[\![S]\!]_v$).

Table 7 presents the translation for the sequential composition of statements in ST. As discussed, when translating statements, we aim to retrieve the list of guarded values for the context variable v. Due to our single-assignment assumption, it is not allowed to have v assigned for more than once in the function block body. Consequently, when given a sequential composition of two statements S_1 and S_2, exactly one of them will return the list of guarded values for the context variable v.

Translation rule: expressions Tables 8–10 present the translation rules for ST expressions, where we use $\oplus_1$ to denote a unary (numerical, relational, or logical) operator, and $\oplus_2$ for a binary operator. As we have seen so far, each translation of an expression requires a context of variable reference (i.e., *init*, *ninit*, or *seq*). We use ρ to denote the value of this context. We consider four categories of expressions: 1) variable referencing; 2) literal expressions; 3) standard function invocations; and 4) operations.

Table 8
ST-to-PVS: variable referencing expressions.

ST expression	PVS expression	Side condition
Variable reference		
v	$v(0)$	$\rho = init$
	v	$\rho = seq$
	$v(t)$	$\rho = ninit$ and $written(v)$
	$v(pre(t))$	$\rho = ninit$ and $\neg written(v)$
Array indexing		
v `[` e `]`	$v(0)$[$[\![e]\!]^{init}$]	$\rho = init$
	v[$[\![e]\!]^{seq}$]	$\rho = seq$
	$v(t)$[$[\![e]\!]^{ninit}$]	$\rho = ninit$ and $written(v)$
	$v(pre(t))$[$[\![e]\!]^{ninit}$]	$\rho = ninit$ and $\neg written(v)$
Referencing function block output		
F `.` Q	$Q(0)$	$\rho = init$
	Q	$\rho = seq$
	$Q(t)$	$\rho = ninit$

Table 9
ST-to-PVS: literal and function invocation expressions.

ST expression	PVS expression	Side condition
Integer, real, or boolean literal		
l	l	$\rho \neq seq$
	LAMBDA $(t:\ tick):\ l$	$\rho = seq$
Standard function invocation		
f `(` p_1 `:=` e_1 `,` p_2 `:=` e_2 `, ...,` p_n `:=` e_n `)`	f `(` $[\![e_1]\!]^{\rho}$ `,` $[\![e_2]\!]^{\rho}$ `, ...,` $[\![e_n]\!]^{\rho}$ `)`	None

Table 10
ST-to-PVS: operation expressions.

ST expression	PVS expression	Side condition
Unary operation		
$\oplus_1\ e$	$\oplus_1\ [\![e]\!]^{\rho}$	None
Binary operation		
$e_1\ \oplus_2\ e_2$	$[\![e_1]\!]^{\rho}\ \oplus_2\ [\![e_2]\!]^{\rho}$	None

where $written(x) \triangleq x \in write(S)$ for any preceding statement S

In the case of variable referencing (Table 8), we may refer to a declared variable of a simple type or of an array type, or to an output variable of some function block that is invoked previously. The treatment of each kind of these variables is similar: depending on the given context ρ of the variable reference (*init*, *ninit*, or *seq*), we generate the references accordingly. In the case of array indexing, we propagate the variable reference context ρ to the translation of its specified index.

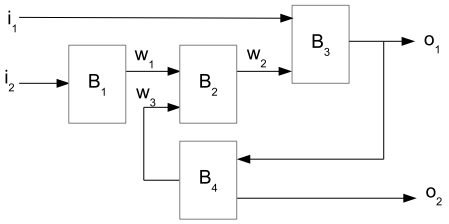
Furthermore, the sequential execution of the source ST code makes it possible to reuse the latest value of a variable that is assigned in a previous statement. To formalize this, we need to make a case distinction when the context ρ is *ninit*: if the variable v has not yet been written yet, then we write $v(pre(t))$ to denote its value from the previous time tick; otherwise, we should refer to its latest value at the current time tick (i.e., $v(t)$).

In the case of literal expressions (Table 9), integer literals (e.g., `2`), real literals (e.g., `2.0`), and Boolean literals (e.g., `TRUE`) can all be directly used in PVS. However, when the context of variable reference suggests that a timed sequence is expected (e.g., in the context of some function block invocation), then we use the lambda expression to create a constant timed sequence.

In the case of operations (Table 10), for all the unary and binary numerical expressions (e.g., `1 + 2`), relational expressions (e.g., `EPS > 0`), and logical expressions (e.g., `e1 & e2`), we can find the obvious corresponding operators in PVS. To translate the operands, we propagate the given context ρ (e.g., $[\![e]\!]^{\rho}$). The case of translating the invocation of a standard function f is also straightforward: pass the translated argument values in the order that is defined in the corresponding standard function definition in PVS.

ST-to-PVS: applications of translation rules

Our example translation in Fig. 7, though informative, is nevertheless contrived. We provide two complete example translations that are applied to the *HYSTERESIS* and *DELAY* function blocks from Annex F [9] in Appendix A. First, Fig. 37 shows the formalized ST implementation for the *HYSTERESIS* block (Fig. 18). This example illustrates the generation of nested PVS tables, mapped from the nested `IF-THEN-ELSE` statement in the source ST code. Second, Fig. 38 shows the formalized ST implementation for the *DELAY* block (Fig. 22). This example illustrates the use of a loop, and the list of generated "guarded values" for local and output variables in the context of non-initial ticks.



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

Fig. 8. A composite FB implementation in FBD and its formalizing predicate.

```
LIMITS_ALARM_IMPL (X, H, L, EPS, QH, Q, QL) : bool =
EXISTS (w1, w2, w3) :
    div(EPS, λ(t1 : tick) : 2.0, w1)
∧   sub(H(L, EPS), w1, w2)
∧   add(L, w1, w3)
∧   HYSTERESIS_REQ(X, w2, w1, QH)
∧   HYSTERESIS_REQ(w3, X, w1, QL)
∧   disj(QH, QL, Q)
```

Fig. 9. Formalizing FBD implementation of the block *LIMITS_ALARM* in PVS.

3.1.5. Formalizing composite FB implementations: FBD

To illustrate the case of formalizing an FBD implementation supplied by IEC 61131-3, let us consider the following FBD of a composite FB and its formalizing predicate. Fig. 8 consists of four internal blocks, B_1, B_2, B_3, and B_4, that are already formalized (i.e., their formalizing predicates $B_1\_REQ, \ldots, B_4\_REQ$ exist). The high-level requirement (as opposed to the implementation supplied by IEC 61131-3) for each internal FB constrains its inputs and outputs, documented by tabular expressions (see 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 inter-connectives (i.e., w_1, w_2 and w_3), such that the conjunction of predicates that formalize the internal components hold.

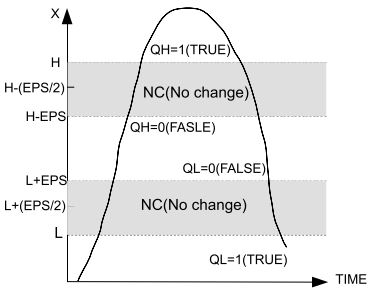
As a more concrete example, consider the FBD implementation of the *LIMITS_ALARM* block that was introduced in Section 2.1 (Fig. 2). In Fig. 9, the predicate *LIMITS_ALARM_IMPL* formalizes the FBD implementation of *LIMITS_ALARM* (Section 2.1). We observe that predicate *LIMITS_ALARM_IMPL*, as well as 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. Furthermore, 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 not included in 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 and/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 and disjoint. 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*. The expected input-output behaviour is depicted in the following Fig. 10, and its tabular requirement (which constrains the relation between inputs X, H, L, EPS and outputs Q, QH, QL) is captured in the three accompanying 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*. Q is set to *FALSE* otherwise. The input-output requirement is captured in the three function tables. We use *NC* to denote "No Change", i.e., the value of variable QH is equal to the value at the previous time tick QH_{-1}. Alternatively we can use the previous value in the condition rows. As a result, we can explicitly write down the current value of this variable in the last result column instead of using *NC* (see an example in Fig. 21).

We will discuss in Section 5.2.3 as to how our formalization process revealed the need for two possible missing assumptions for the *LIMITS_ALARM* block from IEC 61131-3:



Condition	*Result* Q
QH ∨ QL	1
¬(QH ∨ QL)	0

assume: $L + EPS < H - EPS$

Condition	*Result* QH
X > H	1
H - EPS ≤ X ≤ H	NC
X < H − EPS	0

assume: $EPS > 0$

Condition	*Result* QL
X < L	1
L ≤ X ≤ L + EPS	NC
X > L + EPS	0

assume: $EPS > 0$

Fig. 10. Requirement of the block *LIMITS_ALARM* using tabular expressions.

1. Deadband sizes of high and low limits are positive: $EPS > 0$; and
2. Hysteresis zones of high and low limits are non-empty, disjoint, and ordered: $L + EPS < H - EPS$.

We incorporate these assumptions into the formalizing theory of *LIMITS_ALARM* in PVS as follows. For assumption 1, we use the subtype *posreal*, i.e., the set of positive real numbers, to declare the type of the time-dependent input variable *EPS* (i.e., *EPS: [tick → posreal]*). For assumption 2, we define a higher-order dependent type *dependent_high_limit_type* for the type of high limit, which depends on values of the low limit *L* and the deadband size *EPS*. Then, we declare the high limit *H* accordingly (i.e., *H* : *VAR dependent_high_limit_type*).

We now present the PVS theory that formalizes the above intended requirement of the *LIMITS_ALARM* block. All input, output and internal variables are declared as time-dependent functions, taking the current time *t* as one of its parameters.

```
LIMITS_ALARM [ (IMPORTING Time) delta_t : posreal ] : THEORY

  IMPORTING ClockTick [ delta_t ]
  IMPORTING defined_operators [ delta_t ]
  IMPORTING HYSTERESIS [ delta_t ]

  timed_real : TYPE = [ tick → real ]

% proposed assumption (1): EPS > 0
  timed_posreal : TYPE = [ tick → posreal ]

% proposed assumption (2): L + EPS < H - EPS <=> H - L > 2EPS
  dependent_high_limit_type : TYPE =
     [ L : timed_real, EPS : timed_posreal →
                  {H : timed_real | FORALL (t : tick) : H(t) − L(t) > 2 × EPS(t)} ]

  t : VAR tick

% Input variables
  X : VAR timed_real
  L : VAR timed_real
  H : VAR dependent_high_limit_type
  EPS : VAR timed_posreal


% Output variables
  QH : VAR pred [ tick ]
  QL : VAR pred [ tick ]
  Q : VAR pred [ tick ]
```

```
% Internal variables
  w1: VAR [tick → posreal]
  w2: VAR [tick → real]
  w3: VAR [tick → real]
  ...

END LIMITS_ALARM
```

To formalize the intended behaviour of each output, we define a separate function that has a tabular structure. Each of these functions is useful from two perspectives: 1) it is used to define the requirements predicate of the corresponding output variable (see below); and 2) it will be used to generate the witness of output value when conducting the consistency or feasibility proof (Section 4.1).

```
f_QH(X,H,L,EPS)(t): RECURSIVE bool =
   IF init(t) THEN FALSE
   ELSE LET prev = F_QH(X,H,L,EPS)(pre(t)) IN
    TABLE
     %-------------------------------------------------------------------%
     | X(t) > H(L,EPS)(t)                                          | TRUE  ||
     %-------------------------------------------------------------------%
     | X(t) >= sub(H(L,EPS)(t),EPS(t)) ∧ X(t) <= H(L,EPS)(t)       | prev  ||
     %-------------------------------------------------------------------%
     | X(t) < sub(H(L,EPS)(t),EPS(t))                              | FALSE ||
     %-------------------------------------------------------------------%
    ENDTABLE
   ENDIF
 MEASURE rank(t)

f_QL(X,L,EPS)(t): RECURSIVE bool =
   IF init(t) THEN FALSE
   ELSE LET prev = F_QL(X,L,EPS)(pre(t)) IN
    TABLE
     %-----------------------------------------------%
     | X(t) < L(t)                           | TRUE  ||
     %-----------------------------------------------%
     | X(t) <= add(L(t),EPS(t)) ∧ X(t) >= L(t) | prev  ||
     %-----------------------------------------------%
     | X(t) > add(L(t),EPS(t))                | FALSE ||
     %-----------------------------------------------%
    ENDTABLE
   ENDIF
 MEASURE rank(t)

f_Q(QH,QL)(t): bool =
    TABLE
     | QH(t) OR QL(t)          | TRUE ||
     | NOT QH(t) ∧ NOT QL(t)   | FALSE ||
    ENDTABLE
```

Then, each output variable of the *LIMITS_ALARM* block is formalized as a predicate by reusing the above functions.

```
P_QH(X,H,L,EPS,QH): bool =
  FORALL (t:tick): QH(t) = f_QH(X,H,L,EPS)(t)

P_QL(X,L,EPS,QL): bool =
   FORALL (t:tick): QL(t) = f_QL(X,L,EPS)(t)

P_Q(QH,QL,Q): bool =
  FORALL (t:tick): Q(t) = f_Q(QH,QL)(t)
```

Finally, to derive the overall requirement of the *LIMITS_ALARM* block, the above three predicates (corresponding to the tables in Fig. 10, p. 166) are composed using logical conjunctions (Section 3.2).

```
LIMITS_ALARM_REQ(H, X, L, EPS, QH, Q, QL): bool =
  P_QH(X, H, L, EPS, QH) ∧ P_QL(X, L, EPS, QL) ∧ P_Q(QH, QL, Q)
```

For any given input and output trajectories (mapping from ticks to values), the requirements predicate *LIMITS_ALARM_REQ* returns *TRUE* if they satisfy the above three output predicates; otherwise, it returns *FALSE*. This requirements predicate will later be used to verify the correctness of the FBD implementation of *LIMITS_ALARM*. This process can be generalized to verify other FBD implementations in IEC 61131-3.

