An Engineering Methodology for the Formal Verification of Function Block Based Systems

An Engineering Methodology for the Formal Verification of Function Block Based Systems

By

Linna Pang, B.Eng., M.Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

McMaster University

DOCTOR OF PHILOSOPHY (2015) MCMASTER UNIVERSITY
(Computer Science) Hamilton, Ontario

TITLE: An Engineering Methodology for the Formal Verification of Function Block Based Systems

AUTHOR: Linna Pang
B.Eng.,
Beijing Information Technology Institution, China
M.Eng.,
Beijing University of Posts and Telecommunications, China

SUPERVISOR: Dr. Alan Wassyng and Dr. Mark Lawford

NUMBER OF PAGES: xvi, 184

To my parents

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 PLC components that can be composed 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.

Unfortunately, the set of programming notations defined in IEC 61131-3 lack well-defined formal semantics. As a result, tool vendors and users of PLCs may have inconsistent interpretations of the expected system behaviour. To address this issue, we propose an engineering method for formally verifying the conformance of candidate implementations of FBs (and their compositions) to their high-level, input-output requirements. The proposed method is sufficiently general to handle FBs supplied by IEC 61131-3, and industrial FB applications involving real-time requirements. Our method involves several steps. First, we use tabular expressions to ensure the completeness and disjointness of the requirements for the FB. Second, we formalize the candidate implementation(s) of the FB in question. Third, we state and prove theorems regarding the consistency and correctness of the FB. All three steps are performed using the Prototype Verification Systems (PVS) proof assistant.

As a first case study, we apply our approach to the IEC 61131-3 standard to examine the entire library of FBs and their supplied implementations described in structured text (ST) and function block diagrams (FBDs). As a second case study, we apply our approach to two realistic sub-systems taken from the nuclear domain. Applying the proposed method, we identified three kinds of issues: ambiguous behavioural descriptions, missing assumptions, and erroneous implementations. Furthermore, we suggest solutions to these issues.

Acknowledgments

First and foremost, I offer my most sincerest gratitude to my supervisors, Prof. Alan Wassyng and Prof. Mark Lawford, who helped me to shape this thesis throughout with their patience, insight, and industrial experiences. One simply could not wish for a better or more supporting supervisors, who walked me through these years.

Thanks to my other supervisory committee members, Dr. Tom Maibaum and Dr. Ned Nedialkov for their invaluable thoughts and advice on my research work. Thanks to Dr. Jackie Wang for his collaboration on my publications, and helpful comments for improvements to this thesis.

Financial support during my graduate studies has been generously provided by the Ontario Research Fund (ORF) for Research Excellence and McMaster University graduate scholarship.

Also, I would like to thank my friends for their moral support, especially to Lucian M. Patcas, Qinglei Zhang, Hao Lin, Mustafa Elsheikh and Neeraj Singh, throughout the completion of my doctoral journey.

Above all, a very special thank you goes to my beloved parents Boyou and Huifang for being by my side all these years.

Contents

List of Figures

List of Tables

Glossary

$\mathbb{R}^+$	27	Positive real numbers
$\mathbb{N}$	29	Natural numbers ($\{0, 1, 2, ...\}$)
$\vdash$	23	Syntactic entailment
$f_1 \circ f_2$	37	Functional composition
d	30	Time duration of *Held_For* operator
δL	30	Left tolerance of time duration d
δR	30	Right tolerance of time duration d
δt	28	Arbitrarily small time interval between time ticks
FB	17	Function Block
IEC 61131	18	International Electrotechnical Committee standard 61131
PLC	17	Programmable Logic Controller
CPU	17	Central Processing Unit
FBD	18	Function Block Diagram
LD	18	Ladder Diagram
IL	18	Instruction List
ST	18	Structured Text
SFC	18	Sequential Function Chart
TON	44	Timer On-Delay
TOF	45	Timer Off-Delay
TP	46	Timer Pulse
IEC 61499	145	International Electrotechnical Committee standard 61499
SRS	78	Software Requirements Specification
SDD	78	Software Design Description

PVS Glossary

PVS	21	Prototype Verification System
TCC	25	Type Correctness Conditions
$Held\_For$	47	A PVS operator defined for sustained events
$Held\_For\_S$	30	A PVS operator defined for sustained events at sample point
$Held\_For\_I$	31	A PVS operator defined as an intermediate operator between the implementation and its requirements of *Held_For with tolerance*
$Timer\_S$	31	A PVS operator implements the $Held\_For\_S$ at sample level
$Timer\_I$	31	A PVS operator implements the $Held\_For\_I$ at tick level
$TimerGeneral\_I$	31	A PVS general theorem verifying an implementation $Timer\_I$ of $Held\_For\_I$

Chapter 1

Introduction

1.1 General Background and Motivation

Many industrial control systems with traditional analog equipment have been replaced using components that are based upon programmable logic controllers (PLCs) to address increasing market demands for high quality (Bakhmach et al., 2009). Function blocks (FBs) are basic design units that implement the behaviour of a PLC, where each FB is a reusable component for building new, more sophisticated components or systems. The search for higher reliability and predictability may be realized if the FBs are pre-certified with respect to an international standard such as IEC 61131-3 (IEC, 2003; IEC, 2013). The IEC 61131-3 standard defines a set of PLC programming languages, and describes a set of FBs commonly used in the industry as examples of how to use the PLC programming languages (the 2003 version of IEC 61131-3 includes an informative annex – Annex F – with a significant number of FBs, but Annex F was removed in the 2013 version of the Standard). Standards such as DO-178C (Special Committee 205 of RTCA, 2011a) (in the aviation domain) and IEEE 7-4.3.2 (IEEE, 2010) (in the nuclear domain) list acceptance criteria of mission- or safety-critical systems for practitioners to comply with. Two important criteria are that: (1) the system requirements are precise and complete; and (2) the system implementation exhibits behaviour that conforms to these requirements. In one of its supplements, DO-333 (Special Committee

205 of RTCA, 2011b), DO-178C (Special Committee 205 of RTCA, 2011a) advocates the use of formal methods to construct, develop, and reason about the mathematical models of system behaviours.

Formal methods have been successfully employed in the development of many safety-critical systems to provide verifiable specifications for requirements, designs and implementations, and to automate the verification and/or validation of the system. Formal methods can dramatically enhance software quality and reduce (yet not eliminate) reliance on testing. Formal method verification and testing are independent activities. Verification by formal analysis does not have the same challenges that testing does, for example, exhaustive testing is only feasible in very narrow circumstances. Hence, we can gain confidence in the correctness of our system by performing both formal verification and testing. This effectively reduces our reliance on testing as the only means of verification.

There have been many approaches to formalizing and verifying PLC programs. The employment of formal methods to verify computer systems, championed by (Hoare, 1969), has become the core of a "grand challenge" taken up by the computer science community (Woodcock and Banach, 2007). In this regard, we believe that there are three major challenges in formalizing and verifying FB-based systems: (1) creating a unified framework within which requirement and implementation specifications are formalized in a way that facilitates formal proofs; (2) implementing a practical approach for verifying that a FB-based systems conforms to its (hard real-time) requirements; and (3) developing a pre-certified block repository that can be reused safely in practice.

In this thesis, we present a systematic verification approach for FBs, including FBs that exhibit real-time behaviour. IEC 61131-3 (IEC, 2003), including its Annex F, provides a candidate block library for us to apply our approach. We had two reasons for choosing IEC 61131-3 for our case study. First, 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. Second, although the FBs of Annex F are not technically part of the Standard, since the appendix is labeled

"Informative", the entire document, including all annexes, has become a *de facto* standard for FBs as many FB libraries, including those provided by PLC vendors, are heavily based on the FBs from Annex F, as well as those described in the body of the Standard.

Our task is complicated somewhat by the fact that IEC 61131-3 does not adopt a consistent and unified programming notation for defining standard functions and FBs. In many cases, multiple programming languages are used to describe a single FB. The Standard defines five languages for use on PLCs, of which the two most commonly used languages are *Structured Text* (ST) and *Function Block Diagrams* (FBDs). However, these five languages are implementation-oriented notations that lack a unified formal semantics, and are thus not adequate for capturing the precise input-output relation that satisfies both a coverage property (i.e., the input domain covers all cases) and a mutual exclusion property (i.e., no two cases overlap). Moreover, in the absence of high-level, black-box requirements, it is not possible to formally establish that these implementations are consistent with their input-output requirements. Thus, we provide high-level, input-output requirements for FBs that are missing from IEC 61131-3. Such formal, compositional requirements are developed for the purpose of formalizing and verifying sophisticated, composite candidate FB implementations. Moreover, formal descriptions are amenable to mechanized support such as PVS (Owre et al., 1992). 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 a verification separately.

As indicated earlier, two versions of IEC 61131-3 are cited here. The earlier version (IEC, 2003) has been in use since 2003. In 2013, a new version of the Standard was issued (IEC, 2013), 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 our methodology. Our intent is to illustrate how our methodology raised questions about some of IEC 61131-3 FBs. Since the Standard does not attempt to define the required behaviour of each FB in any formal sense, we had to create formalisms based on experience that would be consistent with industrial norms and on what we deduced

was the intended behaviour of the FB. In any case, our primary motivation here is to demonstrate our methods, not to criticize the Standard. We hope that FB users 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 improve the quality of FB-based designs.

1.2 Problem Statement, Objectives, and Methodology

This thesis aims to develop a systematic, formal approach to specifying the requirements and design of a FB, and verifying the conformance of a FB implementation with its requirements. In particular we adopt the formal notation of tabular expressions (a. k. a. function tables or tables) that have proven to be both practical and effective in formally documenting system requirements in industry (Lawford et al., 2000; Wassyng and Janicki, 2003), to formalize the requirements of FBs. We also report on using PVS, a general purpose theorem prover, that provides an integrated reasoning environment with mechanized support for writing specifications using tabular expressions and (higher-order) predicates, to (interactively) prove that implementations satisfy the tabular requirements using sequent calculus style deductions. In this thesis we focus on FBs that are described in the more commonly used languages of ST and FBDs.

More precisely, we set out to achieve the following three objectives:

1. **Specify the black-box input-output relation for each FB.** Currently the intended behaviour of a FB is provided either by a natural language description or a timing diagram in IEC 61131-3. The primary issue for specifying such a relation is identifying the intended behaviour and documenting the behaviour in a rigorous notation. We analyzed the behavioural FB descriptions (i.e., in ST, FBD or timing diagram) and informal descriptions in the Standard of the FB. For timer blocks, we formalized the behaviour using pre-verified timing operators to address the ambiguities introduced by timing diagrams. We then described our

interpretation of the intended behaviour of a FB (i.e., its requirements specification) using tabular expressions. One may disagree with our chosen requirements specification and 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 of the FBs in the Standard. We based our requirements on the informal description of behaviour included in the Standard, as well as on consultation with colleagues who have extensive industrial experience (Wassyng et al., 2011).

2. **Formalize the FB implementations.** We focused on those FBs that are described either in ST, FBD or both. For the FBD implementation (i.e., implementations that contain functions, basic FBs, and/or other pre-developed composite FBs), we analyzed the connection between components and conjoined the requirement predicates for components (obtained by the first objective) in a compositional manner. For the ST implementation, we formalized several translation rules, sufficient for the ST descriptions supplied by IEC 61131-3.

3. **Verify the conformance between the supplied FB implementations (formalized by the second objective) and their intended behaviours (obtained by the first objective).** A potential way to verify the behaviour of FB-based systems is to use formal methods. How to use formal methods in an effective way is still a topic of research in academia and discussion in industry. Such formal verification is reasonable if the complexity of reasoning can be handled well. The verification of FB behaviour can be complicated due to the use of complex design constructs. For example, nested for-loops in the case of ST implementations or complex timing behaviour implemented by the use of timer blocks, in the case of FBD implementations. We used logical implication to verify that an implementation satisfies its requirements. We also proved the equivalence of ST and FBD implementations, if they were both supplied for a given FB. Our verification approach is also applicable to FBs with

more complex timing requirements. To be of practical use, the timing requirements for these blocks incorporate tolerances. To assist others in performing proofs in PVS, we have provided proof strategies for our approach.

1.3 Related Work

In this section we review several related works on formal modelling and verification of PLC programs in the literature. We investigate the related works in two categories, model checking and theorem proving (Section 1.3.1). We also compare our work with the related works in terms of three aspects: extent of case study, value of results and practical implication (Section 1.3.2).

1.3.1 Related Work on Verifying Function Blocks

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:

- by formalizing a subset of the language of instruction lists (ILs) using timed automata, and verifying real-time properties in the Uppaal model checker (Mader and Wupper, 1999);
- by automatically transforming SFC programs into the synchronous data flow language Lustre, which is amenable to mechanized support for checking properties (Kabra et al., 2012);
- by translating ST and FBD into a synchronized data-flow language SIGNAL to compile and analyze the verification of specifications (Jimenez-Fraustro and Rutten, 2001);
- by transforming FBD specifications to Uppaal formal models to verify safety properties of applications in the industrial automation domain (Soliman et al., 2012);

- by providing the formal operational semantics of ILs, which is encoded into the input language for the symbolic model checker Cadence SMV, and then verifying rich behavioural properties written in linear temporal logic (LTL) (Canet et al., 2000);
- by providing 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 (Németh and Bartha, 2009); and
- by translating the algorithms of ladder diagrams (LDs) and timed FBs into finite state automata for which some properties are verified in the model checker SMV (Rossi and Schnoebelen, 2000).

There is also an integration of SMV and Uppaal to handle, respectively, untimed and timed SFC programs (Bauer et al., 2004).

It is well-known that the technique of model checking is only applicable to relatively small systems. In contrast to model checking, another approach is to use the verification environment of a theorem prover:

- to check the correctness of SFC programs, automatically generated from a graphical front-end, in Coq (Blech and Biha, 2013); and
- to formalize PLC programs using higher-order logic to discharge safety properties in HOL (Völker and Krämer, 2002).

Also, an algebraic approach for PLC program verification is presented in (Rousel and Faure, 2002). In (Liu et al., 2010), a trace function method (TFM) based approach is presented to verify the conformance between combined components and system requirements.

1.3.2 Summary and Comparison

These works are similar to ours in that PLC programs are formalized, and the intention is to develop support for mechanized verifications of implementations.

Our work is inspired by (Melham, 1987) in that the overall system behaviour is defined by taking the conjunction of internal components (circuits in (Melham, 1987) or FBs in our case). Our solution to the timing issues of the *PULSE* timer is consistent with (John and Tiegelkamp, 2010). 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 above related work is motivated by the lack of formal semantics for the programming notations defined in the Standard, and attempts to remove ambiguities. Our work is different in three ways: (1) extent of the case study; (2) value of the results; and (3) the practical implication.

Extent of the Case Study. Our approach is able to handle all ST and FBD programs that are listed in (IEC, 2003), including its Annex F, whereas other work (Mader and Wupper, 1999; Jimenez-Fraustro and Rutten, 2001; Canet et al., 2000; Rossi and Schnoebelen, 2000) focuses on limited language constructs or example FBs.

Value of the Results. To our knowledge, there are only a limited number of papers that illustrate the proposed verification approach via a case study, but none of them conducts a case study to the same extent as ours, let alone categorizes all of the issues uncovered. In this thesis, our results are based on the formalization and proofs of all FBs listed in the Standard and its Annex F, whereas others (Bauer et al., 2004; Blech and Biha, 2013) validate their approach via only a limited number of example blocks.

The Practical Implication. Our experiments are conducted on the basis of 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 (Liu et al., 2010). 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 (Kabra et al., 2012; Jimenez-Fraustro and Rutten, 2001; Soliman et al., 2012; Canet et al.,

2000; Németh and Bartha, 2009; Rossi and Schnoebelen, 2000; Völker and Krämer, 2002) that prove certain desired properties of FBs, similar to the additional requirements which we formulate as lemmas. More specifically: (1) work in (Canet et al., 2000) verifies some behavioural properties written in linear temporal logic; (2) work in (Soliman et al., 2012) verifies the FBs against several safety requirements expressed as invariant properties; (3) work in (Németh and Bartha, 2009) proves properties using CTL temporal logic based model checking of safety, liveness, and fairness; and (4) (Bauer et al., 2004) proves SFC programs against a given set of requirements. However, none of these attempts to provide input-output requirements that are provably complete and disjoint.

1.4 Scope

Our aim was to explore and develop a practical and effective approach to the formal verification of FB-based computer system. Our approach relies upon and incorporates existing notations (programming notations listed in IEC 61131-3 and tabular expressions), and the PVS theorem prover that supports higher-order logic. Our verification approach is based on the formalization of requirements and implementation of FB-based computer system. We are dealing with design level verification, not at the level of actual machine level implementation. Hence, the issues of floating point arithmetic are out of scope for this thesis. We can now summarize our approach and contributions of specifying and verifying FBs with reference to Figure 1.1.

As shown on the left of the figure, a FB will typically have a natural language description of the block behaviour accompanied by a detailed implementation given in the ST or FBD languages, 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 (Chapter 3). The ST and FBD implementations are formalized as predicates in PVS as shown on the right of the figure, again making use of tables (Chapter 5). In the case when there are two implementations for a FB, one in a FBD and the other in ST, we attempt to prove their (functional) equivalence in PVS. We

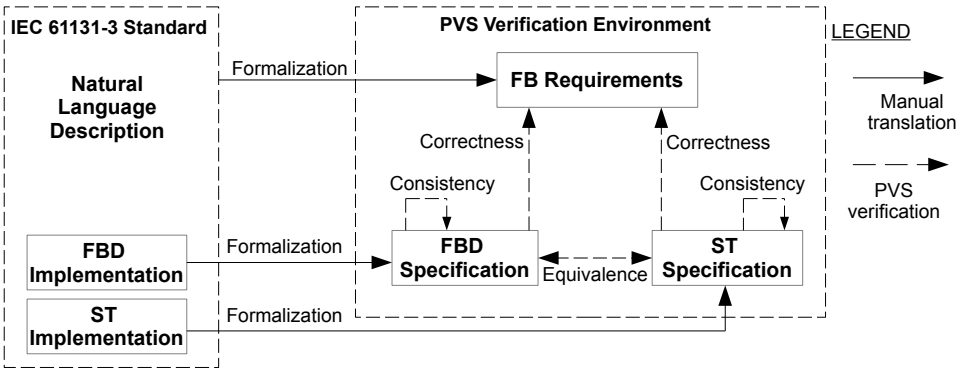


Figure 1.1: Framework

also use PVS to attempt to prove the *consistency*[1] and *correctness* of each implementation with respect to its FB requirements (Chapter 6). We took IEC 61131-3 (IEC, 2003) FBs as a case study and provided proof strategies for verifying consistency and correctness of FBs (Chapter 7). For FBs that are distinguished by their real-time behaviour (i.e., timers), we reused a pre-verified timing operator, *Held_For* (Hu, 2008), to formalize three IEC 61131-3 timer blocks (Chapter 4).

1.5 Contributions

Through the fulfilment of the above objectives, we now summarize our approach and contributions:

1. **Formalization of input-output requirements for FBs**: We formalized the black-box requirements of FBs using tabular expressions. We parameterized the inputs and outputs by time instants, and characterized the temporal relation between inputs and outputs as a universal quantification over discrete time instants. By embedding tabular re-

[1]In this thesis, 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.

quirements in predicates in PVS, the prover generates proof obligations for completeness and disjointness. For real-time FBs (built from timers), we reuse the timing operator *Held_For_I* (Hu, 2008) within requirements tables to formalize timing behaviours.

2. **Formalization of FBD implementations**: We present a compositional approach formalizing the FBD implementations. The implementation specification is a predicate formed by taking the conjunction of the predicates of the internal components used to create the FBD. The interconnections between components are existentially quantified. For real-time FBs (built from timers), we apply the same approach, with the difference that we reuse the timing operator *Timer_I* (Hu, 2008), suited to implementations rather than requirements, to formalize the timer component of the implementation.

3. **Formalization of ST implementations**: We developed a limited set of translation rules that suffice to translate the ST implementations that are supplied by Annex F of IEC 61131-3 (IEC, 2003) as well as those described in the body of the Standard into their equivalent expressions in PVS. Since the IEC 61131-3 FBs are commonly used in industry, our rules can be applied to FBs that are representative of industrial use. This step, that of formalizing the implementation in PVS, allows us to verify the correctness and consistency of ST implementations with respect to their input-output requirements.

4. **Formal verification of FB implementations**: We present a technique verifying the correctness and consistency of a FB implementation with respect to its requirements. We use logical implication to specify a correctness theorem for a FB. The correctness theorem proves that the behaviour of the FB implementation conforms to its requirements. A separate consistency theorem proves that the FB implementation always produces outputs for any given valid inputs. If the FBD and ST implementation are both given for the same FB, we formulate an equivalence theorem verifying the consistency between these two implementations.

5. **Applications to IEC 61131-3 FBs and a realistic real-time FB subsystem**: We applied our methodology to FBs supplied by IEC 61131-3 (IEC, 2003) and two realistic real-time FB subsystems, *Trip Sealed-In* and *Pushbutton*. We identified several kinds of issues in the Standard: ambiguous behaviour, missing input assumptions and inconsistent FB implementations. We found an initialization failure in the FBD implementation of *Trip Sealed-In*. We corrected each issue and verified the proposed fix using our methodology.

6. **Developed proof strategies for consistency and correctness in PVS**: We developed proof strategies that can be used as guidance to discharge the consistency and correctness theorems of those FBs that require real-time requirements and those do not, respectively. Using our proof strategies significantly reduces interactive proof effort in PVS.

1.6 Publications and Industrial Application

This subsection lists all of our publications, technical reports, and industrial application. All of the following are based on discussion with my colleagues.

1.6.1 Journals

- (Pang et al., 2014a) Linna Pang, Chen-Wei Wang, Mark Lawford, and Alan Wassyng. Formal verification of IEC 61131-3 function blocks using tabular expressions. *Science of Computer Programming*. Invited submission for a special issue (ID: SCICO-D-14-00102). Under minor revision.

1.6.2 Refereed Conferences and Workshops

- (Pang et al., 2013a) Linna Pang, Chen-Wei Wang, Mark Lawford, and Alan Wassyng. Formalizing and verifying function blocks using tabular expressions and PVS. *Formal Techniques for Safety-Critical Systems*, *FTSCS2013*, volume 419 of Communications in Computer and Information Science, pages 163–178, Springer, Queenstown, New Zealand,

October, 2013.

- (Pang et al., 2015) Linna Pang, Chen-Wei Wang, Mark Lawford, Alan Wassyng, Josh Newell, Vera Chow and David Tremaine. Formal verification of real-time function blocks using PVS. *Engineering Safety and Security Systems, ESSS2015*. EPTCS 184, 2015, pages 65–79, Oslo, Norway, June, 2015.

1.6.3 Technical Reports

- (Pang et al., 2013b) Linna Pang, Chen-Wei Wang, Mark Lawford, and Alan Wassyng. Formalizing and verifying function blocks using tabular expressions and PVS. Technical Report 11, McMaster Centre for Software Certification (McSCert), McMaster University, August, 2013.

- (Pang et al., 2014b) Linna Pang, Chen-Wei Wang, Mark Lawford, Alan Wassyng, Josh Newell, Vera Chow and David Tremaine. Formal verification of real-time function blocks using PVS. Technical Report 16, McMaster Centre for Software Certification (McSCert), McMaster University, September, 2014.

1.6.4 Industrial Application

Our methodology has been successfully adopted in an on-going industrial project, i.e., Darlington Nuclear Generating Station (DNGS) Shutdown System Number One (SDS1) Replacement Project, Ontario, Canada, to formally verify the correctness of PLC programs.

The process is as follows: (1) the trip computer design is described in a collection of FBD and ST blocks; (2) the design is translated into PVS specifications; and (3) PVS is then run to perform disjointness and completeness checks on the requirements and to verify that the design matches the requirements. As a result, the proposed method successfully identified discrepancies in the requirements and the PLC implementation.

1.7 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces the required mathematical and technical background of the thesis. The proposed systematic verification approach for FB-based system is discussed in Chapters 3, 4, 5, and 6. In particular, Chapters 3 and 4 present a method of formalizing input-output requirements for FBs, including timer blocks. Chapter 5 provides a compositional approach to the formalization of FB implementations. Chapter 6 provides a formal technique for the verification of FB implementations with respect to their tabular requirements. Chapter 7 discusses two case studies using our approach. Chapter 8 concludes our research and suggests future work.

Chapter 2

Preliminaries

In this chapter we introduce notations and concepts that are used throughout this thesis [1]. Section 2.1 presents the basis of tabular expressions. Section 2.2 introduces function blocks and the related standards. Section 2.3 presents an overview of the PVS verification system. Section 2.4 reviews the time model we will use in PVS. Section 2.5 introduces two running examples in the thesis.

2.1 Tabular Expressions

Tabular expressions (Janicki et al., 1997; Parnas, 1983; Parnas et al., 1994; Parnas and Madey, 1995) are a proven and effective approach to describing conditionals and relations, and they are thus ideal for documenting many system requirements. Tabular expressions have also been proven to be of great help both in inspections (Wassyng and Janicki, 2003) and in testing and verification (Wassyng et al., 2011). They are arguably easier to comprehend and to maintain than conventional mathematical expressions. In principle, tabular expressions are a generalization of two dimensional tables. Formal semantics for tabular expressions have been developed, because of its great importance in practice. Reference (Janicki and Wassyng, 2005) presents a relational semantics for tabular expressions, which covers the most common types of tabular expressions used in software practice. Recently, (Jin and

[1] This chapter is based on our published work in (Pang et al., 2015).

Parnas, 2010) presented a new semantics for tabular expressions by using indexing to decouple the appearance of a tabular expression from its semantics.

From practical point of view, only limited types of table are used for most of the cases in relational systems. Normal one-dimensional and structured decision tables are used mostly, and normal two-dimensional tables occasionally. For our purpose of capturing the input-output requirements of function blocks in IEC 61131-3, tabular expressions of the form shown in Figure 2.1 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 Figure 2.1 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.

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

```
IF C1
  IF      C1.1 THEN F = RES1
  ELSEIF  C1.2 THEN F = RES2
  ...
  ELSEIF  C1.m THEN F = RESm
ELSEIF  ...
ELSEIF    Cn   THEN F = RESn
```

Figure 2.1: Semantics of horizontal condition table (HCT)

In documenting input-output behaviours using HCTs as illustrated in Figure 2.1, 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.

2.2 Function Blocks and Related Standards

2.2.1 Programmable Logic Controller and Function Blocks

A programmable logic controller (PLC) is a digital computer used in automation for industrial electromechanical processes. PLCs are widely utilized in hard real-time control systems since output results need to be produced in response of input values within a limited time span.

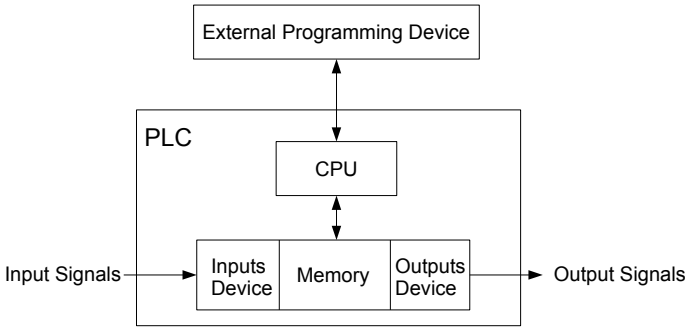


Figure 2.2: Architecture of a PLC

A PLC contains a central processing unit (CPU), main memory, one or more input devices and output devices. An input device converts the signals from the environment to logic levels for the processing unit to read. The processing unit uses the signals from the input device to perform control functions based on application software. An output device transmits the signals to the environment. The architecture of a PLC is shown in Figure 2.2. A PLC operates by continually scanning the application software. One cycle through the program is called a scan time and involves reading the inputs from the environment, executing the application software based on these inputs and then updating the outputs accordingly.

As a basic element to create a PLC program, a function block (FB) is an abstract component that can be implemented either by hardware, software or both. FBs can be reused in different PLC programs. It is only possible to access the interface (i.e., input and output variables) of a FB externally. The FB implementation is hidden from the users. In other words, a FB is a

black-box component that can be reused without knowing the details of its implementation. FBs are fed by input values, perform computations on them according to the behaviour specification (i.e., implementation), and produce output values. All the values of the internal and output variables persist from one execution of the FB to the next. FBs are used for a wide range of PLC-based systems that require retained state.

2.2.2 IEC 61131-3 and PLC Programming Languages

For the purpose of unifying the syntax and semantics of programming languages for PLCs, the International Electrotechnical Committee (IEC) first published IEC 61131-3 in 1993 with revisions in 2003 (IEC, 2003) and 2013 (IEC, 2013). It is considered as one of the most important standards in industrial automation. The Standard has been used predominantly by control engineers to help specify the software parts of PLC-based control systems. Most of our research results were completed before the third edition was released in 2013. Nonetheless, the third edition is fully compatible with the second.

IEC 61131-3 defines a set of programming languages for PLC programming, namely two graphical languages, function block diagram (FBD) and ladder diagram (LD), two textural languages, instruction list (IL) and structured text (ST), and sequential function chart (SFC).

In this thesis we focus on two programming languages that are covered in IEC 61131-3 for writing behavioural descriptions of FBs: ST and FBDs. These two languages are widely used in PLC-based systems. The ST notation is a high-level textural programming language which resembles, in syntax, another high-level programming language, Pascal. FBDs are a graphical programming notation. The fundamental concept behind FBDs is the interconnections among block components, which specify the data flow dependency.

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 as equivalent; instead, we prove that one implementation conforms to the other (but not vice versa). More precisely, the FBD language does not easily support efficient

corresponding constructs for all of the ST iterations (i.e., `FOR`, `WHILE`, and `REPEAT`) and the `EXIT` statement to terminate iteration. On the other hand, the ST language does not support jumps (denoted by "`+-->>LABEL`" in FBD) to transfer program control to the designated network label. For example, the FBD implementation supplied for the *STACK_INT* block (discussed later in Susbection 7.1.3.1) uses three labels `RESET`, `PUSH_STK`, and `POP_STK` to perform the reset, push, and pop operations on the stack. Moreover, the supplied FBD implementation has an explicit execution order for its component FBs (by the use of auxiliary variables), and such specificity is not required in the ST implementation for the same block.

Given the use of PLCs in industry, especially in safety-critical applications, the correctness of PLC implementations plays an important role. Usually, it is achieved by testing, and sometimes formal verification and test.

2.2.3 Classification for Function Blocks

The PLC program units are classified at the level of standard functions, basic function blocks, and composite function blocks. The behaviour of standard function or function block is time-dependent, i.e., variables are parameterized by time instants.

Standard Function

A standard function is a program organization unit that, when executed, outputs exactly one data element as the function result. Standard functions contain no internal state information. Therefore, invocation of a standard function with the same input values always generates the same output values.

The IEC 61131-3 standard 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.

As an example of a timed function, consider the *MOVE* function (declared in Figure 2.3) that takes as input an integer *IN*, and that outputs an

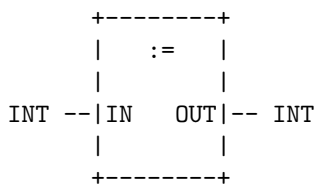


Figure 2.3: Standard function *MOVE* declaration (IEC, 2003)

integer *OUT*. The behaviour of *MOVE* is time-dependent: *OUT* is equal to *IN* at every time t.

Basic Function Block

A basic (or primitive) FB is a program organization unit which, when executed, yields one or more values. All the values of the output (and internal variables) persist from one execution of the FB to the next; therefore, invocation of a FB with the same input values needs not always generate the same output values.

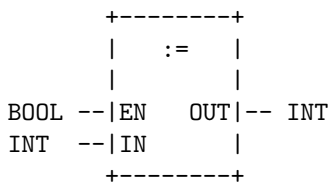


Figure 2.4: Function block *MOVE* declaration (IEC, 2003)

As an example of a basic FB, consider the *MOVE* function block that takes as inputs an enabling condition *EN* and an integer *IN*, and that outputs an integer *OUT*. The behaviour of *MOVE* is time-dependent: at the very first clock tick, *OUT* is initialized to zero; otherwise, at time instant t ($t > 0$), *OUT* is either equal to *IN* at time t, if condition *EN* holds at t, or otherwise *OUT* is equal to *IN* at time $t - \alpha * \delta$ ($\alpha = 1, 2, \ldots$, and δ is the time interval between two time ticks introduced in Section 2.4) where *EN* was last enabled (i.e., a case of "no change" for *OUT*).

Timer blocks are a special kind of basic FBs. The accumulated data regarding the current state of the timer will influence the output values in

response to the same input values. IEC 61131-3 defines three kinds of timers: *TON* (on-delay), *TOF* (off-delay), and *TP* (pulse) timers. They are widely used to specify the timing behaviour of real-time FBs.

Composite Function Block

A composite FB consists of standard functions, basic FBs and any pre-developed composite FBs. Different from basic FBs, composite FBs are often used to solve much complicated problems and the component FBs can be programmed using different languages. It encourages a well-structured software development using either a top-down or bottom-up approach.

For example, The *LIMITS_ALARM* block (Figure 5.4 in Section 5.1.1) is a composite FB consisting of basic FBs and two instances of the pre-developed FB *HYSTERESIS*. The formalization of *LIMITS_ALARM* block is presented in Section 3.4

2.3 PVS Language and Prover

Prototype Verification System (PVS) (Owre et al., 1992) was developed by SRI International's Computer Science Laboratory as an interactive environment for writing specifications and performing proofs. It is intended to be sufficiently used for significant applications to capture the state-of-the-art mechanized formal methods. PVS consists of a specification language, predefined theories, a parser, a type checker, a theorem prover that supports several decision procedures, a symbolic model checker, pre-developed libraries, utilities and documentation with examples in different application areas.

2.3.1 PVS Preliminaries

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. Also, 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 typechecking. 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. The theories can be parameterized with constants, types, and theory instances. 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 providing over 1000 definitions and lemmas. The NASA PVS Library is also a collection of formal developments contributed and maintained by the NASA Langley Formal Methods Team (NASA Langley PVS Libraries Official Website, 2014).

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. 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.

We chose the PVS theorem prover to formalize the input-output requirements of FBs primarily because it supports the syntax and semantics of tables (Section 2.1). 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 (Lawford et al., 2000) that gave us confidence in using the toolset, and for modelling real-time behaviour we reused parts of the PVS theories from (Hu et al., 2009; Hu, 2008). Nonetheless, the techniques presented in this thesis are transferrable 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 basic structure of the PVS sequent calculus is a sequent (Shankar et al., 1999). PVS has a interactive proof checker to perform sequent-style deductions. Syntactically, a PVS sequent is showed as: $P_1, P_2, \ldots, P_m \vdash Q_1, Q_2, \ldots, Q_n$, where P_i, $i = 1, 2, \ldots, m$ are antecedent formulas and Q_j, $j = 1, 2, \ldots, n$ are consequent formulas and $\vdash$ is syntactic entailment. The prover maintains a proof tree and it is the final goal to discharge each leaf of this proof tree by invoking proof commands. Each node of the proof tree is a proof goal that produces its offspring nodes by means of a proof step. The final goal of a PVS sequent is to determine whether at least one of the consequents is a logical consequence of its antecedents. In PVS, a sequent is displayed as:

```
[-1]  P1
...   ...
[-m]  Pm
|-------
[1]   Q1
...   ...
[n]   Qn
```

The antecedents are combined by conjunctives while consequents are connected by disjunctives. Thus, a PVS sequent $P_1, P_2, \ldots, P_m \vdash Q_1, Q_2, \ldots, Q_n$ is logically equivalent to $P_1 \wedge P_2 \wedge \cdots \wedge P_m \vdash Q_1 \vee Q_2 \vee \cdots \vee Q_n$. In our specification, we use $\neg$, $\wedge$, $\vee$, $\Rightarrow$ to denote logical negation, conduction, disjunction and implication and $\forall$, $\exists$ for universal and existential quantifiers.

A sequent can be discharged only if one of the 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 (Hu, 2008). A PVS sequent may be discharged by splitting it into sub-goals and by proving all of these sub-goals.

2.3.2 Naming Convention in PVS

In this section we introduce the naming convention we employ in this thesis.

Theories

Specification in PVS is built from theories, which consists of a theory identifier, a list of formal parameters, a list of imported theories, an exporting clause, an importing clause, a theory body, and an ending identifier. PVS theories may be parameterized to provide generality, reusability, and structure (Owre et al., 1999). We use the closest names as supplied by IEC 61131-3 FBs and other FB subsystems as their PVS theory identifiers. It is more readable to place "_" in between words for theory name. For example, we use *LIMITS_ALARM* as the theory name for block *LIMITS_ALARM*.

Theorems and Lemmas

As the main proof obligations, we name the theorems and lemmas as unambiguously and uniquely as possible. Again, we use "_" to make names more readable. For example, a consistency theorem is a proof obligation goal for each FB. We name the consistency theorem for FB *SR* latch as *SR_CONSISTENCY*.

Variables

We follow the input and output variables names as supplied by IEC 61131-3 and other FB subsystems in a straightforward manner. Most of them are a few letters in upper case convention (e.g., *S1*). We keep the time variable in lower case (i.e., t). In addition, we define the inter-connectors (or wires) that connect components as w's followed by a number (e.g., $w1$).

