J Autom Reasoning
DOI 10.1007/s10817-017-9415-7

Translation of IEC 61131-3 Function Block Diagrams to PVS for Formal Verification with Real-Time Nuclear Application

Josh Newell[1] · **Linna Pang**[1] · **David Tremaine**[1] · **Alan Wassyng**[2] · **Mark Lawford**[2]

Received: 5 December 2016 / Accepted: 16 May 2017

Abstract The trip computers for the two reactor shutdown systems of the Ontario Power Generation (OPG) Darlington Nuclear Power Generating Station are being refurbished due to hardware obsolescence. For one of the systems, the general purpose computer originally used is being replaced by a programmable logic controller (PLC). The trip computer application software has been rewritten using function block diagrams (FBDs), a commonly used PLC programming language defined in the IEC 61131-3 standard. The replacement project's quality assurance program requires that formal verification be performed to compare the FBDs against a formal software requirements specification written using tabular expressions (TEs). The PVS theorem proving tool is used in formal verification. Custom tools developed for OPG are used to translate TEs and FBDs into PVS code. In this paper, we present a method to rigorously translate the graphical FBD language to a mathematical model in PVS using an abstract syntax to represent the FBD constructs. We use an example from the replacement project to demonstrate the use of the model to translate a FBD module into a PVS specification. We then extend that example to demonstrate the method's applicability to a Simulink-based design.

✉ Josh Newell
newelljoeng@gmail.com

Linna Pang
lpang@swi.com

David Tremaine
tremaine@swi.com

Alan Wassyng
wassyng@mcmaster.ca

Mark Lawford
lawford@mcmaster.ca

[1] Systemware Innovation Corporation, Suite 1800, 2300 Yonge Street, Toronto, ON M4P 1E4, Canada

[2] McMaster Centre for Software Certification, McMaster University, 1280 Main Street West, Hamilton, ON L8S 4K1, Canada

Published online: 30 May 2017

Keywords Safety critical systems · IEC 61131-3 · Function block diagrams · Formal specification · PVS · Tabular expressions · Simulink

1 Introduction

Many industrial, safety-critical control systems leverage programmable technologies for their flexibility and scalability.[1] The use of programmable technologies for safety-critical design is now commonplace in nuclear, aerospace and automotive applications, and formal methods can play an important role in ensuring that those applications are safe. In the aviation domain, DO-178C [3] advocates the use of formal methods to create mathematical models for the specification and analysis of system behaviour. In the nuclear industry, IEEE 7-4.3.2 [5] lists acceptance criteria for mission- or safety-critical systems that practitioners need to comply with. In the context of formal methods, two important criteria are: (1) the software requirements are both precise and complete; and (2) the software implementation is correct with respect to specified behaviour. In the Canadian nuclear industry, CE-1001-STD [8] governs the software engineering of safety critical applications. It prescribes not only the formal specification of requirements and design, but also the formal proof of correctness of implementation against requirements. Traditionally, CE-1001-STD has been applied to general purpose computer languages. It is now being applied to the application-oriented language paradigm of programmable logic controllers (PLCs). PLCs provide a higher level of abstraction for the programmer via a set of built-in hierarchical function blocks (FBs) that are safety certified for use in critical applications.

The Ontario Power Generation (OPG) Darlington Nuclear Generating Station (DNGS) in Ontario, Canada uses two diverse, computerised special safety systems for emergency shutdown of the reactor. These are referred to as Shutdown System One and Two (i.e., SDS1 and SDS2). They were completed in the early 1990s and are based on an arrangement of real-time general purpose computers. Each SDS has three redundant trip computers (TCs) in a 2-out-of-3 voting configuration. The TCs are categorized as safety critical and were engineered in compliance with CE-1001-STD, which defines a comprehensive set of development, verification and validation processes. Formal requirements and a design specification were developed and documented using tabular expressions (TEs) [17]. Code was implemented from the TE-based design using general purpose computer languages (diverse between SDS1 and SDS2). In addition to various review and overlapping testing processes, formal proof of correctness of design and code was performed using a theorem prover, Prototype Verification System PVS [12].

Currently, SDS1 and SDS2 are being refurbished to extend the nuclear plant's life and both hardware platforms are being replaced. A safety-certified PLC compliant with IEC 61131-3 [4] was selected for the SDS1 TC replacement. As with the original project, the software requirements are specified using TEs, but the software design is now specified in a function block diagram (FBD) language using built-in IEC 61131-3 FBs provided by a PLC vendor.[2,3]

[1] This paper is an extension of [11] and contains a more extensive example that includes timer functions, a strategy for discharging consistency proofs in PVS, and an application of the methodology, using the same example, with Simulink.

[2] A small portion of the software design is written using structured text (ST), but that is not relevant to the subject of this paper.

[3] The use of IEC 61131-3 compliant built-in FBs was based on a formal specification and subsequent verification of their behaviour; one of many PLC qualification activities.

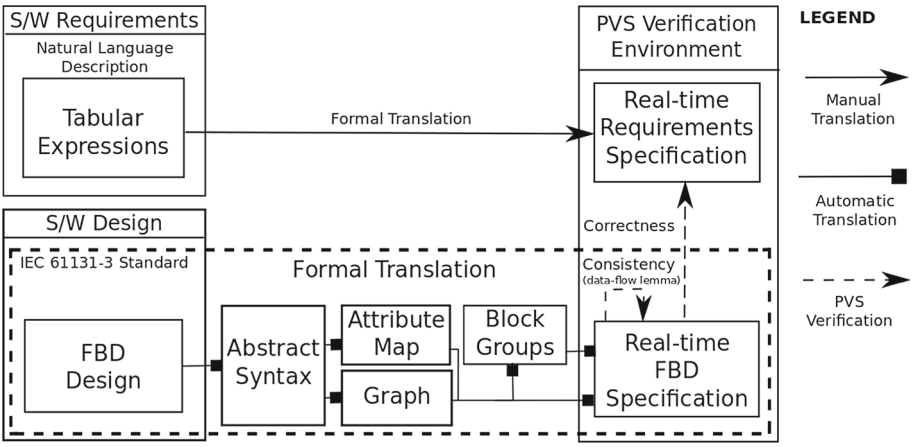

Fig. 1 Framework Diagram

Using the PLC platform, the detailed design automatically generates executable code. PVS is used to formally verify the FBD-based design against the TE-based requirements.

PVS provides an integrated environment with mechanized support for the syntax and semantics of TEs and (higher-order) predicates. Based on [13], an approach was developed for the replacement project to support the formal verification of FBDs. The process is as follows: (1) the translation of real-time requirements, described in TEs, to PVS is accomplished using tools developed and qualified for OPG; (2) the TC design, described in a collection of FBDs, is translated into PVS; and (3) formal proofs for systematic design verification are automated using PVS.

Step (2) of the process is the subject of this paper and is based on our experience with the replacement project. An abstract syntax is created to represent the constructs of a FBD and rigorous translation rules are defined for the general translation of FBDs into PVS specifications.

Figure 1 summarizes the overall verification process and contributions. As shown on the left, the real-time requirements are documented using tabular expressions. The design is written in a FBD language that is compliant with IEC 61131-3. In the center of the diagram, we highlight our main contributions within a dashed rectangle. We define an abstract syntax for FBDs using a FBD design as input. With values from the abstract syntax as input, we define an attribute map and labelled directed graph to represent relationships in the FBD. Given an attribute map and graph, we define an additional data structure, block groups, to reduce the complexity of PVS translation. Shown on the right side of Fig. 1, the FBD specification is produced from our methodology whereas the requirements are separately specified in PVS. Based on [13], our methodology also produces the consistency theorems[4] for FBDs, which are verified in PVS. Sect. 6 discusses a strategy, using a data-flow lemma, to reduce the verification effort needed to discharge consistency theorems for a FBD implementation. The correctness theorems are manually specified and verified in PVS. The strategy used to discharge a correctness proof is discussed in Sect. 7. An example of the method's use and effectiveness is provided from the SDS1 TC project. The example consists of a FBD design

[4] A FBD design is *consistent* if for every input it produces an expected output. Otherwise, a FBD design trivially satisfies any requirement.

Condition		*Result* c_PressureTrip
¬ f_PrsCond	m_Pressure ≥ k_PrsSP	True
	k_PrsSP - k_DeadBand < m_Pressure < k_PrsSP	No Change
	m_Pressure ≤ k_PrsSP - k_DeadBand	False
f_PrsCond		False

assume: $0 < k\_DeadBand << k\_PrsSP$

Fig. 2 Tabular Expression of *c_PressureTrip*

containing both logical and temporal functions. This example is then repeated using Simulink in place of FBDs to demonstrate wider applicability of the method.