Justification of our proposed requirements tables

Our proposed requirements tables for the *LIMITS_ALARM* block are by no means arbitrary and we justify them as follows.

Regarding the functionality of *LIMITS_ALARM*, the most authoritative source we could obtain is a one-line sentence from the standard [9, p. 190]: "This function block implements a high/low limit alarm with hysteresis on both outputs". Despite the fact that this requirement is written using an informal, natural language, a reasonably obvious approach to formalize it is by using the requirements of its component *HYSTERESIS* blocks. Based upon our analysis of the *HYSTERESIS* block (Section 5.2.2), we impose an assumption of non-negative deadband size. If we do not make this assumption, then the hysteresis FB implements a "toggle" of the output value when the input signal is outside of the deadband. See Section 5.2.2 for a detailed explanation. As far as the hysteresis FB is concerned, we believe that practitioners would not expect this toggling behaviour. If it is required, a different, appropriately named FB could be used to produce the toggling behaviour. Furthermore, we also believe that engineers would not expect that at a single system state, the low and high alarms are tripped simultaneously. As a result, we impose another assumption, namely that the two hysteresis regions do not overlap (Section 5.2.3).

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 unambiguous. We now present the two kinds of verification we perform in PVS.

4.1. Verifying the correctness and consistency 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 Section 3.1 is *correct* with respect to its requirement in Section 3.2, we must prove the following in PVS:

$$\begin{aligned}&\forall H, X, L, EPS \bullet \forall QH, Q, QL \bullet \\ &\quad LIMITS\_ALRM\_IMPL(H, X, L, EPS, QH, Q, QL) \\ &\qquad \Rightarrow LIMITS\_ALRM\_REQ(H, X, L, EPS, QH, Q, QL)\end{aligned} \tag{1}$$

The PVS specification of correctness checking is formulated as follows:

```
LIMITS_ALARM_CORRECTNESS: THEOREM
  LIMITS_ALARM_IMPL(H, X, L, EPS, QH, Q, QL) ⇒
    LIMITS_ALARM_REQ(H, X, L, EPS, QH, Q, QL)
```

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:

$$\forall H, X, L, EPS \bullet \exists QH, Q, QL \bullet LIMITS\_ALRM\_IMPL(H, X, L, EPS, QH, Q, QL) \tag{2}$$

The PVS specification of consistency checking is formulated as follows:

```
LIMITS_ALARM_CONSISTENCY : THEOREM
  FORALL (H, X, L, EPS) :
    EXISTS (QH, Q, QL) :
      LIMITS_ALARM_IMPL(H, X, L, EPS, QH, Q, QL)
```

4.2. Verifying the equivalence of implementations

In IEC 61131-3, the 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 Section 3.1 we discussed how to obtain, for a given FB, a predicate for its FBD description (say *FB_FBD_IMPL*) and one for its ST description (say *FB_ST_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:

$$\forall i_1, \ldots, i_m \bullet \forall o_1, \ldots, o_n \bullet \\ FB\_FBD\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \\ \equiv FB\_ST\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \tag{3}$$

In principle, we aim to prove that the ST and FBD implementations of the same FB, if applicable, agree on their external, input-output behaviour. However, the standard allows stateless standard functions (e.g., *MOVE*) to be converted into stateful function blocks, by adding a pair of input *EN* and output *ENO* [9, p. 68], which affects the execution flow of the function at runtime via interrupts. This means that if one implementation uses the stateless version, while the other uses the stateful version, then their runtime implementations may not be provably equivalent (because the implementation that uses the stateful version is possible to be interrupted, which is not possible for the other). Consequently, in this case we are only able to prove that the behaviour of the implementation without interrupts conforms to (i.e., is a subset of) that of the implementation with possible interrupts, by replacing "$\equiv$" with "$\Rightarrow$" in Equation (3).

$$\forall i_1, \ldots, i_m \bullet \forall o_1, \ldots, o_n \bullet \\ FB\_FBD\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \\ \Rightarrow FB\_ST\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \tag{4}$$

As an example, consider the *STACK_INT* block. The ST and FBD implementations are supplied at different levels of abstraction: the FBD description is closer to the hardware level as it uses additional execution control variables (i.e., a pair of enable in/out variables, *EN*/*ENO*) to indicate system errors (Appendix E of IEC 61131-3). Consequently, as explained above, we only need to prove that the lower level FBD implementation conforms to the higher level ST implementation.

Although IEC 61131-3 (2003) had been in use for almost 10 years, while performing this verification on *STACK_INT*, we found that we could not prove the implication (4) without introducing an explicit negation FB, see Section 5.3.1. We believe this is a good example of how the precision of formality can help us find errors that manual inspection often overlooks. In all likelihood a practical implementation of this FB in a real project would eventually fail and the error would be found. However, finding errors early is something we all strive for.

5. Case study: standard IEC 61131-3 including Annex F (version 2.0, 2003)

To justify the value of our approach (Sections 3 and 4), we have formalized and verified all of the FBs from IEC 61131-3, as well as standard functions that are used in these function blocks. Our work so far has revealed a number of possible issues. For the purpose of this paper, we will discuss the issues we found, and our suggestions on how to deal with them. We place issues we found into three categories: ambiguous behaviour (Section 5.1), possible missing input assumptions (Section 5.2), and inconsistent implementations (Section 5.3).

Before discussing each found issue in detail, it is critical for us to remind the reader that: 1) we derive our own input-output requirements table for each function block based on the description in the standard and our experience; and 2) we determine correctness based on these proposed requirements tables.

As is often the case, the mathematically precise requirements are not so much of interest in themselves, but rather that they facilitate very specific discussion on those requirements. Readers may disagree with our version of the required behaviour, but they are very clear as to what we say that required behaviour is. An inviolate assumption in our methodology is that we start with mathematically precise requirements.

5.1. Ambiguous behaviour

5.1.1. Pulse timer in timing diagrams

The block *PULSE* is a timer defined in IEC 61131-3, whose graphical declaration is shown on the LHS of Fig. 11. It takes two inputs (a Boolean condition *IN* and a time duration, *PT*) and produces two outputs (a Boolean value *Q* and a time

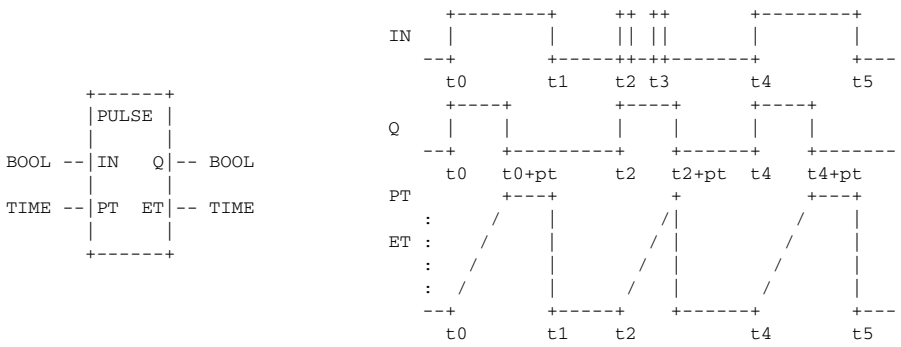


Fig. 11. Declaration of the timer *PULSE* and its definition in timing diagram [9].

Result

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

Result

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

Result

Condition			ET
Q			t − pulse_start_time
¬Q	**¬Held_For_ts(IN, PT, pulse_start_time)**		**0**
	Held_For_ts(IN, PT, pulse_start_time)	IN	PT
		¬IN	0

Fig. 12. Requirement of *PULSE* timer using tabular expressions.

duration, *ET*). 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 time duration, *PT*. The elapsed time that *Q* has remained *TRUE* can also be monitored via output *ET*. IEC 61131-3 presents a timing diagram as depicted on the RHS of Fig. 11, in which 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, and subtle or critical boundary cases are likely to be missing. We formalize the *PULSE* timer using tabular expressions that ensure both completeness and disjointness.

Most of the critical behaviours have been captured by the timing diagrams. However, while developing the tabular expressions we found that there are at least two scenarios that are not covered by the above timing diagram.

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 defaulted to 0 at time $t_2 + PT$ or at time t_k?

We use the three tables in Fig. 12 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 behaviour in these tables now answers the questions left open by the specific version of the timing diagram shown in Fig. 12 ("No", and "$t_2 + PT$"). Even if the questions are not obvious, if a developer suddenly wonders about these specific behaviour, the tabular expressions provide explicit answers, while the timing diagram cannot. The tables have their obvious equivalents in PVS. To make the timing behaviour precise, we define two auxiliary predicates *Held_For* and *Held_For_ts* which are based on the work presented in [18]:

```
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))
```

The predicate *Held_For(P, duration)* holds when the input predicate P holds for at least *duration* units of time. The 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 proposed solution to Scenario 1 (that the rising edge of *IN* at $t_2 + PT$ is missed), 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., $\neg Q$), and condition *IN* has not held for more than *PT* time units (as it became *FALSE* between *t2* and *t3*), so the value of *ET* is defaulted back to 0.

As the *PULSE* timer is not supplied with an 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.

5.1.2. Implicit delay unit

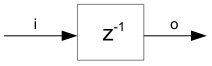
PLC applications often use feedback loops: outputs of an FB are connected as inputs of either another FB, or the FB itself. IEC 61131-3 specifies feedback loops through either a connecting line or shared names of inputs and outputs. However, feedback values (or intermediate output values) cannot be computed instantaneously in reality.

The behaviour of the *SR* block [9, p. 77] may be derived from the following extracts from the standard:

- *It shall be possible ... to determine the order of execution of the elements ... by selection of feedback variables to form an implicit loop.* (p. 137, item 2)
- *No element of a network shall be evaluated until the states of all of its inputs have been evaluated.* (p. 136, item 1)
- *Once the element with a feedback variable as output has been evaluated, the new value of the feedback variable shall be used until the next evaluation of the element.* (p. 136, item 3)

The above first item describes the mechanism that is adopted by the *SR* block. Combining the latter two items implies that there is an implicit delay between the feedback variable value being produced and the time at which it is used as an input. Note, however, for the formal verification for FBs that contain feedback loops, we need to make the unit delay explicit.

Therefore, in our modelling framework of time, we introduce a unit delay block z^{-1} to formalize the above extracts from the standard, and to explicitly inform users that there will be a delay of one unit of time before the newly-evaluated feedback variable value can be used as an input. A unit delay block z^{-1} with its formalization is shown in Fig. 13:



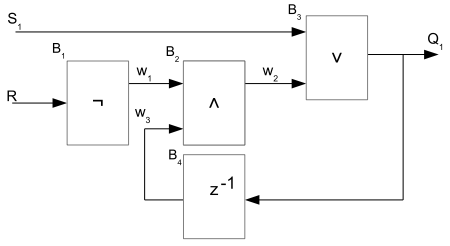
$$z^{-1}(i,o)(t) = \begin{cases} o(t) = i(t-1) & \textbf{if } t > 0 \\ FALSE & \textbf{if } t = 0 \end{cases}$$

Fig. 13. Declaration of the block *Unit Delay* and its formalization.

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 type of i and o can be any defined type, e.g., Boolean type in the example of block *SR*, 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 Fig. 14. The block *SR* 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* itself. Value of Q_1 remains *TRUE* as long as the set flag S_1 is enabled. Q_1 is reset to *FALSE* not only when the reset flag is enabled, but also when the set flag is disabled (so it cannot dominate the output result). Otherwise, Q_1 stays unchanged. There should be a delay between the value of Q_1 which is computed and passed to the next execution cycle. We formalize this by adding the explicit unit delay block z^{-1} and conjoining predicates for the internal blocks (as shown in Fig. 14). Blocks B_1 (formalized by predicate *NEG*), B_2 (*CONJ*), B_3 (*DISJ*), and B_4 (z^{-1}) in Fig. 14 denote the FB of, respectively, logical negation, conjunction, disjunction, and delay. Arrows w_1, w_2, and w_3 are internal connectives that are used to connect those internal blocks.

Adding an explicit unit delay block z^{-1} to formalize feedback loops led us to discharge the correctness and consistency theorems (Section 4) of the FBD implementation in Fig. 14. More precisely, the following theorems, as formulated in (5) and (6), are discharged in PVS, in which *SR_FBD_IMPL*, *SR_REQ* denote the FBD implementation and the tabular requirement of block *SR*.



$$SR\_IMPL(S_1, R, Q_1) \equiv \exists w1, w2, w3 \bullet \left(\begin{array}{l} NEG(R, w1) \\ \wedge\ CONJ(w1, w3, w2) \\ \wedge\ DISJ(S1, w2, Q1) \\ \wedge\ z^{-1}(Q1, w3) \end{array} \right)$$

Fig. 14. FBD implementation of the block *SR* and its formalizing predicate.

$$\forall S_1, R \bullet \forall Q_1 \bullet SR\_FBD\_IMPL(S_1, R, Q_1) \Rightarrow SR\_REQ(S_1, R, Q_1) \tag{5}$$

$$\forall S_1, R \bullet \exists Q_1 \bullet SR\_FBD\_IMPL(S_1, R, Q_1) \tag{6}$$

Our tabular expression for the requirement of the *SR* block is shown in Fig. 15:

		Result
Condition		Q_1
S1		1
¬S1	R	0
	¬R	NC

Fig. 15. Requirement of the *SR* block using a tabular expression.