Proof Commands

The sequent calculus inference rules are used to construct proofs with PVS. The PVS proof commands are powerful, making the proof construction process more illumining and less tedious. Any proof command must be in parentheses either in upper case or lower case. Proof commands can be typed in by the user at the (*Rule*?) prompt, or they can be automatically applied by PVS as part of a proof strategy (Shankar et al., 1999). For example, the (*lift-if*)

command is used to bring the case analyses in an expanded definition to the surface of the proof sequent, where it may be propositionally simplified. A PVS glossary on page xvi provides a quick reference to the used PVS names in this thesis.

2.3.3 Support for Tabular Expressions 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 proven as being well-defined, some useful feedback is returned (i.e., unproven TCCs). 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. In Subsection 7.1.2.2, 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 (Lawford et al., 2000).

2.3.4 Type Correctness Conditions

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. We briefly review failed TCCs that we encountered in our verification process.

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 those unproven TCCs are considered provisional. PVS checks the completeness and disjointness properties for a function table (Section 2.1) 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 in Figure 2.5. The associated TCCs[2] of (*COND-COVERAGE-TCC*) and (*COND-DISJOINT-TCC*) are automatically generated by PVS — see the right in Figure 2.5.

```
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;
```

Figure 2.5: Function tables and TCCs in PVS

Another category of TCCs concerns the well-definedness of expressions, which might be used in tables or other functions. As an example, consider a function $g(x)$ with a real parameter x:

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

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

To model g in PVS, we use the built-in division operator (LHS in Figure 2.6). 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 Figure 2.6).

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;

Figure 2.6: Expressions and well-definedness TCCs in PVS

There are several other categories of TCCs that are automatically generated in PVS: subtype TCCs, existence TCCs, and termination TCCs. Subtype TCCs are generated for expressions whose types are defined using the predicate subtype notation (e.g., positive real numbers *posreal*). Existence TCCs are generated for expressions whose types are declared as non-empty. Termination TCCs are generated to ensure that recursive functions cannot be unfolded an infinite number of times.

2.4 Modelling Time in PVS

2.4.1 Modelling Time in the Physical Domain

As PLCs are widely used in real-time systems, 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:

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

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 (Hu et al., 2009) from (Lawford and Wu, 2000). 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 physical 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 }
```

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* (Hu et al., 2009): e.g., *init* (the very first tick) and $pre(t)$ (the tick preceeding 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 4^{th} 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 (Hu, 2008). It states that predicate P holds at all ticks if (1) P holds at the initial tick t_0; and (2) 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)) AND
     (FORALL (t: noninit_elem): P(pre(t)) ⇒ P(t))
    ⇒ (FORALL (t: tick): P(t))
```

We consider 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.

2.4.2 Modelling Samples in the Software Domain

We use a variable $Sample : \mathbb{N} \to \mathbb{R}_{\geq 0}$ to denote the series of samples over time, such that the time of each sample (i.e., $Sample(n)$, $n \in \mathbb{N}$) maps to a valid clock tick. As shown in Figure 2.7, realistically, the clock tick frequency $\frac{1}{\delta}$ in the physical domain should be significantly larger than the sampling frequency in the software domain. The sample intervals is bounded between *Tmin* and *Tmax*, determined by considering the shortest time after which events must be detected.

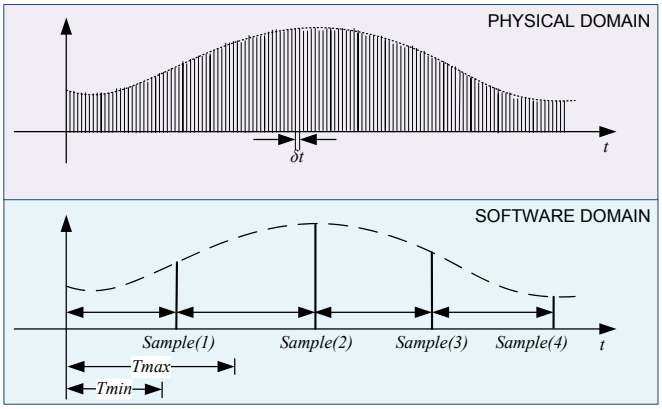


Figure 2.7: Time models in physical and software domains based on tick type (Hu, 2008)

2.4.3 Operators for Specifying Timing Requirements

An infix operator *Held_For* is defined as:

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

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

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

where $d \in [d - \delta L, d + \delta L]$. However, the behaviour of *Held_For* is nondeterministic when P has last been $TRUE$ in-between $[d - \delta L, d + \delta R]$.

To resolve the non-determinism in *Held_For*, two refinement operators are defined: *Held_For_S* and *Held_For_I*. Since it has been verified that the *Held_For_I* operator (with a sustained duration of $d - \delta L$) is a refinement of *Held_For*, meeting the functional timing requirements (FTRs) and performance timing requirements (PTRs), we can use *Held_For_I* to replace the requirement using the *Held_For* operator with tolerance (Hu, 2008). Thus both operators are deterministic by fixing the duration d in the above definition of *Held_For* as $d - \delta L$. We will only see *Held_For_I* in the case studies, but it is defined in terms of *Held_For_S*. *Held_For_S* is a partial function on *tick* that produces values only at points of sampling (i.e., it is undefined on ticks in-between samples).

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

On the other hand, *Held_For_I* is a totalized version of *Held_For_S*: its value at time t, where $Sample(n) \leq t < Sample(n+1)$, is equivalent to that produced at time $Sample(n)$ (i.e., the closest left sample calculated by *Left_Sample*). Function *sup* returns the unique least upper bound of a set.

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

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

2.4.4 Implementing the *Held_For_I* Timing Operator

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

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

where *Timer_S* counts, starting from the closest left sample to the clock tick in question, for how long the monitored condition P has been enabled, and stops counting when *TimeOut* is reached. The output type of *Timer_S* is *tick*, calculated from how many samples P has been held across. The value of *Timer_S* function is updated through a *TimerUpdate* function (Hu, 2008) by passing a condition at both the current and last samples, the previous value of the timer, the timeout value, and the elapsed time since the last update of the timer. The theorem *TimerGeneral_I* is proved to ensure that *Timer_I* is a proper implementation for *Held_For_I*.

$$TimerGeneral\_I\texttt{: THEOREM}\ Held\_For\_I(P,\ timeout - delta\_L,\ Sample)(t) \Leftrightarrow Timer\_I(P,\ timeout - delta\_L,\ Sample) \geq timeout - delta\_L$$

2.5 Running Examples for this Thesis

Two running example FBs are used to exemplify our approach through the thesis, *HYSTERESIS* and *LIMITS_ALARM*. They both are supplied by IEC 61131-3 (IEC, 2003). Block *HYSTERESIS* is a basic FB implemented in ST language, while block *LIMITS_ALARM* is a composite FB (built from two instances of block *HYSTERESIS*) implemented in FBD language. In this section, we will introduce the declarations and informal descriptions for these two FBs provided by the Standard.

2.5.1 Running Example: *HYSTERESIS*

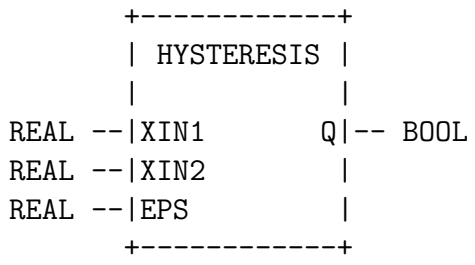


Figure 2.8: Basic FB *HYSTERESIS* declaration (IEC, 2003)

As an example, the block *HYSTERESIS* 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 (Figure 2.8) 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. 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. The requirement of block *HYSTERESIS* is formalized in Section 3.3. We will discuss this example in detail in Subsection 7.1.2.2.

2.5.2 Running Example: *LIMITS_ALARM*

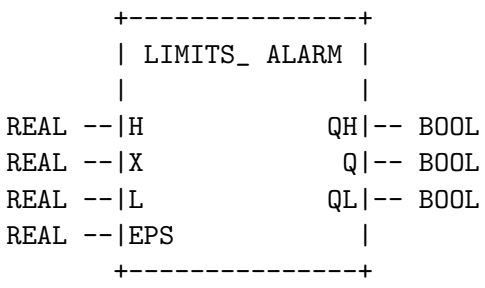


Figure 2.9: Composite FB *LIMITS_ALARM* declaration (IEC, 2003)

As another example FB, the block *LIMITS_ALARM* implements an alarm with high- and low-limit. Its declaration (Figure 2.9) requires four real-valued inputs, high limit H, low limit L, real value X, and hysteresis deadband *EPS* (for both high- and low- limits), and three Boolean outputs, high flag QH, low flag QL and alarm output Q. When variable value X exceeds the high limit H, the high flag QH becomes *TRUE*. Symmetrically, when X goes below the low limit L, the low flag QL becomes *TRUE*. Both flags QH and QL are set to *FALSE* when X is in the exclusive range of $(L+EPS, H-EPS)$. There exists a hysteresis band for the high limit inside which the value of QH remains unchanged: $[H-EPS, H]$. Symmetrically, there exists a hysteresis band for the low limit: $[L, L+EPS]$. Finally, the alarm output Q is set to *TRUE* if and only if either of the flags is set to *TRUE*, or set to *FALSE* otherwise. The input-output requirement is captured in three function tables in Section 3.4. More detailed discussion on this example will be presented in Subsection 7.1.2.3.

Chapter 3

Formalizing Requirements of Function Blocks

This chapter presents a formal approach for defining IEC 61131-3 standard functions and FBs using tabular expressions and PVS [1]. We first introduce a general approach for formalizing input-output black-box requirements for standard functions and FBs in Section 3.1. This approach is then applied to various kinds of standard functions in Section 3.2, basic FBs in Section 3.3, and composite FBs in Section 3.4, respectively.

3.1 A General Approach for Formalizing Input-output Relations

As stated in Section 2.2.2, IEC 61131-3 supplies low-level, implementation-oriented ST and FBD descriptions for FBs. 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 (Section 2.1) 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.

[1] This chapter is based on the published work in (Pang et al., 2013a).

Similar to the basic idea for specifying the behaviour of a hardware device in (Gordon, 1985; Melham, 1989) that stating a combination of values observed on its external variables, the requirement of each standard function or FB is formalized as a predicate constraining input and output variables as parameters, to characterize its input-output relation. To reflect the intended behaviour, the requirement predicate is defined such that the combinations of inputs and outputs values satisfying this constraint are exactly those which can be observed simultaneously on the corresponding input and output variables at the same time tick.

In general, let us consider a FB with m inputs, i_1, i_2, ..., i_m, and n outputs, o_1, o_2, ..., o_n. The black-box input-output requirement of the FB is formalized in function tables (Section 2.1). For output o_p, $p = 1, 2, ...n$, a function table is used to capture the input-output relation between o_p and its dependents, i.e., the combinations of inputs and possibly outputs.

There are three kinds of relation between output o_p and its dependents:

1. The value of output o_p is determined only by a set of the input values at current time tick.

   The requirement of o_p is formalized as a predicate o_p_REQ as follows:

   $$o_p\_REQ : bool \equiv o_p = f(i_1, i_2, \ldots, i_m),$$

   where f is the function table for o_p, depending on those of the inputs.

2. The value of output o_p is determined by the values of a set of inputs at current time tick and the values of a set of other outputs up to the current time tick.

   The requirement of o_p is formalized as a predicate o_p_REQ as follows:

   $$o_p\_REQ : bool \equiv o_p = f(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_{p-1}, o_{p+1}, \ldots, o_n),$$

   where f is the function table for o_p. Different from case 1, the requirement predicate for o_p takes a set of inputs and a set of other outputs as parameters. Thus, the table for f is composed with those tables for its dependent outputs. However, the requirement for a standard function

or FB is not implementable, if two outputs depend on each other at the current time. For example, output o_1 is determined by the value of o_2 at current time t, while o_2 is determined by the value of o_1 at the same time. It is implementable if two outputs depend on each other but not both at the current time. For example, output o_1 is determined by the value of o_2 at previous time t_{-1} [2], while output o_2 is determined by the value of o_1 either at current time t or previous time tick t_{-1}.

3. The value of output o_p is determined by the values of a set of inputs at the current time tick, the values of a set of other outputs up to the current time tick, and the values of itself up to the previous time tick.

   The requirement of o_p is formalized as a predicate o_p_REQ as follows:

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

   where f is the function table for o_p. Different from cases 1 and 2, output o_p depends on the values of a set of inputs, the values of a set of other outputs, and the values of itself at previous time ticks. Thus, the function table for f has to be recursively defined (i.e., with the presence of "NC" in the last result column).

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. We are able to: (1) specify a separate function table that characterizes its relationship with the inputs; and (2) prove its correctness separately.

[2] We use subscription "-1" to denote the value of a variable at previous time tick.

3.2 Formalizing Requirements of Standard Functions

In general, we formalize the behaviour of a standard function f as a relation (i.e., Boolean function or predicate), constraining the permissible combinations of input and output values:

$$f(i_1, i_2, \ldots, i_m) : (o_1, o_2, \ldots, o_n) \triangleq f\_REQ(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n), \quad (3.1)$$

where the symbol $\triangleq$ denotes that function f is formalized using a relation (or predicate) f_REQ. Predicate f_REQ 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.

For example, consider the standard function of *MOVE* (see Figure 2.3), which translates the input-output relation of function *MOVE* into PVS:

```
MOVE(IN, OUT: [tick -> int]): bool =
  FORALL (t: tick): OUT(t) = IN(t)
```

We characterize the temporal relation between *IN* and *OUT* as a universal quantification over discrete time instants. Function [*tick* -> *int*] captures the input and output values at different time instants.

We also support a functional version of the specification for standard functions. For the blocks that consist of standard functions, we formalize them as function compositions of their internal blocks. Given the function f and the formalizations for internal standard functions f_1: $X \rightarrow Y$ and f_2: $Y \rightarrow Z$, we formalize f as functional composition of f_1 and f_2 (denoted by $f_1 \circ f_2$), i.e., $f_2 \circ f_1(x) = f_2(f_1(v_1))$.

For example, the functional version of *MOVE* returns an integer value:

```
MOVE(IN: [tick -> int]): int = IN(t)
```

3.3 Formalizing Requirements of Basic Function Blocks

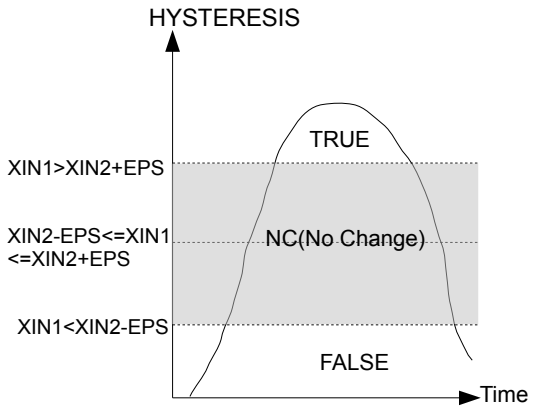
Similarly, a basic FB is formalized as a relation constraining the permissible combinations of input and output values. In general, we formalize the requirement of a basic FB as a Boolean function *BFB_REQ*:

$$\begin{aligned} & BFB\_REQ(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) : bool \\ \equiv \quad & o_1 = f\_o_1(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) \wedge \\ & o_2 = f\_o_2(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) \ \wedge \ \ldots \\ & o_n = f\_o_n(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n), \end{aligned} \tag{3.2}$$

where predicate *BFB_REQ* characterizes the precise relation on the m inputs and the n outputs of the basic FB, and function f_o_i specifies the behaviour of output o_i using a function table. Functions $f\_o_i, i = 1, 2, \ldots, n$ are defined recursively if any values of themselves at previous time ticks are dependent.

Revisiting the running example of basic FB *HYSTERESIS* (Section 2.5.1), we formalize its requirements using function tables. The shaded area on the left (Figure 3.1) denotes the hysteresis deadband (with a size of $2 \times EPS$). The table on the right (Figure 3.1) specifies the requirement for *HYSTERESIS*. Note we use "NC" to denote the case of "no change" for output Q. Alternatively, we can use Q_{-1} to denote the value at previous time tick. At the bottom of tabular requirement, we incorporate with the assumption: $EPS > 0$. Otherwise, the two intervals $[\, XIN2 + EPS, +\infty \,]$ and $[\, -\infty, XIN2 - EPS]$ may overlap (i.e., the first and third rows, $XIN1 > (XIN2 + EPS)$ and $XIN1 < (XIN1 - EPS)$, of tabular requirement are not disjoint). More explanation for this missing assumption will be discussed in Chapter 7.

More precisely, we translate the input-output relation of *HYSTERESIS* into PVS (Figure 3.2). The behaviour of FB *HYSTERESIS* at each time instant is specified as an *IF-THEN-ELSE-ENDIF* statement. The initial case is defined in the *IF* branch separately, while construct *TABLE-ENDTABLE* in the *ELSE* branch specifies the tabular requirement (Figure 3.1). Embedding table in a predicate allows PVS prover to generate proof obligations for disjointness and completeness automatically.



Condition	*Result* Q
XIN1 > (XIN2 + EPS)	TRUE
(XIN2 − EPS) ≤ XIN1 ∧ XIN1 ≤ (XIN2 + EPS)	NC
XIN1 < (XIN2 − EPS)	FALSE

assume: $EPS > 0$

Figure 3.1: Requirement for *HYSTERESIS*: with the assumption $EPS > 0$

```
HYSTERESIS_REQ(XIN1, XIN2: [tick -> real],
               EPS: [tick -> posreal],
               Q: [tick -> bool]): bool =
  FORALL t:
    Q(t) = IF init(t) THEN FALSE
           ELSE LET PREV = Q(pre(t)) IN
             TABLE
               | XIN1(t) < (XIN2(t) - EPS(t))      | FALSE ||
               | ((XIN2(t) - EPS(t)) <= XIN1(t)) &
                  (XIN1(t) <= (XIN2(t) + EPS(t))) | PREV ||
               | (XIN2(t) + EPS(t)) < XIN1(t)      | TRUE  ||
             ENDTABLE
           ENDIF
```

Figure 3.2: PVS formalization of FB *HYSTERESIS* input-output relation

3.4 Formalizing Requirements of Composite Function Blocks

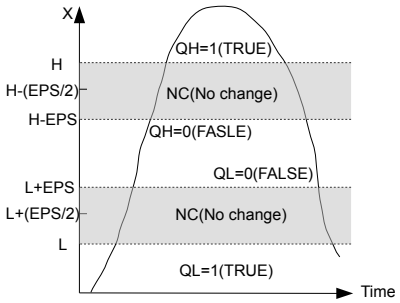
A composite FB is also formalized as a relation constraining the permissible combinations of input and output values. In general, we formalize the

requirement of a composite FB as a Boolean function *CFB_REQ*:

$$\begin{aligned} & CFB\_REQ(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) : bool \\ \equiv \quad & o_1 = f\_o_1(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) \wedge \\ & o_2 = f\_o_2(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) \ \wedge \ \ldots \\ & o_n = f\_o_n(i_1, i_2, \ldots, i_m, o_1, o_2, \ldots, o_n) \end{aligned} \tag{3.3}$$

where predicate *CFB_REQ* characterizes the precise relation on the m inputs and the n outputs of composite FB, function f_o_i specifies the behaviour of output o_i using function table. Functions $f\_o_i, i = 1, 2, \ldots, n$ are defined recursively if any values of themselves at previous time ticks are dependent.

Revisit the example of composite FB *LIMITS_ALARM* (Section 2.5.2), we formalize its requirement using our approach. The expected input-output behaviour is depicted in Figure 3.3, and its tabular requirement (which constrains the relation between inputs *X*, *H*, *L*, *EPS* and outputs *Q*, *QH*, *QL*) is captured in the three surrounding tables for outputs *Q*, *QH*, and *QL*.



Condition	*Result* Q
QH ∨ QL	TRUE
¬(QH ∨ QL)	FLASE

assume: $L + EPS < H - EPS$

Condition	*Result* QH
X > H	TRUE
H - EPS ≤ X ≤ H	NC
X < H − EPS	FALSE

assume: $EPS > 0$

Condition	*Result* QL
X < L	TRUE
L ≤ X ≤ L + EPS	NC
X > L + EPS	FALSE

assume: $EPS > 0$

Figure 3.3: *LIMITS_ALARM* requirement in tabular expression

Let predicates P_QH, P_QL, and P_Q formalize, respectively, the table for *QH*, *QL* and *Q*. Then, we translate the above requirement into PVS as:

```
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)
```

Thus, we formalize the overall requirement for a composite FB by conjuring those predicates that specify the behaviour of outputs. For the given input and output sequences, requirement predicate returns *TRUE* if they match according to the behaviours described in output predicates; otherwise, it returns *FALSE*. This can later be used to verify the correctness of the FBD implementation of *LIMITS_ALARM* (Section 6.1).

3.5 Summary

This chapter introduced an approach for formalizing requirements of standard functions, basic FBs and composite FBs using tabular expressions and PVS. We first introduced a general approach for formalizing input-output relation. We then applied our approach to formalize the requirements of standard functions, basic FBs and composite FBs respectively. We exemplify each kind by providing concrete examples from IEC 61131-3 (IEC, 2003).

As a result, we obtained complete and disjoint black-box requirements of standard functions and FBs. Our formalized requirements then can be used to verify the consistency and correctness of the supplied implementation.

Chapter 4

Formalizing Idealized and Practical IEC 61131-3 Timers

In this chapter we focus on the formalization of IEC 61131-3 timers [1]. We first discuss problems raised by the correctness check for real-time behaviour and the techniques that we use to handle them in general (Section 4.1). We then formalize the idealized behaviour of IEC 61131-3 timers using function tables (Section 4.2). We implement the idealized timers using a pre-verified timing operator *Timer_I* (Section 4.3), which can be used to build real-time FB-based systems. We then discuss the incorporation of timing tolerances with the real-time requirements for timer-based systems (Section 4.4).

4.1 General Description

The correctness for a hard real-time system depends not only on values of its outputs, but also on the times at which they are produced (Rushby, 1992). Especially for safety-critical systems, it is important to specify and verify the real-time behaviour. Such a system often determines the values of outputs by the values of current inputs and the history of both inputs and outputs. For example, based on our knowledge of the Darlington Nuclear Shutdown System Trip Computer Software Redesign Project (Wassyng and Lawford, 2003), "if

[1] This chapter is based on our published work in (Pang et al., 2015).

power and pressure both exceed their maximum allowable limits for 3 seconds, then open the relay for 2 seconds" (Lawford and Wonham, 1995). Any failure or malfunction may result in death or serious injury to people, severe damage to equipment or environmental harm.

IEC 61131-3 provides timer FBs as a special yet important type of basic FB (Section 2.2.3), describing real-time behaviour. There are three types of timer FBs, *TON* (On-delay), *TOF* (Off-delay), and *TP* (Pulse). However, IEC 61131-3 uses timing diagrams to describe the expected (and idealized) behaviour of these timers, which is limited to an incomplete set of use cases. As a result, the lack of formal semantics for the behaviour of the three timers results in difficulties in formalizing and verifying timing behaviour of any real-time system which is built from these timers.

An implementable timing requirement must specify tolerances associated with time duration to account for various factors - e.g., sampling rates, computation time, and latency - that will delay the software controller's response to its operating environment (i.e., the plant). Such a requirement, even when implementable, allows for inconsistent implementations since it is non-deterministic. For example, a common type of functional timing requirements specifies that a monitored condition P must sustain a time duration, say d, with tolerances δL and δR, before being detected by the controller. When input condition P has been enabled in-between $[d-\delta L,\ d+\delta R]$, the controller can take action at any time in-between $[d-\delta L,\ d+\delta R]$. As we will see in our case study (Section 7.2), such a simple requirement can be used as part of specifying more complex real-time behaviour. To resolve such non-determinism, at the requirement level, we adopt a deterministic operator *Held_For_I* (Hu, 2008), which becomes *TRUE* at the first sampling point after the monitored condition P has been enabled for $d - \delta L$ time units. Similarly, at the implementation level we adopt the *Timer_I* operator (Hu, 2008) for counting the elapsed time of some monitored condition (i.e., condition P). We thus implement IEC 61131-3 timers using *Timer_I* operator. The implemented timers can then be used to formalize any FB-based system which is built from IEC 61131-3 timers. The relationship between these two operators is proved as a theorem *TimerGeneral_I* (Hu, 2008): $(P\ \textit{Held\_For\_I}\ (d - \delta L))$ is equivalent

to ($Timer\_I(P) \geq d - \delta L$). More precise definitions are in Chapter 2. The use of the general theorem *TimerGeneral_I* simplifies our verification work for FB-based real-time systems.

4.2 Formalizing Idealized Behaviour for the IEC 61131-3 Timers

4.2.1 Timer On-delay (TON)

The *TON* block is commonly used as a component of safety-critical systems. For example, it can be used to determine if a sensor signal has gone out of its safety range for too long, as we will see in Section 7.2.

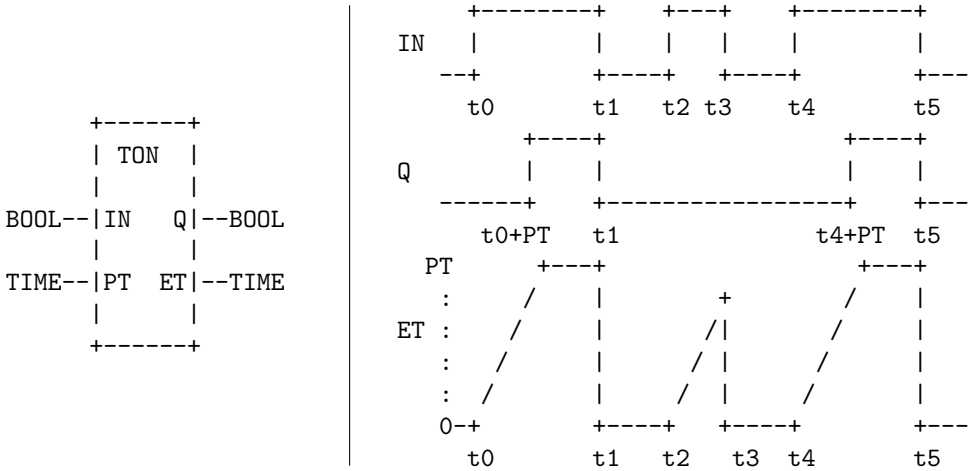


Figure 4.1: Timer *TON* declaration and its timing diagram (IEC, 2003)

Figure 4.1, extracted from IEC 61131-3, shows the input-output declaration (on the LHS) and a timing diagram[2] (on the RHS) illustrating the expected behaviour of the *TON* block. The *TON* block is declared with two inputs (a Boolean condition *IN* and a time period of length *PT*) and two outputs (a Boolean value *Q* and a length *ET* of time period). Timer *TON* monitors the input condition *IN* and sets the output *Q* as *TRUE* whenever *IN* remains enabled for longer than a time period of some input length *PT*; otherwise, it

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

sets output Q as *FALSE*. If the monitored input *IN* has been enabled for some time $t < PT$, then the timer sets the output ET (i.e., elapsed time) with value *t-last_enabled*, where a time stamp *last_enabled* is used to record the exact time (with no delay) that the input condition *IN* just becomes enabled; otherwise, it sets ET with value PT or it sets ET to default time value *0* if *IN* is disabled. We formalize the black-box requirement of the *TON* block (Figure 4.2) using function tables.

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

Condition	*Result* Q
IN $\wedge$ (t - last_enabled $\geq$ PT)	TRUE
IN $\wedge$ (t - last_enabled $<$ PT)	FALSE
$\neg$IN	FALSE

Condition	*Result* ET
IN $\wedge$ (t - last_enabled $\geq$ PT)	PT
IN $\wedge$ (t - last_enabled $<$ PT)	t - last_enabled
$\neg$IN	0

Figure 4.2: Tabular requirements of timer *TON*: idealized behaviour

4.2.2 Timer Off-delay (TOF)

The *TOF* block delays the falling edge of a Boolean input by a specified duration. For example, a *TOF* block can be used to keep cooling fans on for a specific time period after the oven has been turned off.

Figure 4.3, extracted from IEC 61131-3, shows the input-output declaration (on the LHS) and a timing diagram (on the RHS) illustrating the expected behaviour of the *TOF* block. The *TOF* block is declared with two inputs (a Boolean condition *IN* and a time period of length PT) and two outputs (a Boolean value Q and a length ET of time period). Timer *TOF* monitors the input condition *IN* and sets the output Q as *FALSE* whenever *IN* remains disabled for longer than a time period of some input length PT; otherwise, it sets the output Q as *TRUE*. If the monitored input *IN* has been disabled for some time $t < PT$, then the timer sets the output ET (i.e., elapsed time) with

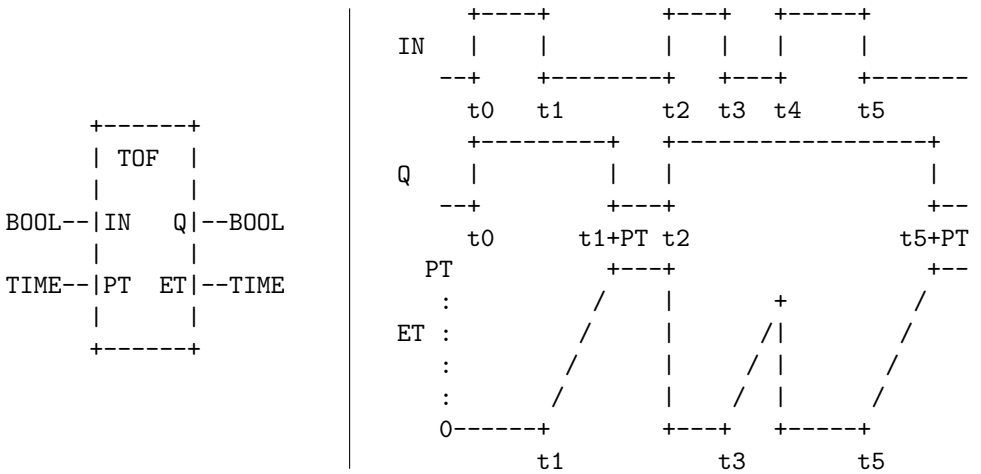


Figure 4.3: Timer *TOF* declaration and its timing diagram (IEC, 2003)

value *t-last_disabled*, where a time stamp *last_disabled* is used to record the exact time (with no delay) that the input condition *IN* just becomes disabled; otherwise, it sets *ET* with value *PT* or it sets *ET* with default time value *0* if *IN* is enabled.

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

Condition	*Result* Q
$\neg IN \wedge$ (t - last_disabled $\geq$ PT)	FALSE
$\neg IN \wedge$ (t - last_disabled $<$ PT)	TRUE
IN	TRUE

Condition	*Result* ET
$\neg IN \wedge$ (t - last_disabled $\geq$ PT)	PT
$\neg IN \wedge$ (t - last_disabled $<$ PT)	t - last_disabled
IN	0

Figure 4.4: Tabular requirements of timer *TOF*: idealized behaviour

4.2.3 Timer Pulse (TP)

Timer pulse acts as a pulse generator: as soon as the input condition *IN* is detected to hold, it generates a pulse to let output *Q* remain *TRUE* for a constant *PT* of time units. The elapsed time that *Q* has remained *TRUE* can be monitored via output *ET*.

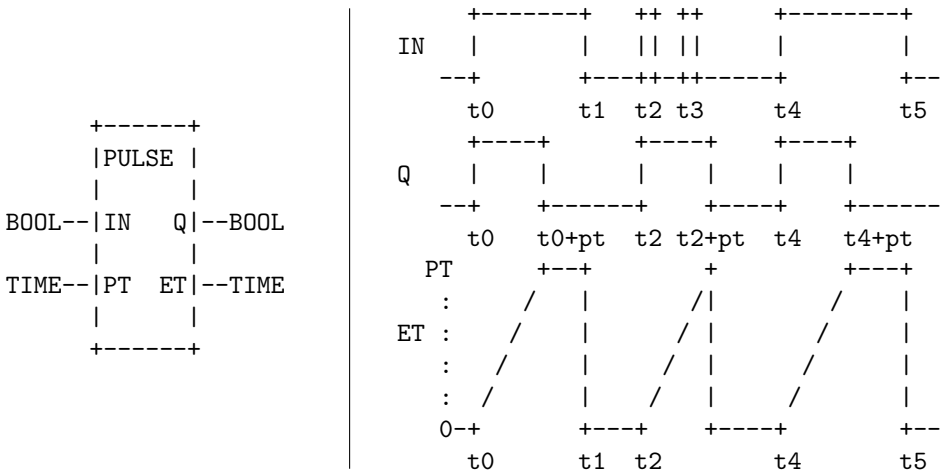


Figure 4.5: Timer *TP* declaration and its timing diagram (IEC, 2003)

Figure 4.5, again extracted from IEC 61131-3, shows the input-output declaration (on the LHS) and a timing diagram (on the RHS) illustrating the expected behaviour of timer *TP*. The 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 $t_3 < t_2 + PT$.

We use the three tables in Figure 4.6 to formalize the behaviour of the *PULSE* timer, where its outputs Q and ET and the internal variable *pulse_start_time* are initialized to, respectively, *FALSE*, 0, and 0. The tables have their obvious equivalents in PVS. To make the timing behaviour precise, we define an auxiliary predicate *Held_For* which is based on the work presented in (Hu et al., 2009). Predicate *Held_For(P, duration)* holds when the input predicate P holds for at least *duration* units of time.

```
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))
```

However, we found two scenarios that are not covered by the above timing diagram supplied by IEC 61131-3. We make the definition of the *PULSE* timer both complete and disjoint using function tables (Figure 4.6). We will

discuss these missing scenarios and our solution in Subsection 7.1.1.1.

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

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

Condition		*Result* ET
Q		t − pulse_start_time
¬Q	¬IN	0
	IN	PT

Figure 4.6: Tabular requirements of timer *TP*: idealized behaviour

4.3 Formalizing Practical Behaviour for the IEC 61131-3 Timers

4.3.1 Timer On-delay (TON)

As we discussed in Section 4.2.1, timer *TON* is implemented using the pre-verified *Timer_I* operator (Figure 4.7). We use *Timer_I* for counting the elapsed time of monitored condition (i.e., input *IN*). We then simplify the function table of output *ET* to $ET(IN, Samsple, PT) = Timer\_I(IN, Sample, PT)$.

Condition	*Result* Q
Timer_I(IN, Sample, PT) ≥ PT	TRUE
Timer_I(IN, Sample, PT) < PT	FALSE

Condition	*Result* ET
Timer_I(IN, Sample, PT) ≥ PT	PT
Timer_I(IN, Sample, PT) < PT	Timer_I(IN, Sample, PT)
¬IN	0

Figure 4.7: Re-formalization of timer *TON*

4.3.2 Timer Off-delay (TOF)

Similarly, we implement timer *TOF* using *Timer_I* operator (Figure 4.8). We then simplify the function table of output *ET* to $ET(\neg IN, Samsple, PT) = Timer\_I(\neg IN, Sample, PT)$.

Condition	*Result* Q
Timer_I(¬IN, Sample, PT) ≥ PT	FALSE
Timer_I(¬IN, Sample, PT) < PT	TRUE

Condition	*Result* ET
Timer_I(¬IN, Sample, PT) ≥ PT	PT
Timer_I(¬IN, Sample, PT) < PT	Timer_I(¬IN, Sample, PT)
IN	0

Figure 4.8: Re-formalization of timer *TOF*

4.3.3 Timer Pulse (TP)

Similar to timers *TON* and *TOF*, we implement timer *TP* using *Timer_I* operator (Figure 4.9).

Condition		*Result* Q
$\neg Q_{-1}$	$\neg IN_{-1} \wedge IN$	TRUE
	$IN_{-1} \vee \neg IN$	FALSE
Q_{-1}	Timer_I(Q, Sample, PT) ≥ PT	FALSE
	Timer_I(Q, Sample, PT) < PT	TRUE

Condition		*Result* ET
Q		Timer_I(Q, Sample, PT)
¬Q	¬IN	0
	IN	PT

Figure 4.9: Re-formalization of timer *TP*

4.4 Formalizing Real-Time Requirements

In Chapter 3 we provide a formal approach for formalizing the requirements of standard functions and FBs, without mentioning the formalization for timing requirements. However, it is critical to specify timing requirements for many real-time safety-critical systems, e.g., medical device or nuclear power plant. The formalized IEC 61131-3 timers (Section 4.3) are used to build such real-time systems. It is important to specifying timing requirement against with which the conformance of such real-time systems is verified.

We follow the approach for formalizing standard functions and FBs (Chapter 3), i.e., using function tables to perform a complete and disjoint analysis on the input domains. To formalize timing requirements, we incorporate timing tolerances on time duration into the requirements model using the *Held_For* operator (Secion 2.4.3). The *Held_For* operator specifies the left and right tolerances on the time duration of the sustained condition, allowing non-deterministic implementations. The *Held_For_I* operator is a refinement of *Held_For*, meeting the functional and performance requirements (Hu, 2008). Thus we can use the *Held_For_I* operator in the FB requirements model which is verified against by the supplied FB implementation.