2 Preliminaries

2.1 Tabular Expressions

Tabular expressions [17] (a.k.a., function tables) are a proven and effective approach for describing conditionals and relations, and thus are ideal for documenting many system requirements. They are arguably easier to comprehend and to maintain than conventional mathematical expressions. Formal semantics for tabular expressions have been well-developed in [7] and are useful for inspections, testing and verification [20,21]. Tabular expressions (TEs) partition the input domain into condition rows in the left column(s), while rows in the right column(s), inside double borders, denote the corresponding output results. Horizontal tabular expressions (HCTs), as one of the table types, were used on the original SDS1 project and continue to be used on the replacement project for specifying software requirements. We may interpret the tabular structure as a list of "if-then-else" statements. Each row defines the input circumstances (being logically conjuncted) under which the output is bound to a particular result value. The completeness and disjointness of TEs can be reasoned about in PVS. As an example, in Figs. 2, 3 and 4 we consider the *c_PressureTrip* requirement, which will be used as a running example. It involves a trip based on high pressure, and trip conditioning based on reactor power and operator input from two pushbuttons. The function *c_PressureTrip* determines the pressure trip value using the trip conditioning status (*f_PrsCond*) and the process variable *m_Presssure* compared against the setpoint value *k_PrsSP*.[5] We present the detailed discussion in Sect. 7.1.

2.2 IEC 61131-3 FBDs

To unify the syntax and semantics of PLC programming languages, the International Electrotechnical Committee (IEC) first published IEC 61131-3 in 1993; its latest version was published in 2013 [4]. The FBD is a graphical language used to describe functions between input and output variables by interconnecting built-in and custom FBs. The DNGS SDS1 trip computer uses built-in IEC 61131-3 FBs as the foundation of the formal software design. The methodology outlined in [13,14], used as a basis for this paper, provides an approach for formally verifying built-in IEC 61131-3 FBs. It also generalizes the approach for verifying generic FBDs using tabular expressions (Sect. 2.1) and the PVS specification language.

[5] The prefixes in this section refer to monitored variables (m_...), controlled variables (c_...), enumerations (e_...), constants (k_...) and functions (f_...).

Condition		*Result* f_PrsCond
EstimatedPower < k_PwrCondSp	f_CondON = Stuck OR f_CondOFF = Stuck	False
	f_CondON = Not Debounced & f_CondOFF = Not Debounced	No Change
	f_CondON = Not Debounced & f_CondOFF = Debounced	False
	f_CondON = Debounced & f_CondOFF = Not Debounced	True
	f_CondON = Debounced & f_CondOFF = Debounced	False
EstimatedPower ≥ k_PwrCondSp		False

assume: $0 < k\_PwrCondSp$

Fig. 3 Tabular Expression of *f_PrsCond*

Condition	*Result* f_CondX
¬ (m_CondX = Pressed)	Not Debounced
[m_CondX = Pressed] & ¬ [(m_CondX = Pressed) Held for k_Debounce]	Not Debounced
(m_CondX = Pressed) Held for k_Debounce & ¬ [(m_CondX = Pressed) Held for k_Stuck]	Debounced
(m_CondX = Pressed) Held for k_Stuck	Stuck

assume: $0 < k\_Debounce << k\_Stuck$

Fig. 4 Tabular Expression of *f_CondX* (*f_CondX* and *m_CondX* are place holders, respectively, for *f_CondON*, *f_CondOFF*, *m_CondON* and *m_CondOFF*.)

Figures 5 and 6 present an example FBD design (seeded with an error) for the requirement described in Figs. 2, 3 and 4, which is further discussed in Sect. 7.2.

2.3 PVS Grammar

The PVS specification language [12] is based on classical higher-order logic equipped with dependent and subtyping mechanisms. PVS has a powerful interactive prover to perform sequent-style deductions. It is used in both academia and industry to analyze formal software specifications and verify the correctness of software against its requirements. We rely on the syntax and semantic mechanisms implemented in PVS to perform systematic design verification on SDS1. To provide a formal translation to PVS, we select a subset of the PVS grammar as a target language for FBD specifications.

3 FBD Abstract Syntax

We propose an abstract mathematical model to represent various FBD components. We consider FBDs as a named collection of variables and networks. In practice, a FBD may consist of several networks used to specify the data-flow and transitions between variables and internal FBs. We allow for negated statements as well as feedback connections to support

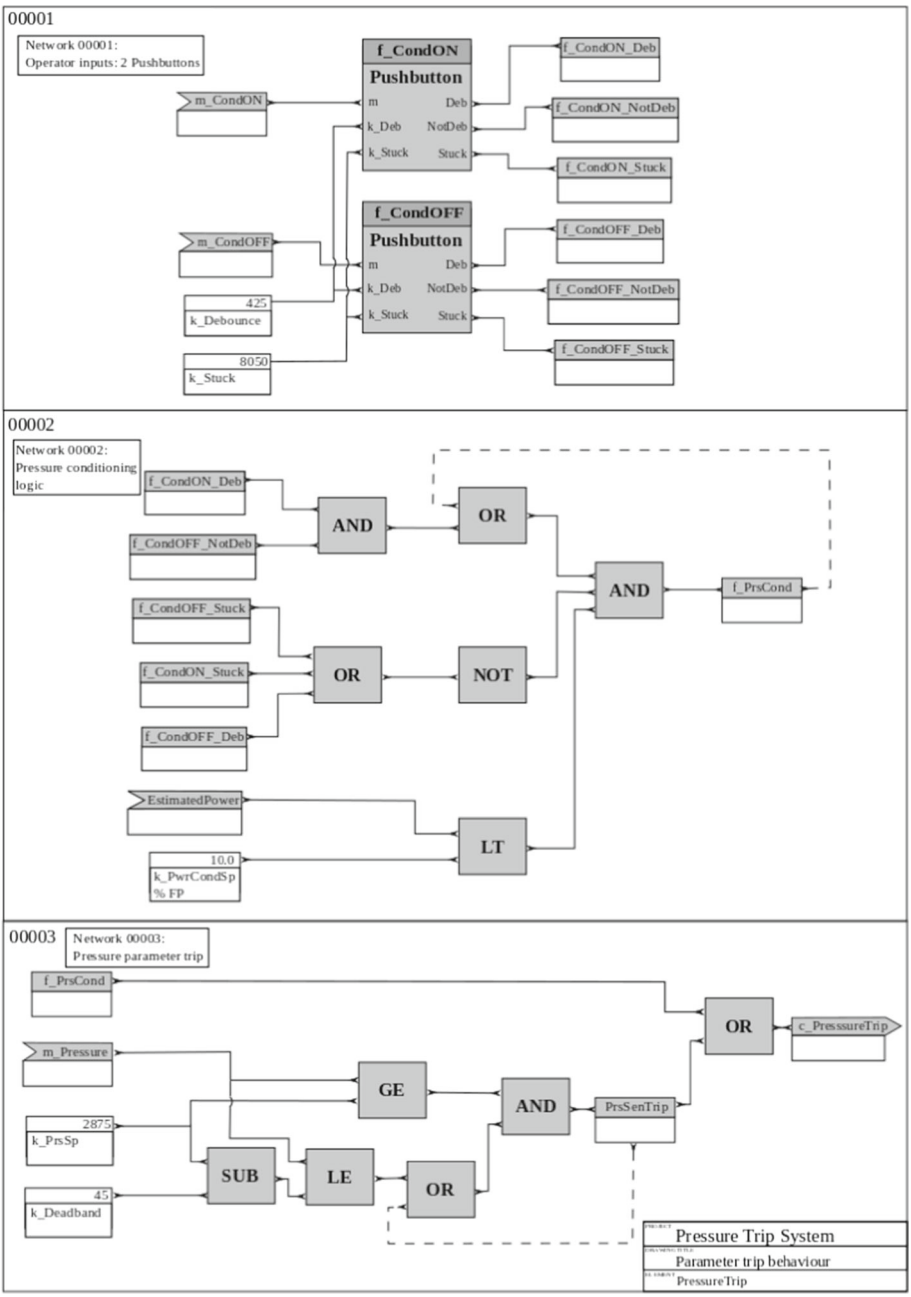

Fig. 5 FBD Design for *PressureTrip* (There are eight internal FBs: subtraction (*SUB*), less than or equal to (*LE*), greater than or equal to (*GE*), less than (*LT*), logical disjunction (*OR*), logical conjunction (*AND*), on-delay timer (*TON*) and logical negation (*NOT*).)