5.2. Possible missing input assumptions

In this section we discuss how we uncover issues from the standard that are related to missing input assumptions. First of all, we make a clear distinction between our intended use of TCCs and lemmas. On the one hand, we use TCCs to ensure that the requirements tables are "healthy" (i.e., complete, disjoint, and well-defined) and can thus be reused as components of other function block theories. For example, to formulate the requirements of the *LIMITS_ALARM* block (Section 3.2), we represent the two instances of the *HYSTERESIS* block by referencing its requirements table, with the obligation to prove that it is both complete and disjoint. As another example, consider the *AVERAGE* block in Section 5.2.5: to formulate any composite function block that uses *AVERAGE* as a component, we are obliged to prove that its requirements table is well-defined (i.e., the denominator *N* is not equal to zero).

On the other hand, for any function block, we may use lemmas to express certain desired properties that are not directly expressed in its requirements table. We consider these lemmas as additional requirements that implementation(s) of the function block in question must also satisfy. For example, consider the *DELAY* block in Section 5.2.4: we use the *IXIN_IXOUT_REL* to assert that the output index *IXOUT* is *always N* samples behind the input index *IXIN*. The formulation of lemma *IXIN_IXOUT_REL* is not arbitrary: it is based on the (informal) requirements that the standards provides for the *DELAY* block [9, p. 187]: "This function block implements an N-sample delay". As a result, any unproven TCCs or lemmas suggest that it is not safe to reuse the function block in question, as there are issues (e.g., missing an explicit input assumption) with either its requirements table or its implementation(s).

Impact of unproven TCCs or lemmas. In our verification framework, each unproven lemma or table with unproven TCCs is not depended upon by theories of other functions. In particular, we do not reuse any unproven lemmas to prove properties of other function blocks. We achieve this by creating a "fixed" version of the FB implementation or requirements table, incorporated with our proposed solution, e.g., an explicit input assumption. In the case of an unproven lemma, we create a new lemma that is identical to the unproven one, except that it references the "fixed" version of FB implementation or requirements table, and it is thus provable. That is, for each "fixed" version of a function block, all the associated TCCs, lemmas, and theorems are proved.

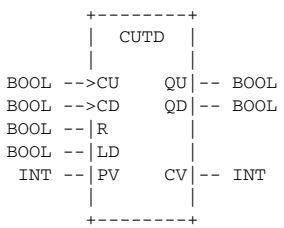
For example, we have two versions of specifications for the *HYSTERESIS* block (Section 5.2.2): one with the non-negative-hysteresis-band-size assumption, and the other one without. The unprovable TCC only affects the correctness of the version of *HYSTERESIS* without such an assumption, and we do not reference this version of *HYSTERESIS* elsewhere. On the other hand, the version of *HYSTERESIS* with an explicitly introduced assumption can be proved correct, and we thus can safely reference it in the context of the *LIMITS_ALARM* block.

As we carefully guided the PVS prover when conducting proofs, for all TCCs and lemmas that we failed to prove, we could: 1) trace back to the original FBs in the standard; 2) decide whether our requirements are incorrect or the supplied implementation is not consistent with the requirements; and 3) uncover issues that we report in this paper. As indicated above, all lemmas and TCCs of the fixed versions of FB theories are proved, and the proofs of all the final results do not make use of any unproven lemmas or TCCs. Furthermore, all proofs are available for inspection of correctness.

Compositionality. The introduction of input assumptions does not break the compositionality of our approach. When an FB in question cannot be proved as satisfying its input-output requirements, we attempt to trace back to its specification and identify the source of failure. For circumstances that lead to undesirable results (e.g., the toggling behaviour of a *HYSTERESIS* block in Section 5.2.2), we propose to precisely character them as input assumptions (e.g., positive hysteresis deadband size). An alternative solution is that of defensive programming: users may modify the FB implementation from the standard, such that it always checks for conditions that will lead to abnormal behaviour, then take the appropriate actions (e.g., flag an error, reset the state, do nothing, etc.).

Input assumptions are useful in that they make those problematic scenarios explicit to users of the FBs, without the need for them to discover them from the source code. In PVS, we formalize input assumptions using predicate subtypes (e.g., *posreal* for the hysteresis deadband size in Section 5.2.2) or dependent types (non-overlapping hysteresis regions for the limits alarm in Section 3.2).

Consequently, TCCs that are specific to these input assumptions will be automatically generated by PVS. That is, adding input restrictions means that there are additional proof obligations to be discharged to make sure that the relevant FBs are




```
FUNCTION_BLOCK CTUD
  VAR_INPUT
    CU, CD : BOOL R_EDGE; (* Value to be counted up/down *)
    R      : BOOL         (* Reset *)
    LD     : BOOL         (* Load value flag *)
    PV     : INT          (* Preset value *)
  END_VAR
  VAR_OUTPUT
    QU : BOOL (* Compare CV with PV for up counter *)
    QD : BOOL (* Compare CV with 0 for down counter *)
    CV : INT  (* Current counted value *)
  END_VAR
  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) ;
END_FUNCTION_BLOCK
```

Fig. 16. Declaration of the block *CTUD* and its ST implementation [9].

invoked with legitimate input values. However, this does not break the compositionality in our approach. For a composite FB, if any of its component FBs is supplied with input values that are not provably legitimate, then the correctness of that composite block cannot be proved, which is a desired outcome.

5.2.1. Limit on the counter blocks

IEC 61131-3 describes three types of counters. An up-down counter (*CTUD*) in IEC 61131-3 is composed of an up counter (*CTU*) and a down counter (*CTD*). The ST implementation and graphical declaration are provided in the standard as shown in Fig. 16.

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 defaulted 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 $QD \equiv (CV <= 0)$. Note that the lines connected to *CU* and *CD* inputs are right-arrowed. In the IEC 61131-3, it denotes the signals from a rising edge detector function block. Similarly, left-arrowed lines denote the signals from a falling edge detector function block. We have formalized and verified these two blocks in PVS.

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:

1. 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[11]: $PVmin < PVmax$.
2. 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. 17. Similarly, we added $PV < PVmax$ and $PVmin < PV$ as assumptions for, respectively, the up and down counters.

We now need to discuss how we arrived at our assumptions. We also need to consider the impact of adopting the ST code as is.

Unlike in the case of the *LIMITS_ALARM* block, in the standard there is no summary of what the intended functionality of the up-down counter (*CUTD*) is. Instead, the standard suggests [9, p. 78] that the ST implementation in its entirety represents

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

Condition				Result CV
R				0
¬R	LD			PV
	¬LD	CU ∧ CD		NC
		CU∧¬CD	$CV_{-1} < PVmax$	CV_{-1}+1
			$CV_{-1} \geq PVmax$	NC
		¬CU∧CD	$CV_{-1} > PVmin$	CV_{-1}-1
			$CV_{-1} \leq PVmin$	NC
		¬CU ∧ ¬CD		NC

assume: $PVmin < PV < PVmax$

Fig. 17. Tabular requirement of the block *CTUD*.

the requirements of *CUTD*: "The operation of these function blocks [e.g., *CUTD*] shall be as specified in the corresponding function block bodies [i.e., ST code]". Consequently, the only authoritative source we can rely on for the purpose of analysis is the ST code itself (RHS in Fig. 16).

Inspecting the variable declaration and implementation body of the ST code, we make the following observations:

1. Since outputs *QU* and *QD* are declared as variables, as opposed to constants, their values are expected to vary according to changes of the state (i.e., *CV* and *PV*). More precisely, the last two lines of the implementation body indicate that $QU \equiv (CV > PV) \wedge QD \equiv (CV \leq 0)$ is an intended invariant.
2. The choice of variable names *PVmax* and *PVmin* suggest that they are, respectively, the upper bound and lower bound for the value of *PV*.
3. The if–then–else statement, executed when $\neg R \wedge \neg LD$, suggests that the value of *PVmax* is used as an upper bound to prevents increments on the counter value *CV* from overflow, and similarly for *PVmin* to prevent decrements on *CV* from underflow. More precisely, the condition $PVmin \leq CV \leq PVmax$ is an intended invariant, and the counter block is effectively disabled (i.e., value of *CV* remains unchanged) when this invariant is violated.

Our observations above are arguably consistent with ones made by any experienced engineer or programmer. Therefore, any violation of them may suggest a possible issue.

Our first proposed assumption $PVmin < PVmax$ is intended to guard the truth of observation 2 above. Our common perception should allow us to assume that for the same monitored quantity (e.g., *PV*), its upper bound is strictly larger than its lower bound, for otherwise it is nearly impossible for the monitored quantity to fall "within the boundaries" (i.e., $PVmax < PVmin \Rightarrow (PVmin \leq PV \leq PVmax \equiv FALSE)$). Even if one may argue that for this particular example, the value *PV* is meant to be chosen from either the interval $[PVmin, PVmax]$ or the interval $[PVmax, PVmin]$, depending on how *PVmin* and *PVmax* are related at runtime. However, this is even more problematic because according to observation 3 above, when $PVmax < PVmin$, choosing a value of *PV* from interval $[PVmax, PVmin]$ will effectively disable the counter block.

Our second proposed assumption $PVmin < PV < PVmax$ is justified by observation 1 above that *PV* should be chosen within its defined boundaries. Without such assumption, say *PV* is always chosen such that $PV > PVmax$, then our observations that $QU \equiv (CV > PV)$, and that $CV \leq PVmax$, imply that *QU* declared as a variable will act like a constant *FALSE*. A similar argument applies to the case of $PV < PVmin$.

Therefore, we think it is more justifiable to impose the two assumptions than not to do so. Nonetheless, the existing ST code in the standard does not prevent users of the *CUTD* block from violating these assumptions, in which case the counter block may, as explained above, become completely disabled and/or always output *QU* and *QD* as *FALSE*.

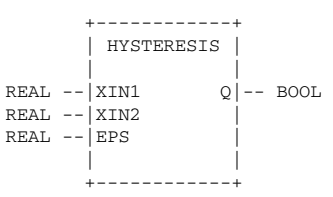
Again, this amply demonstrates the value of our approach: it makes these two input details (subtle yet non-negligible as they greatly impact the resulting behaviour) precise and explicit for users to decide. If the ST code is used as is, users may inadvertently disable the counter simply by inputting values that violate our suggested invariant. No error would be generated. We believe that this kind of control over functionality should always be explicit.

5.2.2. Deadband size of the HYSTERESIS block

The *HYSTERESIS* block implements a Boolean hysteresis: the output value depends not only on the current input values, but also the output value in the past. Its declaration (shown on the LHS in Fig. 18) requires three real-valued input numbers: *XIN1* is typically read from a sensor, *XIN2* specifies its set point, and *EPS* indicates that the deadband (above and below the set point) within which the Boolean output signal value Q should remain unchanged.

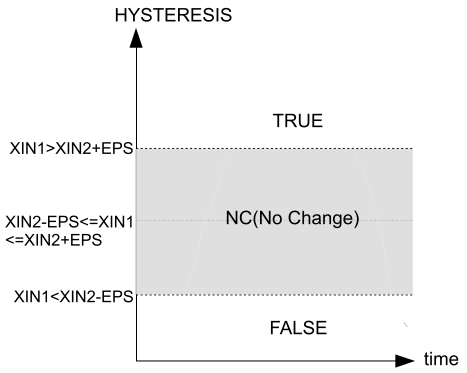
We formalize the requirement of the *HYSTERESIS* block in Fig. 19, with the assumption that the deadband size is non-negative. The shaded area in Fig. 19 denotes the hysteresis deadband (with a size of $2 \times EPS$). If the current sensor value *XIN1* is such that $XIN1 < XIN2 - EPS$, then output Q becomes *FALSE*. Similarly, if it is the case that $XIN1 > XIN2 + EPS$, then Q becomes *TRUE*. For the stability of Q's value, if the sensor value lies within the deadband (i.e., $XIN2 - EPS \leq XIN1 \leq XIN2 + EPS$), then output Q remains unchanged (a case of no change).

For the behaviour specified in Fig. 19, it is necessary to have the assumption about the value of *EPS* being non-negative. Otherwise, the two intervals $XIN1 > (XIN2 + EPS)$ and $XIN1 < (XIN2 - EPS)$ may overlap (i.e., the two constraints are not disjoint) when $EPS < 0$, and an unprovable proof obligation (TCC of Disjointness) is generated in PVS (which we omit here).




```
FUNCTION_BLOCK HYSTERESIS
  (* Boolean hysteresis on difference *)
  (* of REAL inputs, XIN1 - XIN2 *)
  VAR_INPUT XIN1, XIN2, EPS : REAL; END_VAR
  VAR_OUTPUT Q : BOOL := 0; END_VAR
  IF Q THEN IF XIN1 < (XIN2 - EPS) THEN Q := 0; END_IF ;
  ELSIF XIN1 > (XIN2 + EPS) THEN Q := 1 ;
  END_IF ;
END_FUNCTION_BLOCK
```

Fig. 18. Declaration of the block *HYSTERESIS* and its ST implementation [9].