More precisely, we use *Held_For_I* (with the time duration $d - \delta L$) to specify *Held_For* operator with timing tolerances, imposing the constraint that only a single value (i.e., $d - \delta L$ where both are declared constants) is chosen from the duration and is used consistently for detecting sustained events. Hence, the use of operator *Held_For_I* resolves the non-determinism by fixing the level of timing tolerances (i.e., as long as the condition has been activated for or longer than $d - \delta L$, the event is guaranteed to be detected). An example of real-time subsystem *Trip Sealed-in* is in Section 7.2.

4.5 Summary

In this chapter we formalized three IEC 61131-3 timer FBs using tabular expressions. We first provided the complete and disjoint requirements for idealized behaviour of timers. In our approach, missing scenarios of timer *TP*

are detected and resolved. To specify practical timers, we then implemented the idealized timers using the pre-verified timing operator *Timer_I*. The resulting formalization can be used to verify timing requirements. To obtain timing requirements, we discussed the incorporation of timing tolerances into a requirements model of real-time systems using the pre-developed *Held_For_I* operator. As a result, our formalized timers can be composed to build complex real-time systems within our approach. The behaviour of such real-time systems can be formally verified against a timing requirements model. All of the specifications are formalized in the PVS environment.

Chapter 5

Formalizing Implementations of Function Blocks

In this chapter we discuss the formalizations for ST and FBD implementations [1]. We present a compositional approach formalizing FBD implementations (Section 5.1). We first formalize non real-time FBD implementations (Section 5.1.1). Following the same approach, we incorporate timing behaviour with real-time FBD implementations using previously formalized IEC 61131-3 timers (Section 5.1.2). We then present a list of translation rules as a guidance for converting ST implementations into PVS, sufficient for handling ST descriptions supplied by IEC 61131-3 (Section 5.2). Consequently, we have a unified, formal framework to verify the correctness of FBs.

5.1 Formalizing FBD Implementations

5.1.1 Formalizing Non Real-Time FBD Implementation

For non real-time FBD implementation, our formalization of each internal FB as a predicate results in *compositionality*: a predicate that formalizes a composite FB is obtained by taking the conjunction of those that formalize its component FBs.

[1] This chapter is based on the work in (Pang et al., 2014a) (under minor revision), and the published work in (Pang et al., 2013a).

To illustrate the case of formalizing a non real-time FBD implementation supplied by IEC 61131-3, consider the following FBD in Figure 5.1:

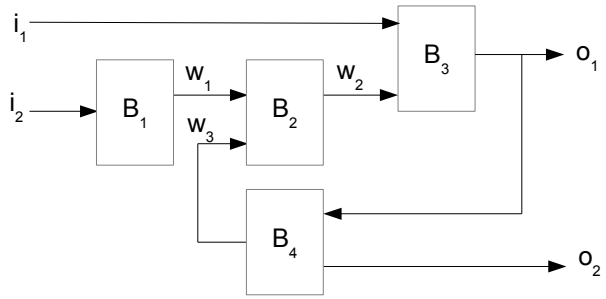


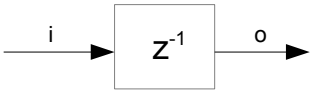
Figure 5.1: Example of non real-time FBD implementation

The example FBD takes as two inputs, i_1 and i_2, and two outputs, o_1 and o_2, consisting of four internal blocks, B_1, ..., B_4. Blocks B_1, ..., B_4 are connected via three wires w_1, w_2, and w_3 (arrowed lines in Figure 5.1). All internal blocks are viewed as black-box components, that have been formalized as predicates B_1_REQ, ..., B_4_REQ. The predicates that formalize the internal components, do not denote those translated from the implementations of IEC 61131-3. Instead, they are input-output relations obtained by our method (presented in Chapter 3). The behavioural descriptions of internal FBs can be written in different languages. For example, the implementation of block B_1 is written in FBD, while the implementation of block B_2 is written in ST. When blocks B_1 and B_2 are used to build more complex FB (as shown in Figure 5.1), their internals (i.e., the implementations of blocks B_1 and B_2) are invisible to block user. Hierarchically, only the interface of FBD is visible to a user who uses it to build other FB, i.e., the implementations of blocks B_1, ..., B_4 and the way they wire up together is hidden from FBD user.

PLC applications often use feedback loops: outputs of a FB are connected as inputs of either another FB, or the FB itself. IEC 61131-3 specifies feedback loops through either a connecting line or shared same names of inputs and outputs. However, feedback values (or of intermediate output values)

cannot be computed instantaneously in practice. For instance, in our example FBD inter-connector w_3 produced by block B_4 depends on o_1 (as an input of block B_4), which indirectly depends on w_3 (through blocks B_2 and B_3). The FBD implementation (Figure 5.1) is unimplementable, since it specifies cyclic dependency (i.e., the value of w_3 is determined by itself at the same time).

We address this issue by introducing a unit delay block z_{-1} (e.g., type of Boolean) with its formalization as shown in Figure 5.2:



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

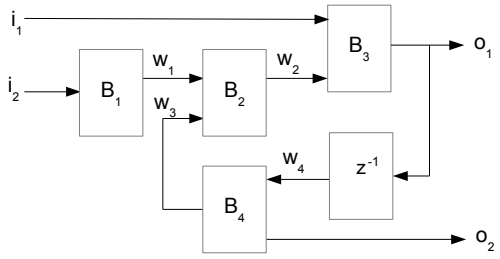
Figure 5.2: Block *Boolean unit delay* declaration and formalization

For any non-initial tick (i.e., $t > 0$), 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 as defined in Figure 5.2 is used in block *SR* in Subsection 7.1.1.2, but have to be the same type. For initial tick (i.e., $t = 0$), it sets the corresponding default value of required type as output value, e.g., block z^{-1} sets the default Boolean value of *FALSE* to output o initially.

We now revise the non real-time FBD (Figure 5.1) by explicitly adding a unit delay block z^{-1} in feedback loop (Figure 5.3a). Accordingly, an additional wire w_4 connects blocks B_4 and z^{-1}. The high-level requirement (as opposed to the implementation supplied by IEC 61131-3) for each internal FB constrains upon its inputs and outputs (i.e., their formalizing predicates B_1_REQ, ..., B_4_REQ exist). To describe the overall behaviour of the revised 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, such that the conjunction of predicates that formalize the internal components hold (predicate *NRT_FBD_IMPL* in Figure 5.3b).

Since we use tabular expressions to specify the predicates for internal

blocks, their behaviours are deterministic. This allows us to easily compose their behaviours using logical conjunction. The conjunction of deterministic components is functionally deterministic.



(a) Revised non real-time FBD example

$$NRT\_FBD\_IMPL(i_1, i_2, o_1, o_2)$$
$$\equiv \exists w_1, w_2, w_3 \bullet$$
$$\begin{pmatrix} B_1\_REQ(i_2, w_1) \\ \wedge \quad B_2\_REQ(w_1, w_3, w_2) \\ \wedge \quad B_3\_REQ(i_1, w_2, o_1) \\ \wedge \quad B_4\_REQ(w_4, w_3, o_2) \\ \wedge \quad z^{-1}(o_1, w_4) \end{pmatrix}$$

(b) Formalizing predicate of non real-time FBD example

Figure 5.3: Revised non real-time FBD example and its formalization

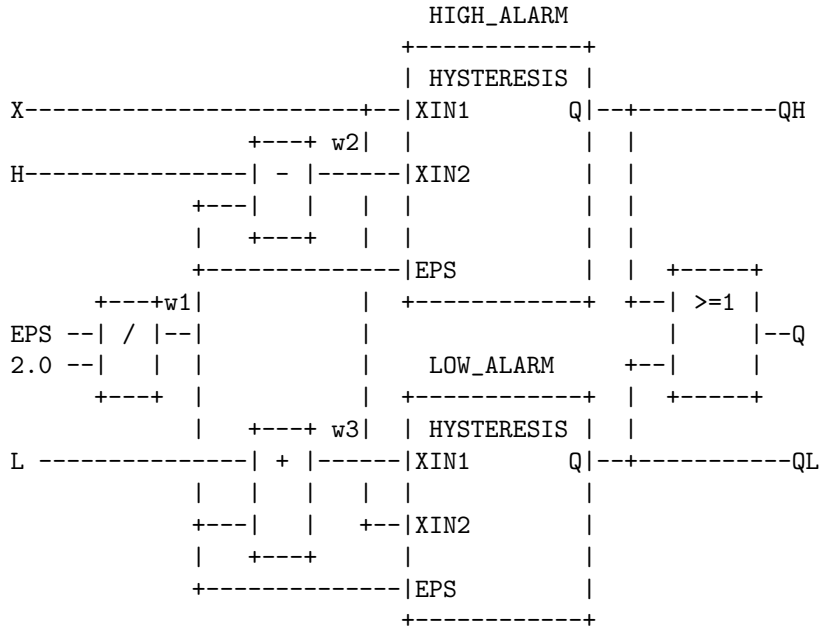


Figure 5.4: FBD implementation of block *LIMITS_ALARM* (IEC, 2013)

For example, consider the FB *LIMITS_ALARM* (Sections 2.5.2), its FBD implementation is shown in Figure 5.4. The FBD implementation of

block *LIMITS_ALARM* consists of six internal blocks, division ("/"), subtraction ("−"), disjunction ("≥ 1"), addition ("+"), and two instances of hysteresis blocks ("*LOW_ALRAM*" and "*HIGH_ALRAM*"). They are wired up via inter-connectors w_1, w_2, and w_3. The predicate formalizes the FBD implementation by taking the conjunction of requirement predicates that formalize its internal blocks.

More precisely, the following mathematical Formula 5.1 specifies the FBD implementation of block *LIMITS_ALARM*. Predicates *DIV*, *SUB*, *DISJ*, *ADD*, *HYSTERESIS_REQ* formalize the input-output requirements of internal blocks division, subtraction, disjunction, addition, and two hysteresis blocks[2]. There are two explanations for Formula 5.1: (1) Since internal blocks are formalized as relations on timed variables, we adopt λ-expression to specify constant input. For example, we specify the constant input of block division (i.e., 2.0) as $\lambda t : 2.0$, where $t \in tick$; and (2) Instead of a timed variable, the first input of block *SUB* is a function H which takes inputs L and EPS as arguments. We will explain this constraint in detail in Subsection 7.1.2.3.

$$
\begin{aligned}
& LIMITS\_ALARM\_IMPL(X, H, L, EPS, QH, Q, QL) : bool \\
\equiv\ & \exists\, w_1, w_2, w_3 \bullet \\
& DIV(EPS, \lambda t : 2.0, w_1) \wedge \\
& SUB(H(L, EPS), w_1, w_2) \wedge \\
& DISJ(QH, QL, Q) \wedge \\
& ADD(L, w_1, w_3) \wedge \\
& HYSTERESIS\_REQ(X, w_2, w_1, QH) \wedge \\
& HYSTERESIS\_REQ(w_3, X, w_1, QL)
\end{aligned}
\tag{5.1}
$$

We formalize Formula 5.1 as predicate *LIMITS_ALARM_IMPL* in PVS (Figure 5.5). Predicates *DIV*, *SUB*, *DISJ*, *ADD*, *HYSTERESIS_REQ* formalize requirements of internal blocks.

5.1.2 Formalizing Real-Time FBD Implementation

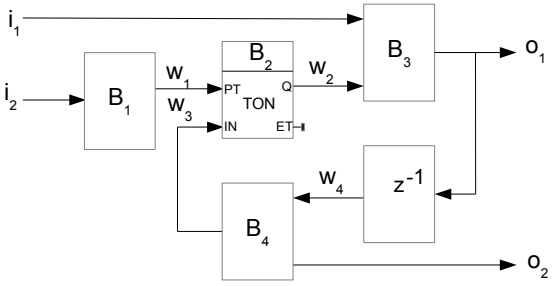
We present a compositional approach for formalizing non real-time FBD implementation (Section 5.1.1). We apply the same compositional approach on

[2]The complete formalizations for internal blocks are in Appendixes C and F

```
LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL): bool =
  EXISTS (w1,w2,w3):
    DIV(EPS,LAMBDA (t1: tick): 2.0,w1) &
    SUB(H(L,EPS),w1,w2) &
    DISJ(QH,QL,Q) &
    ADD(L,w1,w3) &
    HYSTERESIS_REQ(X,w2,w1,QH) &
    HYSTERESIS_REQ(w3,X,w1,QL)
```

Figure 5.5: PVS formalization of the *LIMITS_ALARM* implementation

real-time FBD implementation, with the difference that timing behaviour is incorporated by IEC 61131-3 timers. Hence, at the implementation level we incorporate with IEC 61131-3 timers as internal blocks. Timer blocks adopt operator *Timer_I* (Hu, 2008) for counting the elapsed time of some monitored condition (Chapter 4).



(a) Real-time FBD example

$$
\begin{aligned}
&RT\_FBD\_IMPL(i_1, i_2, o_1, o_2) \\
&\equiv \exists w_1, w_2, w_3, w_4, et \bullet \\
&\begin{pmatrix}
& B_1\_REQ(i_2, w_1) \\
\wedge\ & TON(w_3, w_1, w_2, et) \\
\wedge\ & B_3\_REQ(i_1, w_2, o_1) \\
\wedge\ & B_4\_REQ(w_4, 3_3, o_2) \\
\wedge\ & z^{-1}(o_1, w_4)
\end{pmatrix}
\end{aligned}
$$

(b) Formalizing predicate of real-time FBD example

Figure 5.6: Real-time FBD example and its formalization

To illustrate the case of formalizing a real-time FBD implementation, consider the same FBD as shown in Figure 5.1. We replace internal block B_2 with a IEC 61131-3 timer (e.g., timer *TON*), and leave others unchanged (Figure 5.6a). As a result, blocks B_1, B_3, and B_4 are non real-time blocks, while B_2 is on-delay timer. The high-level requirements for all internal FBs constraining upon their inputs and outputs are already formalized as predicates B_1_REQ, B_3_REQ, B_4_REQ, and *TON* using function tables. To formalize real-time

FBD implementation (i.e., one built from IEC 61131-3 timers), we again take advantage of our compositional formalization to describe the overall behaviour by existentially quantifying over the list of its inter-connectives, such that the conjunction of predicates that formalize the internal components hold.

5.2 Formalizing ST Implementations

As discussed in Section 2.2.2, 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 (IEC, 2003) 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.

5.2.1 Scope 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 (IEC, 2003) 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 (IEC, 2003) (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 (IEC, 2003).

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 (IEC, 2003).

- *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.

- *Non-Nested Iteration*. Each iteration in the body of the ST code is not nested in another iteration. This allows us to formalize each iteration as a recursive function in PVS. As far as the formalization of function blocks in Annex F (IEC, 2003) is concerned, this assumption is always met. However, to relax this assumption, we will need to introduce a mechanism for handling such nested iterations.

Given the above assumptions, and the richness of the specification language and supported libraries of PVS, our ST-to-PVS translation is reasonably straightforward. As we shall see, 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 this section.

```
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;
  ELSIF 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]
 t: VAR tick
 v1: VAR [tick -> [[T1]]]
 v2: VAR [tick -> [[T2]]]
 F_ST_IMPL(v1, v2): bool =
   FORALL t:
     EXISTS (v3, Q: [tick -> [[T2]]):
       f_REQ([[e1]], [[e2]], v3) AND
       G_REQ([[e3]], [[e4]], Q) AND
       v2(t) =
        IF init(t) THEN [[T2_INI]]
        ELSE LET PREV = v2(pre(t)) IN
          TABLE
            | [[e5]]                      | v3(t) ||
            | NOT [[e5]] AND [[e6]]       | Q(t) ||
            | NOT [[e5]] AND NOT [[e6]] |PREV||
          ENDTABLE
        ENDIF
END F
```

Figure 5.7: *ST–to–PVS* translation: a contrived example

5.2.2 An Overview and an Example

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 $[\![\_]\!] : ST \nrightarrow PVS$ 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.

Translation Example Figure 5.7 presents an overview of our *ST–to–PVS* translation. On the left 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 and of default initial value T_2_INI). 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; (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 specified explicitly to bind those argument values.

On the RHS of Figure 5.7 we have a PVS theory[3] 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 Figure 5.7:

- For readability, we retain names of the declared variables of function and function block. To invoke a function or a function block, the name of the relation has the *_REQ* suffix to denote that it is the requirement predicate for the corresponding function or function block in ST.

- The theory is 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). We formalize ST functions and function blocks as input-output relations whose parameters are time-dependent (i.e., function blocks constrain inputs and outputs over time). 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

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

function or function block 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 ST code.

- We define the input-output relation using logical conjunction.

  - The first two conjuncts constrain the input and output values of function and function block invocations, so that their output values (i.e., v_3 and Q) can be referenced later in the table. For each invocation that occurs in the context of some (nested) conditional branch, it is invoked by satisfying the corresponding condition row (e.g., the invocation of function block G is invoked by $\neg[\![e_5]\!] \wedge [\![e_6]\!]$).

  - 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, they are set to the default initial values for associated data types (e.g., $[\![T_2\_INI]\!]$). 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[4]; or (3) itself at the previous time tick if $\neg[\![e_5]\!] \wedge \neg[\![e_6]\!]$.

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

5.2.3 Translation Rules

In this section, we provide the list of translation rules sufficient for translating ST code supplied by Annex F (IEC, 2003) 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 Figure 5.7, 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) T_INI, T_1_INI, T_2_INI, *etc.* denote default initial values of ST types; (7) S, S_1, S_2, *etc.*, denote ST statements; and (8) i denotes a loop counter.

Translation Context Analysis Our translation function $[\![\_]\!]$ often needs to carry around context information from the translation of one component to another. Since all ST variables are mapped into time-variant sequences 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. As a result, given that v is the target variable, and $\rho \in \{init,\ ninit,\ seq\}$ is the context for variable references, we write $[\![\_]\!]^{\rho}_{v}$ to denote the corresponding translation. We drop the context when it is unnecessary for the translation in question to proceed. For example, we write $[\![\_]\!]^{seq}$ to translate 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.

Also, 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 ex-

tract the exact conditions where they occur and make them as table conditions accordingly (e.g., see Figure 5.7 where the invocation of function block G is properly invoked).

We may calculate the *write* set of a given assignment statement S – the set of variables that get assigned – so as to determine if a variable has already been written. If the assignment target matches the context variable v (i.e., $v \in write(S)$), variable v is written (or updated) according to our translation rules; otherwise, we return a special value ϵ. In the context of ST language, variables retain the values from the last execution cycle if they are not written in the current execution cycle (i.e., indicated by returning ϵ).

Translation Rule 1: Function Block Definition We present the translation rules to define function blocks in Table 5.1. The definition of each function block consists of two parts: variable declarations and body definition (denoted as S). Without loss of generality, the function block declares one variable from each of the categories (i.e., input, output, and local variable) with implicit initialization for output variable and explicit initialization for local variable.

As illustrated in Figure 5.7, 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 $delta\_t$. The ST function block body S is translated into the PVS relation F_ST_IMPL that constrains the values of its parameters: the list of inputs and outputs. Inside the definition of this relation, the invoked functions and function blocks are mapped to the corresponding requirements predicates (i.e., each has $\_REQ$ suffix). We use an existential quantification to hide: (1) the list of local variables (i.e., v_3); and (2) return values from functions or function block invocations. For (2), we use q (of Type T_q) and Q (of type T_Q) to denote the list of return values of functions and function blocks that are referenced in S, if any.

In the case of functions and function block invocations, as discussed, we model each function and function block as a relation (a Boolean function) on the lists of inputs and outputs. Each input or output is time-dependent and thus modelled as a timed sequence. For example, the translated argument

ST FB Definition	PVS Theory

FUNCTION_BLOCK F
  VAR_INPUT
  v_1: T_1
  END_VAR
  VAR_OUTPUT
  v_2: T_2
  END_VAR
  VAR
  v_3: T_3 := c
  END_VAR
  S
END_FUNCTION_BLOCK
	F[(IMPORTING $Time$) $delta\_t$: $posreal$]: THEORY
	BEGIN IMPORTING $ClockTick$[$delta\_t$]
	$[\![v_1:\ T_1]\!]$
	$[\![v_2:\ T_2]\!]$
	$F\_ST\_IMPL(v_1, v_2)$: $bool$ =
	FORALL (t: $tick$):
	  EXISTS ($[\![v_3:\ T_3]\!]$, $[\![q:\ T_q]\!]$, $[\![Q:\ T_Q]\!]$):
	  $f\_REQ([\![e_1]\!]^{seq}, \ldots, [\![e_m]\!]^{seq}, q)$ AND
	  $F\_REQ([\![e_1]\!]^{seq}, \ldots, [\![e_n]\!]^{seq}, Q)$ AND
	  $v_3(t)$ = IF $init(t)$ THEN $[\![c]\!]$ ELSE $[\![S]\!]_{v_3}$ ENDIF AND
	  $v_2(t)$ = IF $init(t)$ THEN $[\![S]\!]^{init}_{v_2}$ ELSE $[\![S]\!]_{v_2}$ ENDIF
	END F

Table 5.1: *ST–to–PVS*: function block definition

values of a function or a function block are expected to be timed sequences $[\![e_1]\!]^{seq}$, ..., $[\![e_n]\!]^{seq}$, if the associated inputs are i_1, i_2, ..., i_n. In some cases, multiple function blocks use the same name for their outputs (e.g., both FB_1 and FB_2 has output Q). We resolve the ambiguity by adding their names as prefixes (i.e., FB_1_Q and FB_2_Q). In the same scope of context, these output variables may be referenced to assign values to other variables. In the case where an invocation occurs within some (nested) conditional branch, we often refer to these output variables as the results under the satisfying conditions.

For local and output variables, we consider the initial and non-initial time ticks separately. In the case of the initial time tick, we constrain values of variables according to their specified initial values, if specified[5]. For example, v_3 is explicitly initialized with the constant $[\![c]\!]$ (of type $[\![T_3]\!]$, while v_2 is implicitly initialized with the default value T_2_INI (or equivalently $[\![S]\!]^{init}_{v_2}$). In the case of non-initial time ticks, we generate a constraint that encodes its intended update via $[\![S]\!]_{v_2}$. Given an ST statement S, our translation function effectively "projects" S onto the target variable (e.g., v_2). For example, for output variable v_2, we generate its constraint of intended update via $[\![S]\!]_{v_2}$.

[5]According to the IEC 61131-3 (IEC, 2003), uninitialized variable are defaulted to some values for the associated types (e.g., the default value for Boolean is *FALSE*).

The resulting list of "guarded values", where guards correspond to the branching conditions of the `IF-THEN-ELSE` statements in the source ST code can then be straightforwardly encoded as a *TABLE* expression in PVS. For example, as already seen in Figure 5.7, the projection onto output variable v_2 results in three guarded values. When there are multiple output and local variables, we combine all these constraints via logical conjunctions (e.g., v_2 and v_3).

Translation Rule 2: Interface Declaration We present the translation rules for variable declarations in Table 5.2. We retain 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.

ST Variable Declaration	PVS Variable Declaration
Variable Declaration without Initialization	
`v: T`	v`: VAR [`*tick*` -> `$[\![T]\!]$`]`
Variable Declaration with Initialization	
`v: T := c`	v`: VAR [`*tick*` -> `$[\![T]\!]$`]`

Table 5.2: *ST–to–PVS*: variable declarations

As mentioned above, each ST variable is time-dependent, we thus parameterize the PVS type $[\![T]\!]$ (translated from the ST type `T`) by discrete time ticks (Section 2.4). Since PVS does not handle initialization in the declaration part, our translation rule does not consider whether or not an initial value is specified in the source ST code at the level of variable declarations. Instead, such information is handled at the level of function block definitions (Table 5.1), where the context *init* is passed for translating the specified initial value (i.e., $[\![c]\!]$ of type $[\![T]\!]$).

Translation Rule 3: Data Types We present the translation rules for ST types in Table 5.3. There are four categories of ST types: (1) primitive types (e.g., integers, reals and Booleans); (2) built-in types (e.g., words and time);

(3) structured data type (e.g., aggregation data types[6] and arrays); and (4) user-defined function blocks (e.g., F).

ST Type	PVS Type
Primitive Types	
`INT`	*int*
`REAL`	*real*
`BOOL`	*bool*
Built-In Types	
`WORD`	*bvec*
`TIME`	*tick*
Structured Types	
`STRUCT` $\mathtt{v}_1$`:` $\mathtt{T}_1$`;` $\mathtt{v}_2$`:` $\mathtt{T}_2$`; END_STRUCT`	`[#`v_1`:` $[\![T_1]\!]$`,` v_2`:` $[\![T_2]\!]$`#]`
`ARRAY[`$\mathtt{e}_1$ `..` $\mathtt{e}_2$`] OF T`	`ARRAY[subrange(`$[\![e_1]\!]^{init}$`,` $[\![e_2]\!]^{init}$`) ->` $[\![T]\!]$`]`
User-Defined Function Blocks	
`F`	*F_REQ*

Table 5.3: *ST–to–PVS*: data types

We interpret each type in PVS as follows:

- Primitive types – we adopt the direct corresponding types in PVS.
- Built-in types – we import relevant theories to support their operations (i.e., bit vectors *bvec* from the *bv* prelude library represents type `WORD`, and *tick* in Section 2.4 represents type `TIME`).
- Structured types – we need two structured data types: aggregation data types and arrays. The ST aggregation data types are encoded as PVS records. Each PVS record element corresponds to one element in the ST `STRUCT`. For example, element $\mathtt{v}_1$ (of type $\mathtt{T}_1$) corresponds to v_1 (of type $[\![T_1]\!]$). Any element of a variable (declared of aggregation data type) can be referneced. For example, l is of such type and the first element of l is referneced via l`'`v_1 in PVS. Another structured data type is array which is directly supported in PVS. The *ARRAY* type in PVS is essentially

[6]Aggregation data type is a structured data type which has been declared using `STRUCT` in ST. An example is structure `ANALOG_LIMITS` which implements the declarations of parameters for analog signal monitoring.

a special case of type *FUNCTION* with a contiguous subset of integers for the domain and a proper range type. We use the operator *subrange* supported by PVS to denote an integer range with specified lower and upper bounds. PVS automatically generates the associated subtype TCCs for the validity of indices of any variable declared as of type *ARRAY*. Lower and upper bounds, $\mathtt{e}_1$ and $\mathtt{e}_2$, should be integer expressions, which is guaranteed by our assumption of input type-correctness. As the size of an array remains unchanged 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.

- Function block – we simply reuse its name with the *_REQ* suffix, assuming that its input-output specification (i.e., *F_REQ*) is translated into a PVS theory.

Translation Rule 4: Basic Assignments Table 5.4 presents the translation rules for basic variable assignments in ST. In all cases of assignments, we return a special value ϵ when the assignment target does not match the context variable v. Otherwise, the variable assignment is translated in a straightforward manner: return the translated value of the assignment source (i.e., $[\![e]\!]^{ninit}$). A match in the case of an aggregation variable assignment returns the translated value of the assignment source (i.e., $[\![e_2]\!]^{ninit}$) to the corresponding element (i.e., $v\texttt{'}[\![e_1]\!]^{ninit}$). 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.

ST Statement	PVS Expression	Side Condition
v := e	$[\![e]\!]^{ninit}$	$v \in write(e)$
x := e	ϵ	$x \notin write(e)$
v.$\mathtt{e}_1$:= $\mathtt{e}_2$	$[\![e_2]\!]^{ninit}$	$v.e_1 \in write(e)$
x.$\mathtt{e}_1$:= $\mathtt{e}_2$	ϵ	$x.e_1 \notin write(e)$
v[$\mathtt{e}_1$] := $\mathtt{e}_2$	$v(pre(t))$ `WITH [`$[\![e_1]\!]^{ninit}$ `:=` $[\![e_2]\!]^{ninit}$`]`	v[$\mathtt{e}_1$] $\in write(e)$
x[$\mathtt{e}_1$] := $\mathtt{e}_2$	ϵ	v[$\mathtt{e}_1$] $\notin write(e)$

Table 5.4: *ST–to–PVS*: basic assignments

Translation Rule 5: Conditional Assignments Table 5.5 presents the translation rule for the `IF-THEN-ELSE` conditional statement in ST (see an example in Figure 5.7). In the case of that the context variable v is not written at all by any of the body statements S_i ($0 \leq i \leq n$), we return ϵ. Otherwise, the execution semantics of the ST conditional statement is translated into a list of (disjoint and complete) guards of tables in PVS. Each guard (e.g., $\texttt{NOT}(\bigvee_{j=0}^{k-1} [\![e_j]\!]^{ninit})$ `AND` $[\![e_k]\!]^{ninit}$, $1 \leq k \leq n-1$) is defined as the conjunction of the translated value of the corresponding branching Boolean expression (e.g., $[\![e_k]\!]^{ninit}$) and the translated values of all earlier branching Boolean expressions (e.g., $\texttt{NOT}(\bigvee_{j=0}^{k-1} [\![e_j]\!]^{ninit})$). We use $\bigvee$ as a meta-operator to denote the disjunction of a sequence of expressions occurring in PVS.

ST Statement	PVS Expression	Side Condition

`IF` $\texttt{e}_0$
`THEN` $\texttt{S}_0$
`ELSIF` $\texttt{e}_1$
`THEN` $\texttt{S}_1$
...
`ELSIF` $\texttt{e}_{n-1}$
`THEN` $\texttt{S}_{n-1}$
`ELSE` $\texttt{S}_n$
`END_IF`
	`TABLE`
	$\texttt{\|}[\![e_0]\!]^{ninit}$ $\texttt{\|}[\![S_0]\!]_v^{ninit}$ $\texttt{\|\|}$
	$\texttt{\|NOT(}[\![e_0]\!]^{ninit}\texttt{)AND}[\![e_1]\!]^{ninit}$ $\texttt{\|}[\![S_1]\!]_v^{ninit}$ $\texttt{\|\|}$
	...
	$\texttt{\|NOT(}\bigvee_{j=0}^{n-2}[\![e_j]\!]^{ninit}\texttt{)AND}[\![e_{n-1}]\!]^{ninit}$ $\texttt{\|}[\![S_{n-1}]\!]_v^{ninit}$ $\texttt{\|\|}$
	$\texttt{\|NOT(}\bigvee_{j=0}^{n-1}[\![e_j]\!]^{ninit}\texttt{)}$ $\texttt{\|}[\![S_n]\!]_v^{ninit}$ $\texttt{\|\|}$
	`ENDTABLE`
		$\exists i \bullet v \in write(S_i)$

	ϵ	$\forall i \bullet v \notin write(S_i)$

Table 5.5: *ST–to–PVS*: conditional statement

The resulting PVS table (i.e., Table 5.5) is a list of translated values that are guarded by corresponding branching conditions. 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. If the `ELSE` part is missing from the source ST code, then v retains the value of itself at the previous tick. Accordingly, we specify $v(pre(t))$ as the return value in the PVS table.

Translation Rule 6: Iteration Statements Two types of iterations are used for the purpose of IEC 61131-3 (IEC, 2003): (1) array assignments; and (2) sum of array elements. Similar to the case of translating the conditional

statements, if the context variable v is not written by the loop body statement S, then we return ϵ. Otherwise, the target ST code is encoded as (1) an universally quantified predicate for the case of array assignments; and (2) a recursive addition for the case of the sum of array elements.

For case (1), the loop body S is of form `A[i] := e`, where `A` is an array which elements are assigned by the evaluated values of `e`. The assignments statements are encoded as a conjunction of the assignment for each element, i.e., $\bigwedge_{[\![e_1]\!]^{ninit}}^{[\![e_2]\!]^{ninit}} [\![A]\!](i) = [\![e]\!]^{ninit}$. Equivalently, we generate an universally quantified predicate over array elements, and the indices of the assigned elements are bounded by the translated lower and upper bounds (i.e., $[\![e_1]\!]^{ninit}$ and $[\![e_2]\!]^{ninit}$). For case (2), the loop body S is of form `v := v + A[i]`, where some consecutive elements of array A are added. We formulate a recursive function *loop* implementing the addition of array elements in PVS. Its termination is guaranteed by terminate-TCC proof obligation in PVS. The source ST code is then encoded as an instantiation of function *loop* added by the initial value of v (i.e., $[\![S]\!]_v^{init}$). We instantiate the *loop* 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 $[\![e_2]\!]^{ninit}$); and (3) the translated array A (i.e., $[\![A]\!]$).

Translation Rule 7: Sequential Assignments We present the translation rules for the sequential composition of assignments in Table 5.7. We discuss two scenarios: (1) single assignments; and (2) non-single assignments. For case (1), when translating assignments statements, we aim to retrieve the list of guarded values for the context variable v, if v is only assigned once in the 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. For case (2), we return the list of guarded values generated in statement S_2 for the context variable v, within which v has been updated by the list of guarded values generated in statement S_1, if v is assigned both in bodies S_1 and S_2.

We present the translation rules for ST expressions in Tables 5.8 to 5.11. As we have seen so far, each translation of an expression requires a context of variable reference ρ (i.e., *init*, *ninit*, or *seq*). For the purpose

ST Statement	PVS Expression	Side Condition
FOR i:= e_1 TO e_2 DO A[i] := e; END_FOR	FORALL(i: subrange($[\![e_1]\!]^{ninit}$, $[\![e_2]\!]^{ninit}$)): $[\![A]\!](i)$ = $[\![e]\!]^{ninit}$	$v \in write(S)$
	ϵ	$v \notin write\ (S)$
FOR i:= e_2 TO e_1 BY -1 DO v := v + A[i]; END_FOR	$loop(i, j, [\![A]\!])$: RECURSIVE $[\![T]\!]$ = IF $j = i$ THEN $[\![A]\!](j)$ ELSE $loop(i, j - 1) + [\![A]\!](j)$ ENDIF MEASURE j v = $loop([\![e_1]\!]^{ninit}, [\![e_2]\!]^{ninit}, [\![A]\!]) + [\![S]\!]_v^{init}$	$v \in write(S)$
	ϵ	$v \notin write\ (S)$

where $[\![e_1]\!]^{ninit} < [\![e_2]\!]^{ninit}$

Table 5.6: *ST–to–PVS*: iteration statements

of IEC 61131-3 (IEC, 2003), we consider four categories of expressions: (1) variable referencing (Table 5.8); (2) literal expressions (Table 5.9); (3) standard function invocations (Table 5.10); and (4) operations (Table 5.11).

Translation Rule 8: Variable Referencing Expressions In the case of variable referencing (Table 5.8), the referenced variable may be of a basic type (e.g., *real*), aggregation type, array type, or to an output of some function block

ST Statement	PVS Expression	Side Condition
Single Assignments		
S_1 ; S_2	$[\![S_1]\!]_v$	$v \in write(S_1) \wedge v \notin write(S_2)$
	$[\![S_2]\!]_v$	$v \notin write(S_1) \wedge v \in write(S_2)$
	ϵ	$v \notin write(S_1) \wedge v \notin write(S_2)$
Non-single Assignments		
S_1 ; S_2	$[\![S_2]\!]_{[\![S_1]\!]_v}$	$v \in write(S_1) \wedge v \in write(S_2)$

Table 5.7: *ST–to–PVS*: sequential composition

that was invoked previously. The treatment of each kind of these variables is similar: depending on the given context ρ of the variable referenced, we generate the references accordingly. In the case of aggregation variables and array indexing, we propagate the variable reference context ρ to the translation of its specified element and index. Furthermore, the sequential execution of the source ST code makes it possible to refer to the value of context variable at previous tick. 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, then we write $v(pre(t))$ to denote its value from the previous time tick; otherwise, we refer to its latest value at the current time tick (i.e., $v(t)$).

ST Expression	PVS Expression	Side Condition
Referencing Variable		
v	$v(0)$	$\rho = init$
	v	$\rho = seq$
	$v(t)$	$\rho = ninit \wedge v \in write(S)$
	$v(pre(t))$	$\rho = ninit \wedge v \notin write(S)$
Referencing Elements of Aggregation Date		
v.e	$v\texttt{'}[\![e]\!]^{init}$	$\rho = init$
	$v\texttt{'}[\![e]\!]^{seq}$	$\rho = seq$
	$v(t)\texttt{'}[\![e]\!]^{ninit}$	$\rho = ninit \wedge v \in write(S)$
	$v(pre(t))\texttt{'}[\![e]\!]^{ninit}$	$\rho = ninit \wedge v \notin write(S)$
Referencing Elements of Array		
v[e]	$v(0)\,\texttt{[}[\![e]\!]^{init}\texttt{]}$	$\rho = init$
	$v\,\texttt{[}[\![e]\!]^{seq}\texttt{]}$	$\rho = seq$
	$v(t)\,\texttt{[}[\![e]\!]^{ninit}\texttt{]}$	$\rho = ninit \wedge v \in write(S)$
	$v(pre(t))\,\texttt{[}[\![e]\!]^{ninit}\texttt{]}$	$\rho = ninit \wedge v \notin write(S)$
Referencing Output of Function Block		
F.Q	$Q(0)$	$\rho = init$
	Q	$\rho = seq$
	$Q(t)$	$\rho = ninit \wedge Q \in write(S)$
	$Q(pre(t))$	$\rho = ninit \wedge Q \notin write(S)$

Table 5.8: *ST–to–PVS*: variable referencing expressions

Translation Rule 9: Literal Expressions We present the translation rules for literal expressions in Table 5.9. Integer literals (e.g., `2`), real literals (e.g., `2.0`), time literals (e.g., `delta_t`), and Boolean literals (e.g., `TRUE`) can all be directly used in PVS. However, since we model each variable as a timed sequence, we use the lambda expression to create a constant timed sequence. For example, when the context of a 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 parameterized by time ticks (e.g., $LAMBDA(t : tick) : 2.0$).