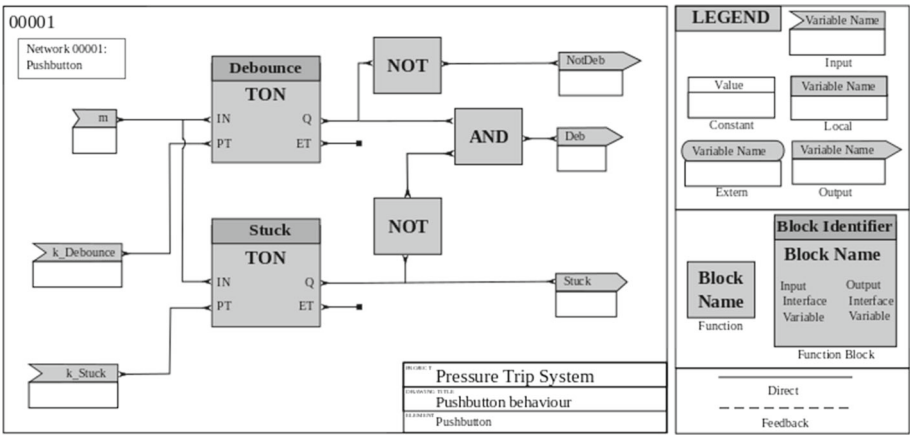

Fig. 6 FBD Design for *Pushbutton*

typical programming practices. In addition, the variable set includes interface properties and a named instance for each internal FB.[6]

Using basic mathematical constructs, we define recursive and terminal components of a FBD. We use the following notations: "×" for Cartesian product, "+" for disjoint union, "{ }" for set, "⟨ ⟩" for sequence, ":" for type definition and "→" for function. We begin by defining the following types: $IDENT$ is an identifier type that has decidable equality; $NAME$ is a type representing text-based labels; $CONN : type = \{direct, feedback\}$ is an enumerated type for direct and feedback connections; $CLS : type = \{input, output, extern, local, wire, constant\}$ is an enumerated type containing six tokens for FBD variable classification; and $INIT : type = IDENT_{init} + \epsilon$ is an initial value that is either a value represented by an identifier or is empty.

$$FBD = IDENT_{fbd} \times VARS \times \{NTWK\} \tag{1}$$
$$VARS = \{DECL\} \tag{2}$$
$$DECL = VAR + (IDENT_{blk} \times NAME_{blk} \times \{VAR\}) \tag{3}$$
$$VAR = IDENT_{var} \times IDENT_{type} \times CLS \times INIT \tag{4}$$
$$NTWK = IDENT_{ntwk} \times \{STM\} \tag{5}$$
$$STM = ELEM \times CONN \times ELEM \tag{6}$$
$$ELEM = STMV + NEG \tag{7}$$
$$STMV = IDENT_{var} + (IDENT_{blk} \times IDENT_{port}) \tag{8}$$
$$NEG = STMV \tag{9}$$

The abstract syntax is a recursive data structure, defined by Eqs. (1)–(9), with an entry value of FBD. A FBD consists of an identifier accompanied by a variable collection and a set of networks. The variable collection $VARS$ is defined by a set of declarations; $DECL$ is either a variable declaration or a block declaration. A variable declaration VAR consists of a variable identifier, a type identifier, and a classification. The second variant of $DECL$ is a block declaration consisting of a block identifier (necessary to uniquely identify a given

[6] Concrete examples are available to assist the reader with the translation rules (Sects. 3, 4 and 5) at http://www.swi.com/research/NFM2016.

instance) and a block name, and a set of variable declarations that describes the interface of the block. The variable names for the interface are referred to as interface variable identifiers $IDENT_{port}$. A network $NTWK$ contains an identifer for the network and a set of statements. A statement consists of a two variable elements and a connector. A variable element $ELEM$ consists of two variants, $STMV$ and NEG. NEG is a recursive reference to $STMV$ and represents a negated interface connection. $STMV$ has two variants. The first represents a FBD variable identifier and the second is a block identifier and an interface variable identifier. Statements represent the connections between variables and blocks.

The abstract syntax is used to generate statements that correspond to FBD components. To produce a graph that models connections between FBD variables and FBs, it is necessary to exclude or rewrite some statements representable in the abstract syntax. Variable-to-variable statements do not represent valid FBD elements and are excluded. Block-to-block statements in the abstract syntax are syntatic sugar that are rewritten. These statements are transformed to a block-to-variable and variable-to-block statements before producing the graph. The variable introduced is refered to as an interconnector, which is necessary for formalization in PVS.

4 FBD Graph Model

In this section we summarize our formalization technique using the abstract syntax (Sect. 3). We use an attribute map and labelled directed graph to represent interconnections in a FBD network. The labels in the graph contain the indices corresponding to those in the attribute map. These are used to retrieve properties for blocks, variables and connections. Given the abstract syntax, we use $VARS$ and $NTWK$ to construct the attribute map and $NTWK$ to construct the graph. We use variable identifiers $IDENT_{var}$ for the indices.

4.1 Attribute Map

The attribute map is an associative structure that relates indices to properties for FBD variables and interface variables. It is created to separate attributes from identifiers. The map is used in conjunction with the graph to retrieve properties for nodes and edges in a FBD network.

$$MAP = \langle (INDX_{var} \rightarrow ATTR_{var}) + (INDX_{fb} \rightarrow ATTR_{fb}) \rangle \tag{10}$$

$$INDX_{var} = IDENT_{var} \tag{11}$$

$$INDX_{fb} = IDENT_{blk} \times IDENT_{port} \tag{12}$$

$$ATTR_{var} = IDENT_{type} \times CLS \times INIT \tag{13}$$

$$ATTR_{fb} = NAME_{blk} \times IDENT_{type} \times CLS \tag{14}$$

The attribute map (Eqs. (10)–(14)) is a sequence of functions from indices to attributes as described by MAP. The map has two possible functional variants. The first function maps the index of a FBD variable to its attributes $ATTR_{var}$: FBD variable type, classification and initial value. The second index is a block identifier and one of its interface variables. The second function maps an index $INDX_{fb}$ to the attributes $ATTR_{fb}$: block name, interface variable type, and interface variable classification. For a given FBD network, a map is defined to store each FBD, interface and interconnector variable.

4.2 Graph Model

A directed graph is mathematically defined as a pair of nodes N, and edges E. Formally, a graph is defined by Eqs. (15) and (16). From the abstract syntax, we construct a graph for each FBD network.

$$G = (N, E) \tag{15}$$
$$E \subseteq N \times N \tag{16}$$
$$L_{node} = IDENT_{var} + IDENT_{blk} \tag{17}$$
$$L_{edge} = IDENT_{port} \times \mathbb{B} \times \mathbb{B} \tag{18}$$

A labelled graph consists of a node and edge labelling function (i.e., $l_{node} : N \rightarrow L_{node}$ and $l_{edge} : E \rightarrow L_{edge}$) that is used to map nodes and edges with their respective labels. We select labels, for the node and edge respectively, as described by Eqs. (17) and (18). L_{node} is either a variable identifier (i.e., $IDENT_{var}$) or a block identifier. L_{edge} contains an interface variable identifier, a boolean flag identifying the edge as a feedback and a boolean flag identifying the negation of an interface connection.

4.3 Block Groups

Given an attribute map and graph for a FBD network, we define an additional data structure that reduces the complexity of our PVS translation by restructuring the data to a format similar to the target expression. The block group data structure (Eqs. (19)–(22)) is motivated by the PVS predicate expression for composite FBDs. In a composite FBD, the predicate for each internal block consists of the internal block name and its associated arguments.