Condition	*Result* Q
$\mathrm{XIN1} > (\mathrm{XIN2} + \mathrm{EPS})$	1
$(\mathrm{XIN2} - \mathrm{EPS}) \leq \mathrm{XIN1} \leq (\mathrm{XIN2} + \mathrm{EPS})$	NC
$\mathrm{XIN1} < (\mathrm{XIN2} - \mathrm{EPS})$	0

assume: $EPS \geq 0$

Fig. 19. Requirement of the block *HYSTERESIS*: with the assumption $EPS \geq 0$.

	Condition	*Result* Q
$\neg Q_{-1}$	$XIN1 > (XIN2 + EPS)$	1
	$XIN1 \leq (XIN2 + EPS)$	NC
Q_{-1}	$XIN1 \geq (XIN2 - EPS)$	NC
	$XIN1 < (XIN2 - EPS)$	0

Fig. 20. ST implementation of block *HYSTERESIS* in tabular expressions: with no assumption on *EPS*.

Nonetheless, in practice, subject to the oscillation on the sensor value *XIN1*, the value of input *EPS* should be positive (and sufficiently large) to create a deadband for stabilizing the value of output *Q*. Therefore, in our PVS models, when proving the correctness of *HYSTERESIS* and blocks that use it (e.g., the *LIMITS_ALARM* block discussed in Section 5.2.3), we adopt a stronger assumption $EPS > 0$ than that for Fig. 19.

We will relax such an assumption of positive deadband size later in this section (in Fig. 21), by considering the behaviour of the *HYSTERESIS* with a negative deadband size.

For the purpose of verification, we translate the ST implementation (on the RHS in Fig. 18) into a PVS predicate that has a tabular structure[12] in Fig. 20. In this complete and disjoint tabular representation of the ST code, there is no assumption about the value of input *EPS* (i.e., whether or not it is non-negative).

However, the behaviour of the ST implementation (Fig. 20) does not conform to that in Fig. 19. The implementation supplied by the standard actually allows a toggling behaviour on the value of output *Q*. In the case of a negative value for *EPS*, the value of output *Q* alternates between *FALSE* and *TRUE* (or 0 and 1) when *XIN2* is within the headband. Let's consider a concrete example. Say $EPS = -2$, $XIN1 = 1$, and $XIN2 = 2$, then by executing the ST code (RHS in Fig. 18 and Fig. 20) multiple times, we obtain alternating (or toggling) results (of 0 and 1) for *Q*.

To understand this toggling behaviour more clearly, we provide an extended tabular requirement that incorporates the case of negative *EPS* (Fig. 21), where the two rows that represent the toggling behaviour are set in boldface.

It may be that developers actually use the toggling feature provided by the functionality in the standard. However, this feature is definitely not what most practitioners would expect from a hysteresis FB. As we said earlier, if we intend to include such behaviour, the behaviour must be explicitly clear. In this particular case, we strongly believe that the toggling behaviour should be implemented in a different FB, not hidden within a hysteresis FB that changes its behaviour depending on whether or not the hysteresis value is positive or negative. The value of our approach is that it made this hidden behaviour explicit, so that we can make decisions on whether or not to include it.

[12] This tabular structure has a straightforward counterpart in PVS.

Condition			Result Q
$EPS \geq 0$	$XIN1 > (XIN2 + EPS)$		1
	$(XIN2 - EPS) \leq XIN1 \leq (XIN2 + EPS)$	Q_{-1}	1
		$\neg Q_{-1}$	0
	$XIN1 < (XIN2 - EPS)$		0
$EPS < 0$	$XIN1 \geq (XIN2 - EPS)$		1
	$(XIN2 + EPS) < XIN1 < (XIN2 - EPS)$	Q_{-1}	**0**
		$\neg Q_{-1}$	**1**
	$XIN1 \leq (XIN2 + EPS)$		0

Fig. 21. Requirement of the block *HYSTERESIS*: with no assumption on *EPS*.

5.2.3. High/low limits of LIMITS_ALARM block

The function block *LIMITS_ALARM* (with two internal blocks *HIGH_ALARM* and *LOW_ALARM*) has been used as a running example in this paper: its declaration in Section 2.1, its FBD implementation formalized in PVS in Section 3.1.5, its tabular requirement in Section 3.2, and its verification conditions in Section 4. In this section we discuss the two missing assumptions of this block.

1. Similar to the case of the *HYSTERESIS* block (Section 5.2.2), we impose an assumption $EPS > 0$ (i.e., positive hysteresis deadband size) to ensure that the two hysteresis zones $[L,\ L+EPS]$ and $[H-EPS,\ H]$ are computed in the right directions and non-empty.
2. We impose another assumption $H - EPS > L + EPS$, or equivalently $H - L > 2EPS$, to separate two hysteresis zones. We reckon that the intention of having both high and low limits is to have two disjoint hysteresis zones. Otherwise, if the two zones overlap, then the high and low alarms may be tripped simultaneously, which would falsify the system property that at any time *only* the high limit or low limit can be tripped.

During the proof of overall correctness, we introduce three lemmas, each corresponding to the correctness of an output variable. This exemplifies the decomposition of the proof for the goal theorem into smaller ones.

```
OUTPUT_QH_CORRECTNESS_CHECKING: LEMMA
  LIMITS_ALARM_IMPL(H, X, L, EPS, QH, Q, QL) => f_QH(X, H, L, EPS, QH)

OUTPUT_QL_CORRECTNESS_CHECKING: LEMMA
  LIMITS_ALARM_IMPL(H, X, L, EPS, QH, Q, QL) => f_QL(X, L, EPS, QL)

OUTPUT_Q_CORRECTNESS_CHECKING: LEMMA
  LIMITS_ALARM_IMPL(H, X, L, EPS, QH, Q, QL) => f_Q(QH, QL, Q)
```

Having introduced the dependent type of *dependent_high_limit_type* (Section 3.2), we are able to prove the invariant property, that high alarm and low alarm can not be tripped at the same time, as a theorem *PROPERTY0*:

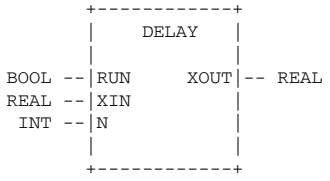
```
PROPERTY0: THEOREM
  LIMITS_ALARM_IMPL(X, H, L, EPS, QH, Q, QL)
    => FORALL (t: tick): NOT (QH(t) ∧ QL(t))
```

With our tabular requirement that incorporates the above two assumptions, we proved that the ST implementation supplied by IEC 61131-3 is both correct and consistent (Section 4). The proof process involved predicates for the 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.

5.2.4. Initialization failure of the DELAY block

The *DELAY* block (declared on the LHS of Fig. 22) generates an *N*-sample delay between the input *XIN* and the output *XOUT*. That is, the value of *XOUT* corresponds to the value of the last *N*th *XIN*. The delay may be disabled, i.e., $XOUT = XIN$, by setting a Boolean input flag *RUN* to *FALSE*.

More precisely, we formulate the requirement of the *DELAY* block using the tabular expressions in Fig. 23. The upper table in Fig. 23 specifies *last_disabled* the latest moment in time when the input flag *RUN* is set to *FALSE*. The lower table in Fig. 23 documents the relation between the inputs (i.e., *N*, *XIN*, and *RUN*) and the output (i.e., *XOUT*). When the delay is disabled (i.e., *RUN* is *FALSE*), the value of *XOUT* is set to that of *XIN* (i.e., no delay is occurring). Otherwise, when the delay is enabled, we differentiate between two cases: whether or not *RUN* is set to *TRUE* for a time period of at least *N* ticks. First, if the delay has been enabled for sufficiently long, the value of *XOUT* is set to that of *XIN N* ticks behind. Second, before the




```
FUNCTION_BLOCK DELAY (* N-sample delay *)
  VAR_INPUT
    RUN : BOOL ; (* 1 = run, 0 = reset *)
    XIN : REAL ;
    N   : INT ;  (* 0 <= N < 128 or manufacturer- *)
  END_VAR (* specified maximum value *)
  VAR_OUTPUT XOUT : REAL; END_VAR (* Delayed output *)
  VAR X : ARRAY [0..127] OF REAL ;(* N-Element queue  with FIFO discipline *)
      I, IXIN, IXOUT : INT := 0 ;
  END_VAR
  IF RUN THEN IXIN := MOD(IXIN + 1, 128) ; X[IXIN] := XIN ;
     IXOUT := MOD(IXOUT + 1, 128) ; XOUT := X[IXOUT];
  ELSE XOUT := XIN ; IXIN := N ; IXOUT := 0;
     FOR I := 0 TO N DO X[I] := XIN; END_FOR;
  END_IF ;
END_FUNCTION_BLOCK
```

Fig. 22. Declaration of the block *DELAY* and its ST implementation [9].

value of delayed *XIN* is ready, the value of *XOUT* is set to that of *XIN* at time (i.e., *last_disabled*) when the *DELAY* block was last disabled.

The ST implementation of the *DELAY* block (shown on the RHS of Fig. 22) uses a circular array *X* to maintain a sliding window of size *N*, as new values of the sample *XIN* are read. Then, the output *XOUT* corresponds to the cell in array *X* that is *N*-position behind the current sample *XIN*. Two auxiliary variables, *IXIN* and *IXOUT*, are used to store indices of cells that store, respectively, *XIN* and *XOUT*. When the input flag *RUN* is set to *TRUE*, indicating that the *N*-sample delay should be in effect, values of both *IXIN* and *IXOUT* are incremented accordingly to slide the window.[13] Otherwise, values of *IXIN*, *IXOUT*, and their in-between cells are reset.

Inspecting its ST implementation, the intended usage of the *DELAY* block requires *RUN* being disabled in order to properly set the two indices. As an example, consider the following use case: 1) disable *RUN* initially ($t = 0$) to properly separate the two indices apart; and 2) enable *RUN* from then on ($t > 0$). For phase 2), there are two cases to consider. Before *N* samples have been collected, the output value should equal to that of the input when *RUN* was last disabled. After *N* samples have been buffered, the proper delay effect should be observed: output value equals to that of the last *N*th input.

However, we discovered that the supplied ST implementation does not prevent users from enabling *RUN* initially, in which case the delay effect will never occur, even after *N* samples have been collected. More precisely, we were unable to prove the following property, which justifies itself by formalizing the informal requirements of the *DELAY* block [9, p. 187]: "This function block implements an N-sample delay", meaning that the value of output should equal that of the input *N*-samples ago.

IXIN_IXOUT_REL: **LEMMA**
$MOD(f\_IXOUT(RUN)(t) + N, 128) = f\_IXIN(RUN, N)(t)$

Recursive functions *f_IXOUT* and *f_IXIN* return the current value of, respectively, *IXOUT* and *IXIN*. Lemma *IXIN_IXOUT_REL* states that, in the context of a circular array of size 128, *IXOUT* is *N* always samples behind *IXIN*. The proof is based upon an induction on time *t* using the induction scheme *time_induction* (see Section 2.4). By reformatting the generated unprovable PVS sequent, we obtained an unprovable predicate: $init(t) \Rightarrow mod(0 + N, 128) = 0$. That is, the initial distance between cells referenced by *IXIN* and *IXOUT* should be *N*, but the initialization in the original implementation in the standard failed to satisfy this constraint.

From the ST implementation in Fig. 22, both *IXIN* and *IXOUT* are initialized to 0. This means that initially they point to same the cell in array *X*. As the *DELAY* block remains enabled (i.e., input *RUN* set to *TRUE*), both *IXIN* and *IXOUT* are incremented and will thus always point to the same cell. Consequently, there is no effect of an *N*-sample delay.

We propose to solve this issue by initializing *IXIN* to *N* instead of 0, such that cells referenced by *IXIN* and *IXOUT* are *N* samples apart. As a result, we are able to prove that the revised implementation satisfies the lemma *IXIN_IXOUT_REL*.

Moreover, the value of *N* may be set to 0, which means there should be 0-sample delay in effect. In this case, both *IXIN* and *IXOUT* will, consistently, always point to the same cell in array *X*. However, allowing such a boundary value for *N* can have dangerous consequence, e.g., the client block *PID* (Section 5.2.6) uses the *DELAY* block as one of its components. As a result, we consider the input of $N = 0$ to be an unacceptable case and redefine the type of *N* by excluding value 0: $\{1, 2, \ldots, 128\}$.

Finally, based on the above reasoning, we formalize the complete tabular requirement for the *DELAY* block (Fig. 23).

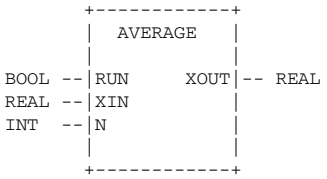
Once we have added the lemma *IXIN_IXOUT_REL* to enforce an invariant not included in the original ST, we clearly have to ask ourselves whether the addition of the lemma is justified. Similar to the case of the *LIMITS_ALARM* block, the most

[13] This circular operation is implemented by a division modulo operator `MOD`.