ST Expression	PVS Expression	Side Condition
Integer, Real, or Boolean Literal		
l	l	$\rho \neq seq$
	`LAMBDA(`t`: tick):` l	$\rho = seq$
Time Literal		
t	0	$\rho = init$
	$[\![t]\!]^\rho$	$\rho \neq init \wedge \exists(n : nat) \bullet n \times delat\_t = t$

Table 5.9: *ST–to–PVS*: literal expressions

Translation Rule 10: Function Invocation Expressions Table 5.10 presents the translation rule for the invocation of a standard function f. For the standard function invocation, we: (1) refer to the output of the predicate form of f (i.e., $[\![q]\!]^\rho$); or (2) pass the translated argument values in the order that is defined in the functional form definition of f (i.e., $f([\![e_1]\!]^\rho, \ldots, [\![e_n]\!]^\rho)$.

ST Expression	PVS Expression	Side Condition
Standard Function Invocation		
f($p_1 := e_1, \ldots, p_n := e_n$)	$[\![q]\!]^\rho$	f invoked as predicate
	f`(`$[\![e_1]\!]^\rho$`, ...,` $[\![e_n]\!]^\rho$`)`	f invoked as function

Table 5.10: *ST–to–PVS*: function invocation expressions

Translation Rule 11: Operations Expressions The translation rules for operations expressions are presented in Table 5.11. For all the unary (denoted by $\oplus_1$) and binary (denoted by $\oplus_2$), we can find the obvious corresponding operators in PVS. For examples, we generate the ST numerical expressions (e.g., `1 + 2`), relational expressions (e.g., `EPS > 0`), and logical expressions (e.g., `e1 & e2`) directly in PVS. To translate the operands, we propagate the given context ρ (e.g., $[\![e]\!]^\rho$).

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

Table 5.11: *ST–to–PVS*: operation expressions

Our example translation in Table 5.1, though informative, is nevertheless contrived. We provide a complete example translation that is applied to the *HYSTERESIS* function block from Annex F (IEC, 2003) in the Appendix F. This example illustrates the generation of nested PVS tables, mapped from the nested `IF-THEN-ELSE` statement in the source ST code.

5.3 Summary

The main contribution of this chapter is a method for formalizing the implementation of FBs using tabular expressions and the PVS theorem prover.

We formalized the FB implementations written in FBD and/or ST. To formalize FBD implementations, we provided a compositional approach to specify the overall behaviour which is built on the formalization of its internal components. For those FBs that are built on timers, our compositional approach is still applicable, except that we use the re-formalized timer component(s). This allows us to verify the correctness of such real-time FBs concerning timing resolution. To formalize the ST implementation, we developed a list of translation rules to convert ST descriptions into PVS, sufficient for formulating all ST specifications supplied by IEC 61131-3 (IEC, 2003). Hence,

using our method, we are able to formalize all FBs written in FBD and ST in IEC 61131-3 (IEC, 2003). In the next chapter, we reason about the correctness of FBs based on our formalizations.

Chapter 6

Proving Correctness and Consistency of Function Blocks

In this chapter we discuss how to ensure the correctness and consistency of FB implementation [1]. In Section 6.1, we verify the *correctness* of FB implementation with respect to its input-output requirements. We discuss the correctness verification for those FBs that require timing behaviour and for those that do not, respectively. In Section 6.2, we then verify the *consistency* of FB implementation. For those FBs supplied with both ST and FBD in the standard, we verify the functional equivalence between these two implementations in Section 6.3. By proving the functional equivalence between two implementations, we only need to prove the correctness theorem with either one implementation.

6.1 Proving Correctness of Function Block Implementation

Given the requirements and implementation predicates of a FB, its correctness can be expressed by a logical implication which asserts that the implementation conforms to its input-output requirements. The requirements of a FB are

[1] This chapter is based on the published works in (Pang et al., 2013a) and (Pang et al., 2015).

formalized as a predicate constraining the allowable input and output values. The implementation of a FB, however, describes the combinations of values which not only appear on the input and output variables (i.e., variables observable by FB user), but also auxiliary variables (i.e., variable not observable by FB user). The correctness theorem is formalized in terms of behaviour observed through input and output values, to expresses a relationship between the implementation and its input-output requirements. We prove the same correctness theorem to verify both non real-time and real-time FBs. This is shown in Figure 1.1 on page 10 as proofs of *correctness*.

Verifying the Correctness of Non Real-Time FB

To be able to reflect the intent of a FB design, the requirements usually specify a more abstract (and probably non-deterministic) view of its intended behaviour than the corresponding implementation. The relationship used to express correctness between an implementation and its requirements therefore, in general, is by logical implication rather than strict logical equivalence.

For a given non real-time FB with input list $i_1, \ldots, i_m$ and output list $o_1, \ldots, o_n$, we formulate the correctness of its implementation (say *FB_IMPL*) with respect to the intended behaviour (say *FB_REQ*) as:

$$\begin{aligned} &\forall\, i_1, \ldots, i_m \bullet \forall\, o_1, \ldots, o_n \bullet \\ &\quad FB\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \Rightarrow FB\_REQ(i_1, \ldots, i_m, o_1, \ldots, o_n) \end{aligned} \tag{6.1}$$

Instead of logical implication, logical equivalence can also be a candidate way of specifying correctness. The correctness of an implementation is then asserted by the following theorem:

$$\begin{aligned} &\forall\, i_1, \ldots, i_m \bullet \forall\, o_1, \ldots, o_n \bullet \\ &\quad FB\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv FB\_REQ(i_1, \ldots, i_m, o_1, \ldots, o_n) \end{aligned} \tag{6.2}$$

The above theorem states that, the behaviour described by implementation is identical to that formalized by its requirement. Since our formalizing requirements of FBs are specified using function tables that are completely deterministic, proving theorem 6.1 entails theorem 6.2.

For example, to prove that the FBD implementation of block *LIMITS_ALARM* (Figure 5.4) is *correct* with respect to its requirements (Figure 3.3), we must prove the following:

$$\begin{aligned} &\forall\, H, X, L, EPS \bullet \forall\, QH, Q, QL \bullet \\ &\quad LIMITS\_ALARM\_IMPL(H, X, L, EPS, QH, Q, QL) \\ &\qquad \Rightarrow LIMITS\_ALARM\_REQ(H, X, L, EPS, QH, Q, QL) \end{aligned} \tag{6.3}$$

The above correctness check is encoded in PVS as follows:

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

Verifying the Correctness of Real-Time FB

We prove the same correctness theorem (Equation 6.1) to verify a FB with timing behaviour. To deal with timing properties, we incorporate the notion of timing tolerances. Verifying the correctness of FBs with or without timers both fit into the same framework, and the only difference occurs when timing requirements are used to monitor elapsed time. The timing tolerance argument incorporated with timing requirements is passed to a pre-verified timing operator.

Figure 6.1 shows the same verification process for real-time FBs. We reuse the operator *Held_For_I* to incorporate the notion of timing tolerances for formalizing input-output requirements (Sections 4.4), and reuse the operator *Timer_I* to formalize IEC 61131-3 timers for formalizing implementations (Section 5.1.2). The requirements specification is documented as a Software Requirements Specification (SRS), and the implementation specification is as a Software Design Description (SDD). The verification goal of correctness for real-time FBs remains consistent within our framework. The relationship between these two timing operators is proved as a general Theorem *TimerGeneral_I* (Hu, 2008): $(P\ Held\_For\_I\ (timeout - \delta L))$ is equivalent

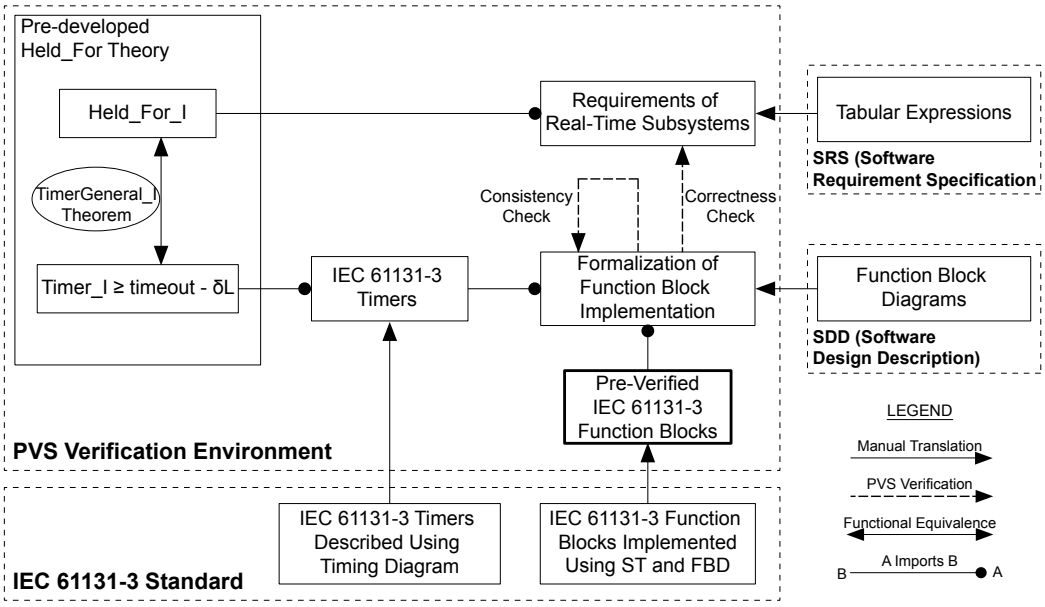


Figure 6.1: Framework of applying the same approach on real-time FBs

to $(Timer\_I\ (P) \geq timeout - \delta L)$. For real-time FB, the use of Theorem *TimerGeneral_I* simplifies the verification work for verifying its correctness. More precise definitions are provided in Chapter 2.

We follow the same proof patterns as to those FBs that do not use timers. We apply our approach to verify a realistic real-time subsystems *Trip Sealed-In* (Section 7.2) from industry. The patterns of correctness proofs are identified in Section 7.3.

6.2 Proving Consistency of Function Block Implementation

For a given implementation, we 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 implementation predicate holds. Otherwise, the implementation trivially implies any requirements. In addition to the correctness theorem (Equation 6.1), proving the consistency theorem ensures that the the correctness theorem is proved not merely because it is inconsistent it-

self. We prove the same consistency theorem to verify both non real-time and real-time FBs. This is shown in Figure 1.1 on page 10 as proofs of *consistency*.

Proving Consistency of Non Real-Time Function Block

Consider the same general non real-time FB (Section 6.1), we formulate the consistency of implementation as:

$$\forall\, i_1, \ldots, i_m \;\bullet\; \exists\, o_1, \ldots, o_n \bullet \mathit{FB\_IMPL}(i_1, \ldots, i_m, o_1, \ldots, o_n) \tag{6.4}$$

An inconsistent implementation (formalized as a predicate) is one which is evaluated to *FALSE* by any allowable values to its input and output variables. If the correctness is formulated as a logical implication, the inconsistent implementation implies any requirement specification. It is also considered as "false implies anything problem" in (Camilleri et al., 1987). In general, consider Equation 6.1, if the implementation predicate *FB_IMPL* is evaluated to *FALSE* by any possible values of the variables $i_1, \ldots, i_m$ and $o_1, \ldots, o_n$, then this implication trivially evaluates to *TRUE*, regardless of what *FB_REQ* is. The truth of inconsistency for the FB implementation makes the proof of its correctness invalid.

For example, a simple FBD consists of a negation (*NOT*) and an addition FB (*AND* or +) as shown in Figure 6.2. It takes as inputs i_1 (of type Boolean), i_1 (of type integer), and an output o (of type integer). Internal variable w connects these two FBs by feeding the output of *NOT* as input of *AND*. For simplicity, we omit time argument t for each variable.

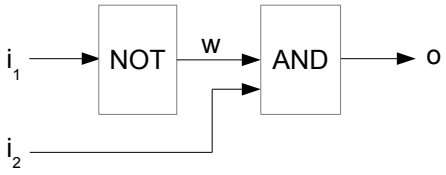


Figure 6.2: Example of an inconsistent FBD

The input-output relations for components are formalized as predicates *REQ_NOT* and *REQ_AND*. The FBD is formalized as *FBD_INCONSISTENCY*

in Equation 6.5:

$$\begin{aligned} & \forall\, i_1, i_2 \bullet \exists\, o \bullet \mathit{FBD\_INCONSISTENCY}(i_1, i_2, o) \\ \equiv\ & \forall\, i_1, i_2 \bullet \exists\, o, w \bullet \mathit{REQ\_NOT}(i_1, w) \ \wedge\ \mathit{REQ\_AND}(w, i_2, o) \\ \equiv\ & \forall\, i_1, i_2 \bullet \exists\, o, w \bullet w = \neg i_1 \wedge o = w + i_2 \\ \equiv\ & \forall\, i_1, i_2 \bullet \exists\, o \bullet o = \neg i_1 + i_2 \end{aligned} \tag{6.5}$$

It is impossible to find such output o that $o = \neg i_1 + i_2$ holds, since the input of *AND* has to be type of integer. Correspondingly, PVS will generate a proof obligation (i.e., *SUBTYPE-TCC*) for the type mismatch.

Practically, the consistency theorem can be proved trivially for simple FBs (i.e., one built from basic FBs, without feedback loops or timers). For example, if we replace *NOT* with *MOVE*, and change the type of i_1 and w from Boolean to integer. We then have the following FBD formalization in Equation 6.6. It is easy to find output o such that $o = i_1 + i_2$ holds. For complex FBs, it may not explicit enough to see the consistency. We consider the proof of consistency is necessary for both simple and complex FBs.

$$\begin{aligned} & \forall\, i_1, i_2 \bullet \exists\, o \bullet \mathit{FBD\_CONSISTENCY}(i_1, i_2, o) \\ \equiv\ & \forall\, i_1, i_2 \bullet \exists\, o, w \bullet \mathit{REQ\_MOVE}(i_1, w) \ \wedge\ \mathit{REQ\_AND}(w, i_2, o) \\ \equiv\ & \forall\, i_1, i_2 \bullet \exists\, o, w \bullet w = i_1 \wedge o = w + i_2 \\ \equiv\ & \forall\, i_1, i_2 \bullet \exists\, o \bullet o = i_1 + i_2 \end{aligned} \tag{6.6}$$

In the case of FB *LIMITS_ALARM*, we must prove the following:

$$\begin{aligned} & \forall\, H, X, L, EPS \bullet \exists\, QH, Q, QL \bullet \\ & \quad \mathit{LIMITS\_ALARM\_IMPL}(H, X, L, EPS, QH, Q, QL) \end{aligned} \tag{6.7}$$

The above consistency check is encoded in PVS 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)
```

Proving Consistency of Real-Time Function Block

As shown in Figure 6.1, we prove the same consistency theorem (Equation 6.4) to verify those FBs that build from IEC 61131-3 timers. We have formalized these timers using the operator *Timer_I*. The FB implementation is formalized as shown in Equation 5.6b (Section 5.1.2). We also follow the same proof patterns for the consistency theorem of real-time FBs. The patterns of consistency proofs are identified in Section 7.3.

6.3 Proving Equivalence between FBD and ST Implementations

In Chapter 5 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 $o_1, \ldots, o_n$. When both ST and FBD implementations are supplied for the same FB (i.e., *STACK_INT*), it may suffice to verify that each of the implementations is *correct* with respect to the requirement. However, as the behaviour of FB implementations is intended to be deterministic in most cases, it would be worth proving that the implementations (if they are given to implement the same algorithm) are equivalent, and generate scenarios (i.e., inconsistent behaviour), if any, where they are not. This is also labelled in Figure 1.1 on page 10 as proofs of *equivalence*.

Consequently, to verify that the two supplied implementations are equivalent, we must prove the following:

$$\begin{aligned} &\forall\, i_1, \ldots, i_m \bullet \forall\, o_1, \ldots, o_n \bullet \\ &\quad FB\_FBD\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \equiv \\ &\qquad\qquad FB\_ST\_IMPL(i_1, \ldots, i_m, o_1, \ldots, o_n) \end{aligned} \tag{6.8}$$

As an example, the *STACK_INT* block implements a last-in-first-out (LIFO) data structure for storing integers. As illustrated in Figure 6.3, it has five inputs (*PUSH*, *POP*, *R1*, *IN*, and *N*) and three outputs (*OUT*, *EMPTY* and *OFLO*).

It outputs:

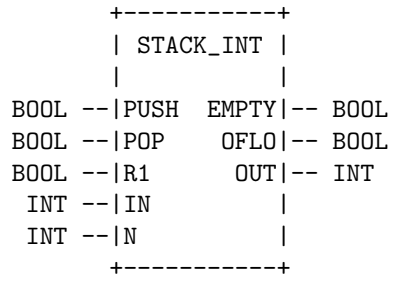
1. an integer value *OUT*, depending upon which operation was just performed;
2. a Boolean value *EMPTY* reporting if the current stack has become empty; and
3. a Boolean value *OFLO* indicating if the operation performed has caused a stack overflow.

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*).

In IEC 61131-3, there are three FBs provided both by ST and FBD implementations (i.e., *WEIGH*, *ALRM_INT* and *STACK_INT*). However, the verification of block *STACK_INT* is an exception in that equivalence check fails. The ST and FBD implementations are supplied at different levels of abstraction. The use of *EN*/*ENO* constrains a specific (sequential) order of executing internal blocks in the FBD implementation. There is no correspoding construct in the ST implementation. Consequently, we only attempt to prove that the implementation predicate of the FBD implementation implies that of the ST implementation in Equation 6.9. More details on block *STACK_INT* is discussed in Subsection 7.1.3.1.

$$\begin{aligned} &\forall\, PUSH, POP, R1, IN, N \bullet \forall\, OUT, EMPTY, OFLO \bullet \\ &\quad STACK\_INT\_FBD\_IMPL(PUSH, POP, R1, IN, N, \\ &\qquad\qquad OUT, EMPTY, OFLO) \\ &\quad \Rightarrow STACK\_INT\_ST\_IMPL(PUSH, POP, R1, IN, N, \\ &\qquad\qquad OUT, EMPTY, OFLO) \end{aligned} \tag{6.9}$$




```
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
```

Figure 6.3: Block *STACK_INT* declaration (IEC, 2003)

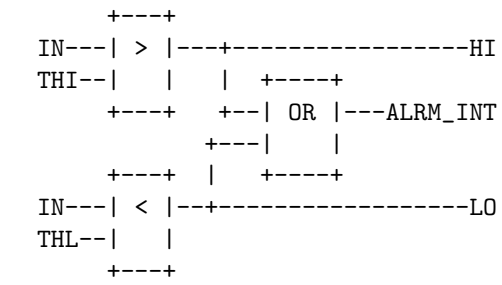
Another example function *ALRM_INT* (declared in Figure 6.4) is supplied both by ST (Figure 6.5a) and FBD (Figure 6.5b). It provides simple high and low level alarming for an input. It takes as input *IN*, high threshold *THI*, and low threshold *TLO*, and outputs *ALRM_INT*, high level alarm *HI*, and low level alarm *LO*. The output *ALRM_INT* is *TRUE* if either a high or low threshold is exceeded or both, and outputs *HI* and *LO* are provided for the high- or low-level alarm conditions.

However, we identify an issue that high and low alarms are allowed to occur simultaneously. Our solution will be discussed in detail later in Section 7.1.2.8.

We prove that the ST implementation is equivalent to the FBD imple-

```
      +-----------+           VAR_INPUT
      | ALRM_INT  |             IN : INT ;
      |           |             THI: INT ; (* High threshold *)
INT --|IN         |-- BOOL      TLO: INT ; (* Low threshold *)
INT --|THI      HI|-- BOOL    END_VAR
INT --|THL      LO|-- BOOL    VAR_OUTPUT
      |           |             HI: BOOL; (* High level alarm *)
      +-----------+             LO: BOOL; (* Low level alarm *)
                               END_VAR
```

Figure 6.4: Block *ALRM_INT* declaration (IEC, 2003)



(a) FBD implementation of function *ALRM_INT*

```
HI  := IN > THI ;
LO  := IN < THL ;
ALRM_INT := THI OR THL ;
```

(b) ST implementation of function *ALRM_INT*

Figure 6.5: FBD and ST implementations of block *ALRM_INT* (IEC, 2013)

mentation as in Equation 6.10:

$$\begin{aligned} &\forall\, IN, THI, TLO \bullet \forall\, HI, LO, OUT \bullet \\ &\quad ALRM\_INT\_FBD\_IMPL(IN, THI, TLO, HI, LO, OUT) \equiv \\ &\quad\quad ALRM\_INT\_ST\_IMPL(IN, THI, TLO, HI, LO, OUT) \end{aligned} \tag{6.10}$$

6.4 Summary

In this chapter we formally verified the correctness of a FB that is described in FBD and ST languages. Based on the formalization of the requirements

and implementation of FBs (Chapters 3, 4 and 5), we formulated three kinds of verification conditions.

We verified the correctness of a FB that is correct with respect to its input-output requirement. The correctness theorem is expressed by a logical implication such that any input and output values that satisfy an implementation predicate also satisfy their requirement predicate. We then ensure that the requirement predicate is satisfied – not only because the implementation predicate is itself inconsistent (i.e., the implementation predicate is evaluated to *FALSE* for any possible values of inputs and outputs). We thus formulated a consistency theorem, which ensures that, for any input values, there exists an output value such that the implementation predicate holds. Moreover, for those FBs supplied both by FBD and ST, we proved their equivalence. As a result, we only need to prove the consistency and correctness for either the ST implementation or the FBD implementation.

In the next chapter, we present case studies from the IEC 61131-3 FB library and a realistic real-time FB subsystem. We also summarize proof patterns for our verification conditions.

Chapter 7

Case Studies

In this chapter we apply our approach to two case studies, the standard IEC 61131-3 block library and a *Trip Sealed-In* subsystem from a safety critical industrial application, to illustrate the value of our approach [1]. In the first case study of the IEC 61131-3 block library, we classify a number of issues into three broad categories: ambiguous behaviour, missing input assumptions, and inconsistent implementations (Section 7.1). We provide our suggested solution for each issue. In the second case study of the *Trip Sealed-In* subsystem, we identify an initialization failure, and suggest a possible solution (Section 7.2). We then summarize the proof patterns that can be used as guidance to discharge the proof obligations of consistency and correctness (Section 7.3).

7.1 Example: IEC 61131-3 Block Library

To justify the value of our approach, we formalized and verified all of the standard functions and FBs supplied by IEC 61131-3 including Annex F (IEC, 2003). As a result, we uncovered blocks with ambiguous behaviour, missing input assumptions, and inconsistent implementations. We will discuss each issue in the following subsections.

[1] This chapter is based on the work in (Pang et al., 2014a) (under minor revision), and the published work in (Pang et al., 2015).

7.1.1 Ambiguous Bahaviour

7.1.1.1 Timer *Pulse* in Timing Diagrams

Block *PULSE* is a timer defined in IEC 61131-3, which has been introduced in Section 4.2.3. We already formalized the black-box behaviour of the *PULSE* timer using function tables (Figure 4.6). In this section we discuss two scenarios that are not specified in the standard, and suggest possible behaviours for these input trajectories.

Most of the critical behaviours have been captured by the timing diagrams in the Standard, but subtle or critical boundary cases are likely to be missing, since timing diagrams specify a limited number of use cases for the intended behaviour. We formalize the *PULSE* timer using function tables that ensure that all possible input sequences are covered (completeness) and are handled in an unambiguous fashion (disjointness). We found that there are at least two scenarios that are not covered by the timing diagram supplied by IEC 61131-3 (Figure 4.5).

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

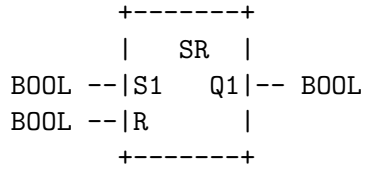
We use three function tables (Figure 4.2.3) to formalize the complete and disjoint behaviour of the *PULSE* timer. As a result, we make explicit assumptions to disambiguate the above two scenarios. Scenario 1 would match the condition row in the upper-left table for output Q, where Q at the previous 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 solution to Scenario 1 (that the rising edge of *IN* at $t_2 + PT$ is missed), Scenario 2 would match the condition row in the lower table for output ET,

where both Q and condition IN at the current time tick do not hold (i.e., $\neg Q \wedge \neg IN$), so the value of ET is reset back to 0.

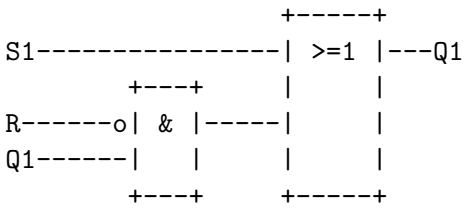
As $PULSE$ timer is not supplied with an implementation, there are no correctness and consistency proofs to be conducted. Nonetheless, obtaining a precise, complete, and disjoint requirements specification is invaluable for anyone creating a concrete implementation.

7.1.1.2 Implicit Delay Unit of the *SR* and *RS* Latches

The semantics of feedback loops is critical for defining the exact behaviour of FBs that use them, and should thus be made explicit and precise. In an effort to achieve this, we introduce mathematical rigour for the purpose of making implicit meanings both explicit and precise. Therefore, in our modelling framework of time, we formalize a delay unit z^{-1} (Section 5.1.1) to explicitly inform users that there will be delay of one unit of time before the newly-evaluated feedback value can be used as an input.



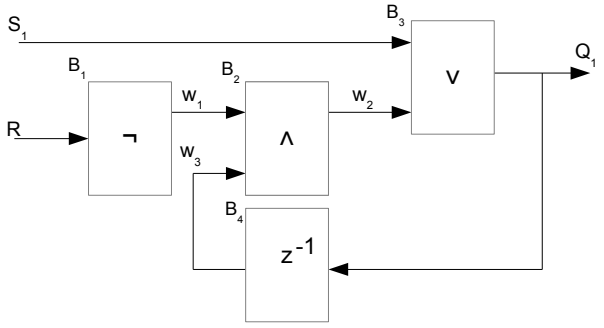
(a) Declaration of function block *SR*



(b) FBD implementation of function block *SR*

Figure 7.1: Block *SR* declaration and FBD implementation (IEC, 2003)

The block SR creates a set-dominant latch (a.k.a., flip-flop) in Figure 7.1. 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



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

Figure 7.2: Block *SR* implementation in FBD and its formalizing predicate

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} (as formalized in Figure 5.2) and conjoining predicates for the internal blocks. IEC 61131-3 uses a circle (e.g., the upper input to conjunction block in Figure 7.1b) to negate the value of Boolean input signal. We explicitly replace such circle with a negation block wherever it occurs.

Thus we revise the FBD implementation of block *SR* in Figure 7.2. Blocks B_1 (formalized by predicate *NEG*), B_2 (*CONJ*), B_3 (*DISJ*), and B_4 (z^{-1}) in Figure 7.2 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 of the FBD implementation in Figure 7.2. More precisely, the following theorems, as formulated in (7.1) and (7.2), are discharged in PVS, in which *SR_FBD_IMPL*, *SR_REQ* denote the FBD implementation and the tabular requirement of block *SR*.

$$\forall S_1, R \bullet \forall Q_1 \bullet SR\_FBD\_IMPL\,(S_1, R, Q_1) \Rightarrow SR\_REQ\,(S_1, R, Q_1) \quad (7.1)$$

$$\forall S_1, R \bullet \exists Q_1 \bullet SR\_FBD\_IMPL\,(S_1, R, Q_1) \tag{7.2}$$

The requirement of *SR* is formalized using a function table (Figure 7.3).

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

Figure 7.3: Requirement of the *SR* block using tabular expression

Symmetrically, we explicitly add a unit delay into the FBD implementation of the reset-dominant latch *RS*. As a result, we prove its consistency and correctness theorems as well.

7.1.2 Missing Input Assumptions

7.1.2.1 Limit on the Counters Blocks *CTU*, *CTD*, and *CTUD*

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 Figure 7.4.

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.

```
                        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 *)
       |  CUTD  |         CV : INT (* Current counted value *)
BOOL -->CU    QU|-- BOOL  END_VAR
BOOL -->CD    QD|-- BOOL
BOOL --|R       |          IF R THEN CV := 0 ;
BOOL --|LD      |          ELSIF LD THEN CV := PV ;
 INT --|PV    CV|-- INT    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
```

Figure 7.4: Block *CTUD* declaration and ST implementation (IEC, 2003)

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 input 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[2]: $PVmin < PVmax$.

2. The range of the preset value PV, with respect to the limits $PVmin$ and

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

$PVmax$, 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: $PVmin < PV < PVmax$.

Condition				*Result* CV
R				0
¬R	LD			PV
	¬LD	CU ∧ CD		NC
		CU∧¬CD	CV_{-1}< PVmax	CV_{-1}+1
			CV_{-1}≥ PVmax	NC
		¬CU∧CD	CV_{-1}> PVmin	CV_{-1}-1
			CV_{-1}≤ PVmin	NC
		¬CU ∧ ¬CD		NC

assume: $PVmin < PV < PVmax$

Figure 7.5: The requirement of block $CTUD$ using tabular expression

Our tabular requirement for the up-down counter that incorporates the missing assumption is shown in Figure 7.5. Similarly, we added $PV < PVmax$ and $PVmin < PV$ as assumptions for, respectively, the up and down counters.

7.1.2.2 Deadband Size of the *HYSTERESIS* Block

As one running example (Section 2.5.1), block $HYSTERESIS$ implements a Boolean hysteresis. The input-output requirement is formalized using function table (Figure 3.1). In Section 3.3, we indicate a missing assumption on the deadband size (i.e., the deadband size should be positive). In this section, we will discuss the found issue in detail.

For the behaviour specified in Figure 3.1, 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 (Figure 7.6).

We differentiate the deadband size defined in the Standard with respect to *EPS_NO* and the deadband size with our suggested assumption with respect to *EPS* in PVS.

```
% Disjointness TCC generated (at line 63, column 7) for
% TABLE
% ------------------------------------------+-------++
% | XIN1(t) < (XIN2(t) - EPS_NO(t))         | FALSE ||
% ------------------------------------------+-------++
% |(XIN2(t) - EPS_NO(t)) <= XIN1(t) AND
%   XIN1(t) <= (XIN2(t) + EPS_NO(t))        |PREV   ||
% ------------------------------------------+-------++
% | (XIN2(t) + EPS_NO(t)) < XIN1(t)         | TRUE  ||
% ------------------------------------------+-------++
% ENDTABLE
% unfinished

HYSTERESIS_REQ_WITHOUT_ASSUMP_TCC1: OBLIGATION
 FORALL (XIN1, XIN2, EPS_NO: [tick[delta_t] → real],
         Q: pred[tick[delta_t]], t):
    NOT init(t) IMPLIES
     (FORALL (PREV: bool):
        PREV = Q(pre(t)) IMPLIES
         NOT (XIN1(t) < (XIN2(t) - EPS_NO(t)) AND
               ((XIN2(t) - EPS_NO(t)) < XIN1(t)) &
               (XIN1(t) <= (XIN2(t) + EPS_NO(t)))) AND
          NOT (XIN1(t) < (XIN2(t) - EPS_NO(t)) AND
                (XIN2(t) + EPS_NO(t)) < XIN1(t)) AND
           NOT ((((XIN2(t) - EPS_NO(t)) <= XIN1(t)) &
                  (XIN1(t) <= (XIN2(t) + EPS_NO(t))))
                 AND (XIN2(t) + EPS_NO(t)) < XIN1(t)))
```

Figure 7.6: Unprovable disjointness TCC for the *HYSTERESIS* block

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*. For example, the deadband size of a trip hysteresis is 50 mV in nuclear domain. Therefore, in our PVS models, when proving the correctness of *HYSTERESIS* and blocks

that use it (e.g., the *LIMITS_ALARM* block discussed in Subsection 7.1.2.3), we adopt a stronger assumption $EPS > 0$ than that for Figure 3.1.

```
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
```

Figure 7.7: ST Implementation for the *HYSTERESIS* Block (IEC, 2003)

We will relax such assumption later in this section (in Figure 7.9), by considering the behaviour of the *HYSTERESIS* block with a negative dead-band size. For the purpose of verification, we translate the ST implementation (as shown in Figure 7.7) into a PVS predicate that has a tabular structure in Figure 7.8. 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 positive).

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

Figure 7.8: ST implementation of block *HYSTERESIS* in tabular expressions: with no assumption on *EPS*

However, the behaviour of the ST code (Figure 7.8) does not conform to that in Figure 3.1. 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* Let's consider a concrete example. Say $EPS = -2$, $XIN1 = 1$, and $XIN2 = 2$, then by executing the ST code (Figures 7.7 and 7.8) multiple times, we obtain alternating (or toggling) results (of *FALSE* and *TRUE*) for Q.

Nonetheless, the toggling behaviour may or may not be what users of IEC 61131-3 expect. In case it is, we provide an extended tabular requirement that incorporates the case of negative *EPS* (Figure 7.9), where the two rows under the condition that $EPS < 0 \wedge (XIN2 + EPS) < XIN1 < (XIN2 - EPS)$ represent the toggling behaviour.

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

Figure 7.9: Requirement of block *HYSTERESIS*: no assumption on *EPS*

7.1.2.3 High/Low Limits of the *LIMITS_ALARM* Block

As another running example (Section 2.5.2), the *LIMITS_ALARM* block implements a high/low limit alarm with hysteresis on both outputs. The input-output requirement is formalized using function tables (Figure 3.3). We formalize the supplied FBD implementation as a conjunction of those predicates that formalize internal components (Equation 5.1). Its verification conditions are in Equations 6.3 and 6.7. In this section we discuss how our formalization process reveals the need for two missing input assumptions of this block.

1. Similar to the case of the *HYSTERESIS* block (Subsection 7.1.2.2), 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. Otherwise, an unprovable disjointedness TCC will be generated.

We solve it by defining a function type *timed_posreal*, i.e., a function from tick to positive real. Variable *EPS* is declared to be of this type. The other two inputs X and L are declared of type *timed_real*, i.e., a function from tick to real.

```
timed_posreal: TYPE = [tick -> posreal]
EPS: VAR timed_posreal
```

2. We impose another assumption $H - EPS > L + EPS$, or equivalently $H - L > 2EPS$, to separate two hysteresis zones. 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.

   We solve this by introducing a dependent type to impose the constraint on the relationship between high limit and low limit. Input H is declared to be of this type.