Block groups require two structures defined by IO_{blk} and GRP_{blk} that depend on the secondary structures $INDX_{blk}$ and ARG. $INDX_{blk}$ consists of a block identifier and block name. ARG associates a FBD variable identifier to an interface variable identifier, with boolean flags for feedback and negation. FB arguments are ordered using the interface variable element index from an attribute map.

$$INDX_{blk} = IDENT_{blk} \times IDENT_{name} \tag{19}$$
$$ARG = IDENT_{var} \times IDENT_{var} \times \mathbb{B} \times \mathbb{B} \tag{20}$$
$$IO_{blk} = INDX_{blk} \times ARG \tag{21}$$
$$GRP_{blk} = INDX_{blk} \times \langle ARG \rangle \tag{22}$$
$$f_{io} \ : \ MAP \rightarrow G \rightarrow N \rightarrow \{IO_{blk}\} \tag{23}$$
$$f_{group} \ : \ MAP \rightarrow \{IO_{blk}\} \rightarrow \langle GRP_{blk} \rangle \tag{24}$$

We present two functions that describe the process for constructing block group values in Eqs. (23) and (24). These functions implement the logic to group and order various elements. Function f_{io} constructs IO_{blk} values from an attribute map, graph and block node. The attribute map retrieves properties for nodes and edges in the graph. Values constructed from variable nodes are not valid. IO_{blk} consists of inputs or outputs for a block. Function f_{group} constructs GRP_{blk} values from a set of IO_{blk} values by extracting inputs or outputs and grouping the block identifier and block name. The resulting GRP_{blk} set is ordered using MAP, as are individual ARG sequences.

5 PVS Translation

We summarize our contributions for translating our mathematical model to PVS expressions. Based on [14], the resulting expression is a predicate with input and output arguments existentially quantified over all its internal FBs. An example of this translation can be found in Fig. 8.

5.1 Identifying Predicate Arguments

The graph maps interconnections between variables and blocks. From this relationship, we determine whether variables are either inputs or outputs in a given FBD network. It is possible the determination differs from the classification property in the attribute map since the classification does not represent the use of a variable in a given network. For example, if a *local* variable is set at the end of network 1 and is used as input in network 2, then it is consistent with its use as an output of network 1 and an input of network 2. Thus, it is not sufficient to rely on the classification value of *local* from the attribute map.

From graph theory, the degree of a node is the number of incident edges to and from a node. Since the graph is directed, we are able to determine the input degree (i.e., deg^{+}) and output degree (i.e., deg^{-}) of a node based on the position of the node in the ordered product of an edge. To find input variables, the graph is searched for all nodes that have an input degree of zero, and nodes that satisfy the variable predicate P_{var} (i.e., nodes that are FBD variables and not blocks). This is precisely described by inference rule (25), which is implemented by our translation process.

$$\frac{n : N \quad P_{var}(n) \quad deg^{+}(n) = 0}{P_{input}(n)} \tag{25}$$

$$\frac{\forall(e : E) : \neg P_{fback}(e) \quad n : N \quad P_{var}(n) \quad deg^{-}(n) = 0}{P_{output}(n)} \tag{26}$$

An output variable is defined as a terminal node in a data-flow. If an output variable is used as feedback in a FBD, then it will have an edge with a feedback property set to *True*, thus the output degree will be non-zero. These edges represent inputs from the previous cycle and satisfy the predicate P_{fback}. To correctly identify output variables, feedback edges are excluded, which causes the output degree to become zero for terminal nodes. This is precisely described by inference rule (26). Using rules (25) and (26) we construct the predicate arguments and resolve the type for each using the attribute map. This information also allows us to construct the expression used in the consistency theorem from [14].

5.2 Identifying Existential Variables

The next step of the predicate formalization is the existential quantification expression of all interconnections between internal blocks. The determination of interconnectors is performed using a similar search predicate from inference rule (26). Feedback edges are excluded to avoid identifying output variables as interconnectors. As a result, the input and output degree of a node should not be zero (i.e., each node has at least one input and one output). This is described by inference rule (27), which is implemented by our translation process.

$$\frac{\forall(e : E) : \neg P_{fback}(e) \quad n : N \quad P_{var}(n) \quad deg^{-}(n) \neq 0 \quad deg^{+}(n) \neq 0}{P_{internal}(n)} \tag{27}$$

Using rule (27), we construct the existential quantification expression over all internal blocks using the attribute map to resolve types. This is the initial component necessary to specify the predicate expression for a composite FBD.

5.3 Function Block Composition

The last step of the composite FBD formalization is a PVS expression consisting of all internal FBs composed by logical conjunction. To define this, we consider several functional structures that produce PVS expressions.

A fold is a higher order function that takes a binary function as input to reduce a recursive data structure to a terminal value. We define a function f_{expr} in Eq. (28) that translates a block grouping (i.e., GRP_{blk}) to a PVS application expression.[7] Considering f_{group}, an ordered list of $Expr$ elements is produced using the function defined by the function f_{exprl} from Eq. (29).

$$f_{expr} \ : \ GRP_{blk} \rightarrow Expr \tag{28}$$

$$f_{exprl} = \text{map}(f_{expr}, f_{group}) \tag{29}$$

$$M_{expr} = (Expr, f_{and}) \tag{30}$$

$$f_{and} \ : \ Expr \rightarrow Expr \rightarrow Expr \tag{31}$$

$$f_{pexpr} = \text{fold}(f_{and}, f_{exprl}) \tag{32}$$

To specify a binary function for the fold, we define a monoid in Eq. (30), with a signature in Eq. (31). f_{and} constructs an "$Expr$ AND $Expr$" value from the two $Expr$ inputs. Each $Expr$ input is a PVS application expression for a given composite block. Using the ordered list of $Expr$ elements, and the binary function from the monoid M_{expr}, the completed conjunctive expression is defined by the function f_{pexpr} in Eq. (32).

6 Proof Strategy for the Consistency Theorem

The consistency theorem is an obligation to show that for a FBD implementation an expected output exists for all legitimate inputs. In this section, we propose a method to automatically discharge the consistency theorem for each FBD specification in PVS. This method involves producing a data-flow expression (Sect. 6.1), a data-flow equivalence lemma and two proofs that discharge the lemma and the consistency theorem (Sect. 6.2).

6.1 Data-Flow Expressions

A basic FBD is an abstraction component that consists of built-in operators (e.g., logical operators). A composite FBD contains, as components, basic FBDs and other pre-developed composite FBDs. For a basic or composite FBD, a data-flow expression is a time-dependent

[7] The application expression consists of the block name applied with ordered arguments. An example of a PVS application expression is *MOVE*(*input*, *output*) where *MOVE* is the block name, and *input* and *output* are the arguments.

function that sequences the data-flow expressions for each internal FB. This requires that a data-flow expression exists for each internal FB in a FBD. For the case where the internal FB is a composite FBD, the data-flow expression produced from this method is used. For the case where the internal FB is a basic FBD, a data-flow expression is predefined in PVS as a function. For a basic built-in FB (i.e., a FB based on logical operators), the specification effort is minimal since the logical FB will have a data-flow expression that is equivalent to the PVS built-in operator. For example, a logical conjunction FB has a data-flow expression that is equivalent to the & operator in PVS. If the built-in FB has more complex behaviour, then a PVS specification will be independently produced that contains a data-flow definition. For example, a timer block *TON* requires a time-dependent recursive function for Q and ET.[8] A comprehensive list of built-in IEC-61131-3 FBs currently specified in PVS can be found in [14].

To construct the data-flow expression for a given output we reorganize components from Sect. 4. To obtain the sequence of internal FB data-flow expressions, we perform a depth-first search (DFS) on the graph starting from the output node and collect all following interconnector nodes. If an interconnector node has a feedback edge and the input is not the node from the DFS, then this interconnector node is replaced with a sub-sequence of interconnector nodes. If g is the Graph and o is an output, then the DFS function returns a sequence of values of the form: $DFS(g, o) = (o, \langle w_j, \ldots, (w_k, \langle w_n, \ldots, w_m \rangle), \ldots, w_i \rangle)$. Each of the sub-sequences corresponds to a recursive PVS function that is used to represent the data-flow expression for an interconnector variable used as feedback. Using the Block Groups described in Sect. 4.3, the corresponding data-flow name and input arguments can be retrieved by matching the output variable to a node from the collected sequence. For every node collected in this sequence the data-flow expression is paired together with the node name. In this form, the sequences are folded into a PVS *let* expression where the alias functions are the data-flow expressions for the interconnector node. The final expression is the data-flow expression for the output, or interconnector variable with feedback as input.