Condition		Result: last_disabled
t = 0		0
t > 0	RUN	NC
	¬RUN	t

Condition		Result: XOUT
¬RUN		XIN
RUN	Held_For(RUN, N·δ)	XIN_{-N}
	¬Held_For(RUN, N·δ)	$XIN_{-rank(t-last\_disabled)}$

Fig. 23. Requirement of the block *DELAY*.




```
FUNCTION_BLOCK AVERAGE
  VAR_INPUT
    RUN : BOOL ; (* 1 = run, 0 = reset *)
    XIN : REAL ; (* Input variable *)
    N   : INT  ; (* 0 <= N < 128 or manufacturer *)
  END_VAR         (* specified maximum value *)
  VAR_OUTPUT XOUT : REAL ; END_VAR (* Averaged output *)
  VAR SUM  : REAL := 0.0; (* Running sum *)
      FIFO : DELAY      ; (* N-Element FIFO *)
  END_VAR
  SUM := SUM - FIFO.XOUT ;
  FIFO (RUN := RUN , XIN := XIN, N := N) ;
  SUM := SUM + FIFO.XOUT ;
  IF RUN THEN XOUT := SUM/N ;
  ELSE SUM := N*XIN ; XOUT := XIN ;
  END_IF ;
END_FUNCTION_BLOCK
```

Fig. 24. Declaration of the block *AVERAGE* and its ST implementation [9].

Condition	Result: last_disabled
RUN	NC
¬RUN	t

Condition			Result: XOUT
¬RUN			XIN
RUN	Held_For(RUN, N·δ)		$\frac{XIN + \sum_{i=1}^{N-1} XIN_{-i}}{N}$
	¬Held_For(RUN, N·δ)	¬RUN_{-1}	XIN_{-1}
		RUN_{-1}	$\frac{XIN + \sum_{i=1}^{\#new\_vals} XIN_{-i} + \sum_{i=1}^{\#old\_vals} XIN_{-rank(t-last\_disabled)}}{N}$

#new_vals = rank(t - last_disabled) - 1
#old_vals = N - #new_vals - 1

Fig. 25. Requirement of the block *AVERAGE*.

authoritative source regarding the functionality of *DELAY* we could obtain from the standard is a one-line sentence [9, p. 187] which says that the value of the output should equal that of the input *N*-samples ago. As the lemma *IXIN_IXOUT_REL* only makes this informal statement formal, we believe our use of it to verify the correctness of the ST code is justified.

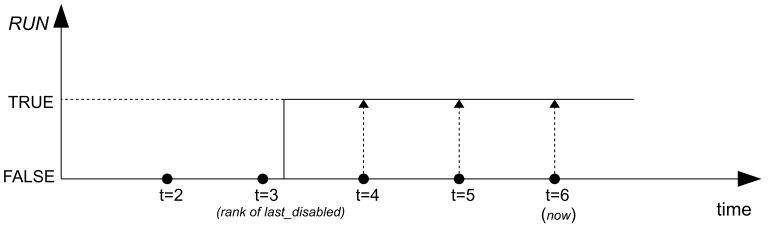
Furthermore, when the lemma *IXIN_IXOUT_REL* failed to prove, we were able to trace back to the original ST code, and confirm that the ST code did not always match the informal statement of its behaviour. More precisely, we found the use case where *RUN* is always enabled, keeping the output and input indices always synchronized on their values, and thus causing the desired delay to never occur. As an example, consider that input *N* is set to 5 for the *DELAY* block, meaning that there should be a 5-sample delay occurring (after the first 5 samples have been buffered). However, the ST code does not prevent the user from enabling *RUN* right from the beginning, and the consequence is that the delay effect will never be observed, even after 5 samples have been collected. We consider this to be non-compliance with its informal requirements [9, p. 187].

5.2.5. Division by zero of the AVERAGE block

The *AVERAGE* block (whose declaration is shown on the LHS in Fig. 24) computes a running average *XOUT* over the last *N* values of the input sample *XIN*. The ST implementation of *AVERAGE* (shown on the RHS in Fig. 24) indicates that it is a composite FB. It references an instance of the *DELAY* block (Section 5.2.4), storing the latest *N* values of the input *XIN*, to maintain an internal sliding window of size *N*. An internal variable *SUM* is used to store the running average, updated by subtracting the oldest value (i.e., output value from the *DELAY* instance) and adding the current value of *XIN*. Furthermore, the output *XOUT* may be calculated differently depending upon the value of a Boolean input flag *RUN*. If *RUN* is *TRUE*, then the value of *XOUT* represents the running average *SUM*/*N* as expected. Otherwise, it is reset to the current value of the input *XIN*.

Based on our understanding of the ST implementation, we formulate the requirement of the *AVERAGE* block in Fig. 25. Similar to the case of the *DELAY* block (Section 5.2.4), the value of *XOUT* is specified using *last_disabled* (i.e., the time when *RUN* was last set to *FALSE*) and the *Held_For* (Section 5.1.1) timing operator. There are four cases to consider:

1. If *RUN* is *FALSE*, then *XOUT* is set to the current value of *XIN*.
2. If *RUN* remains *TRUE* for a time period of at least *N* ticks, then *XOUT* is set to the average of the most recent *N* values of the input sample *XIN*.
3. If *RUN* has just become *TRUE* at the current instant, then *XOUT* is set to the value of *XIN* when *RUN* was last stopped (i.e., XIN_{-1}).
4. If *RUN* has not remained *TRUE* for sufficiently long, then *XOUT* is set to the average over: 1) samples taken since after the moment in which *RUN* was last *FALSE* (i.e., instant *last_disabled*); and 2) a number of copies of the value of *XIN* at instant *last_disabled* (i.e., $XIN_{-rank(t-last\_disabled)}$[14]). The obvious constraint is that is that the total number of samples from 1) and 2) equals *N*.
   As an example, consider the following scenario, where currently $t = 6$ and *RUN* has become and remained *TRUE* since when $t = 4$:



In the above scenario, the resulting average *XOUT* should be

$$\frac{XIN + (XIN_{-1} + XIN_{-2}) + XIN_{-3} \times 2}{5}$$

where XIN_{-1} and XIN_{-2} denote values of *XIN* when, respectively, $t = 5$ and $t = 4$. Say the sliding window size is 5, so we need to count two copies of the value of *XIN* at instant *last_disabled* (i.e., XIN_{-3}).

However, the range of *N* (i.e., {0, 1, 2, ..., 128}) includes the possibility of zero. This means that when *RUN* is *TRUE* and the value of *N* happens to be set zero, the value of the running average will be undefined due to a division by zero. This issue is reflected by an unprovable PVS proof obligation:

```
% Subtype TCC generated (at line 72, column 31) for n
% expected type nznum
% unfinished
Average_impl_st_TCC1: OBLIGATION
  FORALL (run: pred[tick[delta_t]], n: DelayUnits[delta_t], t: tick[delta_t]):
    run(t) AND NOT init(t) => n /= 0;
```

The above proof obligation is generated when the implementation predicate is type-checked. It states that when input *RUN* is *TRUE* and the current tick is not the initial tick, the value of *N* can not be zero. However, this sequent is unprovable. We propose to solve this issue by constraining the type of *N* such that the value of zero is excluded: $\{1, 2, \ldots, 128\}$.

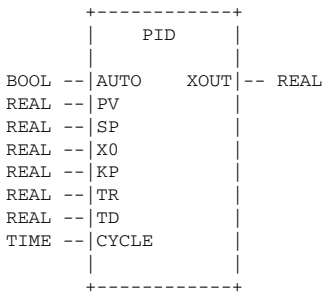
Are there other options for solving the issue of possible division by zero? The ST code could be modified so that it defensively handles this issue by checking for the value of *N* being zero, and either flagging an error or returning some default result. However, we think it is better to explicitly state this input assumption and thus warn the intended users of the *AVERAGE* block that they need to cope with this possibility.

5.2.6. Division by zero of the PID block

The *PID* (proportional-integral-derivative) block, whose declaration is shown on the LHS in Fig. 26, implements the classical three-term controller for closed-loop feedback control. The output signal from the *PID*, based upon its internal three-term computation, is used in many industrial applications where stable control is required using the feedback of the input process value.

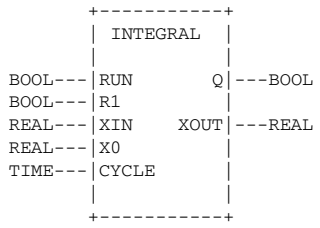
At each current time instant *t*, the *PID* controller computes an "error" value as the difference between values of a measured process (*PV*) variable and a desired set point (*SP*). The controller then outputs a control signal (*XOUT*) as the result of a weighted sum of three terms: 1) the proportional term (depending on the current error); 2) the integral term (depending on errors accumulated from past); and 3) the derivative term (predicting error in the future). The computation also depends on other inputs constants: *KP* (proportionality constant), *TR* (reset time), *TD* (derivative time), and *CYCLE* (sampling period). At the top level, we formalize the requirements of the *PID* block as a one-line equation, resembling the last statement of its ST implementation (shown on the RHS in Fig. 26):

[14] Here $rank(t - last\_disabled)$ denotes the number of ticks occurring between *last_disabled* and now.




```
FUNCTION_BLOCK PID
  VAR_INPUT
    AUTO : BOOL ;  (* 0 - manual , 1 - automatic *)
    PV   : REAL ;  (* Process variable *)
    SP   : REAL ;  (* Set point *)
    X0   : REAL ;  (* Manual output adjustment - *)
                   (* Typically from transfer station *)
    KP   : REAL ;  (* Proportionality constant *)
    TR   : REAL ;  (* Reset time *)
    TD   : REAL ;  (* Derivative time constant *)
    CYCLE: TIME ;  (* Sampling period *)
  END_VAR
  VAR_OUTPUT XOUT : REAL; END_VAR
  VAR ERROR : REAL ;       (* PV - SP *)
      ITERM : INTEGRAL ;   (* FB for integral term *)
      DTERM : DERIVATIVE ; (* FB for derivative term *)
  END_VAR
  ERROR := PV - SP ;
  (*** Adjust ITERM so that XOUT := X0 when AUTO = 0 ***)
  ITERM (RUN := AUTO, R1 := NOT AUTO, XIN := ERROR,
         X0 := TR * (X0 - ERROR), CYCLE := CYCLE) ;
  DTERM (RUN := AUTO, XIN := ERROR, CYCLE := CYCLE) ;
  XOUT := KP * (ERROR + ITERM.XOUT/TR + DTERM.XOUT*TD) ;
END_FUNCTION_BLOCK
```

Fig. 26. Declaration of the block *PID* and its ST implementation [9].




```
VAR_INPUT
  RUN   : BOOL ;  (* 1 = integrate, 0 = hold *)
  R1    : BOOL ;  (* Overriding reset *)
  XIN   : REAL ;  (* Input variable *)
  X0    : REAL ;  (* Initial value *)
  CYCLE: TIME ;  (* Sampling period *)
END_VAR
VAR_OUTPUT
  Q     : BOOL ;  (* NOT R1 *)
  XOUT  : REAL ;  (* Integrated output *)
END_VAR
```

Fig. 27. Declarations of the block *INTEGRAL* [9].

Condition	*Result* Q
R1	0
¬R1	1

Condition		*Result* XOUT
R1		$X0$
¬R1	RUN	$XOUT_{-1} + XIN * CYCLE$
	¬RUN	$XOUT_{-1}$

Fig. 28. Requirement of the block *INTEGRAL*.

$$XOUT = KP \times ((PV - SP) + \frac{ITERM.XOUT}{TR} + DTERM.XOUT \times TD)^{15}$$

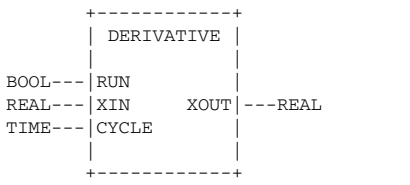
where *ITERM* and *DTERM* are instances of, respectively, the *INTEGRAL* (Fig. 27 and Fig. 28) block and the *DERIVATIVE* (Fig. 29 and Fig. 30) block. Indeed, formalizing the requirements of these two functional units is also our contribution. As components of the composite *PID* block, these two FBs are used to compute, respectively, the integral and derivative terms. We write *ITERM.XOUT* and *DETERM.XOUT* to denote output values resulting from their last invocations.

The *INTEGRAL* block (Fig. 27 and Fig. 28) implements the integral of values of input *XIN* over time. The strategy of implementation is an approximation using partitions with right endpoints (with an input sampling period *CYCLE*). The integral result *XOUT* is reset to a preset value *X0* if the Boolean input flag *R1* is enabled. The integral is calculated if another input flag *RUN* is also enabled; otherwise, no new partitions are added (i.e., *XOUT* remains unchanged). Another output *Q* is set to *TRUE* while the integral is not reset; otherwise, *Q* is set to *FALSE*.

The *DERIVATIVE* block (Fig. 29 and Fig. 30) computes the differentiation of values of input *XIN* with respect to time. The rate of change is computed on the basis of: 1) an input sampling period *CYCLE*; and 2) values of input *XIN* at present and at the previous three clock ticks (i.e., *XIN* and XIN_{-i}, $i \in \{1, 2, 3\}$). The derivative result *XOUT* is reset to 0.0 if a Boolean flag *RUN* is disabled.