```
dependent_high_limit_type: TYPE =
 [L: timed_real, EPS: timed_posreal ->
  {H: timed_real | FORALL (t: tick): H(t) - L(t) > 2*EPS(t)}]
H: VAR dependent_high_limit_type
```

During the proof of overall correctness, we introduce three lemmas, each corresponding to the correctness of an output variable. *P_QH*, *P_QL*, and *P_Q* are predicates specifying the intended behaviour of outputs *QH*, *QL*, and *Q*. 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*, 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*.

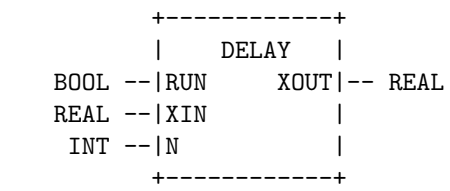
```
PROPERTY0: THEOREM
  LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL)
     => FORALL (t: tick): NOT (QH(t) AND QL(t))
```

Incorporating these two assumptions with our tabular requirement, we proved that the ST implementation supplied by IEC 61131-3 is both correct and consistent.

7.1.2.4 Initialization Failure of the *DELAY* Block

The *DELAY* block (Figure 7.10a) 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 Figure 7.11. The upper table in Figure 7.11 specifies *last_disabled*, the latest moment in time when the input flag *RUN* is set to *FALSE*. The lower table in Figure 7.11 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 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.



(a) Declaration of function block *DELAY*

```
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]          (* N-Element queue  *)
          OF REAL;                (* 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
```

(b) ST implementation of function block *DELAY*

Figure 7.10: Block *DELAY* declaration and ST implementation (IEC, 2003)

The ST implementation of the *DELAY* block (Figure 7.10b) 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[3]. Otherwise, values of *IXIN*, *IXOUT*, and their in-between cells are reset.

Inspecting its ST implementation, the intended usage of the *DELAY*

[3]This circular operation is implemented by a division modulo operator *mod*.

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 sample 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 (IEC, 2003, p187): "This function function block implements an N-sample delay", meaning that the value of output should equal to 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 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 Figure 7.10, both *IXIN* and *IXOUT* are initialized to 0. This means that initially they point to the 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* (Subsection 7.1.2.6) 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, ..., 128}.

Finally, based on the above reasoning, we formalize the complete tabular requirement for the *DELAY* block (Figure 7.11).

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

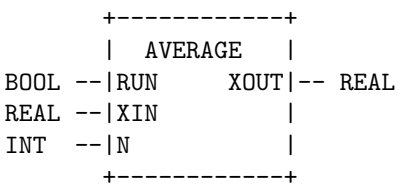
Condition		*Result* XOUT
¬RUN		XIN
RUN	Held_For(RUN, N·δ)	XIN_{-N}
	¬Held_For(RUN, N·δ)	$XIN_{-rank(t-last\_disabled)}$

Figure 7.11: Requirement of block *DELAY*

7.1.2.5 Division by Zero of the *AVERAGE* Block

The *AVERAGE* block (whose declaration is shown in Figure 7.12a) computes a running average *XOUT* over the last N values of the input sample *XIN*. The ST implementation of *AVERAGE* (shown in Figure 7.12b) indicates that it is a composite FB. It references an instance of the *DELAY* block (Subsection 7.1.2.4), 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*.



(a) Declaration of function block *AVERAGE*

```
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 ; (* Averaged output *)
  END_VAR
  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
```

(b) ST implementation of function block *AVERAGE*

Figure 7.12: Block *AVERAGE* declaration and ST implementation (IEC, 2003)

Based on our understanding of the ST implementation, we formulate the requirement of the *AVERAGE* block in Figure 7.13. Similar to the case of the the *DELAY* block (Subsection 7.1.2.4), 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 4.2.3) timing operator. There are four cases to consider:

1. If *RUN* is *FALSE*, then *XOUT* is set to the current value of *XIN*.

Condition	*Result* last_disabled
RUN	NC
¬RUN	t

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

#new_vals = *rank*(*t* - *last_disabled*) - 1

#old_vals = *N* - *#new_vals* - 1

Figure 7.13: Requirement of block *AVERAGE*

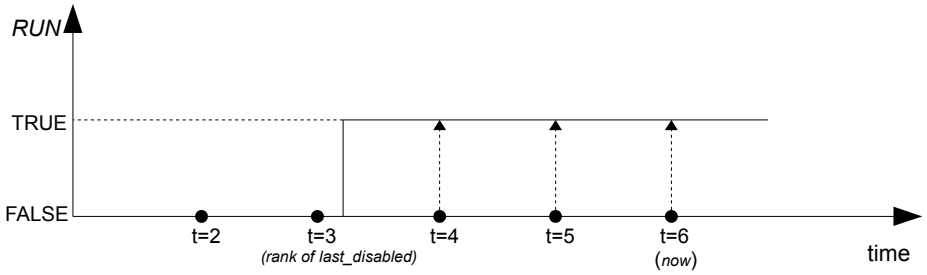
2. If *RUN* remains *TRUE* for a time period of at least *N* ticks, then *XOUT* is set to the average of the most recent *N* values of input sample *XIN*.

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., $XIN_{-rank(t-last\_disabled)}$[4]). The obvious constraint 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 $t = 4$:

[4]Here $rank(t - last\_disabled)$ denotes the number of ticks occurring between *last_disabled* and now (i.e., the current time tick).



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, ..., 128}.

7.1.2.6 Division by Zero of the *PID* Block

A *PID* (proportional-integral-derivative) controller (see Figure 7.14) is a widely used control loop feedback mechanism. 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. It calculates an error value as the difference between a measured process variable (PV) and a desired setpoint (SP) to minimize the error by adjusting the process. The output of the *PID* controller is calculated by summing up the proportional, integral, and derivative terms. The *PID* output is as follows:

$$output(t) = K_p e(t) + K_i \int_0^t e(\tau)\,\mathrm{d}\tau + K_d \frac{\mathrm{d}}{\mathrm{dt}} e(t),$$

where K_p is proportional gain, K_i is integral gain, K_d is derivative gain, e is error, t is time, and τ is variable of integration.

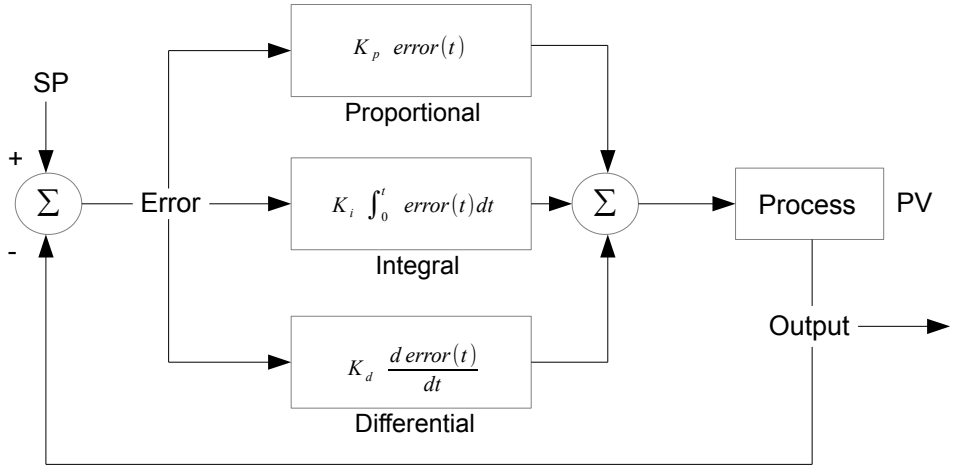
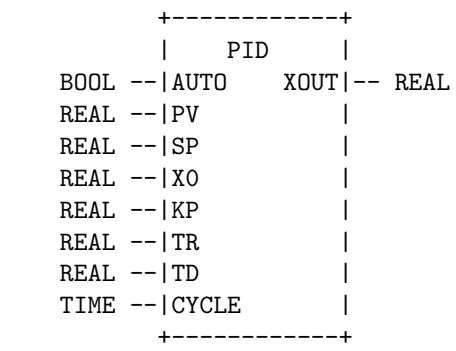


Figure 7.14: A block diagram of a *PID* controller

The *PID* block (declared in Figure 7.15a) implements this classical three-term controller for closed-loop feedback control. 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



(a) Declaration of function block *PID*

```
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
```

(b) ST implementation of function block *PID*

Figure 7.15: Block *PID* declaration and ST implementation (IEC, 2003)

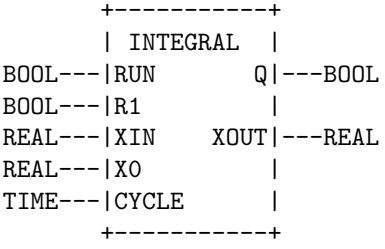
past); and (3) the derivative term (predicting error in the future). The computation also depends on other inputs constants: *KP* (proportionality constant), *TR* (reset time), *TD* (derivative time), and *CYCLE* (sampling period). At

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 in Figure 7.15b):

$$XOUT = KP \times (\ (PV - SP) + \frac{ITERM.XOUT}{TR} + DTERM.XOUT \times TD\ ) \quad [5]$$

where *ITERM* and *DTERM* are instances of, respectively, the *INTEGRAL* (Figure 7.16 and Figure 7.17) block and the *DERIVATIVE* (Figure 7.18 and Figure 7.19) 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 (Figure 7.16 and Figure 7.17) 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*.




```
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
```

Figure 7.16: Declarations of block *INTEGRAL* (IEC, 2003)

The *DERIVATIVE* block (Figure 7.18 and Figure 7.19) computes the differentiation of values of input *XIN* with respect to time. The rate of change

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

	Result
Condition	Q
R1	0
¬R1	1

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

Figure 7.17: Requirement of block *INTEGRAL*

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.

```
       +------------+
       | DERIVATIVE |
BOOL---|RUN         |
REAL---|XIN     XOUT|---REAL
TIME---|CYCLE       |
       +------------+
```

```
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
```

Figure 7.18: Declarations of block *DERIVATIVE* (IEC, 2003)

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

Figure 7.19: Requirement of block *DERIVATIVE*

As indicated from the ST implementation of *PID* (Figure 7.15b) and tabular requirements of *INTEGRAL* and *DERIVATIVE* (Figure 7.16 to 7.19), an input Boolean flag *AUTO* is set to distinguish cases in the computation. If *AUTO* is set *TRUE*, the controller attempts 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 (Subsection 7.1.2.5), 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.

7.1.2.7 Incorrect Length Setting of the *DIFFEQ* Block

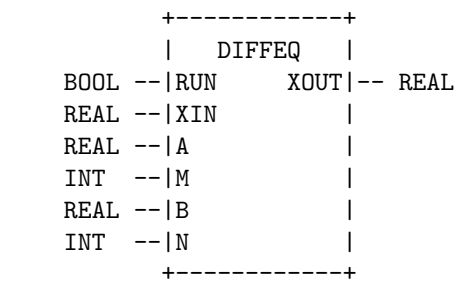
The *DIFFEQ* block (whose declaration is shown in Figure 7.20a[6]) implements the difference equation, an invariant on the present and past input and output values. The output *XOUT* represents the weighted sum of values drawn from three categories: (1) the current value of input *XIN*; (2) the previous N values of *XIN*; and (3) the previous M values of *XOUT*. More precisely:

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

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 Figure 7.20. 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 Figure 7.20). 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,

[6]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.



(a) Declaration of function block *PID*

```
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
```

(b) ST implementation of function block *DIFFEQ*

Figure 7.20: Block *DIFFEQ* declaration and ST implementation (IEC, 2003)

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.

	Result
Condition	XOUT
RUN	$\mathrm{B}_0 \cdot \mathrm{XIN} + \sum_{i=1}^{N} \mathrm{B}_i \cdot \mathrm{XIN}_{-i} + \sum_{j=1}^{M} A_j \cdot \mathrm{XOUT}_{-j}$
¬RUN	$\mathrm{B}_0 \cdot \mathrm{XIN}$

Figure 7.21: Tabular requirements of the *DIFFEQ* block

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]
```

7.1.2.8 High/Low Limits on the *ALRM_INT* Function

The function *ALRM_INT* (declared in Figure 6.4) is introduced as an example in Section 6.3. We prove the equivalence between the supplied ST and FBD implementations for function *ALRM_INT*.

Similar to one issue of block *LIMITS_ALARM* (Subsection 7.1.2.3), we identify a missing input assumption. The high and low alarm is allowed to be tripped simultaneously that contradicts the intended behaviour of both alarming levels. We resolve this by incorporating an assumption $THI > TLO$.

As a result, we are able to prove the following invariant property, that high and low alarm can not be tripped at the same time:

```
HI_LO: LEMMA
 THI(t) > TLO(t) =>
   (f_HI(INP,THI)(t) AND NOT f_LO(INP,TLO)(t)) OR
   (NOT f_HI(INP,THI)(t) AND f_LO(INP,TLO)(t)) OR
   (NOT f_HI(INP,THI)(t) AND NOT f_LO(INP,TLO)(t))
```

7.1.3 Inconsistent Implementations

7.1.3.1 Missing Internal Component of the *STACK_INT* Block

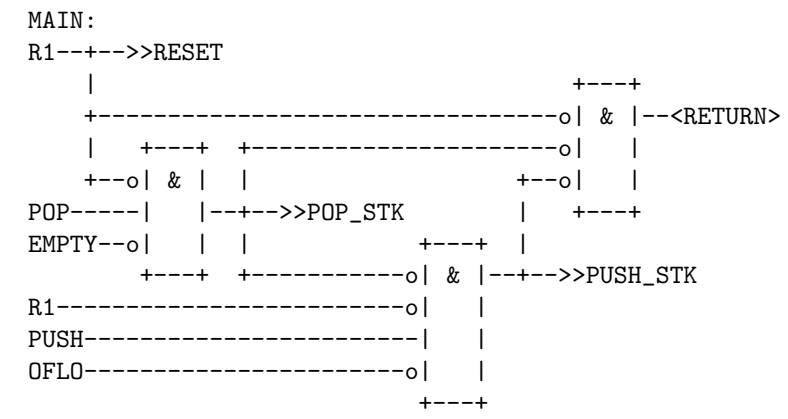
The *STACK_INT* block implements a stack of up to 128 integers (as introduced in Section 6.3). IEC 61131-3 supplies both ST and FBD implementations for the *STACK_INT* block. In this section we discuss the issue of a missing internal component of the FBD implementation, and suggest our solution.

Figure 7.22a lists the complete ST implementation, and Figure 7.22b shows the `MAIN` part of the FBD implementation. 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 Figure 7.23. We formulate this "jumps-to" mechanism in the conceivable 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 Figure 7.23), 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.

```
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 ;
```

(a) ST implementation of block *STACK_INT*



(b) The FBD implementation of block *STACK_INT*: *MAIN* part

Figure 7.22: ST and FBD implementations of block *STACK_INT* (IEC, 2003)

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.

   The ST and FBD implementations are actually not specified at the same level of abstraction. The use of *EN*/*ENO* constrains a specific

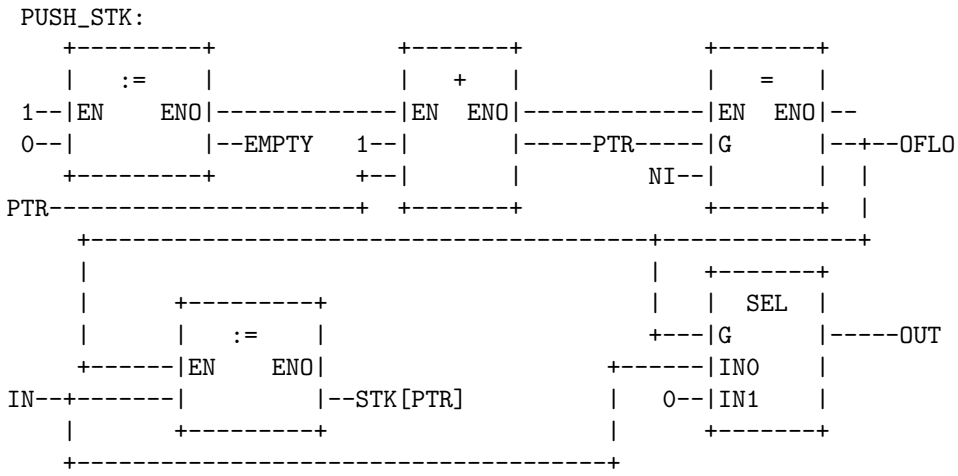


Figure 7.23: The FBD implementation of the *STACK_INT* block (IEC, 2003): *PUSH_STK* part

(sequential) order of executing internal blocks in the FBD implementation. However, there is no such constraint[7] 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 Equation 6.9.

2. A missing FB in the FBD implementation.

   However, we failed to prove Equation 6.9. In other words, we found inconsistency between these two implementations. By looking into the proofs, the following unprovable sequent was generated in PVS[8].

   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 ending in "*!n*" ($n = 1,2,\ldots$) are skolem constants (i.e., arbitrary constants of the corresponding types) that are used to

[7]In fact, there is no mechanism to translate one programming language to another that is semantically equivalent in IEC 61131-3. Furthermore, it may be impossible to translate one to another, since the lack of corresponding elements. For example, a pair of *EN* and *ENO* is used in FBD, but it cannot be used in ST.

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

eliminate quantifiers. The *COND* construct is a multi-way extension to the polymorphic "if-then-else" construct in PVS. *t!2*, *PTR!1*, *INP!1*, and *PUSH!1* are all arbitrary (yet type-correct) constants. At the $t!2^{th}$ tick (Section 2.3) of time, an input request *PUSH!1* is made to push an integer *INP!1* onto a stack *STK!1*, and the push operation moves the internal stack pointer to a new position *PTR!1*.

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

In the above sequent, the antecedent is inferred from the behaviour of the FBD implementation (Figure 7.23), and the consequence from that of the ST implementation (Figure 7.22a). 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* the and the lower *MOVE* block (Figure 7.23). 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.

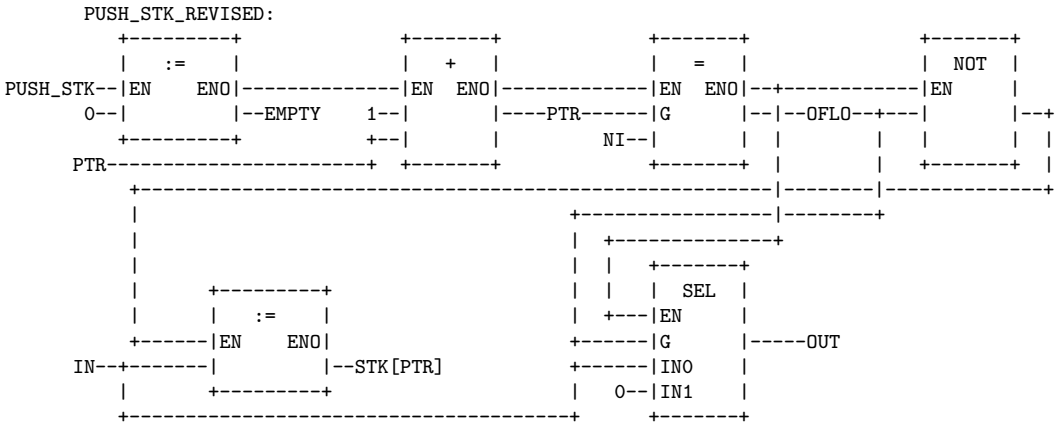


Figure 7.24: *PUSH_STK* part of the FBD implementation of block *STACK_INT*: a negation block added

With the revised FBD implementation for `PUSH_STK` (Figure 7.24 with modifications), we are able to prove Equation 6.9. We also proved that both the ST and FBD implementations are consistent (Section 6.2). 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:

$$\begin{aligned}&\forall\, PUSH, POP, R1, IN, N \bullet \forall\, OUT, EMPTY, OFLO \bullet \\ &\quad STACK\_INT\_ST\_IMPL\,(PUSH,POP,R1,IN,N,OUT,EMPTY,OFLO) \\ &\quad\quad \Rightarrow \\ &\quad\quad\quad STACK\_INT\_REQ\,(PUSH,POP,R1,IN,N,OUT,EMPTY,OFLO)\end{aligned} \tag{7.3}$$

Finally, we provide the complete requirement of the *STACK_INT* block in tabular expressions in the Appendix. A table is created for each output variable: *EMPTY* (Figure E.4), *OFLO* (Figure E.5), and *OUT* (Figure E.6). In fact, we found that the state of internal variables is necessary for us to define the behaviour of the stack: *NI* (Figure E.1), *PTR* (Figure E.2), and *STK*(*PTR*) (Figure E.3).

7.2 Example: the *Trip Sealed-In* Subsystem

In this section we include a complete verification work for a real-time FB subsystem that is generalized from the nuclear industry. We apply the same approach to verify the consistency and correctness of the *Trip Sealed-In* subsystem. During the verification process, we identified an initialization error and suggest our solution.

7.2.1 Overview: Informal Description

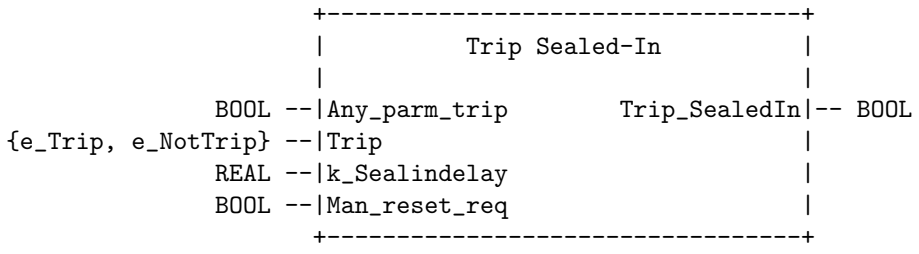


Figure 7.25: Input-output declaration of *Trip Sealed-In* subsystem

The *Trip Sealed-In* subsystem is a generic subsystem which monitors: (1) a set of sensor values; and (2) an alarm value produced by some other subsystem. It signals an alarm (denoted by the output *Trip_SealedIn*), which may be manipulated by other subsystems, when two conditions are met. First, any of the monitored sensor values goes out of its safety range (called a parameter trip and denoted by an input condition *Any_parm_trip*). Second, the monitored input alarm is signalled continuously for longer than some preset constant *k_Seqlindelay*[9] amount of time (denoted by an input value *Trip* of enumerated type {*e_Trip*,*e_NotTrip*}). Once the alarm *Trip_SealedIn* is activated, it is not deactivated until all monitored sensor values fall back within their safety ranges, and when a manual reset is requested (denoted as an input *Man_reset_req*).

[9]The *k_* name prefix is reserved for system-wide constants.

7.2.2 Tabular Requirements with Timing Tolerances

We use a function table (Figure 7.26) to perform a complete and disjoint analysis on the input domains. To incorporate timing tolerances into the requirements of *Trip Sealed-In*, we use the non-deterministic *Held_For* operator (Chapter 2) to specify a sustained window of a time duration $[k\_Sealindelay - \delta L, k\_Sealindelay + \delta R]$.

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

Figure 7.26: The *Trip Sealed-In* subsystem: (non-deterministic) requirements of with tolerances

However, for the purpose of verification in PVS, we reformulate the non-deterministic behaviour of Figure 7.26 in a recursive function[10] using the deterministic *Held_For_I* operator to impose the constraint that only a single value (i.e., *k_Sealindelay− delta_L* where both are declared constants) is chosen from the duration and is used consistently for detecting sustained events.

```
Channel_trip_sealedin_REQ_f
        (Any_parameter_tripped: pred[tick],
         c_ChanTrip : timed_trip,
         Manual_reset_request: pred[tick])(t: tick): RECURSIVE bool =

 IF init(t) THEN TRUE
 ELSE LET
    ChanTrip_status = LAMBDA (t: tick): c_ChanTrip(t) = e_Trip,
    PREV = Channel_trip_sealedin_REQ_f
                        (Any_parameter_tripped,
                        c_ChanTrip,
                        Manual_reset_request)(pre(t)) IN
```

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

```
TABLE
%-------------------------------------------------+--------%
|Any_parameter_tripped(t) &
 Held_For_I(ChanTrip_status,
          k_Sealindelay_req - delta_L,
             Sample_t)(t)                          |  TRUE ||
%-------------------------------------------------+--------%
|Any_parameter_tripped(t) &
 NOT (Held_For_I(ChanTrip_status,
                  k_Sealindelay_req - delta_L,
                  Sample_t)(t))                    | PREV ||
%-------------------------------------------------+--------%
|NOT Any_parameter_tripped(t) & Manual_reset_request(t)   | FALSE ||
%-------------------------------------------------+--------%
|NOT Any_parameter_tripped(t) & NOT Manual_reset_request(t)| PREV||
%-------------------------------------------------+--------%
ENDTABLE
ENDIF
MEASURE rank(t)
```

Using the above recursive function *Channel_trip_sealedin_REQ_f*, over all ticks, we have a deterministic requirements (Figure 7.27) for the *Trip Sealed-In* subsystem.

Compared with Figure 7.26, the use of the operator *Held_For_I* resolves the non-determinism by fixing the level of timing tolerance (i.e., as long as the alarm input *Trip* has been activated for or longer than $k\_Sealindelay - \delta L$, the *Trip Sealed-In* subsystem is guaranteed to detect it and act accordingly).

7.2.3 Formalization on FBD Implementation

For a given FBD implementation (Figure 7.28), we prove that it should satisfy the intended behaviour (Figure 7.27).

We use the IEC 61131-3 *TON* timer (see Chapter 4 for its formalization incorporated with tolerances) to implement the use of the *Held_For_I* operator

```
Channel_trip_sealedin_REQ_P(Any_parameter_tripped: pred[tick],
                            c_ChanTrip: timed_trip,
                            Manual_reset_request: pred[tick],
                            Channel_trip_sealedin: pred[tick]): bool =

FORALL (t: tick):
 Channel_trip_sealedin(t) =
      Channel_trip_sealedin_REQ_f(Any_parameter_tripped,
                                  c_ChanTrip,
                                  Manual_reset_request)(t)
```

Figure 7.27: The *Trip Sealed-In* subsystem: (deterministic) requirements with tolerances in PVS

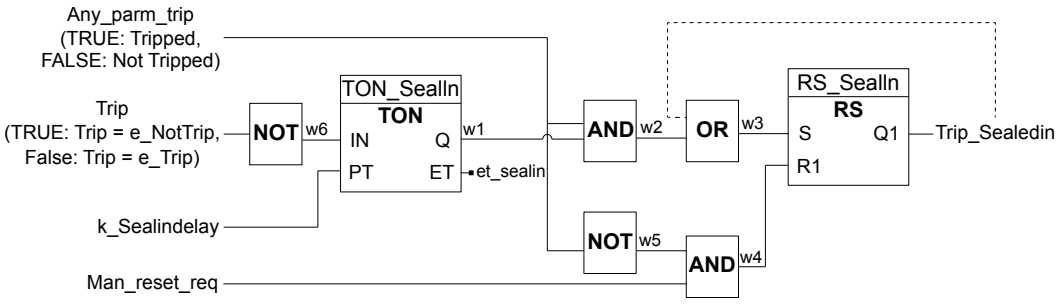


Figure 7.28: *Trip Sealed-In* implementation in FBD

(subject to a correctness proof below). As the recursive function used to define the requirements depends on the value of itself (at the previous time tick), a feedback loop (dashed line) is specified in the FBD implementation.

The use of the left-most *NOT* (negation) block in Figure 7.28 addresses the mismatch between type at the requirements level (i.e., {*e_Trip*,*e_NotTrip*}) and that at the FB implementation level (i.e., Boolean): somehow the engineers interpret value *e_Trip* as *FALSE* and *e_NotTrip* as *TRUE*, so a conversion is necessary to make sure the *Trip Sealed-In* has the consistent interpretation. The requirements that the alarm output *Trip_Sealedin* is deactivated (or reset) when there is no parameter trips, and when a manual reset is requested, is implemented using a standard block *RS* (reset-dominant flip flop).

To prove that the proposed FBD implementation of *Trip Sealed-In* (Figure 7.28) is both consistent and conforms to its requirements (Figure 7.27),

we first follow our approach to formalize it by composing, using conjunction, the formalizing predicates[11] of all component blocks (all inter-connectors are hidden using an existential quantification.):

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

7.2.4 Proofs of Consistency and Correctness

First, we prove that the FBD implementation (Figure 7.28) is feasible by instantiating Equation 6.4:

$$
\begin{aligned}
&\forall\ Any\_parm\_trip, Trip, Man\_reset\_req \bullet \exists\ Trip\_SealedIn \bullet \\
&\quad Trip\_sealedin\_IMPL(Any\_parm\_trip,\ AbstParmTrip\_timed(Trip), \\
&\qquad\qquad Man\_reset\_req,\ Trip\_SealedIn)
\end{aligned}
$$

The abstraction function *AbstParmTrip_timed* handled the mismatched types of input *Trip* at levels of requirement and implementation (i.e., *e_NotTrip* mapped to *TRUE* while *e_Trip* mapped to *FALSE*). We discharge the above consistency proof using proper instantiations.

Second, we prove that the FBD implementation is correct with respect to Figure 7.27, considering timing tolerances, by instantiating Equation 6.1:

$$
\begin{aligned}
&\forall\ Any\_parm\_trip,\ Trip,\ Man\_reset\_req,\ Trip\_SealedIn \bullet \\
&\quad Trip\_sealedin\_REQ(Any\_parm\_trip,\ Trip,\ Man\_reset\_req,\ Trip\_SealedIn) \\
&\qquad \Rightarrow Trip\_sealedin\_IMPL(Any\_parm\_trip,\ AbstParmTrip\_timed(Trip), \\
&\qquad\qquad Man\_reset\_req,\ Trip\_SealedIn)
\end{aligned}
$$

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

As there is a feedback loop in the FBD implementation (Figure 7.28), our strategy of discharging the correctness theorem is by mathematical induction (using the *time_induction* proposition in Section 2.4) over tick values. In both the base and inductive cases, we have to expand the definition of the *Timer_I* operator, because it is used to formalize the requirements of the *TON* timer that contributes to the FBD implementation.

7.2.5 Proof Discussion

In this section we present the proof patterns of consistency and correctness for the *Trip Sealed-In* subsystem. They exemplify the proof strategies later in Section 7.3.

For the consistency proof, we explicitly formulate the requirements of internal components in functional form to assist instantiation steps. The consistency proof can be discharged easily with proper instantiations (using these separately defined functions) to existentially quantified inter-connectors. The completion of consistency proof proves the feasibility of the design.

For the correctness theorem, we perform three important steps.

1. An inductive proving approach (i.e., *time_induction*) is applied. The mathematical induction allows us to apply induction over tick values. We prove that the requirement specification and the corresponding implementation specification are equivalent at all ticks within acceptable timing tolerances.

2. For the base step (i.e., initial case of *t=0*), expending the definition of the *SEL* block and applying basic PVS commands are sufficient. For the inductive step, an important general theorem *TimerGeneral_I* is reused and instantiated properly. A large amount of proving effort can be simplified.

3. For the inductive step, after instantiating theorem *TimerGeneral_I* we encounter a situation where the form of the λ-expression of inter-connector w_6 mismatches the form that appears as the first argument of *Held_For_I* operator. The occurrence of λ-expressions can be difficult to deal with

in PVS. We introduce an auxiliary lemma (i.e., *PROPERTY0*) to prove their logical equivalence. Lemma *PROPERTY0* can be proved by *extensionality* axiom for the required types (i.e., the types of two λ-expressions). The PVS specification of *PROPERTY0* is as follows:

```
PROPERTY0: LEMMA
FORALL c_Trip:
(LAMBDA (t: tick[delta_t]):
NOT COND (c_Trip(t) = e_Trip) -> FALSE,ELSE -> TRUE ENDCOND)=
    (LAMBDA (t: tick[delta_t]): c_Trip(t) = e_Trip)
```

However, by trying to prove the base case in Step 2 (when $t = 0$), we found that the initial value of output *Q1* of the *RS_Sealin* block and the initial value of the subsystem output *Trip_SealedIn* — these two values are directly connected in the initial FBD implementation (Figure 7.28) — are inconsistent. According to the SRS, value of *Trip_SealedIn* is initialized to *TRUE*, whereas that of *Q1* is *FALSE*. We resolve this issue of inconsistency by suggesting a revised FBD implementation (Figure 7.29) and prove that it is correct with respect to Figure 7.27.

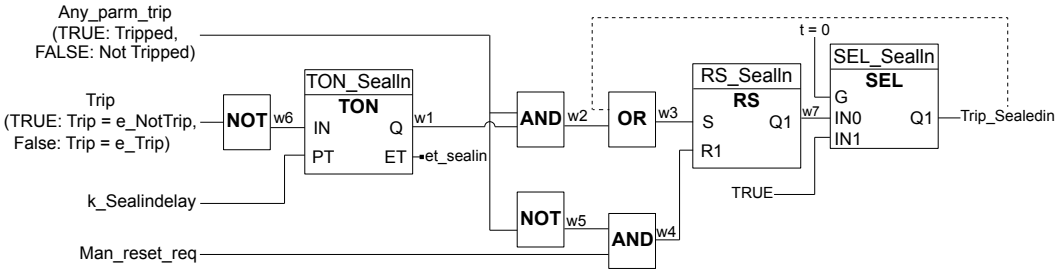


Figure 7.29: Revised *Trip Sealed-In* implementation in FBD

To revise the given FBD implementation, we add an IEC 61131-3 selection block (i.e., *SEL_Sealin*), acting as a multiplexer to discriminate the value of *Q1* (at the initial tick and at the non-initial tick) that is output as *Trip_SealedIn*. If input *G* is *TRUE* (i.e., initial case of *t=0*), output *Trip_SealedIn* is set to *TRUE*; otherwise, it is set to the value from input *INO* (i.e., from output *Q1* of block *RS_Sealin*). By imposing block

SEL_Sealin and applying the above three proof steps, the correctness theorem of the *Trip Sealed-In* subsystem can be completed.

7.3 Lessons learned: Proof Strategies

In this section we summarize the proof strategies applied in our verification approach. In general, there are two typical kinds of proofs in our work: those that require induction and those that do not. PLC programs often use feedback loops, i.e., outputs of a FB are connected as inputs of either another FB or the FB itself. Since the feedback values (or of intermediate output values) cannot be computed instantaneously in practice, the current feedback values depend on the values of themselves at earlier time ticks. In most of cases, only the depending value at a previous time tick is sufficient to generate the current output value. The proofs that involve such values usually require induction. For proofs by induction, we apply an induction scheme (i.e., *time_induction*) over time ticks. For proofs that do not require induction, we apply definition expansion, arithmetic, equality, and quantifier reasoning. We follow certain proof structures to discharge the proofs of consistency and correctness.

Some of our proofs that only depend on skolemization, propositional simplification, rewriting and linear arithmetic can be handled automatically, e.g., use of PVS proof command *grind* is sufficient. More complex proofs, on the other hand, require manual control. For the cases splits, the instantiation of universally quantified formulas and application of additional lemmas, manual control is necessary. To control the proofs as much as possible, we do not rely on powerful proving strategies (e.g., *induct-and-simplify* that combines of *induct* and repeated simplification) in key steps until the proof task can be handled automatically.

We first discuss the proof strategies for consistency theorem in Section 7.3.1. We then provide the proof strategies for correctness theorem in Section 7.3.2. They can be used to guide any verification work for PLC programs that are formalized based on our approach.

7.3.1 Proof Structure for Consistency Theorem

For a given FBD, in general, we perform the following key steps to prove the consistency theorem.

1. Applying *skolem*! to introduce Skolem constants on the universally quantified input variables in the consequent.
2. Using *inst*'s to instantiate existentially quantified output variables in the consequent with pre-defined functions. The instantiation for each output is the functional version of the requirement predicate for this output.
3. Using *inst*'s to instantiate existentially quantified inter-connector variables in the consequent with pre-defined functions. The internal variable connects an output of a component to an input (or more inputs) of other component(s). The instantiation for each internal variable is the functional version of the requirement predicate for the connecting output.
4. Applying *split* to split the conjunctive predicates of internal components.
5. Applying *expand*'s to expand the requirement predicate for each internal component, and then introducing Skolem constant to existentially quantified t in the consequent via *skolem*!.
6. Using a combination of basic commands to complete the proof.

For the steps 2 and 3, the functional versions of requirement predicates are formulated in the corresponding theory separately. They are defined recursively if any earlier time ticks of value is depended on. Otherwise, they are non-recursive functions. By importing the theory of a component, they can be used by calling the function names. Alternatively, one can explicitly instantiate universally quantified internal or output variable by involving such functional definition within a proof command. However, it is error-prone for complex functionality. It is also tedious if multiple instances of a component are used in the same (or different) FBD.