The consistency theorem has a generic solution for composite FBDs. The strategy required to solve the theorem is to skolemize the input variables and instantiate the output variables using data-flow expressions for each output. The main body of the predicate requires further instantiation for each interconnector variable using a data-flow expression. The final proof sequent can be discharged using a brute force strategy that repeatedly performs expansion and simplification until the built-in FB specifications are discharged. This technique begins to perform poorly as the size and nesting level of a composite FBD increases. During the preliminary stages of the SDS1 Trip Computer Replacement Project, we discovered the limitation of this technique when specifying large FBDs that produced multiple outputs and contained over 19 FBs with 4 layers of nesting. A solution to this issue required us to use a FB lemma equating the predicate definition to data-flow expressions used for the outputs. We name this FBD property the data-flow equivalence lemma. The data-flow equivalence lemma is a statement that the predicate, universally quantified over all inputs and outputs, is equivalent to the conjunction of all data-flows equated to the output variables. This allows us to recursively use pre-verified lemmas for each FB and avoids the need to expand the internal definitions. The general form of the lemma is:

$$\forall(i_1, \ldots, i_m) : \forall(o_1, \ldots, o_n) : FBD_{pred}(i_1, \ldots, i_m, o_1, \ldots, o_n)$$

[8] Timer TON (On-delay) is commonly used as a component of safety-critical systems. It monitors the input condition *IN* and sets the output Q as *True* whenever *IN* remains enabled for longer than a period of some input length *PT*. If the input *in* has been enabled for some time $t < PT$, then the timer sets the output *ET* (i.e., elapsed time) with value t; otherwise, it sets *ET* with value *PT*.

$$\iff (o_1 = FBD_{df_1}(i_1, \ldots, i_m) \wedge \ldots \wedge o_n = FBD_{df_n}(i_1, \ldots, i_m)) \tag{33}$$

where i_k, $k = 1, \ldots, m$, are FBD inputs, o_l, $l = 1, \ldots, n$, are FBD outputs, FBD_{pred} is the predicate definition and FBD_{df_l} is a data-flow expression for output o_l, $l = 1, \ldots, n$.

6.2 Data-Flow Expression Equivalence and Consistency Proofs

The proofs for both the data-flow equivalence lemma and the consistency theorem depend on the composition of a given FBD. PVS does not have sufficient support for n-ary relations. This limitation makes it difficult to specify a generic strategy to discharge a proof consisting of arbitrary n-ary input and output. For example, PVS cannot support a general n-ary theorem and strategy that encompasses network 00002 (7 inputs and 1 output) and 00003 (4 inputs and 1 output) from Fig. 5. As a result, we generate a proof script for both the data-flow equivalence lemma and consistency theorem that use the techniques outlined below.

The data-flow equivalence lemma consists of a predicate universally quantified over all inputs and outputs. The initial step is to skolemize the universal quantification. Secondly, the equivalence relation is split into two propositions resulting in two proof branches. The first proof branch is of the form:

$$\frac{FBD_{pred}(i_1, \ldots, i_m, o_1, \ldots, o_n)}{o_1 = FBD_{df_1}(i_1, \ldots, i_m) \wedge \cdots \wedge o_n = FBD_{df_n}(i_1, \ldots, i_m)}$$

By expanding FBD_{pred}, the existential quantification is skolemized over all interconnector variables, w_p, $p = 1, \ldots, j$, and the logical conjunction is flattened. The form of the sequent is now:

$$\frac{FB_{pred_1}(i_1, \ldots, i_m, w_1, \ldots, w_j, o_1, \ldots, o_n), \ldots, \quad FB_{pred_n}(i_1, \ldots, i_m, w_1, \ldots, w_j, o_1, \ldots, o_n)}{o_1 = FBD_{df_1}(i_1, \ldots, i_m) \wedge \cdots \wedge o_n = FBD_{df_n}(i_1, \ldots, i_m)}$$

In this form, the data-flow lemma can be used for each block and can be instantiated with the appropriate variables. The propositions in the antecedent are replaced with the respective data-flow definitions. With the form of this sequent, it is now appropriate to use a brute force strategy. The splitting action will result in a branch comparing an individual output data-flow to the data-flow of each internal FB. Since each output data-flow consists of a composition of block data-flows, this proof will complete without the aforementioned performance concerns. The second proof branch is of the form:

$$\frac{o_1 = FBD_{df_1}(i_1, \ldots, i_m), \ldots, o_n = FBD_{df_n}(i_1, \ldots, i_m)}{FBD_{pred}(i_1, \ldots, i_m, o_1, \ldots, o_n)}$$

By expanding FBD_{pred}, the existential quantification is instantiated using data-flow expressions for each interconnector variable:

$$\frac{o_1 = FBD_{df_1}(i_1, \ldots, i_m), \ldots, o_n = FBD_{df_n}(i, \ldots, i_m)}{FB_{pred_1}(i_1, \ldots, i_m, f_{dfw_1}, \ldots, f_{dfw_j}, \ldots, o_1, \ldots, o_n) \wedge \cdots \wedge \quad FB_{pred_n}(i_1, \ldots, i_m, f_{dfw_1}, \ldots, f_{dfw_j}, \ldots, o_1, \ldots, o_n)}$$

We can now use the data-flow lemma for each block, by instantiating it with the appropriate variables and replacing the propositions in the succedent with the respective data-flow definitions. The proof can be completed using a brute force strategy. The splitting action will result in a branch comparing the data-flow for each block to all individual output data-flows. Since each output data-flow consists of a composition of block data-flows, this proof will mitigate the performance concern.

The consistency theorem can be proved using a strategy that includes the data-flow equivalence lemma. The theorem form as follows:

$$\forall(i_1, \ldots, i_m) : \exists(o_1, \ldots, o_n) : FBD_{pred}(i_1, \ldots, i_m, o_1, \ldots, o_n)$$

Introducing the data-flow equivalence lemma, the sequent appears as:

$$\frac{\forall(i_1, \ldots, i_m) : \forall(o_1, \ldots, o_n) : FBD_{pred}(i_1, \ldots, i_m, o_1, \ldots, o_n) \quad \Longleftrightarrow \bigwedge_i (o_i = FBD_{df_i}(i_1, \ldots, i_m))}{\forall(i_1, \ldots, i_m) : \exists(o_1, \ldots, o_n) : FBD_{pred}(i_1, \ldots, i_m, o_1, \ldots, o_n)}$$

Skolemizing the succedent and instantiating the antecedent eliminates quantification over inputs. Both the succedent and antecedent are now quantified in such a way that requires instantiation. Instantiating each output with its corresponding data-flow expression results in a sequent as follows:

$$\frac{FBD_{pred}(i_1, \ldots, i_m, FBD_{df_1}, \ldots, FBD_{df_n}) \quad \Longleftrightarrow \bigwedge_i (FBD_{df_i}(i_1, \ldots, i_m) = FBD_{df_i}(i_1, \ldots, i_m))}{FBD_{pred}(i_1, \ldots, i_m, FBD_{df_1}, \ldots, FBD_{df_n})}$$

Replacing $FBD_{pred}(i_1, \ldots, i_m, FBD_{df_1}, \ldots, FBD_{df_n})$ in the succedent with the term in the succedent results in:

$$\frac{}{\bigwedge_i (FBD_{df_i}(i_1, \ldots, i_m) = FBD_{df_i}(i_1, \ldots, i_m))}$$

This final form is trivially proved by splitting the expression.

7 Nuclear Industry Case Study

The DNGS SDS1 TCs monitor a diverse set of nuclear and process parameter sensors that cover all critical design basis accident scenarios. In the case of anomalous behaviour, the TCs respond via control logic to signal a reactor trip. Signals from three redundant SDS1 TCs are connected to 2-out-of-3 voting logic that ultimately initiates a reactor trip.[9] The SDS1 TC software requirements are formalized using TEs and the software is designed using FBDs. First, we present a simplified example of verifying a parameter trip requirement. Second, we demonstrate the application of our formal translation rules and discuss the verification results from applying PVS.

7.1 Parameter Trip Setpoint Requirements

In this example, we consider the requirements of a generalized parameter trip. The TC is designed to provide coverage of a pressure input *m_Pressure*. The parameter will not trip if it has been conditioned out. There are two possible mechanisms for conditioning the parameter: low reactor power and operator input through two pushbuttons.

The TE as shown in Fig. 2, Sect. 2.1 specifies that *c_PressureTrip* generates a trip response, if the pressure input (*m_Pressure*) is above or equal to the setpoint (*k_PrsSP*) if the parameter is not conditioned out. It will not generate a trip response if the pressure input is below or equal to the setpoint minus the deadband value. The deadband value is assumed to be positive (or else the tabular expression is ill-formed), and much smaller in value than the absolute

[9] SDS2 uses diverse sensors and technologies to cause a reactor trip if SDS1 were to fail.

value of the setpoint (or else it affects behaviour rather than simply reducing noise). The value of *c_PressureTrip* does not change at all if the pressure input is in the deadband region. A trip response will not be generated if the parameter has been conditioned out. Note that, since the function value may be left unchanged, an initial value must be provided. In keeping with the safety priority of the system, the initial value in this case would be tripped (represented as *True*).