As indicated from the ST implementation of *PID* (Fig. 26) and tabular requirements of *INTEGRAL* and *DERIVATIVE* (Fig. 27 to 30), an input Boolean flag *AUTO* is set to distinguish cases in the computation. If *AUTO* is set *TRUE*, the controller attempts

[15] Also, we write x_{-n} to denote the previous value of variable x at the last nth tick.

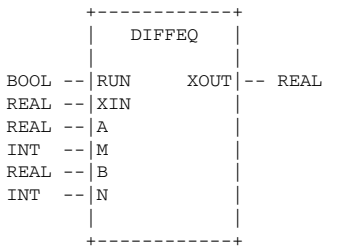



```
VAR_INPUT
  RUN   : BOOL ;  (* 0 = reset *)
  XIN   : REAL ;  (* Input to be differentiated *)
  CYCLE : TIME ;  (* Sampling period *)
END_VAR
VAR_OUTPUT
  XOUT  : REAL ;  (* Differentiated output *)
END_VAR
```

Fig. 29. Declarations of the block *DERIVATIVE* [9].

	Result
Condition	XOUT
R1	$\frac{3.0\times(XIN-XIN_{-3})+(XIN_{-1}-XIN_{-2})}{10.0\times CYCLE}$
¬R1	0.0

Fig. 30. Requirement of the block *DERIVATIVE*.




```
FUNCTION_BLOCK DIFFEQ
  VAR_INPUT
    RUN : BOOL ;                    (* 1 = run, 0 = reset *)
    XIN : REAL ;
    A   : ARRAY[1..127] OF REAL ; (* Output coefficients *)
    M   : INT  ;                    (* Length of output history *)
    B   : ARRAY[0..127] OF REAL ; (* Input coefficients *)
    N   : INT  ;                    (* Length of input history *)
  END_VAR
  VAR_OUTPUT XOUT : REAL := 0.0 ; END_VAR
  VAR (* NOTE : Manufacturer may specify other array sizes *)
    XI  : ARRAY [0..127] OF REAL; (* Input history *)
    XO  : ARRAY [0..127] OF REAL; (* Output history *)
    I   : INT ;
  END_VAR
  XO[0] := XOUT ; XI[0] := XIN ;
  XOUT  := B[0] * XIN ;
  IF RUN THEN
    FOR I := M TO 1 BY -1 DO
      XOUT := XOUT + A[I] * XO[I] ; XO[I] := XO[I-1];
    END_FOR;
    FOR I := N TO 1 BY -1 DO
      XOUT := XOUT + B[I] * XI[I] ; XI[I] := XI[I-1];
    END_FOR;
  ELSE
    FOR I := 1 TO M DO XO[I] := 0.0; END_FOR;
    FOR I := 1 TO N DO XI[I] := 0.0; END_FOR;
  END_IF ;
END_FUNCTION_BLOCK
```

Fig. 31. Declaration of the block *DIFFEQ* and its ST implementation [9].

to output *XOUT* closer to the desired set point value. Otherwise, another input *X0*, typically suppled by a transfer station, is used for a manual output adjustment.

However, observing the ST implementation of the *PID* block, the integral term is calculated through a division (of the output value *XOUT* from the FB instance *ITERM*) by the reset time *TR*. The type of *TR*, the set of real numbers, includes the possibility of zero that will lead to an undefined integral term. Similar to the case of the *AVERAGE* block (Section 5.2.5), this issue is reflected by an unprovable proof sequent generated by PVS, requiring that the value of *TR* can not be zero. As a result, our proposed solution is to redefine the data type for *TR* to exclude the value of zero. This serves the same purpose as an assumption that TR not be equal to zero.

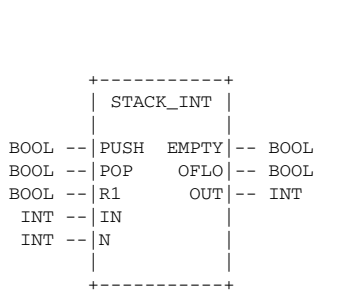
5.2.7. Inconsistent length setting of the DIFFEQ block

The *DIFFEQ* block (whose declaration is shown on the LHS in Fig. 31[16]) implements the difference equation, an invariant on the present and past input and output values. The output *XOUT* represents the weighted sum of values drawn from three categories: 1) the current value of input *XIN*; 2) the previous *N* values of *XIN*; and 3) the previous *M* values of *XOUT*. More precisely:

[16] The textual comments in the standard are in fact mistakenly placed to annotate the variables of input and output histories and coefficients, but since it does not affect the semantic verification, the corrected version is presented for readability.

Condition	Result: XOUT
RUN	$B_0 \cdot XIN + \sum_{i=1}^{N} B_i \cdot XIN_{-i} + \sum_{j=1}^{M} A_j \cdot XOUT_{-j}$
¬RUN	$B_0 \cdot XIN$

Fig. 32. Requirements of the block *DIFFEQ*.




```
FUNCTION_BLOCK STACK_INT
  VAR_INPUT PUSH, POP: BOOL R_EDGE ; (* Basic stack operations *)
                    R1 : BOOL ;      (* Over-riding reset *)
                    IN : INT ;       (* Input to be pushed *)
                    N  : INT ;       (* Maximum depth after reset *)
  END_VAR
  VAR_OUTPUT EMPTY : BOOL := 1 ;     (* Stack empty *)
             OFLO  : BOOL := 0 ;     (* Stack overflow *)
             OUT   : INT  := 0 ;     (* Top of stack data *)
  END_VAR
  VAR STK : ARRAY[0..127] OF INT;    (* Internal stack *)
      NI  : INT :=128 ;              (* Storage for N upon reset *)
      PTR : INT := -1 ;              (* Stack pointer *)
  END_VAR
(* Function Block body *)
END_FUNCTION_BLOCK
```

Fig. 33. Declaration of the block *STACK_INT* [9].

$$XOUT(t) = B_0 \cdot XIN(t) + \sum_{i=1}^{N} B_i \cdot XIN(t-i) + \sum_{j=1}^{M} A_j \cdot XOUT(t-j)$$

where A and B coefficients are inputs to the *DIFFEQ* block. Based on this formula, we formalize the requirement of the *DIFFEQ* block accordingly in Fig. 32. When the Boolean input flag *RUN* is set to *FALSE*, the value of *XOUT* is calculated by $B_0 \cdot XIN(t)$, just as if the input and output histories were empty.

The sum function (i.e., $\sum$) is implemented using a for-loop in the ST implementation (shown on the RHS in Fig. 31). When the input flag *RUN* is set to *TRUE*, two for-loops are used to compute the weighted sum of the (present and past) input and output values using coefficients stored in, respectively, the input array B and array A. Otherwise, another two for-loops are used to reset the input and output histories as all 0's.

However, observing the ST implementation, the type of input history length N (i.e., *INT*) is inconsistent with the length of input coefficients array, i.e., 128. More precisely, an issue of out-of-bound array indices would occur if $N \leq 0$ or $N > 127$. A similar issue also applies to the type of output history M and the length of the output coefficients array. Consequently, the implementation predicate in PVS cannot be type-checked. We propose to solve this problem by constraining the types of M and N: from *INT* to the interval between 1 and 127.

Moreover, lengths of the coefficient arrays A and B depend upon values of, respectively, M and N. To specify such constraints, we use dependent types in PVS:

```
M, N: subrange(1,127)
A: VAR ARRAY[subrange(0,M) -> real]
B: VAR ARRAY[subrange(0,N) -> real]
```

What other options are there for dealing with the issue of possible array index out of bound? Our reasoning was similar to that for the *AVERAGE* and *PID* blocks (Section 5.2.5 and Section 5.2.6). As accessing an area of memory that is outside the defined domain (i.e., the input history array) is dangerous, it should definitely be avoided. Again, depending on the specific application context of users, the check on a valid length of the input history array may be performed by either the implementor or the user of *DIFFEQ*. We have made that explicit by typing the variables appropriately.

5.3. Inconsistent implementations

5.3.1. Missing internal component of the STACK_INT block

The *STACK_INT* block implements a last-in-first-out (LIFO) data structure for storing integers. As illustrated in Fig. 33, it has five inputs (*PUSH*, *POP*, *R1*, *IN*, and *N*) and three outputs (*OUT*, *EMPTY* and *OFLO*). It may perform 1) a push operation (set by both the Boolean flag *PUSH* and the integer value *IN*), subject to a limit *N* on its maximum depth; 2) a pop operation (set by the Boolean flag *POP*); or 3) a reset operation (set by both the Boolean flag *R1* and the new maximum depth *N*). It outputs: 1) an integer value *OUT*, depending upon which operation was just performed; 2) a Boolean value *EMPTY* reporting if the current stack has become empty; and 3) a Boolean value *OFLO* indicating if the operation performed has caused a stack overflow.

```
IF R1 THEN
    OFLO := 0; EMPTY := 1; PTR := -1;
    NI := LIMIT (MN:=1,IN:=N,MX:=128); OUT := 0;
ELSIF POP & NOT EMPTY THEN
    OFLO := 0; PTR := PTR-1; EMPTY := PTR < 0;
    IF EMPTY THEN OUT := 0;
    ELSE OUT := STK[PTR];
    END_IF ;
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 ;
```

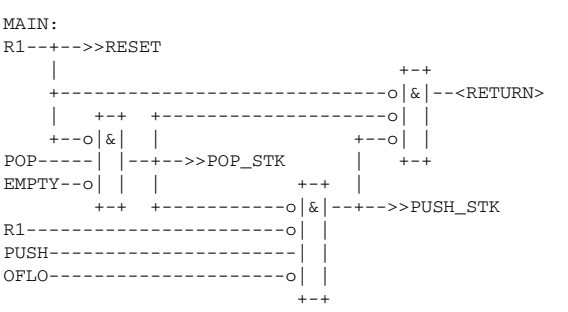


Fig. 34. ST and FBD implementations of the block *STACK_INT* [9].

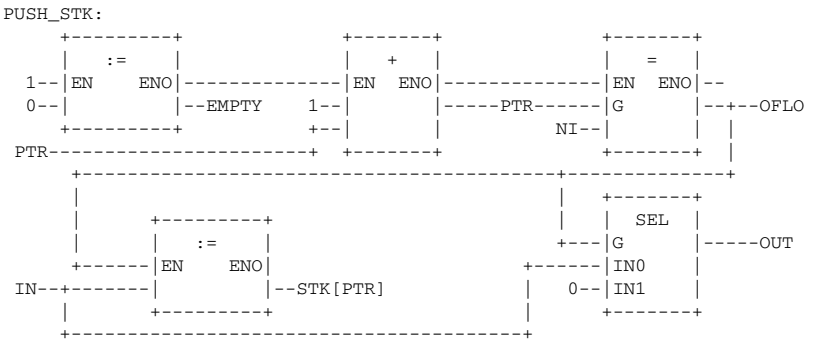


Fig. 35. *PUSH_STK* part of implementation of block *STACK_INT* in FBD [9].

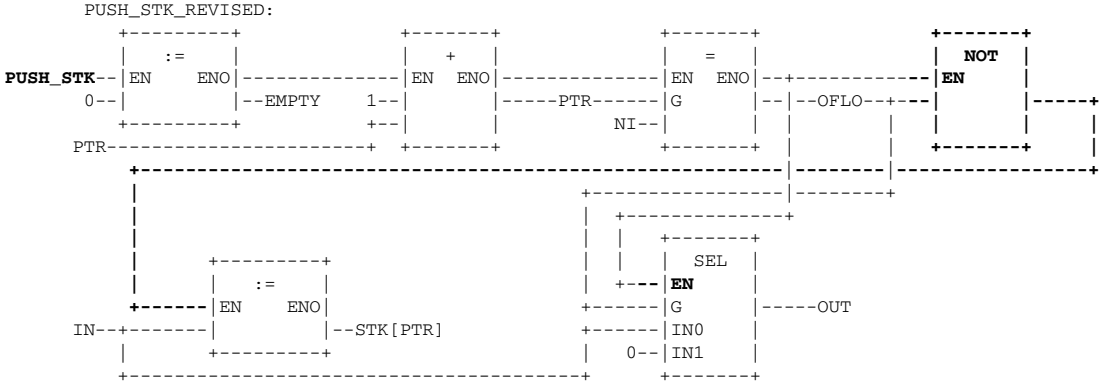


Fig. 36. *PUSH_STK* part of the FBD implementation of block *STACK_INT*: a negation block added.

IEC 61131-3 supplies both ST and FBD implementations for the *STACK_INT* block. Fig. 34 lists the complete ST implementation (on the LHS) and the `MAIN` part of the FBD implementation (on the RHS). For the FBD implementation, there are four separate parts connected with each other. The `MAIN` part is connected with three other sub-parts: `RESET`, `POP_STK` and `PUSH_STK`. Conditions for connecting these sub-parts correspond to those of the "if–then–else" statements in the ST implementation. The control of execution flow is transferred from the `MAIN` to each sub-part using the "jumps-to" notation (analogous to the standard go-to statement), i.e., `-->>RESET`, `-->>POP_STK`, and `-->>PUSH_STK`. The jumped-to locations are defined using labels, e.g., `PUSH_STK` in Fig. 35. We formulate this "jumps-to" mechanism in an easy to understand and straightforward manner: by defining a Boolean flag for each possible entry point.