Consider the given FBD in Figure 7.30, which is constructed using logical conjunctions to compose the requirements predicates of the four internal

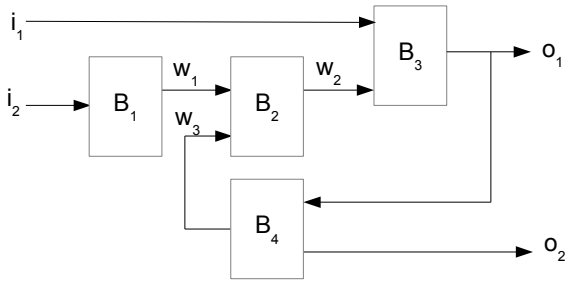


Figure 7.30: Example of a FBD implementation

FBs, B_1, B_2, B_3, and B_4. Their requirements predicates are formalized as B_1_REQ, ..., B_4_REQ, and their corresponding functional forms are f_1_REQ, ..., f_4_REQ. The variables $w_i, i = 1, 2, 3$ represent inter-connectors between internal blocks. They are hidden from the user using existential quantifiers. The consistency theorem can be proved when, for any values of input variables i_1 and i_2, there exists the values of output variables o_1 and o_2 such that the implementation predicate *FB_IMPL* is satisfied. The consistency theorem can be proved as follows:

$$
\begin{aligned}
& \forall\, i_1, i_2 \bullet \exists\, o_1, o_2 \bullet FB\_IMPL(i_1, i_2, o_1, o_2) \\
\equiv\ & < \textit{expand the definition of FB\_IMPL} > \\
& \forall\, i_1, i_2 \bullet \exists\, o_1, o_2 \bullet \\
& (\exists\, w_1, w_2, w_3 \bullet \\
& \quad B_1\_REQ(i_2, w_1) \wedge B_2\_REQw_1, w_3, w_2) \wedge \\
& \quad B_3\_REQ(i_1, w_2, o_1) \wedge B_4\_REQ(o_1, w_3, o_2) \\
\equiv\ & < \textit{expand the definitions of } B_i\_REQ \textit{ with } f_i\_REQ,\ i = 1, ..., 4 > \\
& \forall\, i_1, i_2 \bullet \exists\, o_1, o_2 \bullet \\
& \exists\, w_1, w_2, w_3 \bullet \\
& \quad w_1 = f_1\_REQ(i_2) \wedge w_2 = f_2\_REQ(w_1, w_3) \wedge \\
& \quad o_1 = f_3\_REQ(i_1, w_2) \wedge w_3 = f_4\_REQ\_w_3(o_1) \wedge o_2 = f_4\_REQ\_o_2(o_1)
\end{aligned}
$$

$\equiv$ $< by\ (\exists x \bullet\ x = term \wedge expr[x]) = expr[term/x]\ to\ eliminate\ w_1 >$

$\forall\ i_1, i_2\ \bullet \exists\ o_1, o_2 \bullet$

$\exists\ w_2, w_3 \bullet$

$w_2 = f_2\_REQ(f_1\_REQ(i_2), w_3) \wedge w_3 = f_4\_REQ\_w_3(o_1) \wedge$

$o_1 = f_3\_REQ(i_1, w_2) \wedge o_2 = f_4\_REQ\_o_2(o_1)$

$\equiv$ $< by\ (\exists x \bullet\ x = term \wedge expr[x]) = expr[term/x]\ to\ eliminate\ w_2 >$

$\exists\ i_1, i_2\ \bullet \forall\ o_1, o_2 \bullet$

$\exists\ w_3 \bullet$

$w_3 = f_4\_REQ\_w_3(o_1) \wedge$

$o_1 = f_3\_REQ(i_1, f_2\_REQ(f_1\_REQ(i_2), w_3)) \wedge$

$o_2 = f_4\_REQ\_o_2(o_1)$

$\equiv$ $< by\ (\exists x \bullet\ x = term \wedge expr[x]) = expr[term/x]\ to\ eliminate\ w_3 >$

$\forall\ i_1, i_2\ \bullet \exists\ o_1, o_2 \bullet$

$(o_1 = f_3\_REQ(i_1, f_2\_REQ(f_1\_REQ(i_2), f_4\_REQ\_w_3(o_1))) \wedge$

$o_2 = f_4\_REQ\_o_2(o_1))$

$\equiv$ $< by \forall - elimination,\ let\ i_1^{'}\ and\ i_2^{'}\ denote\ skolemization\ constants >$

$\exists\ o_1, o_2 \bullet$

$(o_1 = f_3\_REQ(i_1^{'}, f_2\_REQ(f_1\_REQ(i_2^{'}), f_4\_REQ\_w_3(o_1))) \wedge$

$o_2 = f_4\_REQ\_o_2(o_1))$

$\equiv$ $< since\ requirement\ is\ both\ disjoint\ and\ complete, the\ witness\ chosen\ for\ o_1$

$and\ o_2\ is\ recursive\ function\ f_2\_REQ\ and\ f_4\_REQ\_o\_2 >$

$TRUE$

To assist the proof of consistency theorem, we formulate function for each and internal variable and output variable explicitly. We have functions f_1_REQ, f_2_REQ, $f_4\_REQ\_w_3$, for w_1, w_2, and w_3, and functions f_3_REQ, $f_4\_REQ\_o_2$ for o_1 and o_2. The behaviour of an internal or output variable that depends on the value of itself at earlier time ticks (implemented by feedback loops), is defined using a recursive function. For example, output o_1 depends on the value of itself at a previous time tick. The corresponding func-

tion $f_3\_REQ\,(i_1, f_2\_REQ(f_1\_REQ(i_2), f_4\_REQ\_w_3(o_1)))$ specifying its behaviour is defined recursively.

7.3.1.1 Proof Strategy of Consistency for IEC 61131-3 Function Block *LIMITS_ALARM*

For the *LIMITS_ALARM* block (Subsection 7.1.2.3), where outputs QH and QL depend on the value of itself at a previous time tick, and output Q does not, we have recursive functions f_QH, f_QL, and non-recursive function f_Q specified for the instantiations of outputs QH, QL, and Q. We also have a functional form of component *DIV*, *SUB*, *ADD* for inter-connecters w_1, w_2, and w_3. These are used in the key steps 2 and 3 to discharge the consistency theorem for block *LIMITS_ALARM*.

In Figure 7.31, we summarize the proof structure of the consistency theorem for *LIMITS_ALARM* (i.e., theorem *LIMITS_ALARM_CONSISTENCY*). The key proof steps are as follows:

1. Applying *skolem*! to introduce skolem constants *X!1*, *H!1*, *L!1*, and *EPS!1* on the input variables *X*, *H*, *L*, and *EPS* in the consequent.

2. Using *inst*'s to instantiate the output variables *QH*, *QL*, and *Q* with *f_QH*, *f_QL*, and *f_Q* in the consequent.

3. Using *inst*'s to instantiate inter-connectors w_1, w_2, and w_3 with functional form of basic FB *DIV*, *SUB*, and *ADD* in the consequent. By instantiating w_1, a subtype TCC is generated to constrain it as positive reals. It is proved by applying *typepred* on *EPS* and *assert* to use decision logic.

4. Applying *split* to split the conjunctive predicates for components, two instances of *HYSTERESIS* (high and low alarms), *SUB*, *ADD*, *DISJ*, and *DIV*.

5. Applying *expand*'s to expand the requirement predicate for each component, and then using *skosimp* on t in the consequent.

6. Using a combination of basic commands[12] to complete the proof.

   (a) For two instances of *HYSTERESIS*, *expand*'s the functional definitions of its arguments and complete the rest by basic commands.

   (b) For *DISJ*, expand *f_QH* and complete the rest by basic commands.

   (c) For *SUB*, *ADD*, and *DIV*, it is proved by the propositional axiom rules.

7.3.1.2 Proof Strategy of Consistency for Subsystem *Trip Sealed-In*

For the *Trip Sealed-In* subsystem (Section 7.2), where the output *Trip_SealedIn* is fed back as input of component *OR*, we have (a revised version of) recursive function *Sealin_IMPL_f_Chan_Trip_Sealedin_revised* for output *Trip SealedIn*, and non-recursive functions for all inter-connectors.

In Figure 7.32, we summarize the proof structure of the consistency theorem for subsystem *Trip Sealed-In* (i.e., theorem *Channel_trip_sealedin_consistency*). The key proof steps are as follows:

1. Applying *skosimp* to introduce three skolem constants *Any_parameter_trip!1*, *c_ChanTrip!1*, and *Manual_reset_request!1* on the universally quantified input variables *Any_parameter_trip*, *c_ChanTrip*, and *Manual_reset_request*.

2. Using *inst* to instantiate the output variable *Trip_SealedIn* with the revised version of *Sealin_IMPL_f_Chan_Trip_Sealedin_revised* in the consequent. By instantiating *Trip_SealedIn*, two subtype TCCs are generated for time period *k_Sealindelay_impl*. We prove them by applying *typepred* on *delta_L* and *delta_R* and *assert* to use decision logic.

3. Using *inst*'s to instantiate inter-connecters w_1, ..., w_7 with functions f_w_1, ..., f_w_7 in the consequent. By instantiating w_1, w_2, w_3, and w_7, subtype TCCs are generated for time period *k_Sealindelay_impl*. We also prove them by applying *typepred* on *delta_L* and *delta_R* and *assert* to use decision logic.

[12] Those proofs that do not require induction can be proved by basic commands of definition expansion, and arithmetic equality, and quantifier reasoning.

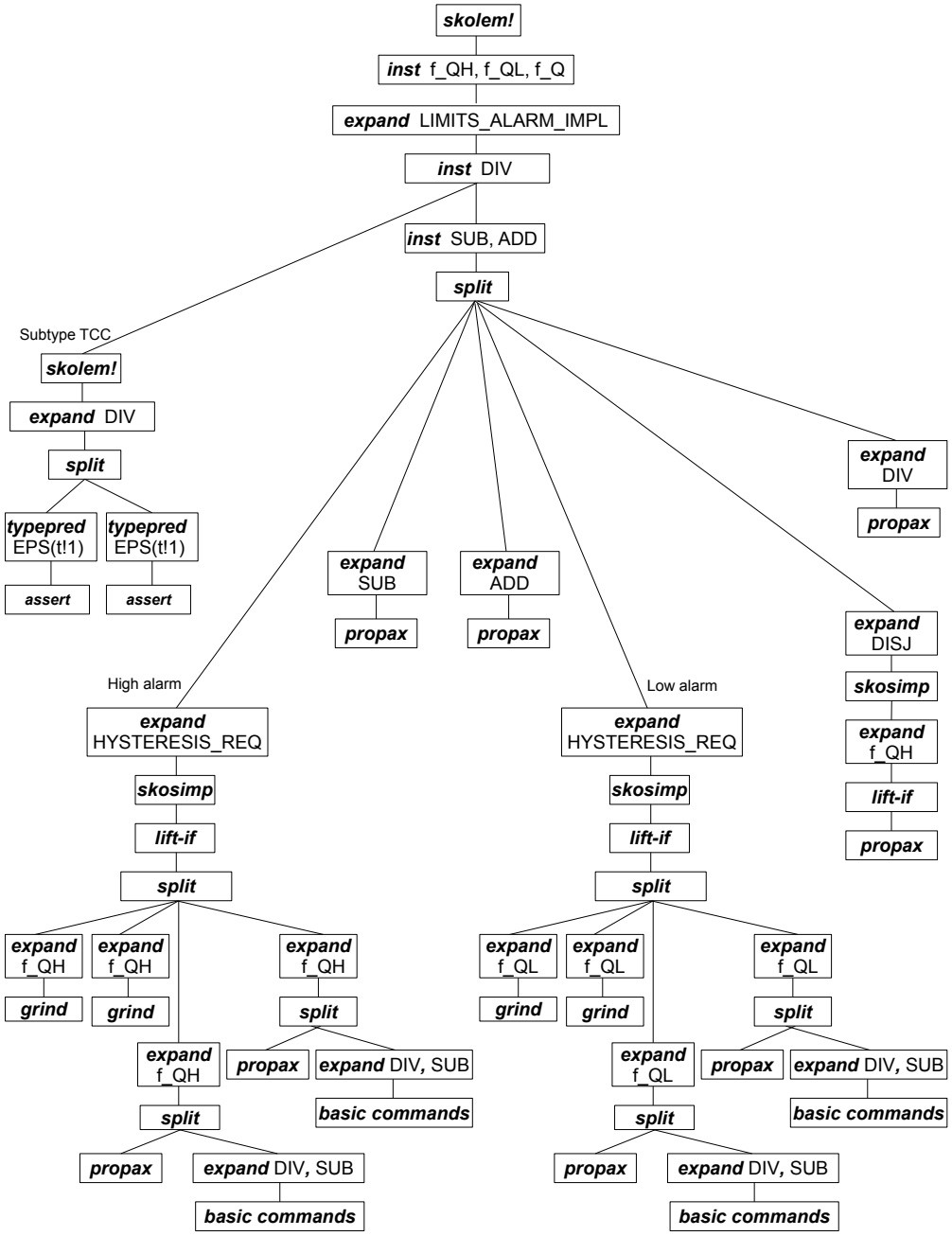


Figure 7.31: Consistency proof structure for block *LIMITS_ALARM*

4. Applying *split* to split the conjunctive predicates for components, two*NEG*'s, *TON*, two *CONJ*s, *DISJ*, *RS*, and *SEL*.

5. Applying *expand*'s to expand the requirement predicate for each component, and then using *skosimp* on t in the consequent.

6. Using a combination of basic commands to complete the proof.

   (a) For blocks *RS SEL*, *expand*'s the functional definitions of its arguments and complete the rest by basic commands.

   (b) For *NEG*'s, *TON*, *DISJ*, and *CONJ*'s, we prove them using the propositional axiom rules.

7.3.2 Proof Structure for Correctness Theorem

For a given FBD, in general we perform the following key steps to prove the correctness theorem.

1. Applying *skosimp*[13] to introduce Skolem constants on the universally quantified inputs and outputs in the consequent, and then disjunctively simplify the implication into implementation predicate in the antecedent and requirement predicate in the consequent.

2. Using *expand*'s to expand requirement and implementation predicates.

3. Using *skolem*! to introduce Skolem constant for each existentially quantified inter-connector in the antecedent.

4. Applying induction scheme *time_induction* on time tick t, generating two branches: base case and inductive step.

   (a) Base case: using *skolem*! to introduce Skolem constant *t!1* on t and *expand*'s on the predicate requirement of each component. Then using *inst*'s to instantiate Skolem constant *t!1* on each universally quantified formula of component. Complete the rest of the proof by decision logic.

[13]Proof command *skosimp* is a compound of *skolem*! and *flatten*.

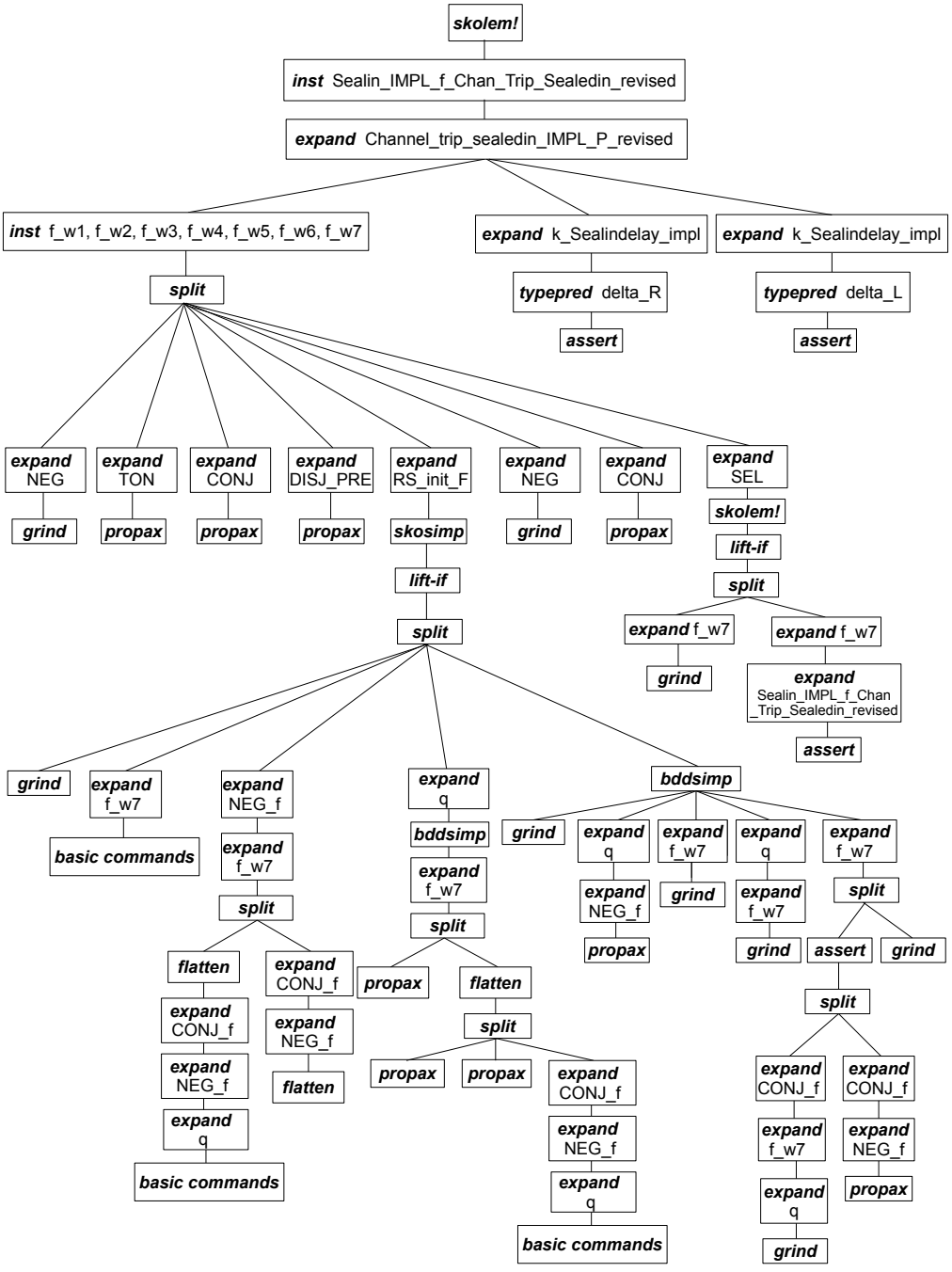


Figure 7.32: Consistency proof structure for the *Trip Sealed-In* subsystem

(b) Inductive step: the same strategies as for base case first, then *expand* the recursive function in the consequent. For real-time FBs, repeatedly using lemma *TimerGeneral_I* with proper instantiations to link the requirement and implementation domains for operator *Held_For*. Complete the rest of the proof by decision logic.

5. For those proofs that do not require induction, we can skip Step 4 by using a combination of basic commands to complete the proof.

Consider the FBD in Figure 5.1 again. The correctness theorem is proved for any values of input variables, i_1 and i_2, and output variables, o_1 and o_2, such that the implementation predicate *FB_IMPL* implies the requirement predicate *FB_REQ*. The requirements predicate *FB_REQ* is formalized using a function table for each output. The tables for output o_1 and o_2 are $f\_REQ\_o_1$ and $f\_REQ\_o_2$. They are defined as $f\_REQ\_o_1 = f_3\_REQ(i_1, f_2\_REQ(f_1\_REQ(i_2), f_4\_REQ\_w_3(o_1))$ and $f\_REQ\_o_2 = f_4\_REQ\_o_2(o_1)$. The correctness theorem can be proved as follows:

$$
\begin{aligned}
& \forall\, i_1, i_2 \bullet \forall\, o_1, o_2 \bullet FB\_IMPL(i_1, i_2, o_1, o_2) \Rightarrow FB\_REQ(i_1, i_2, o_1, o_2) \\
\equiv\ & < \textit{expand the definitions of FB\_IMPL and FB\_REQ} > \\
& \forall\, i_1, i_2 \bullet \forall\, o_1, o_2 \bullet \\
& (\exists\, w_1, w_2, w_3 \bullet \\
& \quad B_1\_REQ(i_2, w_1) \wedge B_2\_REQ(w_1, w_3, w_2) \wedge \\
& \quad B_3\_REQ(i_1, w_2, o_1) \wedge B_4\_REQ(o_1, w_3, o_2)) \Rightarrow \\
& (o_1 = f\_REQ\_o_1(i_1, i_2, o_1) \wedge o_2 = f\_REQ\_o_2(o_1)) \\
\equiv\ & < \textit{expand the definitions of } B_i\_REQ \textit{ with } f_i\_REQ,\ i = 1, ..., 4 > \\
& \forall\, i_1, i_2 \bullet \forall\, o_1, o_2 \bullet \\
& (\exists\, w_1, w_2, w_3 \bullet \\
& \quad w_1 = f_1\_REQ(i_2) \wedge w_2 = f_2\_REQ(w_1, w_3) \wedge \\
& \quad o_1 = f_3\_REQ(i_1, w_2) \wedge w_3 = f_4\_REQ\_w_3(o_1) \wedge o_2 = f_4\_REQ\_o_2(o_1)) \Rightarrow \\
& (o_1 = f\_REQ\_o_1(i_1, i_2, o_1) \wedge o_2 = f\_REQ\_o_2(o_1))
\end{aligned}
$$

$$
\begin{aligned}
\equiv \quad & < by\ (\exists x \bullet\ x = term \wedge expr[x]) = expr[term/x]\ to\ eliminate\ w_1 > \\
& \forall\, i_1, i_2\ \bullet \forall\, o_1, o_2 \bullet \\
& (\exists\, w_2, w_3 \bullet \\
& \quad w_2 = f_2\_REQ(f_1\_REQ(i_2), w_3) \wedge w_3 = f_4\_REQ\_w_3(o_1) \wedge \\
& \quad o_1 = f_3\_REQ(i_1, w_2) \wedge o_2 = f_4\_REQ\_o_2(o_1)) \Rightarrow \\
& (o_1 = f\_REQ\_o_1(i_1, i_2, o_1) \wedge o_2 = f\_REQ\_o_2(o_1)) \\
\equiv \quad & < by\ (\exists x \bullet\ x = term \wedge expr[x]) = expr[term/x]\ to\ eliminate\ w_2 > \\
& \forall\, i_1, i_2\ \bullet \forall\, o_1, o_2 \bullet \\
& (\exists\, w_3 \bullet \\
& \quad w_3 = f_4\_REQ\_w_3(o_1) \wedge \\
& \quad o_1 = f_3\_REQ(i_1, f_2\_REQ(f_1\_REQ(i_2), w_3)) \wedge \\
& \quad o_2 = f_4\_REQ\_o_2(o_1)) \Rightarrow \\
& (o_1 = f\_REQ\_o_1(i_1, i_2, o_1) \wedge o_2 = f\_REQ\_o_2(o_1)) \\
\equiv \quad & < by\ (\exists x \bullet\ x = term \wedge expr[x]) = expr[term/x]\ to\ eliminate\ w_3 > \\
& \forall\, i_1, i_2\ \bullet \forall\, o_1, o_2 \bullet \\
& (o_1 = f_3\_REQ(i_1, f_2\_REQ(f_1\_REQ(i_2), f_4\_REQ\_w_3(o_1))) \wedge \\
& \quad o_2 = f_4\_REQ\_o_2(o_1)) \Rightarrow \\
\equiv \quad & < by\ \forall - elimination,\ let\ i_1^{'},\ i_2^{'},\ o_1^{'}\ and\ o_2^{'}\ denote\ skolemization\ constants > \\
& (o_1^{'} = f_3\_REQ(i_1^{'}, f_2\_REQ(f_1\_REQ(i_2^{'}), f_4\_REQ\_w_3(o_1^{'}))) \wedge \\
& \quad o_2^{'} = f_4\_REQ\_o_2(o_1^{'})) \Rightarrow \\
& (o_1^{'} = f\_REQ\_o_1(i_1^{'}, i_2^{'}, o_1^{'}) \wedge o_2^{'} = f\_REQ\_o_2(o_1^{'})) \\
\equiv \quad & < by\ the\ definitions\ of\ f\_REQ\_o_1\ and\ f\_REQ\_o_2 > \\
& TRUE
\end{aligned}
$$

The requirements predicates both for the FBD and its components are already formalized. The above proof illustrates the process of proving the correctness for a FBD in terms of its requirements.

For those FBs whose outputs are fed back into the FB itself, we apply mathematical induction on time ticks (i.e., *time_induction*) to prove the correctness theorem; on the other hand, for those FBs whose outputs are not

fed back into the FB itself, we prove the correctness theorem by standard predicate logic. For any reasonably complex FBs supplied by IEC 61131-3, an output variable either is fed back as an input to the FB itself or is connected as an input to another internal FB.

7.3.2.1 Proof Strategy of Correctness for IEC 61131-3 Function Block *LIMITS_ALARM*

For the *LIMITS_ALARM* block (Subsection 7.1.2.3), we split the proof of overall correctness into three lemmas. Each proves the correctness of one output variable. For example, lemma *OUTPUT_QH_CORRECTNESS_CHECKING* proves the correctness of output *QH*.

In Figure 7.33, we summarize the proof structure of the correctness of output variable *QH*. The key proof steps are as follows:

1. Applying *skosimp* to introduce Skolem constants, *H!1*, *L!1*, *EPS!1*, *QH!1*, *QL!1*, *Q!1*, on the universally quantified inputs and outputs, *H*, *L*, *EPS*, *QH*, *QL*, *Q* in the consequent, and then disjunctively simplify the implication into the form that *LIMITS_ALARM_IMPL* in the antecedent and *P_QH* in the consequent.

2. Using *expand*'s to expand *LIMITS_ALARM_IMPL* and *P_QH*.

3. Using *skosimp* to introduce Skolem constants, *w1!1*, *w2!1*, and *w3!1*, for existentially quantified inter-connectors, $w1$, $w2$, and $w3$, in the antecedent, and then split the conjuncts of components in the antecedent.

4. Applying induction scheme *time_induction* on time tick t, generating two branches: base case and inductive step.

   (a) Base case:

      i. Using *skosimp* to introduce Skolem constant *t!1* on t, and then simplify the implication in the consequent.

      ii. Using *expand*'s to expand the requirement predicate of each component, *SUB*, *DIV*, and *HYSTERESIS_REQ*.

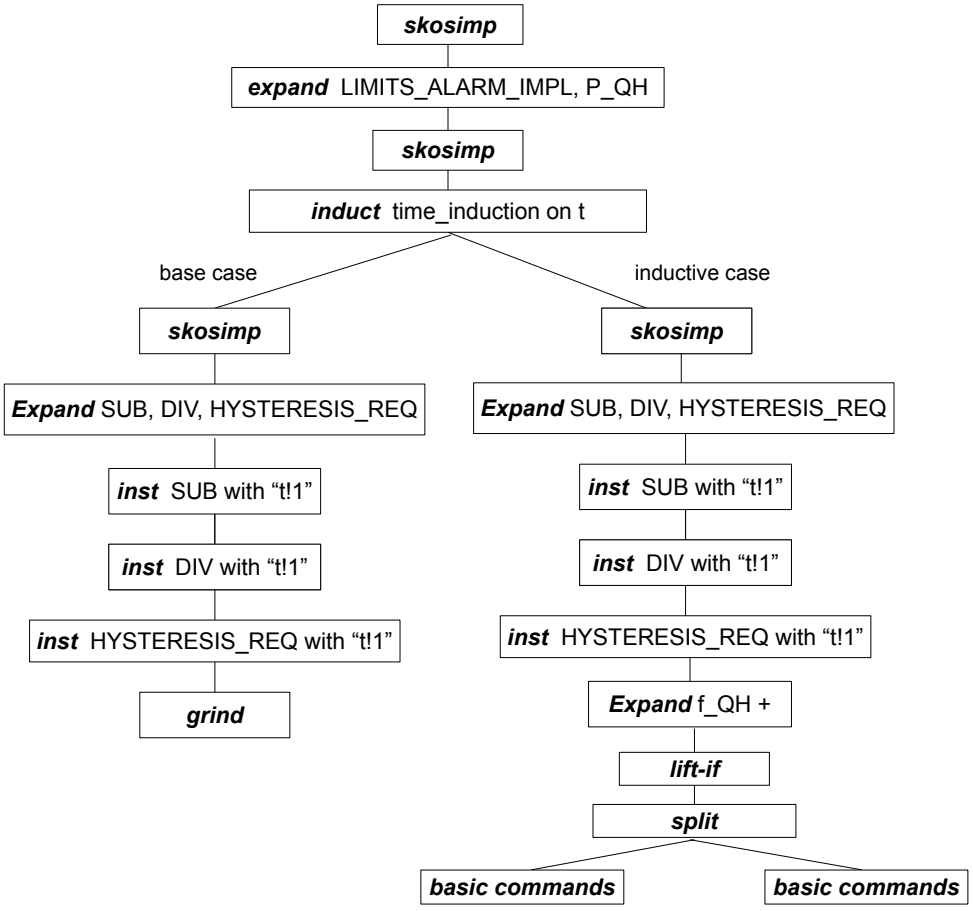


Figure 7.33: Correctness proof structure for output *QH* of *LIMITS_ALARM*

iii. Using *inst*'s to introduce Skolem constant *t!1* on each universally quantified formula of component.

iv. Completing the rest of the proof by decision logic.

(b) Inductive step: the same strategies as for base case first, then *expand* the recursive function *f_QH* in the consequent[14]. Completing the rest of the proof by decision logic.

[14]Proof command (*expand "expr" +*) expands expression *expr* throughout the formulas occurred in the consequent.

7.3.2.2 Proof Strategy of Correctness for Subsystem *Trip Sealed-In*

For the subsystem *Trip Sealed-In*, we prove the correctness of which timing behaviour is considered (Section 7.2).

In Figure 7.34, we summarize the proof structure of the correctness theorem for *Trip Sealed-In* (i.e., theorem *Channel_trip_sealedin_correctness*). The key proof steps are as follows:

1. Applying *skosimp* to introduce Skolem constants, *Any_parameter_trip!1*, *c_ChanTrip!1*, and *Manual_reset_request!1*, *Channel_trip_sealedin!1*, on universally quantified inputs and outputs, *Any_parameter_trip*, *c_ChanTrip*, and *Manual_reset_request*, *Channel_trip_sealedin*, in the consequent, and then disjunctively simplify the implication into the form that implementation predicate *Channel_trip_sealedin_IMPL_P_revised* in the antecedent and requirement predicate *Channel_trip_sealedin_REQ_P* in the consequent.

2. Using *expand*'s to expand the (revised version of) implementation predicate of *Channel_trip_sealedin_IMPL_P_revised*, and the requirement predicate of *Channel_trip_sealedin_REQ_P*.

3. Using *skosimp* to introduce Skolem constants, *w1!1*, ..., and *w7!1*, for existentially quantified inter-connectors, $w1$, ..., and $w7$, in the antecedent, and then split the conjuncts of components in the antecedent.

4. Applying induction scheme *time_induction* on time tick t, generating two branches: base case and inductive step.

   (a) Base case:

      i. Using *skosimp* to introduce Skolem constant *t!1* on t, and then simplify the implication in the consequent.

      ii. Using *expand*'s to expand the functional form of the tabular requirement *Channel_trip_sealedin_REQ_f*, and the component *SEL* that initializes the subsystem.

      iii. Using *inst* to introduce Skolem constant *t!1* on t for the universally quantified formula of *SEL*.

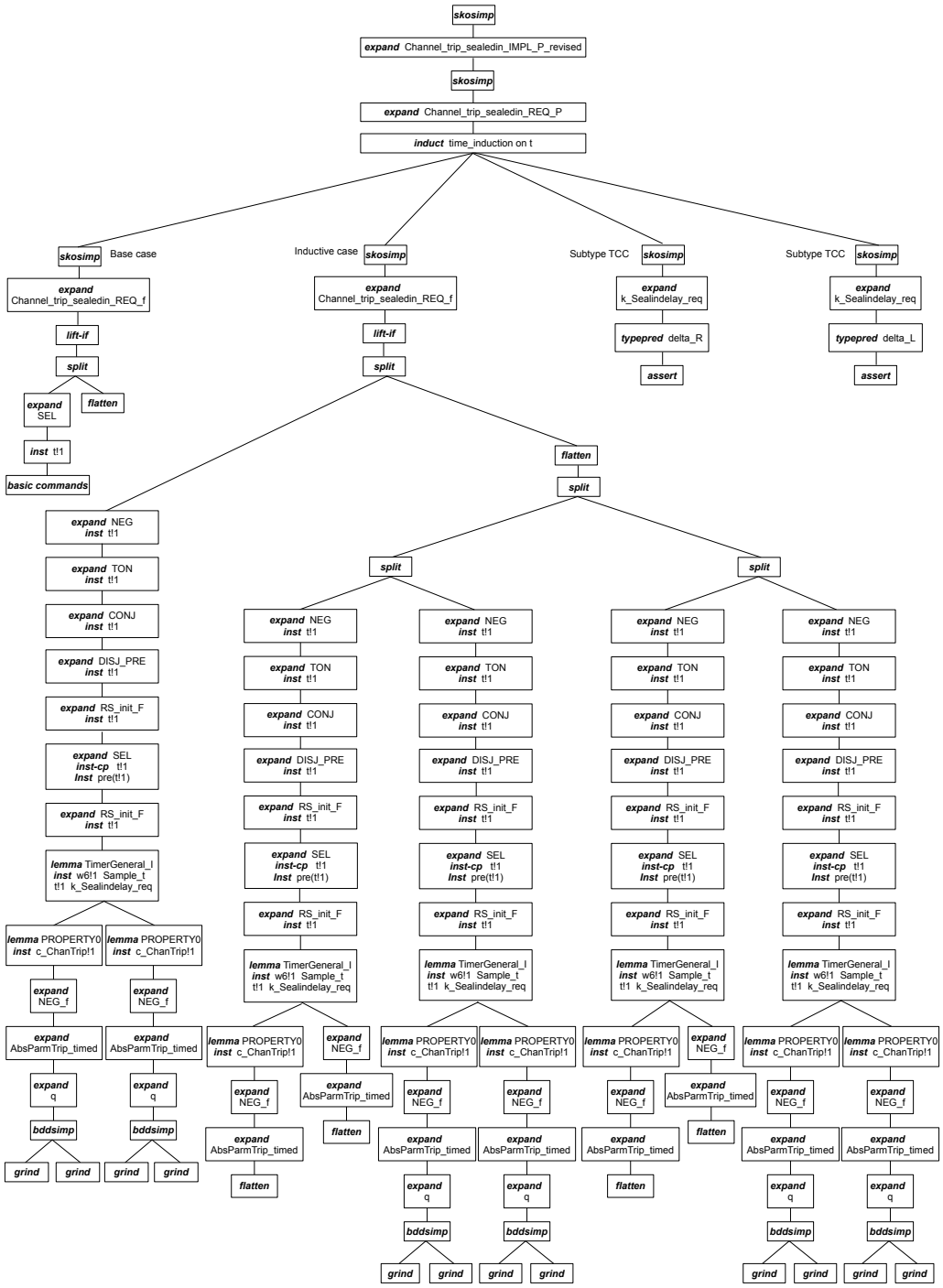


Figure 7.34: Correctness proof structure for the *Trip Sealed-In* subsystem

iv. Completing the rest of the proof by decision logic.

(b) Inductive step:

i. Repeat the first step of base case.

ii. Using *expand*'s to expand the functional form of the tabular requirement *Channel_trip_sealedin_REQ_f* in the consequent. After simplification, the resulting proof goals can be proved in the same pattern. For each proof goal, expanding each component by *expand*, and then introducing Skolem constant *t!1* on t via *inst*. For the component *SEL* of which the output is fed as input back to the subsystem, using *inst-cp* to remain a copy of its requirement predicate and then instantiate another constant *pre(t!1)* by *inst*. By using lemma *TimerGeneral_I* that links the *Held_For* operator in implementation and requirement domains, and lemma *PROPERTY0* that deals with the λ-expressions occurred in *Held_For*, the rest of the proof is completed by proper instantiations and decision logic.

iii. Along with the use of induction, two subtype TCCs are generated for time period *k_Sealindelay_req*. They can be proved using *typepred*'s on *delta_L* and *delta_R* and decision logic.

7.4 Summary

In this chapter, we demonstrated the verification process for the consistency and correctness of the IEC 61131-3 block library and the *Trip Sealed-In* subsystem using our methodology. We provided proof strategies to guide the proofs for consistency and correctness.

For the verification of IEC 61131-3 block library, we identifed a number of different kinds of issues: ambiguous behaviour, missing input assumptions, and inconsistent implementations. We grouped the problematic FBs according to the kinds of issues we found. We discussed each issue in detail and suggested possible solutions to each. For the verification of the *Trip Sealed-In* subsystem (i.e., one built from IEC 61131-3 *TON* timer), we formalized the timing

requirement (using the pre-developed timing operator *Held_For_I* in physical domain) and its FBD implementation (using another pre-developed timing operator *Timer_I* in the software domain) in PVS. We identified an initialization issue and suggested a solution. We were then able to prove its consistency and correctness theorems. For the proof strategies, we provided the key steps to achieve consistency and correctness. We applied our methodology to function block *LIMITS_ALARM* and subsystem *Trip Sealed-In* to illustrate the application of our proof strategies.

The results we achieved on those IEC standard functions and FBs will benefit PLC developers enabling them to reuse these FBs safely, since this represents a one time cost that afterwards can be used in any application. These identified proof strategies are amenable to proof scripts that will facilitate the automated verification work of the consistency and correctness theorems for other FB-based control applications.

Chapter 8

Conclusion

In this thesis we have investigated formal reasoning about function block based PLC control systems, creating a framework in which the consistency and correctness of PLC programs written in industry standard IEC 61131-3 function block diagram (FBD) and structured text (ST) programming languages can be verified. In the 2003 version of IEC 61131-3 (IEC, 2003), an appendix also supplies a block library consisting of some of the most commonly used standard functions and FBs. The main goal of this thesis was to provide a systematic verification approach to function block based control systems utilizing the block library by first formalizing and verifying the block library. As a result, those pre-verified FBs can be safely reused in other PLC applications. We performed our verification work in the environment of the theorem prover PVS, applying it to both real- and non real-time FBs. The basis of higher-order logic, formal verification, tabular expression and an understanding of PLCs are required to apply our proposed method in real applications.

8.1 Highlight of the Contributions

In this section we highlight the contributions related to the proposed formal reasoning approach for FB-based control systems.

We formalized the input-output requirements of standard functions and FBs as predicates using tabular expressions, a proven and effective approach

describing conditionals and relations. The formalized black-box requirements predicate captures the constraints characterizing the relation between input and output variables at each time instant. With the use of pre-verified timing operator *Held_For*, we also formalized three types of timers supplied by IEC 61131-3 that can be used to design real-time FB subsystems. The resulting requirements for standard functions and FBs are well-defined, i.e., they are both complete and disjoint.

We formalized the FB implementations that are described in ST and/or FBD. For the ST implementation, we developed a list of translation rules as guidance to convert ST description into PVS, sufficient for all ST descriptions in IEC 61131-3. For the FBD implementation, we provided a compositional approach to specify the overall behaviour predicate, which is based on the formalizations of its internal FBs.

Utilizing PVS formalizations for requirements and implementation of a FB, we perform two kinds of verification. We first verify the correctness of a FB, i.e., the given FB implementation is functionally correct with respect to its intended behaviour (black-box requirements). The proof for the overall correctness can be decomposed into the correctness proof for each output variable. We then verify the consistency of a FB, i.e., for each input combination, there exists at least one corresponding assignment to the outputs such that implementation predicate holds. In addition, we verify the equivalence of ST and FBD implementations if both are supplied for a FB.