The TE as shown in Fig. 3, Sect. 2.1 specifies that *f_PrsCond* will be conditioned out, if the operator has requested conditioning (*f_CondON* is debounced and *f_CondOff* is not debounced) and there is low estimated power. It will not perform conditioning if *f_CondOFF* has been debounced or if either pushbutton is stuck. This logic protects the system from erroneous pushbutton combinations and ensures conditioning only occurs during operator request at low power.

The TE as shown in Fig. 4, Sect. 2.1 specifies a generic requirement for both the condition ON and condition OFF pushbuttons. The requirement is derived from a TE presented in [15], which describes the behaviour of a pushbutton as a time delay function with a debounce and stuck state. When the pushbutton is not pressed, the resulting state is considered not debounced. The definition makes use of the *Held for* operator [14] as described by Eq. (34). For simplicity, the proposed example does not include timing tolerances. We approximate continuous time, as a type *tick*, defined as a discrete series of equally-distributed clock ticks with an arbitrarily small positive interval δ: $tick = \{t_n : \mathbb{R}_{\geq 0} \mid \delta \in \mathbb{R}_{>0} \wedge (\exists n : \mathbb{N} \bullet t_n = n \times \delta)\}$. The infix operator, *P Held For* (d), states that a monitored boolean condition P will be sustained over a positive time duration d. More precisely,

$$\exists (t_j : tick) : (t - t_j \geq d) \wedge (\forall (t_n : tick \mid t_n \geq t_j \wedge t_n \leq t) : P(t_n)) \tag{34}$$

7.2 Design and Formal Translation

An example design (Fig. 5, Sect. 2.2) uses several built-in IEC 61131-3 FBs to specify the functional behaviour. The design is decomposed into three networks to specify the functionality of the pushbuttons, conditioning and trip logic respectively. As an example, we select Network 00003 since it uses a feedback connection for the hysteresis effect. It is important to note that the target PLC treats "de-energised" (`false = 0`) as the safe state, therefore `c_PressureTrip = false` in the design domain is equivalent to *c_PressureTrip* = *True* in the requirements domain. The example also includes built-in timer blocks used to implement the pushbutton behaviour in Fig. 6. *ET* outputs are left unconnected since they are not used. The idealized behaviour of the pushbutton is described in [15], which uses a time-dependent *last_enabled* function to calculate the rising edge.

For this example, we use the prototype translator to demonstrate our translation rules. Mapping this diagram to an abstract syntax is performed by preparing an ASCII input file and using a simple parser. We have implemented a function to modify block-to-block connections by introducing an additional "wire" variable. These variables are added to an attribute map and are used in the labels of a graph, as illustrated in Fig. 7. A graph and attribute map is produced for each network. A top level PVS expression for the FBD will use the expressions generated from each graph.

The translation rules are further applied and the resulting PVS code is illustrated in Fig. 8. Using the input and output identification rules from Eqs. (25) and (26), inputs and outputs of the graph in Fig. 7 are respectively: *k_PrsSP*, *k_DeadBand*, *m_Pressure*, *f_PrsCond* and *c_PressureTrip*. *k_PrsSP* and *k_DeadBand* are not included as arguments. These variables are constant values represented in PVS by a constant function. The existential identification rule

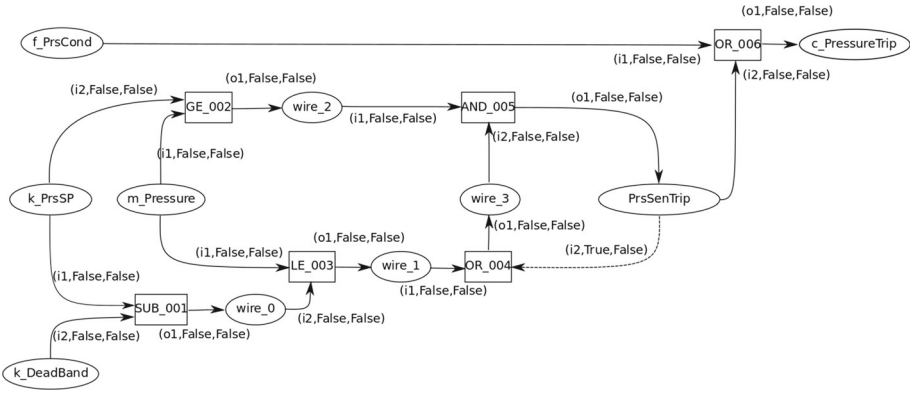

Fig. 7 Labelled Directed Graph for *Network 3: c_PressureTrip*

from Eq. (27) yields the internal variables: *wire0*, *wire1*, *wire2*, *wire4* and *PrsSenTrip*. Lastly, the conjunction of internal blocks *SUB*, *GE*, *LE*, *OR*[10] and *AND* completes the expression as shown.[11]

The translation rules for the data-flow expressions are applied and the resulting PVS code for an interconnector variable is illustrated in Fig. 9. Using the techniques described in Sect. 6.1, data-flow expressions for each FB are sequently composed and assigned to the interconnector variables. The final expression is the data-flow expression for the FB connected to the output variable. Since a feedback connection is used, the data-flow expression is defined as a recursive function with a measure function, $rank(t)$, that specifies the ordering relation over the time sequence. This function is then used to discharge the data-flow equivalence lemma and the consistency theorem as described in Sect. 6.2.

7.3 Verification

CE-1001-STD [8] specifies a set of complementary and overlapping verification processes, one of them being systematic design verification (SDV). The objective of SDV is to verify that all functions in the design are equivalent to their corresponding functions in the requirements using mathematical techniques or rigorous argument. SDV uses a specialization of the four variable model [16] to confirm the satisfaction of Eq. (35).

$$OUT \circ SOF \circ IN \vdash REQ \tag{35}$$

For the purposes of our example, *REQ* is the TE from Figs. 2, 3 and 4 and other supporting information (not shown) that defines the monitored and controlled variables, the constants, and the enumerated types. *SOF* is the FBD from Figs. 5 and 6, and other supporting information (not shown) that defines the input and output variables and constants used. *IN* and *OUT* are functions that translate monitored variables to input variables and output variables to controlled variables, respectively (an example of such a translation for *c_PressureTrip* is shown in Sect. 7.2). Our verification was performed in PVS using built-in table constructor (`cond`) to specify the requirements [21]. We then created a PVS specification containing a

[10] The underscore (..._) is used for generated names that conflict with PVS keywords.

[11] The FBD is formalized over a discrete time series of equally distributed samplings, i.e., ticks. The *pre* operator returns the previous time sample.

```
PressureTrip_3 ( m_Pressure : [ tick → DINT ] ,
                 f_PrsCond : [ tick → BOOL ] ,
                 c_PressureTrip : [ tick → BOOL ] ) : bool =
  forall( t : tick ):
  exists ( wire0 : [ tick → DINT ] ,
           wire1 : [ tick → BOOL ] ,
           wire2 : [ tick → BOOL ] ,
           wire3 : [ tick → BOOL ] ,
           PrsSenTrip : [ tick → BOOL ] ) :
  if init ( t ) then
   c_PressureTrip ( t ) = false and
   PrsSenTrip ( t ) = false
  else
   OR_ ( f_PrsCond ,
         PrsSenTrip ,
         c_PressureTrip )
       ( t ) and
   AND_ ( lambda ( t1 : noninit_elem ) :
            not wire2 ( t ) ,
          wire3 ,
          PrsSenTrip )
        ( t ) and
   GE ( m_Pressure ,
        k_PrsSP ,
        wire2 )
      ( t ) and
   LE ( m_Pressure ,
        wire0 ,
        wire1 )
      ( t ) and
   OR_ ( wire1 ,
         lambda ( t1 : noninit_elem ) :
           PrsSenTrip ( pre ( t1 ) ) ,
         wire3 )
       ( t ) and
   SUB ( k_PrsSP ,
         k_DeadBand ,
         wire0 )
       ( t )
 endif
```

Fig. 8 Generated PVS for *Network 3: PressureTrip*

theorem in the form of Eq. (35). The requirements and design specify behaviour using recursion, which requires the proof to begin with the *time_induction* scheme from the *ClockTick* theory. The formulae are expanded until the comparison is performed on low level operators. For the *Held for* and *TON* operators, a pre-verified lemma is used to discharge the sequents as was done in [15].