Of particular interest is the `PUSH_STK` part of the FBD implementation (shown in Fig. 35), which is built up from four components: *MOVE* (:=), *ADDITION* (+), *EQUATION* (=) and *SELECTION* (*SEL*). Enabling input (*EN*) and output (*ENO*) are Boolean flags used to constrain the data flow in the FBD. The *MOVE* block, if enabled, passes on the input value as the output. The *ADDITION* block outputs the result of adding two input numbers. The *EQUATION* block outputs *TRUE* if two input numbers are equal. The *SELECTION* block selects one of the two input values based upon an input Boolean flag.

We found two issues in the *STACK_INT* block: 1) non-equivalent ST and FBD implementations; and 2) a missing FB in the FBD implementation.

Issue 1: Non-Equivalent Implementations. The ST and FBD implementations are actually not specified at the same level of abstraction. The use of *EN*/*ENO* constrains a specific (sequential) order of executing internal blocks in the FBD implementation. However, there is no such constraint in the ST implementation (as we parallelize assignment whenever possible). Consequently, we only attempt to prove that the implementation predicate of the FBD implementation implies that of the ST implementation in (7).

$$\begin{aligned}&\forall\, PUSH, POP, R1, IN, N \bullet \forall\, OUT, EMPTY, OFLO \bullet \\ &\quad STACK\_INT\_FBD\_IMPL(PUSH, POP, R1, IN, N, OUT, EMPTY, OFLO) \\ &\quad \Rightarrow STACK\_INT\_ST\_IMPL(PUSH, POP, R1, IN, N, OUT, EMPTY, OFLO)\end{aligned} \tag{7}$$

Issue 2: Missing FB in the FBD implementation. However, we failed to prove Equation (7) and the following unprovable proof sequent was generated in PVS.[17]

```
STACK_INT_fbd_implies_st_original.3.2.2.1 :
[-1] ...
  .
  .
  .
[-13] COND init(t) -> STK(PTR(t)) = 0,
          OFLO(t) -> STK(PTR(t)) = INP(t),
          ELSE -> STK(PTR(t)) = STK(PTR(pre(t)))
      ENDCOND
  |-------
[1] NOT R1(t) ∧ NOT (POP(t) ∧ NOT EMPTY(pre(t)))
      ∧ PUSH(t) ∧ NOT OFLO(pre(t)) ∧ NOT OFLO(t)
      IMPLIES STK(PTR(t)) = INP(t)
[2] init(t)
```

As introduced in Section 2.3, this proof sequent can be discharged by proving that the conjunction of antecedents implies the disjunction of consequents. Variables in the sequent above are skolem constants (i.e., arbitrary constants of the corresponding types) that are used to eliminate quantifiers. The *COND* construct is a multi-way extension to the polymorphic *IF–THEN–ELSE* construct in PVS. *t*, *PTR*, *INP*, and *PUSH* are all arbitrary (yet type-correct) constants. At the *t*th tick (Section 2.3) of time, an input request *PUSH* is made to push an integer *INP* onto a stack *STK*, and the push operation moves the internal stack pointer to a new position *PTR*.

In the above sequent, the antecedent is inferred from the behaviour of the FBD implementation (Fig. 35), and the consequence from that of the ST implementation (LHS in Fig. 34). Inspecting the sequent, we identified a missing negation from the antecedent. From the consequence, we observe that the push operation is performed and the pointer is updated accordingly (i.e., $STK(PTR(t)) = INP(t)$) when the stack would not overflow (i.e., $\neg OFLO(t)$). On the other hand, from the antecedent, the same push operation is not associated with the wrong guard (i.e., $OFLO(t)$), meaning that the push operation is performed when the stack is already full.

Similarly, by inspecting the FBD and ST code, we found that there is a missing negation block *NOT* between the *EQUATION* and the lower *MOVE* block (Fig. 35). That is, output *OFLO* from the *EQUATION* block (i.e., whether or not there is a stack overflow) should be negated so that it can be passed as the enabling condition of the lower MOVE block.

With the revised FBD implementation for `PUSH_STK` (illustrated in Fig. 36 with highlighted modifications), we are able to prove Equation (7). We also proved that both the ST and FBD implementations are consistent (Section 4.1). For the correctness theorem, as the logical implication is transitive, we only need to prove that the more abstract ST implementation conforms to the requirement:

$$\begin{aligned}&\forall\, PUSH, POP, R1, IN, N \bullet \forall\, OUT, EMPTY, OFLO \bullet \\ &\quad STACK\_INT\_ST\_IMPL(PUSH, POP, R1, IN, N, OUT, EMPTY, OFLO) \\ &\quad \Rightarrow STACK\_INT\_REQ(PUSH, POP, R1, IN, N, OUT, EMPTY, OFLO)\end{aligned} \tag{8}$$

Finally, we provide the complete requirement of the *STACK_INT* block in tabular expressions in the Appendix B. A table is created for each output variable: *EMPTY* (Fig. 42), *OFLO* (Fig. 43), and *OUT* (Fig. 44). In fact, we found that the state of internal variables is necessary for us to define the behaviour of the stack: *NI* (Fig. 39), *PTR* (Fig. 40), and *STK(PTR)* (Fig. 41).

[17] For clarity, we omit the irrelevant lines in this proof sequent.

We had a dilemma here regarding these inconsistent implementations. The obvious alternative is to fix the ST implementation for *STACK_INT*, making it agree with the behaviour of the FBD implementation. However, we would then have two implementations that consistently exhibit dangerous behaviour (e.g., pushing an item onto the stack only when it is already overflowed). Therefore, we think our proposed solution of fixing the problematic behaviour of the FBD implementation is more reasonable than the alternative.

6. Related work

There are many articles on formalizing and verifying PLC programs specified by programming languages covered in IEC 61131-3, such as sequential function charts (SFCs). Some approaches do this using 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 [25]; to automatically transform SFC programs into the synchronous data flow language of Lustre, amenable to mechanized support for checking properties [26]; to translate ST and FBD into a synchronized data-flow language SIGNAL to compile and analyze the verification of specifications [27]; 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) [29]; 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 [30]; to translate the algorithms of ladder diagrams (LDs) and timed FBs into finite state automata in which some properties are verified in SMV [31]; There is also an integration of SMV and Uppaal to handle, respectively, untimed and timed SFC programs [32].

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 [33]; and to formalize PLC programs using higher-order logic and to discharge safety properties in HOL [34]. 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 [35]. In [36], a trace function method (TFM) based approach is presented to solve the same problem as ours.

Our work is inspired by [37] in that the overall system behaviour is defined by taking the conjunction of those of internal components (circuits in [37] or FBs in our case). Our proposed solutions to the timing issues of the *PULSE* timer are consistent with [38]. 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.

The related work is motivated by the lack of formal semantics for the programming notations defined in the standard, and attempts to remove ambiguities. However, the issues we found are not reported in the related work. We situate our work from three aspects: 1) extent of case study; 2) value of results; and 3) practical implication.

Extent of case study. Our approach is able to handle all ST and FBD programs that are listed in [9], including its Annex F, whereas other work (e.g., [25,27,29,31]) focuses on limited language constructs or example FBs.

Value of results. To our knowledge, there is only a limited number of papers that illustrate the proposed verification approach via a case study, but none of them conduct a case study to the same extent as ours, let alone categorize the uncovered issues. In this paper, our results are based on the formalization and proofs of all FBs listed in the standard and its Annex F, whereas others (e.g., [32,33]) validate their approach via only a limited number of example blocks.

Practical implication. Our experiments are conducted on a mature theorem prover, and of a practical timing theory that are tailored to the execution context of FBs, whereas some related work does not even have tool support (e.g., [36]). Our results show that with the assistance of function tables and PVS, verification can be conducted in an industrial context with manageable mathematical artifacts (e.g., background theories, specifications, theorems, proofs, etc.). Nonetheless, there are existing works (e.g., [26–29,39,31,34]) that prove certain desired properties of FBs, similar to the additional requirements which we formulate as lemmas. More specifically: 1) work in [29] verifies some behavioural properties written in linear temporal logic; 2) work in [28] verifies the FBs against several safety requirements expressed as invariant properties; 3) work in [39] proves properties using CTL temporal logic based model checking of safety (i.e., boundness), liveness, and fairness; and 4) [32] proves SFC programs against a given set of requirements. However, none of those attempts to provide input-output requirements that are provably complete and disjoint.

7. Conclusion and future work

Many industrial control systems, especially safety-critical systems, require a high degree of confidence in the safety of the system. For those systems that are based upon PLCs, IEC 61131-3 provides several standard programming languages to describe the behaviour of the implemented function blocks (FBs). Since FBs have been growing in popularity, we present an approach to formalizing and verifying FBs using tabular expressions in the Prototype Verification System (PVS) environment. For STs, we provide a list of translation patterns. For the purpose of our verification, our rules of *ST-to-PVS* translation are designed to handle only the syntactic constructs of the ST language that are exploited in Annex F. That is, constructs that are supported by the ST language but not used in Annex F of the standard [9] (e.g., `CASE` statement, `WHILE` and `REPEAT` loops, etc.) are not currently handled by our translation rules. As mentioned earlier, our translation is still very useful since the Annex F example function blocks are commonly used in industry. As future work, we intend to extend our translation rules

to cover the remaining programming constructs of the ST language, so as to be able to verify FBs using those constructs. For FBDs, we formalize each basic FB as a predicate, allowing for deriving the semantics of composite FBs via logical compositions. More importantly, for demonstrating how we can argue the correctness of the ST and FBD implementations, we provide a black-box input-output requirement specification for each FB in the form of tabular expressions. Finally, as a demonstration of the applicability and value of our methodology, we formally verified the consistency and correctness of the whole FB library in IEC 61131-3 2003 and its informative annex in PVS.

Using our approach, we identified several kinds of potential issues in the FBs in the IEC 61131-3 standard and in the informative annex (Annex F) of the 2003 version: ambiguous block behaviour, missing assumptions, initialization failure, division by zero, mismatched variable types, and erroneous implementations. We provided our suggested solutions for each of these issues to demonstrate that the methodology can help us identify inconsistencies, ambiguities, missing information and even technical errors, that may be difficult or too time consuming to find through manual analysis or by simply examining the FBs in order to understand their behaviour. As indicated, the primary purpose of this work was to demonstrate that this type of formal approach can work on real industrial examples, and that the results are both useful and based on sound principles. As a side benefit, if methodologies like this move into accepted practice, and applied to published libraries of functions and FBs, the FBs in international standards such as IEC 61131-3 could be reused safely in PLC-based control system development, and hardware manufacturers' implementations of the FBs could be tested against the black-box requirement specifications of the FBs. This would strengthen the benefits of being in compliance with such a standard, for developers and manufacturers alike.

In future work, we will extend our list of ST-to-PVS translation patterns to cover more syntactic features. We will also adapt, and possibly extend, the approach for verifying FBs described in another FB related standard, IEC 61499 that fits well with distributed control systems.

Acknowledgements

This work is supported by funding from the Ontario Research Fund – Research Excellence program. The authors are grateful to Neeraj Kumar Singh and Hao Wang for their technical advice throughout the development of this paper. We would like to thank the anonymous reviewers for their detailed and very helpful comments. We took them to heart and think that the changes inspired by the reviews significantly improved this paper.

Appendix A. Example ST-to-PVS translations for Section 3.1.4

A.1. The HYSTERESIS function block

The ST implementation for the *HYSTERESIS* block is presented in Fig. 18.

```
HYSTERESIS [(IMPORTING Time) delta_t: posreal] : THEORY
BEGIN
  IMPORTING ClockTick[delta_t]
  XIN1 : VAR [tick -> real]
  XIN2 : VAR [tick -> real]
  EPS  : VAR [tick -> bool]
  Q    : VAR [tick -> bool]
  HYSTERESIS_st_impl (XIN1, XIN2, EPS, Q): bool =
    FORALL (t: tick):
      IF init(t) THEN
        Q(0) = FALSE
      ELSE
        Q(t) = TABLE
               | Q(pre(t))      | TABLE
                                  | XIN1(t) < (XIN2(t) - EPS(t))       | FALSE      ||
                                  | NOT XIN1(t) < (XIN2(t) - EPS(t)) | Q(pre(t)) ||
                                ENDTABLE ||
               | NOT Q(pre(t)) | TABLE
                                  | XIN1(t) > (XIN2(t) + EPS(t))       |  TRUE      ||
                                  | NOT XIN1(t) > (XIN2(t) + EPS(t)) | Q(pre(t)) ||
                                ENDTABLE ||
           ENDTABLE
      ENDIF
END HYSTERESIS
```

Fig. 37. Example *ST-to-PVS* translation: the *HYSTERESIS* block.