We applied our verification approach on the standard functions and FBs supplied by the IEC 61131-3 standard, which has been widely used for over two decades in industrial automation. Our effort has led to the discovery of several kinds of issues in the standard: ambiguous behaviour, missing input assumptions, and inconsistent implementations. We provide our suggested solution to each. As a result, with the incorporation of our solutions, the IEC 61131-3 standard functions and FBs can be reused safely in any PLC-based control system development. Table 8.1 summarizes the issues found in the IEC 61131-3 standard, as well as our suggested solutions.

To assist the verification work, we provided proof strategies for the consistency and correctness theorems. These proof strategies have been suc-

Issue	FB Subsystem	Solution
Ambiguous Behaviour	*PULSE* Timer	Using function tables to formalize its behaviour which ensures both completeness and disjointness.
	SR and *RS* latches	Explicitly adding unit delay block z_{-1} to formalize feedback loop.
Missing Input Assumption	*LIMITS_ALARM*	Imposing two assumptions to ensure that two hysteresis zones (1) are computed in the right directions; and (2) do not overlap.
	Up Counter *CTU*	Adding assumption to constrain the preset value PV has to less than maximal value $PVmax$.
	Down Counter *CTD*	Adding assumption to constrain the preset value PV has to larger than minimal value $PVmin$.
	Up-down Counter *CTUD*	Adding assumption to constrain the preset value PV has to be in-between minimal and maximal valves $PVmin$ and $PVmax$.
	HYSTERESIS	Adding assumption on deadband size to eliminate unintended toggling behaviour.
	DIFFEQ	Using dependent types to specify the constraint on the lengths of input and output history.
	N-sample *DELAY*	Redefining the length of N to exclude the case of 0-delay.
	AVERAGE	Constraining the type of N to exclude value 0.
	PID	Constraining the type of TR to exclude value 0.
	ALRM_INT	Adding assumption to constrain the high and low limits.
Inconsistent Implementations	*STACK_INT*	Adding one negation block *NEG* to correct the supplied FBD.

Table 8.1: Summary for problematic FBs in IEC 61131-3

cessfully applied on all case studies in this thesis. We distinguish the proofs by those who require induction and those that do not. For those proofs that require induction, we adopted an induction scheme (i.e., *time_induction*) over time ticks; for those proofs that do not require induction, we completed the proofs using definition expansion, arithmetic, equality, and quantifier reasoning. We illustrated the applications of the proof structures on block *LIMITS_ALARM* from IEC 61131-3 and a generalized real-time FB subsystem *Trip Sealed-In* from an industrial application.

Overall, we manually formalized and verified 29 FBs from the IEC 61131-3 standard into PVS, 16 FBs in ST, 4 FBs in FBDs, 5 FBs in both of ST and FBDs, 3 FBs in timing diagrams and 1 FB is described in natural language. We performed 115 proof tasks for consistency, correctness, functional equivalence (if applicable), and some required properties. Additionally, 350 proof obligations of TCCs are automatically generated in PVS. We also applied our methodology to two realistic sub-systems taken from the nuclear domain, i.e., the *Trip Sealed-In* subsystem and the *Pushbutton* subsystem. We proved 43 TCCs and 40 proof tasks in PVS. Only simple TCCs can be proved automatically, while others need human intervention.

The methodology has been tried successfully on real industrial problems in the nuclear industry, and its applicability has been demonstrated in it being chosen for use by the companies performing this work in a regulated environment.

8.2 Limitations and Future Research

In this thesis we have not provided complete translation rules for arbitrary *ST–to–PVS*. No attempt was made to deal with ST constructs such as `WHILE` and `REPEAT`, and more complex ST structures such as nested `IF` statements in `FOR` loops, or vice versa. The provided rules can be further extended to formally translate more general ST program into PVS, not just the rules that are sufficient for the examples of the IEC 61131-3 standard.

Requirement validation is concerned with checking whether the required behaviour of the PLC software system will result in a system that

meets its users expectations. It is necessary to ensure that the requirements specification contains no errors and that it specifies the intended behaviour correctly. The work reported in this thesis involves providing and making assumptions about the requirements specifications of IEC 61131-3 FBs, based on experience that would be consistent with industrial norms. It helps to validate the requirements specifications with the engineers and users of FBs. Since the requirements specifications of FBs are specified in tabular expressions, it helps the engineers and users to comprehend and maintain the requirements specifications.

Although the proof strategies are used as guidance for the proofs of consistency and correctness theorems, the proofs still involve a certain amount of moderately tedious interactions. Developing proof tactics to provide a more automated method for proving consistency and correctness theorems would greatly facilitate the use of our approach in larger practical applications. One possible solution to this problem is the use of advanced user-defined proof scripts supported by PVS to achieve greater levels of automation and customization. Similar to LCF-style tactics, the capabilities of the PVS proof checker can be extended by defining proof scripts. The proof scripts are often used to discharge TCCs, make simplifications, rewrite decision procedures, and perform proofs by induction. These proof scripts can be made based on: (1) our suggested proof patterns; (2) the construction of definitions, lemmas, and theorems; and (3) recovering from failed proof attempts.

As a candidate successor of IEC 61131-3, IEC 61499 provides a generic model for distributed control systems. The basic concept of function block in this standard has more features, e.g., separation of data and event flow and object-orientation. We may adapt, and possibly extend, our approach for verifying IEC 61499 function blocks in the future.

Appendix A

Time Theory

This appendix contains the PVS file for *Time* theory (Hu, 2008).

```
Time: THEORY

BEGIN

  time: TYPE+ = nonneg_real

  non_initial_time: TYPE+ = posreal

END Time
```

Appendix B

ClockTick Theory

The *ClockTick* theory imports the *Time* theory, and defines the *tick* type as a subtype of the *time* type. Variable *delta_t* is passed as a parameter that can be instantiated when the theory is imported. The *ClockTick* theory is updated from (Hu, 2008).

```
ClockTick[delta_t: posreal]: THEORY

BEGIN

  IMPORTING Time

  n: VAR nat

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

  x: VAR tick

  init(x): bool = (x = 0)

  snd(x): bool = (x = delta_t)

  trd(x) : bool = (x = 2 * delta_t)

  fth(x) : bool = (x = 3 * delta_t)

  noninit_elem: TYPE = {x | NOT init(x)}

  from_2nd_tick: TYPE = { x | NOT init(x) & NOT snd(x) }

  from_3rd_tick: TYPE = { x | NOT init(x) & NOT snd(x) & NOT trd(x) }

  y: VAR noninit_elem

  yy: VAR from_2nd_tick
```

```
  yyy: VAR from_3rd_tick

  pre(y): tick = y - delta_t

  pre_2(yy): tick = yy - 2 * delta_t

  pre_3(yyy): tick = yyy - 3 * delta_t

  next(x): tick = x + delta_t

  rank(x): nat = x / delta_t

  time_induct: LEMMA
    FORALL (P: pred[tick]):
      (FORALL x, n: rank(x) = n IMPLIES P(x)) IMPLIES (FORALL x: P(x))

  time_induction: PROPOSITION
    FORALL (P: pred[tick]):
      (FORALL (t: tick): init(t) IMPLIES P(t)) AND
       (FORALL (t: noninit_elem): P(pre(t)) IMPLIES P(t))
       IMPLIES (FORALL (t: tick): P(t))

  tick_PROPERTY0: LEMMA
    FORALL (n1, n2: nat):
      n1 * delta_t > n2 * delta_t IFF n1 * delta_t - delta_t >= n2 * delta_t

  tick_PROPERTY1: LEMMA FORALL (t: tick | t > 0): t > x IFF pre(t) >= x

END ClockTick
```

Appendix C

Defined_operators Theory

The *defined_operators* theory defines basic operators.

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

BEGIN

  IMPORTING ClockTick[delta_t]

  Condition_type: TYPE = pred[tick]

  t, ts, te, duration: VAR tick

  neg(i, out: bool): bool = (out = NOT i)

  conj(i1, i2, out: bool): bool = (out = (i1 & i2))

  conj_3(i1, i2, i3, out: bool): bool = (out = (i1 & i2 & i3))

  conj_4(i1, i2, i3, i4, out: bool): bool = (out = (i1 & i2 & i3 & i4))

  disj(i1, i2, out: bool): bool = (out = (i1 OR i2))

  disj(i1: pred[tick], i2: pred[tick], out: pred[tick]): bool =
     FORALL (t:tick): out(t) = (i1(t) OR i2(t))

  disj_4(i1, i2, i3, i4, out: bool): bool = (out = (i1 OR i2 OR i3 OR i4))

  Z(i, out: pred[tick])(t): bool =
     IF init(t) THEN (out(t) = False) ELSE out(t) = i(pre(t)) ENDIF

  add(i1, i2: int): int = (i1 + i2)

  add(i1, i2: real): real = (i1 + i2)
```

```
add(i1, i2, out: real): bool = (out = (i1 + i2))

add(i1: [tick->real], i2: [tick->real], out: [tick->real]): bool =
   FORALL (t: tick): (out(t) = (i1(t) + i2(t)))

add(en, eno: pred[tick], i1, i2, out: [tick->int]): bool =
  FORALL t:
  IF init(t) THEN (out(t) = -1) & NOT eno(t)
  ELSE TABLE
         %-------------------------------------------------------------%
         | en(t)     | (out(t) = (i1(pre(t)) + i2(pre(t)))) & eno(t) ||
         %-------------------------------------------------------------%
         | NOT en(t) | (out(t) = i1(pre(t))) & NOT eno(t)            ||
         %-------------------------------------------------------------%
        ENDTABLE
  ENDIF

add_1(en, eno: pred[tick], i, out: [tick->int]): bool =
  FORALL t:
  IF init(t) THEN (out(t) = -1) & NOT eno(t)
  ELSE TABLE
        %-------------------------------------------------%
        | en(t)     | (out(t) = (i(pre(t)) + 1)) & eno(t) ||
        %-------------------------------------------------%
        | NOT en(t) | (out(t) = i(pre(t))) & NOT eno(t)   ||
       %-------------------------------------------------%
        ENDTABLE
  ENDIF

mul(i1, i2: int): int = (i1 * i2)

div(i1: real, i2: {i:real| i /= 0 }): real = i1 / i2

div(i1: real, i2: {i:real| i /= 0 }, out: real): bool = (out = (i1 / i2))

div(i1: [tick->real], i2: [tick->{i:real| i /= 0 }],
    out: [tick->real]): bool =
   FORALL (t:tick): (out(t) = (i1(t) / i2(t)))

sub(i1, i2: int): int = (i1 - i2)

sub(i1, i2: real): real = (i1 - i2)

sub(i1, i2, out: real): bool = (out = (i1 - i2))

sub(i1: [tick->real], i2: [tick->real], out: [tick->real]): bool =
   FORALL (t: tick): (out(t) = (i1(t) - i2(t)))

sub(en, eno: pred[tick], i1, i2, out: [tick->int]): bool =
```

```
 FORALL t:
 IF init(t) THEN (out(t) = -1) & NOT eno(t)
 ELSE TABLE
       %-------------------------------------------------------------%
       | en(t)     | (out(t) = (i1(pre(t)) - i2(pre(t)))) & eno(t) ||
       %-------------------------------------------------------------%
       | NOT en(t) | (out(t) = i1(pre(t))) & NOT eno(t)            ||
       %-------------------------------------------------------------%
       ENDTABLE
 ENDIF

sub_1(en, eno: pred[tick], i, out: [tick->int]): bool =
 FORALL t:
 IF init(t) THEN (out(t) = -1) & NOT eno(t)
 ELSE TABLE
       %-------------------------------------------------%
       | en(t)     | (out(t) = (i(pre(t)) - 1)) & eno(t) ||
       %-------------------------------------------------%
       | NOT en(t) | (out(t) = i(pre(t))) & NOT eno(t)   ||
       %-------------------------------------------------%
      ENDTABLE
 ENDIF

move(i,out: int): bool = (out = i)

move_int(en, eno: pred[tick], i, out: [tick->int]): bool =
 FORALL t:
 IF init(t) THEN (out(t) = 0) & NOT eno(t)
 ELSE TABLE
       %-------------------------------------------------%
       | en(t)     | (out(t) = i(t)) & eno(t)            ||
       %-------------------------------------------------%
       | NOT en(t) | (out(t) = out(pre(t))) & NOT eno(t) ||
       %-------------------------------------------------%
      ENDTABLE
ENDIF

move_bool(en, eno: pred[tick], i, out: pred[tick]): bool =
 FORALL t:
 IF init(t) THEN (out(t) = False) & NOT eno(t)
 ELSE TABLE
       %-------------------------------------------------%
       | en(t)     | (out(t) = i(t)) & eno(t)            ||
       %-------------------------------------------------%
       | NOT en(t) | (out(t) = out(pre(t))) & NOT eno(t) ||
       %-------------------------------------------------%
      ENDTABLE
ENDIF
```

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

gt(i1, i2: int): bool = (i1 > i2)

gt(i1, i2: real): bool = (i1 > i2)

gt(i1, i2: int, out: bool): bool = (out = (i1 > i2))

eq(i1, i2: int): bool = (i1 = i2)

eq(en, eno: pred[tick], i1, i2: [tick->int], out: pred[tick]): bool =
 FORALL t:
 IF init(t) THEN NOT out(t) & NOT eno(t)
 ELSE TABLE
       %-------------------------------------------------------------%
       | en(t) & (i1(t) = i2(t))  | out(t) & eno(t)                ||
       %-------------------------------------------------------------%
       | en(t) &  (i1(t) /= i2(t))| NOT out(t) & eno(t)            ||
       %-------------------------------------------------------------%
       | NOT en(t)               | (out(t) = out(pre(t))) & NOT eno(t)||
       %-------------------------------------------------------------%
      ENDTABLE
ENDIF

eq(en: pred[tick], i1, i2: [tick->int], out: pred[tick]): bool =
 FORALL t:
 IF init(t) THEN NOT out(t)
 ELSE TABLE
        %-------------------------------------------------%
        | en(t) & (i1(t) = i2(t))   | out(t)             ||
        %-------------------------------------------------%
        | en(t) &  (i1(t) /= i2(t)) | NOT out(t)         ||
        %-------------------------------------------------%
        | NOT en(t)                 | out(t) = out(pre(t))||
        %-------------------------------------------------%
      ENDTABLE
ENDIF

ge(i1, i2: int): bool = (i1 >= i2)
```

```
ge(i1, i2: real): bool = (i1 >= i2)

le(i1, i2: int): bool = (i1 <= i2)

le(i1, i2: real): bool = (i1 <= i2)

lt(i1, i2: int): bool = (i1 < i2)

lt(i1, i2: int, out: bool): bool = (out = (i1 < i2))

lt(i1, i2: real): bool = (i1 < i2)

lt(en, eno: pred[tick], i1, i2: [tick->int], out: pred[tick]): bool =
 FORALL t:
 IF init(t) THEN NOT out(t) & NOT eno(t)
 ELSE TABLE
       %-------------------------------------------------------------%
       | en(t) & (i1(t) < i2(t)) | out(t) & eno(t)                  ||
       %-------------------------------------------------------------%
       | en(t) & (i1(t) >= i2(t))| NOT out(t) & eno(t)              ||
       %-------------------------------------------------------------%
       | NOT en(t)               | (out(t) = out(pre(t))) & NOT eno(t)||
       %-------------------------------------------------------------%
      ENDTABLE
ENDIF

ne(i1, i2: int): bool = (i1 /= i2)

sel(g: bool, i1, i2: int): int = IF g = FALSE THEN i1 ELSE i2 ENDIF

sel(en, eno, g: pred[tick], in0, in1: int, out: [tick->int]): bool =
 FORALL t:
 IF init(t) THEN (out(t) = 0) & NOT eno(t)
 ELSE TABLE
       %-----------------------------------------------------------%
       | en(t) & NOT g(t) | (out(t) = in0) & eno(t)                ||
       %-----------------------------------------------------------%
       | en(t) & g(t)     | (out(t) = in1) & eno(t)                ||
       %-----------------------------------------------------------%
       | NOT en(t)        | (out(t) = out(pre(t))) & NOT eno(t)    ||
       %-----------------------------------------------------------%
      ENDTABLE
ENDIF

sel(g: pred[tick], in0, in1: [tick->int], out: [tick->int]): bool =
 FORALL t:
 IF init(t) THEN out(t) = 0
 ELSE TABLE
```

```
        %-----------------------------%
        | NOT g(t) | out(t) = in0(t) ||
        %-----------------------------%
        | g(t)     | out(t) = in1(t) ||
        %-----------------------------%
       ENDTABLE
 ENDIF

 min(i1, i2: int): int = IF i1 <= i2 THEN i1 ELSE i2 ENDIF

 max(i1, i2: int): int = IF i1 >= i2 THEN i1 ELSE i2 ENDIF

 limit(i, mn, mx: int): int = min(max(i,mn),mx)

 limit(en, eno: pred[tick], i, mn, mx, out: [tick->int]): bool =
  FORALL t:
  IF init(t) THEN (out(t) = 128) & NOT eno(t)
  ELSE TABLE
         %-------------------------------------------------------------%
         | en(t)     | (out(t) = min(max(i(t),mn(t)),mx(t))) & eno(t) ||
         %-------------------------------------------------------------%
         | NOT en(t) | (out(t) = out(pre(t))) & NOT eno(t)            ||
         %-------------------------------------------------------------%
        ENDTABLE
    ENDIF

 mux(k: {i:int| i >= 0 & i <= 3 }, i0, i1, i2, i3: int): int =
   TABLE
     %-------------%
     | k = 0 | i0 ||
     %-------------%
     | k = 1 | i1 ||
     %-------------%
     | k = 2 | i2 ||
     %-------------%
     | k = 3 | i3 ||
     %-------------%
   ENDTABLE

 Held_For(P: Condition_type, duration: tick)(t): 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: Condition_type, duration, ts: tick)(t): bool =
   (t - ts >= duration) &
        (FORALL (t_n: tick | t_n >= ts & t_n <= t): P(t_n))

END defined_operators
```

Appendix D

Timers Blocks

This appendix contains the PVS formalizations for timer blocks *TON*, *TOF*, and *PULSE*.

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

BEGIN

  IMPORTING ClockTick[delta_t]

  t: VAR tick

  IN: VAR pred[tick]

  PT: VAR [tick -> tick]

  last_enabled(IN)(t): RECURSIVE tick =
    IF init(t) THEN 0
    ELSE
     TABLE
      %-----------------------------------------------------%
      | NOT IN(pre(t)) & IN(t)  |              t            ||
      %-----------------------------------------------------%
      | IN(pre(t)) OR NOT IN(t) | last_enabled(IN)(pre(t)) ||
      %-----------------------------------------------------%
     ENDTABLE
    ENDIF
  MEASURE rank(t)

  q(IN,PT)(t): bool =
    IF init(t) THEN False
    ELSE
     TABLE
      %-----------------------------------------------------%
      | IN(t) & (t - last_enabled(IN)(t) >= PT(t)) | TRUE  ||
```

```
      %-----------------------------------------------------%
      | IN(t) & (t - last_enabled(IN)(t) <  PT(t)) | FALSE ||
      %-----------------------------------------------------%
      | NOT IN(t)                                  | FALSE ||
      %-----------------------------------------------------%
     ENDTABLE
    ENDIF

  ET(IN,PT)(t): tick =
    IF init(t) THEN 0
    ELSE
     TABLE
      %-------------------------------------------------------------------%
      | IN(t) & t - last_enabled(i)(t) >= PT(t) |             PT(t)      ||
      %-------------------------------------------------------------------%
      | IN(t) & t - last_enabled(i)(t) < PT(t)  |t - last_enabled(IN)(t) ||
      %-------------------------------------------------------------------%
      | NOT IN(t)                               |            0           ||
      %-------------------------------------------------------------------%
     ENDTABLE
    ENDIF

END TON



TOF[(IMPORTING Time) delta_t:posreal]: THEORY

BEGIN

  IMPORTING ClockTick[delta_t]

  t: VAR tick

  IN: VAR pred[tick]

  PT: VAR [tick -> tick]

  last_disabled(IN)(t): RECURSIVE tick =
    IF init(t) THEN 0
    ELSE
     TABLE
      %-----------------------------------------------------%
      | IN(pre(t)) & NOT IN(t)  |               t           ||
      %-----------------------------------------------------%
      | NOT IN(pre(t)) OR IN(t) | last_disabled(IN)(pre(t)) ||
      %-----------------------------------------------------%
     ENDTABLE
    ENDIF
  MEASURE rank(t)
```

```
  Q(IN,PT)(t): bool =
    IF init(t) THEN True
    ELSE
     TABLE
      %-----------------------------------------------------------%
      | NOT IN(t) & (t - last_disabled(IN)(t) >= PT(t)) | FALSE ||
      %-----------------------------------------------------------%
      | NOT IN(t) & (t - last_disabled(IN)(t) <  PT(t)) | TRUE  ||
      %-----------------------------------------------------------%
      | IN(t)                                           | TRUE  ||
      %-----------------------------------------------------------%
     ENDTABLE
    ENDIF

  ET(IN,PT)(t): tick =
    IF init(t) THEN 0
    ELSE
     TABLE
      %-------------------------------------------------------------------------%
      |NOT IN(t) & t - last_disabled(IN)(t)>=PT(t)|          pt(t)         ||
      %-------------------------------------------------------------------------%
      |NOT IN(t) & t - last_disabled(IN)(t)< PT(t)|t - last_disabled(IN)(t)||
      %-------------------------------------------------------------------------%
      | IN(t)                                     |          0             ||
      %-------------------------------------------------------------------------%
     ENDTABLE
    ENDIF
  END TOF


PULSE[(IMPORTING Time) delta_t:posreal]: THEORY

BEGIN

  IMPORTING ClockTick[delta_t]

  IMPORTING defined_operators[delta_t]

  t: VAR tick

  IN: VAR pred[tick]

  PT: VAR [tick -> tick]

  Q: VAR pred[tick]

  ET: VAR [tick -> tick]
```

```
  Q(IN,PT,Q)(t): bool =
    IF init(t) THEN FALSE
      ELSE
        TABLE
         %-------------------------------------------------%
         | NOT Q(pre(t)) & NOT IN(pre(t)) & IN(t)    | TRUE  ||
         %-------------------------------------------------%
         | NOT Q(pre(t)) & (IN(pre(t)) OR NOT IN(t)) | FALSE ||
         %-------------------------------------------------%
         | Q(pre(t)) & NOT Held_For(Q,PT(t))(t)      | TRUE  ||
         %-------------------------------------------------%
         | Q(pre(t)) & Held_For(Q,PT(t))(t)          | FALSE ||
         %-------------------------------------------------%
        ENDTABLE
    ENDIF

  pulse_start_time(Q)(t): RECURSIVE tick =
    IF init(t) THEN 0
    ELSE
     TABLE
      %-------------------------------------------------------%
      | NOT Q(pre(t)) & Q(t)  |              t              ||
      %-------------------------------------------------------%
      | Q(pre(t)) OR NOT Q(t) | pulse_start_time(Q)(pre(t)) ||
      %-------------------------------------------------------%
     ENDTABLE
    ENDIF
  MEASURE rank(t)

  ET(IN,PT,Q)(t): tick =
    IF init(t) THEN 0
    ELSE
     TABLE
      %-------------------------------------------------%
      |Q(t)                 | t - pulse_start_time(Q)(t) ||
      %-------------------------------------------------%
      |NOT Q(t) & NOT IN(t) |              0             ||
      %-------------------------------------------------%
      |NOT Q(t) & IN(t)     |            PT(t)           ||
      %-------------------------------------------------%
     ENDTABLE
    ENDIF

END PULSE
```

Appendix E

Block *STACK_INT*

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.

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.

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

Figure E.1: Requirement for internal *NI* of block *STACK_INT*

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*.

Output Variables

The output *EMPTY* is a Boolean flag indicating if the current stack is empty.

Condition			*Result* PTR
R1			-1
¬R1	POP ∧ ¬EMPTY$_{-1}$		PTR$_{-1}$-1
	¬POP ∨ EMPTY$_{-1}$	PUSH ∧ ¬OFLO$_{-1}$	PTR$_{-1}$+1
		¬PUSH ∨ OFLO$_{-1}$	NC

Figure E.2: Requirement for internal *PTR* of block *STACK_INT*

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

Figure E.3: Requirement for internal *STK* of block *STACK_INT*

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.

Condition			*Result* EMPTY
R1			TRUE
¬R1	POP ∧ ¬EMPTY$_{-1}$	PTR < 0	TRUE
		PTR ≥ 0	FALSE
	¬POP ∨ EMPTY$_{-1}$	PUSH ∧ ¬OFLO$_{-1}$	FALSE
		¬PUSH ∨ OFLO$_{-1}$	NC

Figure E.4: Requirement for output *EMPTY* of block *STACK_INT*

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*[1] and a push operation is performed. 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., $\neg OFLO$).

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

Condition				*Result* OFLO
R1				FALSE
¬R1	POP ∧ ¬EMPTY$_{-1}$			FALSE
	¬POP ∨ EMPTY$_{-1}$	PUSH ∧ ¬OFLO$_{-1}$	PTR = NI	TRUE
			PTR ≠ NI	FALSE
		¬PUSH ∨ OFLO$_{-1}$		NC

Figure E.5: Requirement for output *OFLO* of block *STACK_INT*

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*).

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

Figure E.6: Requirement for output *OUT* of block *STACK_INT*

The PVS formalization for block *STACK_INT* is as follows. It imports two theories, `Time` (Appendix A), and `defined_operators` (Appendix C).

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

BEGIN

  IMPORTING ClockTick[delta_t]

  IMPORTING defined_operators[delta_t]

  t: VAR tick

  inp, n, out: VAR [tick -> int]

  r1, oflo, empty, pop, push, ret1, ret2, ret3: VAR pred[tick]

  index: TYPE = {ind: int | ind >= -1 & ind <= 127 }

  index_revised: TYPE = {ind: int | ind >= -1 & ind <= 128 }

  storage_size: TYPE = {size: posint | size >= 1 & size <= 128}

  stack: TYPE = ARRAY[index -> int]

  stack_revised: TYPE = ARRAY[index_revised -> int]

  ni: VAR [tick -> storage_size]

  ptr: VAR [tick -> index]

  ptr_revised: VAR [tick -> index_revised]

  stk: VAR stack

  stk_revised: VAR stack_revised

  move_int(en, eno: pred[tick], i, out: [tick->int])(t): bool =
    IF init(t) THEN (out(t) = 0) & (eno(t) = FALSE)
    ELSE TABLE
         %-------------------------------------------------%
         | en(t)     | (out(t) = i(t)) & (eno(t) = TRUE)   ||
         %-------------------------------------------------%
         | NOT en(t) |                   (eno(t) = FALSE)  ||
         %-------------------------------------------------%
        ENDTABLE
    ENDIF

  move_int_en(en: pred[tick], i, out: [tick->int])(t): bool =
```

```
  IF init(t) THEN out(t) = 0
  ELSE TABLE
       %------------------------------%
       | en(t)     | (out(t) = i(t)) ||
       %------------------------------%
       | NOT en(t) |      TRUE       ||
       %------------------------------%
      ENDTABLE
  ENDIF

move_int_ptr(en, eno: pred[tick], i, out: [tick->int])(t): bool =
  IF init(t) THEN (out(t) = -1) & (eno(t) = FALSE)
  ELSE TABLE
       %-------------------------------------------------%
       | en(t)     | (out(t) = i(t)) & (eno(t) = TRUE)   ||
       %-------------------------------------------------%
       | NOT en(t) |                   (eno(t) = FALSE)  ||
       %-------------------------------------------------%
      ENDTABLE
  ENDIF

move_bool_oflo(en, eno: pred[tick], i, out:pred[tick])(t): bool =
  IF init(t) THEN (out(t) = FALSE) & (eno(t) = FALSE)
  ELSE TABLE
       %-------------------------------------------------%
       | en(t)     | (out(t) = i(t)) & (eno(t) = TRUE)   ||
       %-------------------------------------------------%
       | NOT en(t) |                   (eno(t) = FALSE)  ||
       %-------------------------------------------------%
      ENDTABLE
  ENDIF

move_bool_empty(en, eno: pred[tick], i, out: pred[tick])(t): bool =
  IF init(t) THEN (out(t) = TRUE) & (eno(t) = FALSE)
  ELSE TABLE
       %-------------------------------------------------%
      | en(t)     | (out(t) = i(t)) & (eno(t) = en(t))  ||
       %-------------------------------------------------%
       | NOT en(t) |                   (eno(t) = en(t))  ||
       %-------------------------------------------------%
      ENDTABLE
  ENDIF

sub_1_ptr(en, eno: pred[tick], out: [tick->int])(t): bool =
  IF init(t) THEN (out(t) = -1) & (eno(t) = FALSE)
  ELSE TABLE
       %-------------------------------------------------------------%
       | en(t) & (out(pre(t)) = -1)   | (out(t) = (out(pre(t))))   &
                                        (eno(t) = FALSE)           ||
```

```
        %-------------------------------------------------------------%
        | en(t) & (out(pre(t)) /= -1)  | (out(t) = out(pre(t)) - 1) &
                                         (eno(t) = TRUE)            ||
        %-------------------------------------------------------------%
        | NOT en(t)                    | (eno(t) = FALSE)           ||
        %-------------------------------------------------------------%
       ENDTABLE
  ENDIF

add_1_ptr(en, eno: pred[tick], out, sp: [tick->int])(t): bool =
  IF init(t) THEN (out(t) = -1) & (eno(t) = FALSE)
  ELSE TABLE
        %-------------------------------------------------------------%
        | en(t) & (out(pre(t)) = sp(t))  | (out(t) = out(pre(t)))     &
                                           (eno(t) = FALSE)           ||
        %-------------------------------------------------------------%
        | en(t) & (out(pre(t)) /= sp(t)) | (out(t) = out(pre(t)) + 1) &
                                           (eno(t) = TRUE)            ||
        %-------------------------------------------------------------%
        | NOT en(t)                      | (eno(t) = FALSE)           ||
        %-------------------------------------------------------------%
       ENDTABLE
  ENDIF

lt_empty(en,eno:pred[tick], i1,i2:[tick->int], out:pred[tick])(t): bool =
  IF init(t) THEN (out(t) = TRUE) & (eno(t) = FALSE)
  ELSE TABLE
        %-------------------------------------------------------------%
        | en(t) & (i1(t) < i2(t))  |(out(t) = TRUE)  & (eno(t) = TRUE) ||
        %-------------------------------------------------------------%
        | en(t) & (i1(t) >= i2(t)) |(out(t) = FALSE) & (eno(t) = TRUE) ||
        %-------------------------------------------------------------%
        | NOT en(t)                |                   (eno(t) = FALSE)||
        %-------------------------------------------------------------%
       ENDTABLE
  ENDIF

sel_update(en,eno,g:pred[tick], in0,in1:int, out:[tick->int])(t): bool =
  IF init(t) THEN (out(t) = 0) & (eno(t) = FALSE)
  ELSE TABLE
        %-----------------------------------------------------%
        | en(t) & NOT g(t) | (out(t) = in0) & (eno(t) = TRUE)  ||
        %-----------------------------------------------------%
        | en(t) & g(t)     | (out(t) = in1) & (eno(t) = TRUE)  ||
        %-----------------------------------------------------%
        | NOT en(t)        |                  (eno(t) = FALSE) ||
        %-----------------------------------------------------%
       ENDTABLE
  ENDIF
```

```
sel_update_en(en,g:pred[tick], in0,in1: int, out:[tick->int])(t): bool =
  IF init(t) THEN (out(t) = 0)
  ELSE TABLE
       %-----------------------------------%
       | en(t) & NOT g(t) | (out(t) = in0)  ||
       %-----------------------------------%
       | en(t) & g(t)     | (out(t) = in1)  ||
       %-----------------------------------%
       | NOT en(t)        |     TRUE        ||
       %-----------------------------------%
      ENDTABLE
  ENDIF

eq_oflo(en: pred[tick], i1, i2: [tick->int], out: pred[tick])(t): bool =
  IF init(t) THEN (out(t) = 0)
  ELSE TABLE
       %-----------------------------------------------%
       | en(t) & (i1(t) = i2(t))   | (out(t) = TRUE)  ||
       %-----------------------------------------------%
       | en(t) &  (i1(t) /= i2(t)) | (out(t) = FALSE) ||
       %-----------------------------------------------%
       | NOT en(t)                 |       TRUE       ||
       %-----------------------------------------------%
      ENDTABLE
  ENDIF

neg_en(en: pred[tick], i, out: pred[tick])(t): bool =
  IF init(t) THEN out(t) = FALSE
  ELSE TABLE
       %---------------------------------%
       | en(t)     | out(t) = NOT i(t) ||
       %---------------------------------%
       | NOT en(t) | out(t) = FALSE    ||
      %---------------------------------%
      ENDTABLE
  ENDIF

Z_update(i, out: [tick->index])(t): bool =
    IF init(t) THEN (out(t) = -1) ELSE out(t) = i(pre(t)) ENDIF

f_stk_ptr(r1, pop, empty, push, oflo,
                 inp, ptr_revised)(t): RECURSIVE stack_revised =
   IF init(t) THEN LAMBDA (ptrr: index_revised): 0
   ELSE TABLE
        %-------------------------------------------------------------%
        | NOT r1(t) & NOT (pop(t) & NOT empty(pre(t))) &
           push(t) & NOT oflo(pre(t)) & NOT oflo(t)
              |f_stk_ptr(r1,pop,empty,push,oflo,inp,ptr_revised)(pre(t))
```

```
                                    WITH [(ptr_revised(t)) := inp(t)]||
          %-------------------------------------------------------------%
          | r1(t) OR (pop(t) & NOT empty(pre(t))) OR
             NOT push(t) OR oflo(pre(t)) OR oflo(t)
              |f_stk_ptr(r1,pop,empty,push,oflo,inp,ptr_revised)(pre(t))||
          %-------------------------------------------------------------%
          ENDTABLE
     ENDIF
    MEASURE rank(t)

unifun_empty_oflo_ptr(r1,pop,push,ni)(t):RECURSIVE[bool,bool,index_revised]=
    IF init(t) THEN (TRUE,FALSE,-1)
    ELSE LET PREV = unifun_empty_oflo_ptr(r1, pop, push, ni)(pre(t)) IN
      TABLE
       %-----------------------------------------------------------------%
       | r1(t)                                 |     (TRUE,FALSE,-1)    ||
       %-----------------------------------------------------------------%
       | NOT r1(t) & (pop(t) & NOT PREV`1) &
         (PREV`3 >= 0)                         | (TRUE,FALSE,PREV`3 - 1)||
       %-----------------------------------------------------------------%
       | NOT r1(t) & (pop(t) & NOT PREV`1) &
         (PREV`3 = -1)                         | (PREV`1,PREV`2,PREV`3) ||
       %-----------------------------------------------------------------%
       | NOT r1(t) & (NOT pop(t) OR PREV`1) & (push(t) &
         NOT PREV`2) & (PREV`3 = ni(t) - 1)   | (FALSE,TRUE,PREV`3 + 1)||
       %-----------------------------------------------------------------%
       | NOT r1(t) & (NOT pop(t) OR PREV`1) & (push(t) &
         NOT PREV`2) & (PREV`3 /= ni(t) - 1)  | (PREV`1,PREV`2,PREV`3) ||
       %-----------------------------------------------------------------%
       | NOT r1(t) & (NOT pop(t) OR PREV`1) &
         (NOT push(t) OR PREV`2)              | (PREV`1,PREV`2,PREV`3) ||
       %-----------------------------------------------------------------%
     ENDTABLE
    ENDIF
   MEASURE rank(t)

  f_ni(r1, n)(t): RECURSIVE storage_size =
    IF init(t) THEN 128 ELSIF r1(t) THEN limit(1,n(t),128)
    ELSE f_ni(r1,n)(pre(t))
    ENDIF
   MEASURE rank(t)

  f_out(r1, pop, push, empty, oflo, stk, ptr, inp, ni, out)(t): bool =
   out(t) =
    IF init(t) THEN 0
    ELSE LET PREV = out(pre(t)) IN
     TABLE
      %-----------------------------------------------------------------%
      | r1(t)                                                 |     0     ||
```

```
    %-------------------------------------------------------------------%
    | NOT r1(t) & (pop(t) & NOT empty(pre(t))) & empty(t)|     0     ||
    %-------------------------------------------------------------------%
    | NOT r1(t) & (pop(t) & NOT empty(pre(t))) &
      NOT empty(t)                                      |stk(ptr(t))||
    %-------------------------------------------------------------------%
    | NOT r1(t) & (NOT pop(t) OR empty(pre(t))) &
      (push(t) & NOT oflo(pre(t))) & (ptr(t) /= ni(t))   |  inp(t)   ||
    %-------------------------------------------------------------------%
    | NOT r1(t) & (NOT pop(t) OR empty(pre(t))) &
      (push(t) & NOT oflo(pre(t))) & (ptr(t) = ni(t))    |    0      ||
    %-------------------------------------------------------------------%
    | NOT r1(t) & (NOT pop(t) OR empty(pre(t))) &
      NOT (push(t) & NOT oflo(pre(t)))                   |   PREV    ||
    %-------------------------------------------------------------------%
    ENDTABLE
   ENDIF

ret1(ret1,empty,oflo,out)(t): bool =
 IF init(t) THEN TRUE ELSIF ret1(t) THEN
  out(t) = out(pre(t)) & empty(t) = empty(pre(t)) & oflo(t) = oflo(pre(t))
 ELSE TRUE ENDIF
 ENDIF

STACK_INT_FBD(push,pop,r1,inp,n,empty,oflo,out)(t): bool =
 EXISTS (ptr_revised,ni,stk_revised):
  IF init(t) THEN (empty(t) = True) & (oflo(t) = False) & (out(t) = 0) &
   ptr_revised(t) = -1 & ni(t) = 128 & stk_revised(ptr_revised(t)) = 0
  ELSE
    EXISTS (d1,d2,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,
    w16,w18,w19,w20:pred[tick],ret1,ret2,ret3,pop_stk,push_stk:pred[tick]):
      Z(empty,d1)(t) & neg(r1(t),w1(t)) & neg(d1(t),w2(t)) &
      conj_3(w1(t),pop(t),w2(t),pop_stk(t)) & neg(r1(t),w3(t)) &
      neg(pop_stk(t),w4(t)) & neg(push_stk(t),w5(t)) &
      conj_3(w3(t),w4(t),w5(t),ret1(t)) &
      neg(pop_stk(t),w6(t)) & neg(r1(t),w7(t)) &
      Z(oflo,d2)(t) & neg(d2(t),w8(t)) &
      conj_4(w6(t),w7(t),push(t),w8(t),push_stk(t)) &
      move_int(r1,w9,LAMBDA t:0,out)(t) &
      move_int_ptr(w9,w10,LAMBDA t:-1,ptr_revised)(t) &
      move_bool_oflo(w10,w11,LAMBDA t:False,oflo)(t) &
      move_bool_empty(w11,w12,LAMBDA t:True,empty)(t) &
      limit(w12,ret2,n,LAMBDA t:1,LAMBDA t:128,ni)(t) &
      sub_1_ptr(pop_stk,w13,ptr_revised)(t) &
      lt_empty(w13,w14,ptr_revised,LAMBDA t:0,empty)(t) &
      sel_update(w14,w15,empty,stk_revised(ptr_revised(t)),0,out)(t) &
      move_bool_oflo(w15,ret3,LAMBDA t:False,oflo)(t) &
      move_bool_empty(push_stk,w18,LAMBDA t:False,empty)(t) &
      add_1_ptr(w18,w19,ptr_revised,lambda t:ni(t)-1)(t) &
```

```
        eq_oflo(w19,ptr_revised,ni,oflo)(t) &
        move_int_en(w16,inp,lambda t:stk_revised(ptr_revised(t)))(t) &
        neg_en(w20,oflo,w16)(t) &
        sel_update_en(w20,oflo,inp(t),0,out)(t) &
        ret1(ret1,empty,oflo,out)(t)
    ENDIF

  STACK_INT_ST(push,pop,r1,inp,n,empty,oflo,out): bool =
    FORALL t:
    EXISTS (ptr_revised,ni,stk_revised):
     ptr_revised(t) = unifun_empty_oflo_ptr(r1,pop,push,ni)(t)`3 &
     ni(t) = f_ni(r1,n)(t) &
     stk_revised = f_stk_ptr(r1,pop,empty,push,oflo,inp,ptr_revised)(t) &
     empty(t) = unifun_empty_oflo_ptr(r1,pop,push,ni)(t)`1 &PREV
     oflo(t) = unifun_empty_oflo_ptr(r1,pop,push,ni)(t)`2 &
     out(t) = f_out(r1,pop,push,empty,oflo,stk_revised,ptr_revised,inp)(t)

  STACK_INT_REQ(push,pop,r1,inp,n,empty,oflo,out): bool =
    FORALL t:
    EXISTS (ptr_revised,ni,stk_revised):
     empty(t) = unifun_empty_oflo_ptr(r1,pop,push,ni)(t)`1 &
     oflo(t) = unifun_empty_oflo_ptr(r1,pop,push,ni)(t)`2 &
     ptr_revised(t) = unifun_empty_oflo_ptr(r1,pop,push,ni)(t)`3 &
     ni(t) = f_ni(r1,n)(t) &
     stk_revised = f_stk_ptr(r1,pop,empty,push,oflo,inp,ptr_revised)(t) &
     out(t) = f_out(r1,pop,push,empty,oflo,stk_revised,ptr_revised,inp)(t)

  STACK_INT_FBD_IMPLIES_ST: LEMMA
    STACK_INT_FBD(push,pop,r1,inp,n,empty,oflo,out)
       IMPLIES
          STACK_INT_ST(push,pop,r1,inp,n,empty,oflo,out)

  STACK_INT_ST_CONSISTENCY: THEOREM
    FORALL (pop,push,r1,inp,n):
      EXISTS (empty,oflo,out):
        STACK_INT_ST(push,pop,r1,inp,n,empty,oflo,out)

  STACK_INT_FBD_CONSISTENCY: THEOREM
    FORALL (pop,push,r1,inp,n):
      EXISTS (empty,oflo,out):
        STACK_INT_FBD(push,pop,r1,inp,n,empty,oflo,out)

  STACK_INT_CORRECTNESS: THEOREM
    FORALL (pop,push,r1,inp,n):
      FORALL (empty,oflo,out):
         STACK_INT_ST(push,pop,r1,inp,n,empty,oflo,out)
            IMPLIES
               STACK_INT_REQ(push,pop,r1,inp,n,empty,oflo,out)
END STACK_INT
```

Appendix F

Block *HYSTERESIS*

This appendix contains the PVS specification for block *HYSTERESIS* (Section 7.1.2.2). The *ClockTick* theory is imported by the *HYSTERESIS* theory.