After further reduction, we discovered an unprovable sequent that prevented us from discharging the proof. Upon investigation, we recognize the design failed to add a negation between the *GE* and *AND* blocks in Network 00003 of Fig. 5 (the seeded error mentioned in Sect. 2.2). This is a clear demonstration of how formal verification detects subtle design flaws that could potentially result in unintended behaviour. The application of the approach[12] for SDV on the DNGS SDS1 TC replacement project helped identify design pattern inconsisten-

[12] The approach was qualified by trial use, inspection and acceptance testing.

```
PrsSenTrip_df ( m_Pressure : [ tick → DINT ] ,
                f_PrsCond : [ tick → BOOL ] )
                ( t : tick ) : recursive BOOL =
  if init( t ) then
   false
  else
  let wire0 = SUB_df ( k_PrsSP ,
                       k_DeadBand ) ,
      wire1 = LE_df ( m_Pressure ,
                      wire0 ) ,
      wire2 = GE_df ( m_Pressure ,
                      k_PrsSP ) ,
      wire3 = OR_df ( wire1 ,
                      lambda ( t1 : noninit_elem ) :
                       PrsSenTrip_df( m_Pressure,
                                      f_PrsCond )
                                    ( pre ( t ) ) ) in
      AND_df ( lambda ( t1 : noninit_elem ) :
                not wire2 ( t1 ) ,
               wire3 )
       ( t )
   endif
measure rank ( t )
```

Fig. 9 Generated PVS for *Network 00003: PressureTrip*

cies that led to an improved FBD-based design approach, uncovered inconsistencies in TEs that led to a more precise requirements specification, and identified an omitted conversion in the FBD for performing an average power calculation. PVS was used to verify all FBDs in the design, which accounted for 80% of the overall SDV effort. Our approach was used to automatically discharge 70% of the proof obligations. The most complicated FBD, a module with 20 FBs and 39 variables, and modules with real-time properties, required some user interaction with PVS to discharge the proof.

8 Simulink Model of a Nuclear Industrial Example

The method described in this paper was created for specific application to IEC 61131-3 compliant FBDs and the PLC digital computers they are used to program. Looking beyond to more general application, we repeated the example given in Sect. 7, this time using Simulink as the graphical design language. Designs created using Simulink can be translated into a general purpose programming language applicable to various target platforms.

Figures 10, 11, 12, 13 and 14 depict a Simulink design (model) of the *c_PressureTrip* requirements using TEs (Sect. 2.1). The Simulink structure and primitives are as close to the FBD design as possible, but some changes were necessary due to language differences. The respective changes replace the dashed-line with a a unit delay (i.e., z^1) for feedback loops, and the built-in TON/TOF blocks with the timing delay blocks for debounce/stuck behaviour.

The method, unchanged, was applied to the Simulink design to create an attribute map, labelled directed graphs for each Simulink module, block groups and finally the PVS code. This PVS representation of the design was then fed, along with the PVS representation from the TEs, to the PVS prover. All proofs were discharged successfully.

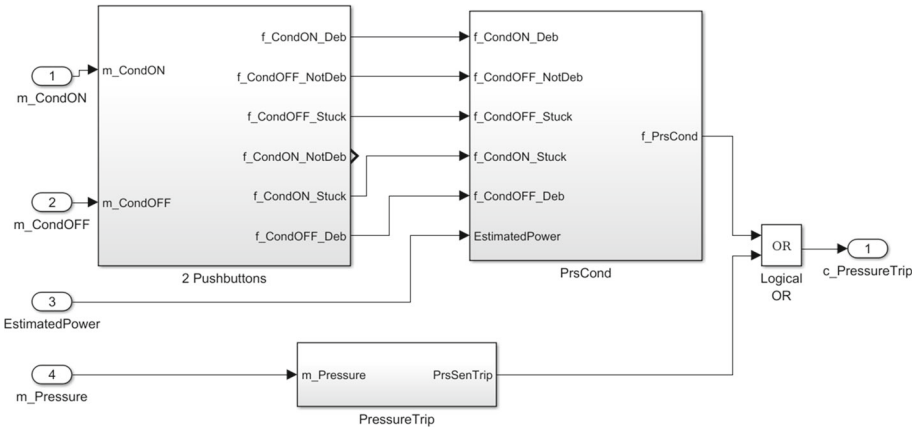

Fig. 10 Simulink Model for c_PressureTrip

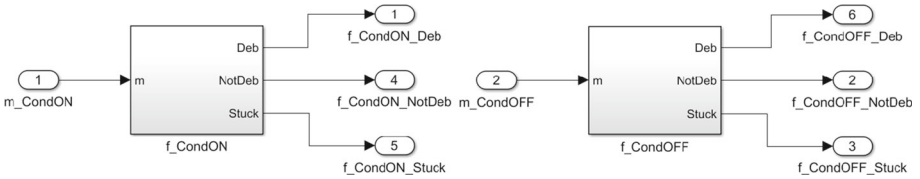

Fig. 11 Simulink Model for 2 Pushbuttons

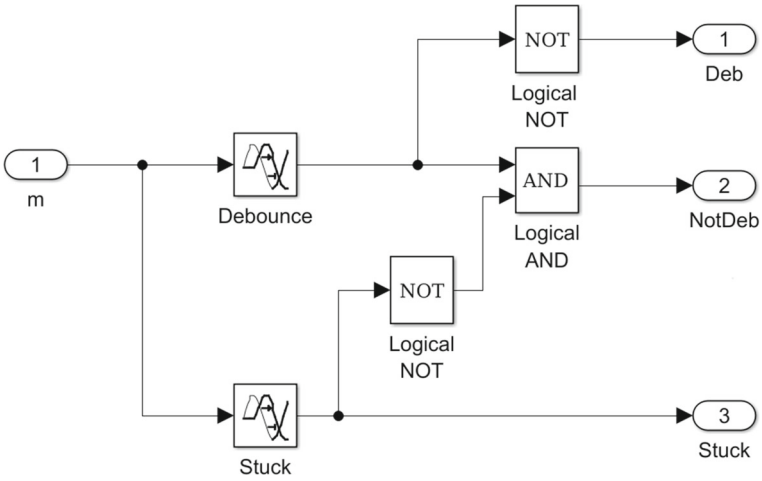

Fig. 12 Simulink Model for Pushbutton Design

9 Related Work

IEC 61131-3 provides definitions for five PLC languages[13] and various research work has produced formalization and verification of PLC programs. In terms of the formal verification

[13] Function block diagram (FBD), structured text (ST), instruction list (IL), ladder diagram (LD) and sequential function chart (SFC).