A.2. The DELAY function block

The ST implementation for the *DELAY* block is presented in Fig. 22.

```
DELAY [(IMPORTING Time) delta_t: posreal] : THEORY
BEGIN
  IMPORTING ClockTick[delta_t]
  IMPORTING structures@for_iterate
  RUN   : VAR [tick -> bool]
  XIN   : VAR [tick -> real]
  N     : VAR [tick -> int]
  XOUT  : VAR [tick -> real]
  DELAY_st_impl (RUN, XIN, N, XOUT): bool =
    EXISTS (X: [tick -> ARRAY[subrange(0, 127) -> real]],
            I: [tick -> int], IXIN: [tick -> int], IXOUT: [tick -> int]):
      FORALL (t: tick):
        IF init(t) THEN
          I(0) = 0 AND IXIN(0) = 0 AND IXOUT(0) = 0
        ELSE
          X(t) = TABLE
                   | RUN(t)       | X(pre(t)) WITH [(IXIN(t)) := XIN(t)] ||
                   | NOT RUN(t)   | for(0, N(t), X(pre(t)),
                                    LAMBDA (i : subrange(0, N(t)),
                                            X_t: ARRAY[subrange(0, 127) -> real]):
                                      X_t WITH [(i) := XIN(t)])          ||
                ENDTABLE
          AND
          IXIN(t) = TABLE
                      | RUN(t)       | mod(IXIN(t)    +1, 128)  ||
                      | NOT RUN(t)   |          N(t)             ||
                    ENDTABLE
          AND
          IXOUT(t) = TABLE
                       | RUN(t)       | mod(IXOUT(t)   +1, 128)  ||
                       | NOT RUN(t)   |             0             ||
                     ENDTABLE
          AND
          XOUT(t) = TABLE
                      | RUN(t)       | X(t)(IXOUT(t))  ||
                      | NOT RUN(t)   |      XIN(t)     ||
                    ENDTABLE
        ENDIF
END DELAY
```

Fig. 38. Example *ST-to-PVS* translation: the *DELAY* block.

Appendix B. Tabular requirement of the *STACK_INT* block from Section 5.3.1

To define the requirements of the *STACK_INT* block, we consider its three output variables (*EMPTY*, *OFLO*, and *OUT*) and three internal variables (*NI*, *PTR*, and *STK*). The stack is represented using a zero-based array *STK* with a preset size NI. A pointer *PTR* (of *STK*) references the last item pushed onto the stack.

B.1. Internal variables

The value of *NI* restricts the maximum capacity of the stack. Its value may be set upon a reset operation, where an internal function *LIMIT* is used to return a value *N*, bounded by some preset (lower and upper) limits. Its value stays unchanged until another reset operation is requested.

Since indices of the array representation of the stack start with 0, the initial value of the pointer value *PTR* (for an empty stack) is set to -1. The pointer position may shift to the left or to the right when, respectively, a pop operation (from a non-empty stack) or a push operation (not resulting in a stack overflow) is performed.

For the array representation *STK* of stack, we are only interested in querying the value stored at index *PTR*. When a valid push operation occurs (not resulting in a stack overflow), the value of *STK(PTR)* is set to that of the input *IN*.

Condition	Result: NI
R1	LIMIT(1,N,128)
¬R1	NC

Fig. 39. Requirement for internal *NI* of the block *STACK_INT*.

Condition			Result: PTR
R1			-1
¬R1	POP $\wedge$ ¬EMPTY$_{-1}$		PTR$_{-1} - 1$
	¬POP $\vee$ EMPTY$_{-1}$	PUSH $\wedge$ ¬OFLO$_{-1}$	PTR$_{-1} + 1$
		¬PUSH $\vee$ OFLO$_{-1}$	NC

Fig. 40. Requirement for internal *PTR* of the block *STACK_INT*.

Condition	Result: STK(PTR)
¬ R1 $\wedge$ ¬ (POP $\wedge$ ¬ EMPTY$_{-1}$) $\wedge$ PUSH $\wedge$ ¬ OFLO$_{-1}$ $\wedge$ ¬ OFLO	IN
R1 $\vee$ (POP $\wedge$ ¬ EMPTY$_{-1}$) $\vee$ ¬ PUSH $\vee$ OFLO$_{-1}$ $\vee$ OFLO	NC

Fig. 41. Requirement for internal *STK* of the block *STACK_INT*.

Condition			Result: EMPTY
R1			1
¬R1	POP $\wedge$ ¬EMPTY$_{-1}$	PTR $<$ 0	1
		PTR $\geq$ 0	0
	¬POP $\vee$ EMPTY$_{-1}$	PUSH $\wedge$ ¬OFLO$_{-1}$	0
		¬PUSH $\vee$ OFLO$_{-1}$	NC

Fig. 42. Requirement for output *EMPTY* of the block *STACK_INT*.

Condition				Result: OFLO
R1				0
¬R1	POP $\wedge$ ¬EMPTY$_{-1}$			0
	¬POP $\vee$ EMPTY$_{-1}$	PUSH $\wedge$ ¬OFLO$_{-1}$	PTR $=$ NI	1
			PTR $\neq$ NI	0
		¬PUSH $\vee$ OFLO$_{-1}$		NC

Fig. 43. Requirement for output *OFLO* of the block *STACK_INT*.

B.2. Output variables

The output *EMPTY* is a Boolean flag indicating if the current stack is empty. The current stack may be reinitialized (to be an empty stack) by a reset operation (by enabling another Boolean flag *R1*). When a push operation occurs, as long as there was not previously a stack overflow (i.e., $\neg OFLO_{-1}$), then the stack remains (or becomes) non-empty (i.e., $\neg EMPTY$). When a pop operation occurs, if the stack was previously left with only one item, then the stack becomes empty (by setting the internal pointer *PTR* to -1); otherwise, when more than one items were previously left, then the stack remains non-empty.

The output *OFLO* is a Boolean flag indicating if the current operation has resulted in a stack overflow. Obviously, a stack overflow can occur only when the stack previously reached its maximum capacity *NI*[18] and a push operation is performed.

[18] The internal pointer variable starts at 0, so when it reaches *NI* -1, the stack is full.

Condition				Result: OUT
R1				0
¬R1	POP ∧ ¬EMPTY$_{-1}$	EMPTY		0
		¬EMPTY		STK(PTR)
	¬POP ∨ EMPTY$_{-1}$	PUSH ∧ ¬OFLO$_{-1}$	¬OFLO	IN
			OFLO	0
		¬PUSH ∨ OFLO$_{-1}$		NC

Fig. 44. Requirement for output *OUT* of block *STACK_INT*.

Once there is a stack overflow, the value of *OFLO* holds until a reset operation is requested. Otherwise, the stack remains in its normal state (i.e., ¬*OFLO*).

The output *OUT* indicates the top of the stack. The value of *OUT* is set to 0 when either 1) the stack is reinitialized to be empty; 2) the stack is currently empty; or 3) the current push operation results in a stack overflow. Otherwise, popping from a non-empty stack (with more than one item) results in *OUT* being set to where the current *PTR* points to (i.e., *STK*(*PTR*)); pushing onto a stack results in *OUT* being set to the value just added to the stack (i.e., input *IN*).

References

[1] E. Bakhmach, O. Siora, V. Tokarev, S. Reshetytskyi, V. Kharchenko, V. Bezsalyi, FPGA-based technology and systems for I&C of existing and advanced reactors, in: Proceedings of an International Conference on Opportunities and Challenges for Water Cooled Reactors in the 21 Century, International Atomic Energy Agency, 2009, p. 173, IAEA-CN-164-7S04.

[2] Special Committee 205 of RTCA, DO-178C: software considerations in airborne systems and equipment certification, 2011, http://dx.doi.org/10.1109/IEEESTD.2010.5542302.

[3] IEEE, 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), 2010, http://dx.doi.org/10.1109/IEEESTD.2010.5542302.

[4] D.L. Parnas, J. Madey, Functional documents for computer systems, Sci. Comput. Program. 25 (1995) 41–61.

[5] D.L. Parnas, J. Madey, M. Iglewski, Precise documentation of well-structured programs, IEEE Trans. Softw. Eng. 20 (1994) 948–976.

[6] M. Lawford, J. McDougall, P. Froebel, G. Moum, Practical application of functional and relational methods for the specification and verification of safety critical software, in: Proceedings of AMAST 2000, in: LNCS, vol. 1816, Springer, 2000, pp. 73–88.

[7] A. Wassyng, R. Janicki, Tabular expressions in software engineering, in: Proceedings of ICSSEA'03, vol. 4, Paris, France, 2003, pp. 1–46.

[8] S. Owre, J.M. Rushby, N. Shankar, PVS: a prototype verification system, in: CADE, in: LNCS, vol. 607, 1992, pp. 748–752.

[9] IEC, 61131-3 Ed. 2.0 en:2003: programmable controllers — part 3: programming languages, 2003, International Electrotechnical Commission.

[10] IEC, 61131-3 Ed. 3.0 en:2013: programmable controllers — part 3: programming languages, 2013, International Electrotechnical Commission.

[11] L. Pang, C.-W. Wang, M. Lawford, A. Wassyng, Formalizing and verifying function blocks using tabular expressions and PVS, in: Formal Techniques for Safety-Critical Systems, in: Communications in Computer and Information Science, vol. 419, Spring. 2013, pp. 163–178.

[12] R. Janicki, D.L. Parnas, J. Zucker, Tabular representations in relational documents, in: Relational Methods in Computer Science, Springer Verlag, 1997, pp. 184–196.

[13] D.L. Parnas, A generalized control structure and its formal definition, Commun. ACM 26 (1983) 572–581.

[14] R. Janicki, A. Wassyng, Tabular expressions and their relational semantics, Fundam. Inform. 67 (2005) 343–370.

[15] Y. Jin, D.L. Parnas, Defining the meaning of tabular mathematical expressions, Sci. Comput. Program. 75 (2010) 980–1000.

[16] A. Wassyng, M. Lawford, T.S.E. Maibaum, Software certification experience in the Canadian nuclear industry: lessons for the future, in: EMSOFT, 2011, pp. 219–226.

[17] NASA Langley PVS libraries official website, 2014, http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html.

[18] X. Hu, M. Lawford, A. Wassyng, Formal verification of the implementability of timing requirements, in: FMICS, in: LNCS, vol. 5596, Springer, 2009, pp. 119–134.

[19] X. Hu, Proving implementability of timing properties with tolerance, Ph.D. thesis, McMaster University, Department of Computing and Software, 2008.

[20] C. Eles, M. Lawford, A tabular expression toolbox for Matlab/Simulink, in: NASA Formal Methods, 2011, pp. 494–499.

[21] N. Shankar, S. Owre, J.M. Rushby, D.W.J. Stringer-Calvert, PVS Prover Guide, Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.

[22] M. Lawford, H. Wu, Verification of real-time control software using PVS, in: Proceedings of the 2000 Conference on Information Sciences and Systems, vol. 2, Dept. of Electrical Engineering, Princeton University, Princeton, NJ, 2000, pp. TP1–13–TP1–17.

[23] A. Camilleri, M. Gordon, T. Melham, Hardware verification using higher-order logic, Technical report UCAM-CL-TR-91, Cambridge Univ. Computer Lab, 1986.

[24] C. Muñoz, R. Demasi, Advanced theorem proving techniques in PVS and applications, in: Tools for Practical Software Verification, in: LNCS, vol. 7682, Springer, 2012, pp. 96–132.

[25] A. Mader, H. Wupper, Timed automaton models for simple programmable logic controllers, in: ECRTS, IEEE, 1999, pp. 114–122.

[26] A. Kabra, A. Bhattacharjee, G. Karmakar, A. Wakankar, Formalization of sequential function chart as synchronous model in Lustre, in: NCETACS, 2012, pp. 115–120.

[27] F. Jimenez-Fraustro, E. Rutten, A synchronous model of IEC 61131 PLC languages in SIGNAL, in: 13th Euromicro Conference on Real-Time Systems, 2001, pp. 135–142.

[28] D. Soliman, K. Thramboulidis, G. Frey, Transformation of function block diagrams to Uppaal timed automata for the verification of safety applications, Annu. Rev. Control (2012).

[29] G. Canet, S. Couffin, J.J. Lesage, A. Petit, P. Schnoebelen, Towards the automatic verification of PLC programs written in instruction list, in: IEEE International Conference on Systems, Man and Cybernetics, 2000, pp. 2449–2454.

[30] E. Németh, T. Bartha, Formal verification of safety functions by reinterpretation of functional block based specifications, in: FMICS, Springer, 2009, pp. 199–214.

[31] O. Rossi, P. Schnoebelen, Formal modeling of timed function blocks for the automatic verification of ladder diagram programs, in: 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, ADPM, Dortmund, Germany, 2000, pp. 177–182.

[32] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, O. Stursberg, Verification of PLC programs given as sequential function charts, in: Integration of Software Specification Techniques for Applications in Engineering, in: LNCS, vol. 3147, Springer, Berlin Heidelberg, 2004, pp. 517–540.
[33] J.O. Blech, S.O. Biha, On formal reasoning on the semantics of PLC using Coq, CoRR, arXiv:1301.3047, 2013.
[34] N. Völker, B.J. Krämer, Automated verification of function block-based industrial control systems, Sci. Comput. Program. 42 (2002) 101–113.
[35] J.-M. Roussel, J. Faure, An algebraic approach for PLC programs verification, in: 6th International Workshop on Discrete Event Systems, 2002, pp. 303–308.
[36] Z. Liu, D. Parnas, B. Widemann, Documenting and verifying systems assembled from components, Front. Comput. Sci. China 4 (2010) 151–161.
[37] T. Melham, Abstraction mechanisms for hardware verification, in: VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers, 1987, pp. 129–157.
[38] K.H. John, M. Tiegelkamp, IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids, 2nd ed., Springer, 2010.
[39] E. Németh, T. Bartha, Formal verification of safety functions by reinterpretation of functional block based specifications, in: Formal Methods for Industrial Critical Systems, in: LNCS, vol. 5596, Springer, Berlin/Heidelberg, 2009, pp. 199–214.