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

BEGIN

  IMPORTING ClockTick[delta_t]

  t: VAR tick

  XIN1, XIN2, EPS_NO: VAR [tick -> real]

  EPS: VAR [tick -> posreal]

  Q: VAR pred[tick]

  HYSTERESIS_IMPL_ST(XIN1,XIN2,EPS,Q): bool =
   FORALL t:
    Q(t) =
      IF init(t) THEN False
      ELSIF Q(pre(t)) & XIN1(t) < (XIN2(t) - EPS(t)) THEN FALSE
      ELSIF XIN1(t) > (XIN2(t) + EPS(t)) THEN TRUE
      ELSE Q(pre(t))
      ENDIF

  f_Q(XIN1,XIN2,EPS)(t): RECURSIVE bool =
      IF init(t) THEN False
      ELSIF f_Q(XIN1,XIN2,EPS)(pre(t)) & XIN1(t) < (XIN2(t) - EPS(t)) THEN FALSE
      ELSIF XIN1(t) > (XIN2(t) + EPS(t)) THEN TRUE
      ELSE f_Q(XIN1,XIN2,EPS)(pre(t))
      ENDIF
      MEASURE rank(t)
```

```
  HYSTERESIS_REQ(XIN1,XIN2,EPS,Q): bool =
    FORALL t:
    Q(t) =
      IF init(t) THEN FALSE
      ELSE LET PREV = Q(pre(t)) IN
       TABLE
        %-------------------------------------------------------------%
        | XIN1(t) < (XIN2(t) - EPS(t))                     | FALSE ||
        %-------------------------------------------------------------%
        | ((XIN2(t) - EPS(t)) <= XIN1(t)) &
           (XIN1(t) <= (XIN2(t) + EPS(t)))                | PREV  ||
        %-------------------------------------------------------------%
        | (XIN2(t) + EPS(t)) < XIN1(t)                     | TRUE  ||
        %-------------------------------------------------------------%
       ENDTABLE
      ENDIF

  HYSTERESIS_REQ_WITHOUT_ASSUMPTION(XIN1,XIN2,EPS_NO,Q): bool =
    FORALL t:
    Q(t) =
      IF init(t) THEN FALSE
      ELSE LET PREV = Q(pre(t)) IN
       TABLE
        %-------------------------------------------------------------%
        | XIN1(t) < (XIN2(t) - EPS_NO(t))                  | FALSE ||
        %-------------------------------------------------------------%
        | ((XIN2(t) - EPS_NO(t)) <= XIN1(t)) &
           (XIN1(t) <= (XIN2(t) + EPS_NO(t)))             | PREV  ||
        %-------------------------------------------------------------%
        | (XIN2(t) + EPS_NO(t)) < XIN1(t)                  | TRUE  ||
        %-------------------------------------------------------------%
       ENDTABLE
      ENDIF

  HYSTERESIS_CONSISTENCY: THEOREM
   FORALL (XIN1,XIN2,EPS):
    EXISTS Q:
      HYSTERESIS_IMPL_ST(XIN1,XIN2,EPS,Q)

  HYSTERESIS_CORRECTNESS: THEOREM
      HYSTERESIS_IMPL_ST(XIN1,XIN2,EPS,Q)
           IMPLIES
                HYSTERESIS_REQ(XIN1,XIN2,EPS,Q)

END HYSTERESIS
```

Appendix G

Block *LIMITS_ALARM*

This appendix contains the PVS specification for block *LIMITS_ALARM* (Section 7.1.2.3). It imports three theories, *ClockTick*, *defined_operators* and *HYSTERESIS*.

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

BEGIN

  IMPORTING ClockTick[delta_t]

  IMPORTING defined_operators[delta_t]

  IMPORTING HYSTERESIS[delta_t]

  timed_real: TYPE = [tick -> real]

  timed_posreal: TYPE = [tick -> posreal]

  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

  X, L: VAR timed_real

  H: VAR dependent_high_limit_type

  EPS: VAR timed_posreal

  QH, QL, Q: VAR pred[tick]

  w1: VAR [tick -> posreal]
```

```
w2, w3: VAR [tick -> real]

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

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)
```

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

  LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL): bool =
    EXISTS (w1,w2,w3):
      DIV(EPS,LAMBDA (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)

  OUTPUT_QH_CORRECTNESS_CHECKING: LEMMA
    LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL) IMPLIES P_QH(X,H,L,EPS,QH)

  OUTPUT_QL_CORRECTNESS_CHECKING: LEMMA
    LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL) IMPLIES P_QL(X,L,EPS,QL)

  OUTPUT_Q_CORRECTNESS_CHECKING: LEMMA
    LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL) IMPLIES P_Q(QH,QL,Q)

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

  LIMITS_ALARM_CORRECTNESS: LEMMA
    LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL)
      IMPLIES
          LIMITS_ALARM_REQ(X,H,L,EPS,QH,Q,QL)

  PROPERTY0: THEOREM
    LIMITS_ALARM_IMPL(X,H,L,EPS,QH,Q,QL)
      IMPLIES
       FORALL (t: tick):  NOT (QH(t) & QL(t))

END LIMITS_ALARM
```

Appendix H

The *Trip Sealed-In* Subsystem

This appendix contains the PVS specification for the *Trip Sealed-In* subsystem. It consists of seven parts, *abstractions*, *comlibrary*, *srslibrary*, *sddlibrary*, *srsfunctions*, *sddfunctions*, and *obligations*.

Common Library (*comlibrary*)

Common library includes common imports used by both the SRS and the SDD. It consists of seven theories: *Time* theory (Appendix A), *ClockTick* theory (Appendix B), *SampleInstance* theory (Appendix B in (Hu, 2008)), *FeasibilityResults* theory (Appendix C in (Hu, 2008)), *SampleInstanceOnTick* theory (Appendix E in (Hu, 2008)), *Held_For* theory (Appendix F in (Hu, 2008)), and *TimerGeneral* (Appendix G in (Hu, 2008)) theory.

SRS Library (*srslibrary*)

SRS library consists of the formalizations for SRS common functions, SRS constants, and SRS types.

```
SRSTypes: THEORY

BEGIN

  t_real: TYPE = real

  t_integer: TYPE = int

  y_trip: TYPE = {e_Trip, e_NotTrip}

  y_pb: TYPE = {e_NotPressed, e_Pressed}
```

```
  y_pbdesign: TYPE = {e_pbDebounced, e_pbStuck, e_pbNotDebounced}

END SRSTypes


SRSCommonFunctions: THEORY

BEGIN

  IMPORTING SRSTypes

  trunc(r: real) : int =  IF 0 <= r THEN floor(r) ELSE ceiling(r) ENDIF

END SRSCommonFunctions


RSUBRANGE: THEORY

BEGIN

rsubrange(x, y: real): TYPE = {z:real | (x <= z) & (z <= y)}

END RSUBRANGE


SRSConstants: THEORY

BEGIN

  IMPORTING SRSTypes

  IMPORTING comlibrary@Time

  k_Sealindelay_req : non_initial_time = 150

  k_Sealindelay_deltaL_req : non_initial_time = 25

  k_Sealindelay_deltaR_req : non_initial_time = 50

END SRSConstants
```

SRS Functions (*srsfunctions*)

Theory *Channel_trip_sealedin_req* formalizes the black-box requirements in the SRS. It imports: the *Time* theory (Appendix A); SRS types and constants from *srslibrary*; and *Held_For* and *SampleInstanceOnTick* theories from *comlibrary*.

```
Channel_trip_sealedin_req
       [(IMPORTING comlibrary@Time)
         K: non_initial_time,
         TL, TR: {t: time | t < K},
         delta_t: {tk: non_initial_time | tk < K - TL AND tk < TR + TL},
         (IMPORTING srslibrary@SRSConstants)
         delta_L, delta_R: {t: time | t < k_Sealindelay_req}]: THEORY

BEGIN

  IMPORTING comlibrary@SampleInstanceOnTick[K, TL, TR, delta_t]

  IMPORTING comlibrary@Held_For[K, TL, TR, delta_t, delta_L, delta_R]

  IMPORTING srslibrary@SRSTypes

  IMPORTING srslibrary@SRSConstants

  timed_trip: TYPE = [tick -> y_trip]

  Channel_trip_sealedin_REQ_f
          (Any_parameter_tripped: pred[tick],
           c_ChanTrip: timed_trip,
           Manual_reset_request: pred[tick])(t: tick): RECURSIVE bool =
      IF init(t) THEN TRUE
      ELSE LET ChanTrip_status = LAMBDA (t: tick): c_ChanTrip(t) = e_Trip,
               PREV = Channel_trip_sealedin_REQ_f
                                (Any_parameter_tripped,
                                 c_ChanTrip,
                                 Manual_reset_request)(pre(t)) IN
       TABLE
       %-------------------------------------------------------------%
       |Any_parameter_tripped(t) &                 %               %
        Held_For_I(ChanTrip_status,                 %               %
                   k_Sealindelay_req - delta_L,     %               %
                   Sample_t)(t)                     |    TRUE       ||
       %-------------------------------------------------------------%
       |Any_parameter_tripped(t) &                 %               %
        NOT (Held_For_I(ChanTrip_status,            %               %
                        k_Sealindelay_req - delta_L,%               %
                        Sample_t)(t))               |     PREV      ||
       %-------------------------------------------------------------%
       |NOT Any_parameter_tripped(t) &             %               %
            Manual_reset_request(t)                 |    FALSE      ||
       %-------------------------------------------------------------%
       |NOT Any_parameter_tripped(t) &             %               %
        NOT Manual_reset_request(t)                 |     PREV      ||
       %-------------------------------------------------------------%
       ENDTABLE
```

```
    ENDIF
  MEASURE rank(t)

 Channel_trip_sealedin_REQ_P(Any_parameter_tripped: pred[tick],
                             c_ChanTrip: timed_trip,
                             Manual_reset_request: pred[tick],
                             Channel_trip_sealedin: pred[tick]): bool =
   FORALL (t: tick):
     Channel_trip_sealedin(t) =
          Channel_trip_sealedin_REQ_f(Any_parameter_tripped,
                                      c_ChanTrip,
                                      Manual_reset_request)(t)

END Channel_trip_sealedin_req
```

SDD Library (*sddlibrary*)

SDD library consists of the formalizations of SDD core functions (i.e., theory *Defined_operators*), SDD constants, SDD types, formalization for component FBs (*TON* timer and FB *RS*).

```
SDDTypes : THEORY

BEGIN

  REAL : TYPE = real

  DINT : TYPE = integer

  BOOL : TYPE = bool

END SDDTypes


SDDConstants : THEORY

BEGIN

  IMPORTING SDDTypes

  IMPORTING comlibrary@Time

  k_Sealindelay_impl : non_initial_time = 150

END SDDConstants
```

```
RS_Latch[(IMPORTING comlibrary@Time) delta_t: non_initial_time]: THEORY

BEGIN

  IMPORTING comlibrary@ClockTick[delta_t]

  f_RS_init_F(r1, s: pred[tick])(t: tick): RECURSIVE bool =
       IF init(t) THEN FALSE
       ELSE
         LET PREV = f_RS_init_F(r1, s)(pre(t)) IN
         TABLE
          %-----------------------------------%
          | r1(t)                    | FALSE ||
          %-----------------------------------%
          | NOT (r1(t)) & s(t)       | TRUE  ||
          %-----------------------------------%
          | NOT (r1(t)) & NOT (s(t)) | PREV  ||
          %-----------------------------------%
         ENDTABLE
       ENDIF
     MEASURE rank(t)


  RS_init_F(r1, s, q: pred[tick]): bool =
   FORALL (t: tick):
    q(t) =
       IF init(t) THEN FALSE
       ELSE
         LET PREV = q(pre(t)) IN
         TABLE
          %-----------------------------------%
          | r1(t)                    | FALSE ||
          %-----------------------------------%
          | NOT (r1(t)) & s(t)       | TRUE  ||
          %-----------------------------------%
          | NOT (r1(t)) & NOT (s(t)) | PREV  ||
          %-----------------------------------%
         ENDTABLE
       ENDIF

end RS_Latch



ton[(IMPORTING comlibrary@Time)
     K: non_initial_time, TL,TR: {t: time | t < K},
     delta_t: {tk: non_initial_time | tk < K - TL AND tk < TR + TL},
     delta_L, delta_R: time]: THEORY

BEGIN
```

```
  IMPORTING comlibrary@TimerGeneral[K, TL, TR, delta_t, delta_L, delta_R]

  q(i: pred[tick], pt: non_initial_time)(t: tick ): bool =
    TABLE
     %-----------------------------------+-------++
     | Timer_I(i, Sample_t, pt)(t) >= pt  | TRUE  ||
     %-----------------------------------+-------++
     | Timer_I(i, Sample_t, pt)(t) <  pt  | FALSE ||
     %-----------------------------------+-------++
    ENDTABLE

  et(i: pred[tick], pt: non_initial_time)(t: tick): tick =
    Timer_I(i, Sample_t, pt)(t)

  TON(i: pred[tick], pt: non_initial_time,
      q: pred[tick], et: [tick -> tick]): bool =
    FORALL(t: tick): q(t) = q(i, pt)(t) & et(t) = et(i, pt)(t)

END ton
```

SDD Functions (*sddfunctions*)

Theory *Sealin_impl* formalizes the FBD implementation in SDD. It imports: the *Time* theory (Appendix A); SDD core functions, constants, *ton* theory and *RS_Latch* theory from *sddlibrary*.

```
Sealin_impl[(IMPORTING comlibrary@Time)
             K: non_initial_time,
             TL, TR: {t: time | t < K},
             delta_t: {tk: non_initial_time | tk < K - TL AND tk < TR + TL},
             (IMPORTING sddlibrary@SDDConstants)
             delta_L, delta_R: {t: time | t < k_Sealindelay_impl}]: THEORY

BEGIN

  IMPORTING sddlibrary@SDDCore

  IMPORTING sddlibrary@SDDConstants

  IMPORTING sddlibrary@ton[K, TL, TR, delta_t, delta_L, delta_R]

  IMPORTING sddlibrary@RS_Latch[delta_t]

  Channel_trip_sealedin_IMPL_P_original
                      (Any_parm_trip: pred[tick],
                       ChanTrip: pred[tick],
```

```
                    Man_Reset_Req: pred[tick],
                    Chan_Trip_Sealedin: pred[tick]): bool =
  EXISTS(w1, w2, w3, w4, w5, w6: pred[tick],
         et_sealin: [tick -> tick]):
     NEG(Any_parm_trip, w5)                                        &
     w6 = NEG_f(ChanTrip)                                          &
     TON(w6, k_Sealindelay_impl - delta_L, w1, et_sealin)          &
     CONJ(Any_parm_trip, w1, w2)                                   &
     DISJ_PRE(Chan_Trip_Sealedin, w2, w3)                          &
     CONJ(w5, Man_Reset_Req, w4)                                   &
     RS_init_F(w4, w3, Chan_Trip_Sealedin)

Channel_trip_sealedin_IMPL_P_revised
                   (Any_parm_trip: pred[tick],
                    ChanTrip: pred[tick],
                    Man_Reset_Req: pred[tick],
                    Chan_Trip_Sealedin: pred[tick]): bool =
  EXISTS(w1, w2, w3, w4, w5, w6, w7: pred[tick],
         et_sealin: [tick -> tick]):
     NEG(Any_parm_trip, w5)                                        &
     w6 = NEG_f(ChanTrip)                                          &
     TON(w6, k_Sealindelay_impl - delta_L, w1, et_sealin)          &
     CONJ(Any_parm_trip, w1, w2)                                   &
     DISJ_PRE(Chan_Trip_Sealedin, w2, w3)                          &
     CONJ(w5, Man_Reset_Req, w4)                                   &
     RS_init_F(w4, w3, w7)                                         &
     SEL(LAMBDA (t: tick): init(t), w7,
         LAMBDA (t: tick): TRUE, Chan_Trip_Sealedin)

Sealin_IMPL_f_Chan_Trip_Sealedin_original
                   (Any_parm_trip: pred[tick],
                    ChanTrip: pred[tick],
                    Man_Reset_Req: pred[tick])(t: tick): RECURSIVE bool =
   IF init(t) THEN FALSE
   ELSE
      (Sealin_IMPL_f_Chan_Trip_Sealedin_original
                   (Any_parm_trip,
                    ChanTrip,
                    Man_Reset_Req)(pre(t)) OR
      (Sealin_IMPL_f_Chan_Trip_Sealedin_original
                   (Any_parm_trip,
                    ChanTrip,
                    Man_Reset_Req)(pre(t)) OR
        CONJ_2_f(Any_parm_trip,
                 q(NEG_f(ChanTrip),
                 k_Sealindelay_impl - delta_L))(t))) &
        NEG_f(CONJ_2_f(NEG_f(Any_parm_trip), Man_Reset_Req))(t)
   ENDIF
  MEASURE rank(t)
```

```
  Sealin_IMPL_f_Chan_Trip_Sealedin_revised
                    (Any_parm_trip: pred[tick],
                     ChanTrip: pred[tick],
                     Man_Reset_Req: pred[tick])(t: tick): RECURSIVE bool =
    IF init(t) THEN TRUE
    ELSE
       (Sealin_IMPL_f_Chan_Trip_Sealedin_revised
                    (Any_parm_trip,
                     ChanTrip,
                     Man_Reset_Req)(pre(t)) OR
       (Sealin_IMPL_f_Chan_Trip_Sealedin_revised(Any_parm_trip,
                     ChanTrip,
                     Man_Reset_Req)(pre(t)) OR
         CONJ_2_f(Any_parm_trip,
                  q(NEG_f(ChanTrip),
                   k_Sealindelay_impl - delta_L))(t))) &
         NEG_f(CONJ_2_f(NEG_f(Any_parm_trip), Man_Reset_Req))(t)
    ENDIF
   MEASURE rank(t)

 Sealin_IMPL_f_w7(Any_parm_trip: pred[tick],
                  ChanTrip: pred[tick],
                  Man_Reset_Req: pred[tick])(t: tick): RECURSIVE bool =
    IF init(t) THEN FALSE
    ELSE
       (Sealin_IMPL_f_Chan_Trip_Sealedin_revised
                    (Any_parm_trip,
                     ChanTrip,
                     Man_Reset_Req)(pre(t)) OR
       (Sealin_IMPL_f_Chan_Trip_Sealedin_revised
                    (Any_parm_trip,
                     ChanTrip,
                     Man_Reset_Req)(pre(t)) OR
         CONJ_2_f(Any_parm_trip,
                  q(NEG_f(ChanTrip),
                   k_Sealindelay_impl - delta_L))(t))) &
         NEG_f(CONJ_2_f(NEG_f(Any_parm_trip), Man_Reset_Req))(t)
    ENDIF
  MEASURE rank(t)

END Sealin_impl
```

Abstraction Functions (*abstractions*)

Theory *AbstractionFunctions* formalizes the abstraction functions between the SRS and the SDD. It consists of: the *ClockTick* (Appendix B) theory, SRS

types, constants, and common functions from *srslibrary*, and SDD types from *sddlibrary*.

```
AbstractionFunctions[(IMPORTING comlibrary@Time) delta_t:posreal]: THEORY

BEGIN

  IMPORTING comlibrary@ClockTick[delta_t]

  IMPORTING srslibrary@SRSTypes

  IMPORTING sddlibrary@SDDTypes

  IMPORTING srslibrary@SRSConstants

  IMPORTING srslibrary@SRSCommonFunctions

  AbstDINT(x: int): DINT = x

  AbstRealToDINT(x: real) : DINT = AbstDINT(trunc(x))

  AbstTrip(x: y_trip): bool =
     COND ( x = e_Trip ) -> TRUE, ( x = e_NotTrip ) -> FALSE ENDCOND

  AbstParmTrip(x: y_trip): bool =
     COND (x = e_Trip) -> FALSE, (x = e_NotTrip) -> TRUE ENDCOND

  AbstParmTrip_timed(x : [tick -> y_trip]) : pred[tick] =
   LAMBDA (t: tick):
       COND (x(t) = e_Trip) -> FALSE, x(t) = e_NotTrip) -> TRUE ENDCOND

END AbstractionFunctions
```

Proof Obligations (*obligations*)

Theory *Channel_trip_sealedin_obl* formalizes the auxiliary lemmas and proof obligations. It imports theory *Time* (Appendix A), the requirement theory *Trip_sealedin_req* from *srsfunctions*, the FBD implementation theory *Sealin_impl* from *sddfunctions*, the abstraction functions theory *AbstractionFunctions* from *abstractions*, and SDD constants theory *SDDConstants* from *sddlibrary*.

```
Channel_trip_sealedin_obl: THEORY

BEGIN

  IMPORTING comlibrary@Time
```

```
K: non_initial_time = 35

TL: {t: time | t < K & t < 4 & t > 1}

TR: {t: time | t < K & t < 10 & t > 1}

delta_t: {tk: non_initial_time | tk < K - TL & tk < TR + TL}

delta_L, delta_R: {t: time | t < 50}

IMPORTING srsfunctions@Trip_sealedin_req[K,TL,TR,delta_t,delta_L,delta_R]

IMPORTING sddfunctions@Sealin_impl[K,TL,TR,delta_t,delta_L,delta_R]

IMPORTING abstractions@AbstractionFunctions

IMPORTING sddlibrary@SDDConstants

PROPERTY0: LEMMA
  FORALL (c_ChanTrip: timed_trip):
   (LAMBDA (t: tick[delta_t]):
     NOT COND (c_ChanTrip(t) = e_Trip) -> FALSE, ELSE -> TRUE ENDCOND) =
       (LAMBDA (t: tick[delta_t]): c_ChanTrip(t) = e_Trip)

PROPERTY1: LEMMA
  FORALL (A, B: bool): (A <=> B) => (NOT A <=> NOT B)

Channel_trip_sealedin_consistency_check_revised: THEOREM
  FORALL(Any_parameter_trip: pred[tick],
         c_ChanTrip: timed_trip,
         Manual_reset_request: pred[tick]):
    EXISTS(Channel_trip_sealedin: pred[tick]):
     Channel_trip_sealedin_IMPL_P_revised
                (Any_parameter_trip,
                 AbstParmTrip_timed(c_ChanTrip),
                 Manual_reset_request,
                 Channel_trip_sealedin)

Channel_trip_sealedin_correctness_check_original: THEOREM
  FORALL(Any_parameter_trip: pred[tick],
         c_ChanTrip: timed_trip,
         Manual_reset_request: pred[tick],
         Channel_trip_sealedin: pred[tick]):
     Channel_trip_sealedin_IMPL_P_original
                (Any_parameter_trip,
                 AbstParmTrip_timed(c_ChanTrip),
                 Manual_reset_request,
                 Channel_trip_sealedin)
```

```
            =>
          Channel_trip_sealedin_REQ_P(Any_parameter_trip,
                                      c_ChanTrip,
                                      Manual_reset_request,
                                      Channel_trip_sealedin)

  Channel_trip_sealedin_correctness_check_revised: THEOREM
    FORALL(Any_parameter_trip: pred[tick],
           c_ChanTrip: timed_trip,
           Manual_reset_request: pred[tick],
           Channel_trip_sealedin: pred[tick]):
       Channel_trip_sealedin_IMPL_P_revised(Any_parameter_trip,
                                            AbstParmTrip_timed(c_ChanTrip),
                                            Manual_reset_request,
                                            Channel_trip_sealedin)
            =>
          Channel_trip_sealedin_REQ_P(Any_parameter_trip,
                                      c_ChanTrip,
                                      Manual_reset_request,
                                      Channel_trip_sealedin)

END Channel_trip_sealedin_obl
```

Bibliography

Bakhmach, E., O.Siora, Tokarev, V., Reshetytskyi, S., Kharchenko, V., and Bezsalyi, V. (2009). FPGA - based technology and systems for I&C of existing and advanced reactors. International Atomic Energy Agency, page 173. IAEA-CN-164-7S04.

Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., and Stursberg, O. (2004). Verification of PLC programs given as sequential function charts. In Integration of Software Specification Techniques for Applications in Engineering, volume 3147 of LNCS, pages 517–540. Springer Berlin Heidelberg.

Blech, J. O. and Biha, S. O. (2013). On formal reasoning on the semantics of PLC using Coq. Computing Research Repository, abs/1301.3047.

Camilleri, A., Gordon, M., and Melham, T. (1987). Hardware verification using higher-order logic. In Borrione, D., editor, From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference on From HDL Descriptions to Guaranteed Correct Circuit Designs, pages 43–67. North-Holland.

Canet, G., Couffin, S., Lesage, J. J., Petit, A., and Schnoebelen, P. (2000). Towards the automatic verification of PLC programs written in instruction list. In IEEE International Conference on Systems, Man and Cybernetics, pages 2449–2454.

Gordon, M. J. (1985). Why higher-order logic is a good formalism for specifying and verifying hardware. University of Cambridge, Computer Laboratory.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580.

Hu, X. (2008). Proving implementability of timing properties with tolerance. PhD thesis, McMaster University, Department of Computing and Software.

Hu, X., Lawford, M., and Wassyng, A. (2009). Formal verification of the implementability of timing requirements. In Formal Methods for Industrial Critical Systems, volume 5596 of LNCS, pages 119–134. Springer.

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

IEC (2013). 61131-3 Ed. 3.0 en:2013: Programmable Controllers — Part 3: Programming Languages. International Electrotechnical Commission.

IEEE (2010). 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).

Janicki, R., Parnas, D. L., and Zucker, J. (1997). Tabular representations in relational documents. In Relational Methods in Computer Science, pages 184–196. Springer Verlag.

Janicki, R. and Wassyng, A. (2005). Tabular expressions and their relational semantics. Fundamenta Informaticae, 67(4):343–370.

Jimenez-Fraustro, F. and Rutten, E. (2001). A synchronous model of IEC 61131 PLC languages in SIGNAL. In Euromicro Conference On Real-Time Systems, pages 135–142.

Jin, Y. and Parnas, D. L. (2010). Defining the meaning of tabular mathematical expressions. Science of Computer Programming, 75(11):980 – 1000.

John, K. H. and Tiegelkamp, M. (2010). IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids. Springer, 2nd edition.

Kabra, A., Bhattacharjee, A., Karmakar, G., and Wakankar, A. (2012). Formalization of sequential function chart as synchronous model in Lustre. In Emerging Trends and Applications in Computer Science, pages 115–120.

Lawford, M., McDougall, J., Froebel, P., and Moum, G. (2000). Practical application of functional and relational methods for the specification and verification of safety critical software. In Algebraic Methodology and Software Technology, volume 1816 of LNCS, pages 73–88. Springer.

Lawford, M. and Wonham, W. (1995). Equivalence preserving transformations of timed transition models. IEEE Transactions on Automatic Control, 40:1167–1179.

Lawford, M. and Wu, H. (2000). Verification of real-time control software using PVS. In Ramadge, P. and Verdu, S., editors, Information Sciences and Systems, volume 2, pages TP1–13–TP1–17, Princeton, NJ. Dept. of Electrical Engineering, Princeton University.

Liu, Z., Parnas, D., and Widemann, B. (2010). Documenting and verifying systems assembled from components. Frontiers of Computer Science in China, 4(2):151–161.

Mader, A. and Wupper, H. (1999). Timed automaton models for simple programmable logic controllers. In Euromicro Conference On Real-Time Systems, pages 114–122. IEEE.

Melham, T. (1987). Abstraction mechanisms for hardware verification. In VLSI Specification, Verification and Synthesis, pages 129–157. Kluwer Academic Publishers.

Melham, T. F. (1989). Formalizing abstraction mechanisms for hardware verification in higher order logic. PhD thesis, University of Cambridge.

NASA Langley PVS Libraries Official Website (2014). http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html.

Németh, E. and Bartha, T. (2009). Formal verification of safety functions by reinterpretation of functional block based specifications. In Cofer, D. and Fantechi, A., editors, Formal Methods for Industrial Critical Systems, volume 5596 of LNCS, pages 199–214. Springer Berlin / Heidelberg.

Owre, S., Rushby, J. M., and Shankar, N. (1992). PVS: A prototype verification system. In 11th International Conference on Automated Deduction (CADE), volume 607 of LNCS, pages 748–752.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. (1999). PVS language reference. Computer Science Laboratory, SRI International, Menlo Park, CA.

Pang, L., Wang, C.-W., Lawford, M., and Wassyng, A. (2013a). Formalizing and verifying function blocks using tabular expressions and PVS. In 4th International Workshop on Formal Techniques for Safety-Critical Systems, volume 419 of Communications in Computer and Information Science, pages 163–178. Springer.

Pang, L., Wang, C.-W., Lawford, M., and Wassyng, A. (2013b). Formalizing and verifying function blocks using tabular expressions and PVS. Technical Report 11, McSCert.

Pang, L., Wang, C.-W., Lawford, M., and Wassyng, A. (2014a). Formal verification of IEC 61131-3 function blocks using tabular expressions. Science of Computer Programming. Invited submission for a special issue (ID: SCICO-D-14-00102). Under minor revision.

Pang, L., Wang, C.-W., Lawford, M., and Wassyng, A. (2014b). Formalizing and verifying function blocks using tabular expressions and PVS. Technical Report 16, McSCert.

Pang, L., Wang, C.-W., Lawford, M., Wassyng, A., Newell, J., Chow, V., and Tremaine, D. (2015). Formal verification of real-time function blocks using PVS. In Proceedings 4th International Workshop on Engineering Safety and Security Systems, ESSS 2015, Oslo, Norway, June 22, 2015., pages 65–79.

Parnas, D. L. (1983). A generalized control structure and its formal definition. Communications of the ACM, 26:572–581.

Parnas, D. L. and Madey, J. (1995). Functional documents for computer systems. Science of Computer Programming, 25(1):41–61.

Parnas, D. L., Madey, J., and Iglewski, M. (1994). Precise documentation of well-structured programs. IEEE Transactions on Software Engineering, 20:948–976.

Rossi, O. and Schnoebelen, P. (2000). 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), pages 177–182, Dortmund,Germany.

Roussel, J.-M. and Faure, J. (2002). An algebraic approach for PLC programs verification. In 6th International Workshop on Discrete Event Systems, pages 303–308.

Rushby, J. (1992). Formal methods for dependable real-time systems. In International Symposium on Real-Time Embedded Processing for Space Applications, pages 355–366, Les Saintes-Maries-de-la-Mer, France. CNES, the French Space Agency, Cépaduès-éditions, Toulouse, France.

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

Soliman, D., Thramboulidis, K., and Frey, G. (2012). Transformation of function block diagrams to Uppaal timed automata for the verification of safety applications. Annual Reviews in Control.

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

Special Committee 205 of RTCA (2011b). DO-333: Formal Methods Supplement to DO-178C and DO-278A.

Völker, N. and Krämer, B. J. (2002). Automated verification of function block-based industrial control systems. Science of Computer Programming, 42(1):101 – 113.

Wassyng, A. and Janicki, R. (2003). Tabular expressions in software engineering. In International Conference on Software & Systems Engineering and their Applications, volume 4, pages 1–46, Paris, France.

Wassyng, A. and Lawford, M. (2003). Lessons learned from a successful implementation of formal methods in an industrial project. In Araki, K., Gnesi, S., and Mandrioli, D., editors, International Symposium of Formal Methods Europe Proceedings(FME), volume 2805 of LNCS, pages 133–153. Springer-Verlag.

Wassyng, A., Lawford, M., and Maibaum, T. S. E. (2011). Software certification experience in the canadian nuclear industry: Lessons for the future. In International Conference on Embedded Software (EMSOFT), pages 219–226.

Woodcock, J. and Banach, R. (2007). The verification grand challenge. Journal of Universal Computer Science, 13(5):661–668.