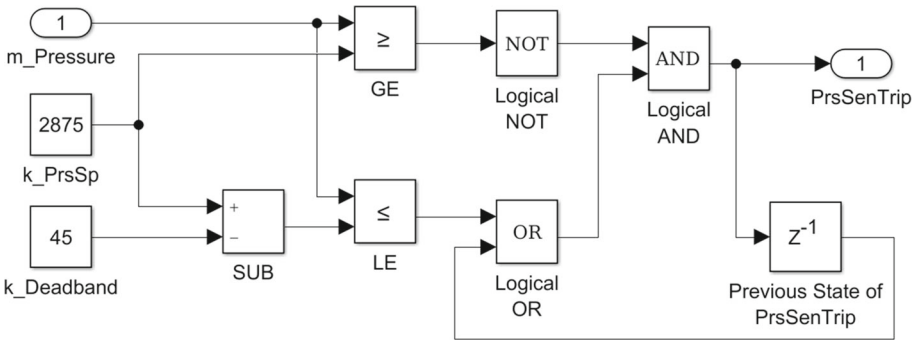

Fig. 13 Simulink Model for PrsSenTrip

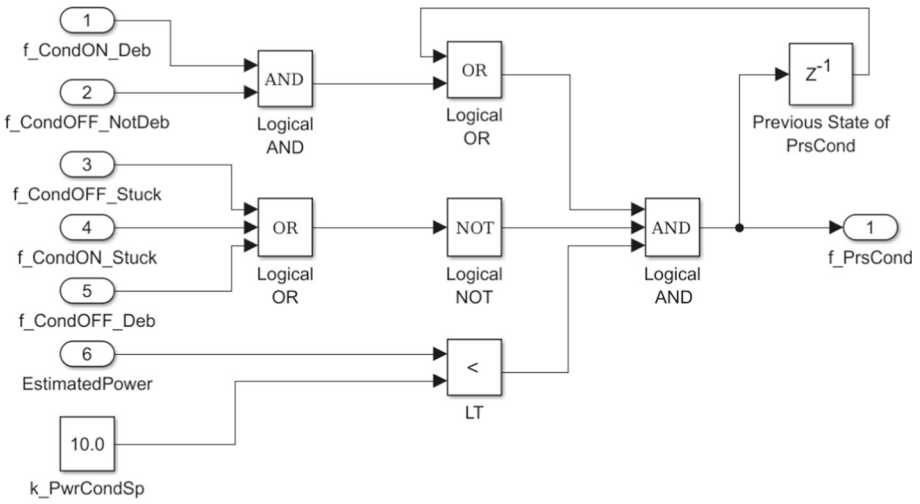

Fig. 14 Simulink Model for f_PrsCond

of PLC programs written in these languages, there are typically two main approaches to prove or disprove the correctness of a design with respect to a certain formal requirements specification or required property: model checking and theorem proving. We summarize the most relevant works to this paper in Table 1 in terms of: the main verification approach, the set of supported IEC 61131-3 languages, industrial scalability, supporting front-end and verification tool.

In the case of model checking, Németh and Bartha [10] provides 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. Soliman et al. [18] transforms FBD descriptions to its logically equivalent Uppaal models that perform the verification of safety applications in the industrial automation domain. Jimenez-Fraustro and Rutten [6] translates ST and FBD into a synchronized data-flow language SIGNAL to compile and reason about the verification of specifications. Darvas et al. [2] presents two complementary solutions for the formal verification of safety-critical PLC programs (written in ID, IL and FBD) based on model checking and equivalence checking. Nellen et al. [9] proposes two CEGAR-based techniques for the reachability analysis for SFC-based chemical

Table 1 Summary of Related Work

References	IEC 61131-3 Language(s)	Scalability	Front-end	Tool(s)
Theorem proving				
[1]	SFC, IL, LD, and FBD	Small	Yes	Coq
[19]	SFC, ST and FBD	Small	No	HOL
[11]	FBD	Large	Yes	PVS
Model checking				
[10]	FBD	Small	No	Design/CPN
[18]	FBD	Small	No	Uppaal
[6]	ST and FBD	Small	No	SIGNAL
[2]	IL, ID and FBD	Large	Yes	nuXmv and UPPAAL
[9]	SFC	Medium	No	SpaceEx

plants. In the case of theorem proving, Blech and Biha [1] uses Coq to check the correctness of SFC programs, which is automatically generated from a graphical front-end. Völker and Krämer [19] formalizes PLC programs using higher-order logic and uses HOL to discharge safety properties.

In the case of model checking, there is difficulty scaling up to industrial-size applications. In theorem proving, complex formalisms can be handled, but the process of proofs is not fully automated and adds additional overhead to industrial scale applications. Thus, the strengths and weaknesses for model checking and theorem proving are complementary. To balance this issue, our technique has been successfully used in an on-going nuclear industrial application, and it is novel in that: (1) we translate a FBD design to a formal PVS model; (2) the resulting PVS model can be verified against TE-based requirements input to PVS; and (3) we propose a method to prove the consistency theorem for a FBD in PVS.

10 Conclusion and Future Work

In an earlier version of this paper [11], we extended the work presented in [13] with an industrial-scaled methodology for the systematic translation of FBD designs compliant with IEC 61131-3 into the PVS formal specification language. The approach was developed for OPG and is in current use as part of the verification of the DNGS SDS1 TCs. In combination with PVS, this work has proven effective in uncovering subtle inconsistencies in applying design patterns, inconsistencies in the real-time requirements documented using TEs, and non-conformance between a more complex FBD design and its real-time requirements. In this revision of the paper, we propose a proof strategy to reduce the effort required to produce and discharge the consistency theorem for a FBD design example. As an extension, we demonstrate applicability to a Simulink-based design using the same example.

As on-going and future work, we first aim to improve our translation rules using operational semantics to verify the transformation itself and FBD well-formedness conditions to provide more precision for potential tool designers. Secondly, we are currently implementing proof scripts to increase the level of automation, which has potential application in other industrial domains, e.g., aerospace. Thirdly, we intend to apply the methodology to generate specifications in Coq to make use of custom tactics and use the environment to

certify the translation process. Lastly, we plan to extend our formalization technique to other IEC 61131-3 compliant languages, e.g., Structured Text (ST).

Acknowledgements We would like to thank OPG for their permitting us to describe the work related to the DNGS TC replacement project. The methodology and tools described herein are the property of OPG. Particularly, we thank Ivan Dimitrov, Section Manager, Safety Related Computers, Computers and Control Design, and Mike Viola, SDS Replacement Project Manager, for their valued oversight and assistance. Special thanks to Lucian Patcas for his thorough review.

References

1. Blech, J.O., Biha, S.O.: On formal reasoning on the semantics of PLC using Coq. CoRR abs/1301.3047 (2013)
2. Darvas, D., Majzik, I., Blanco Viñuela, E.: Formal Verification of Safety PLC Based Control Software, pp. 508–522. Springer, Cham (2016)
3. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Special Committee 205 of RTCA (2011)
4. IEC: 61131-3 Ed. 3.0 en:2013: Programmable Controllers—Part 3: Programming Languages. International Electrotechnical Commission (2013)
5. IEEE 7-4.3.2: Standard for Digital Computers in Safety Systems of Nuclear Power Generating Stations (Revision of IEEE Std 7-4.3.2-2003). The Institute of Electrical and Electronics Engineers (IEEE) (2010)
6. Jimenez-Fraustro, F., Rutten, E.: A synchronous model of IEC 61131 PLC languages in SIGNAL. In: Euromicro Conference On Real-Time Systems, pp. 135–142 (2001)
7. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Sci. Comput. Program. **75**(11), 980–1000 (2010)
8. Joannou, P., Harauz, J., Viola, M., Cirjanic, R., Chan, D., Whittall, R., Tremaine, D., Moum, G.: Standard for Software Engineering of Safety Critical Software Revised for Application Oriented Language. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 3 (2014)
9. Nellen, J., Driessen, K., Neuhäußer, M., Ábrahám, E., Wolters, B.: Two CEGAR-based approaches for the safety verification of PLC-controlled plants. Inf. Syst. Front. **18**(5), 927–952 (2016)
10. Németh, E., Bartha, T.: Formal verification of safety functions by reinterpretation of functional block based specifications. In: Formal Methods for Industrial Critical Systems. Springer, pp. 199–214 (2009)
11. Newell, J., Pang, L., Tremaine, D., Wassyng, A., Lawford, M.: Formal translation of IEC 61131-3 function block diagrams to PVS with nuclear application. In: NASA Formal Methods—8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7–9, 2016, Proceedings. pp. 206–220 (2016)
12. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: International Conference on Automated Deduction. LNCS, vol. 607, pp. 748–752 (1992)
13. Pang, L.: An Engineering Methodology for the Formal Verification of Function Block Based Systems. Ph.D. thesis, McMaster University, Department of Computing and Software (2015)
14. Pang, L., Wang, C., Lawford, M., Wassyng, A.: Formal verification of function blocks applied to IEC 61131-3. Sci. Comput. Program. **113**, 149–190 (2015)
15. Pang, L., Wang, C., Lawford, M., Wassyng, A., Newell, J., Chow, V., Tremaine, D.: 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, pp. 65–79 (2015)
16. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
17. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Softw. Eng. **20**, 948–976 (1994)
18. Soliman, D., Thramboulidis, K., Frey, G.: Transformation of function block diagrams to Uppaal timed automata for the verification of safety applications. Annu. Rev. Control. **36**, 338–345 (2012)
19. Völker, N., Krämer, B.J.: Automated verification of function block-based industrial control systems. Sci. Comput. Program. **42**(1), 101–113 (2002)
20. Wassyng, A., Janicki, R.: Tabular expressions in software engineering. In: International Conference on Software & System Engineering and their Applications, vol. 4, pp. 1–46 (2003)
21. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: FME 2003: Formal Methods, LNCS, vol. 2805, pp. 133–153. Springer, Berlin, Heidelberg (2003)