Formal Translation of IEC 61131-3 Function Block Diagrams to PVS with Nuclear Application

Josh Newell[1(✉)], Linna Pang[1], David Tremaine[1], Alan Wassyng[2], and Mark Lawford[2]

[1] Systemware Innovation Corporation, Toronto M4P 1E4, Canada
{jnewell,lpang,tremaine}@swi.com
[2] McMaster Centre for Software Certification, McMaster University, Hamilton L8S 4K1, Canada
{wassyng,lawford}@mcmaster.ca

Abstract. The trip computers for the two reactor shutdown systems of the Ontario Power Generation (OPG) Darlington Nuclear Power Generating Station (DNGS) 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 (SRS) written using tabular expressions (TEs). The PVS theorem proving tool is used in the 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.

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

1 Introduction

Many industrial, safety-critical control systems leverage programmable technologies for their flexibility and scalability. 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 [2] advocates the use of formal methods to create mathematical models for the specification

S. Rayadurgam and O. Tkachuk (Eds.): NFM 2016, LNCS 9690, pp. 206–220, 2016.
DOI: 10.1007/978-3-319-40648-0_16

and analysis of system behaviour. In the nuclear industry, IEEE 7-4.3.2 [1] 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 [7] 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 can be 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 design specification were developed and documented using tabular expressions (TEs) [13]. In addition to various review and overlapping testing processes, formal proof of correctness was performed using a theorem prover Prototype Verification System PVS [9].

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[1,2]. Using the PLC platform, the detailed design automatically generates executable code. PVS is used to formally verify the design against the requirements.

PVS provides an integrated environment with mechanized support for the syntax and semantics of TEs and (higher-order) predicates. Based on [10], an approach was developed for the replacement project to support the formal verification of FBDs. The process is as follows: (1) the trip computer design, described in a collection of FBDs, is translated into PVS; (2) the requirements described in tabular expressions are translated into PVS; and (3) formal proofs for systematic design verification are automated using PVS.

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

[2] The use of IEC 61131-3 compliant built-in FBs eased formal specification and subsequent verification of their behavior; one of many PLC qualification activities.

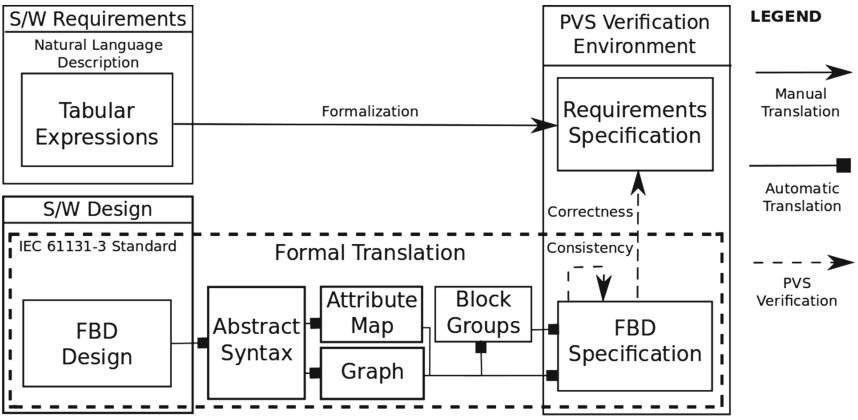

Fig. 1. Framework diagram

Step (1) of the process is the subject of this paper and is based on our experience in 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 requirements are documented using tabular expressions. The design is written in a FBD language that is complaint 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 requirements are formalized in PVS whereas the FBD specification is produced from our methodology. Based on [10], our technique also produces the consistency theorems[3] for FBDs, which are verified manually in PVS. The correctness theorems are manually specified and verified in PVS. The future automation of consistency and correctness proofs is discussed in Sect. 8.

2 Preliminaries

2.1 Tabular Expressions

Tabular expressions [13] (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

[3] A FBD design is consistent if for every input there exists an output that satisfies the internal relationships. Otherwise, a FBD design trivially satisfies any requirement.

maintain than conventional mathematical expressions. Formal semantics for tabular expressions have been well-developed in [6] and are useful for inspections, testing and verification [17,18]. Tabular expressions were used on the original SDS1 project and continue to be used on the replacement project for specifying software requirements. As an example of a tabular expression (Fig. 2), we consider the *c_PressParmTrip* requirement that will be used as a running example. The function calculates the parameter trip value using the process variable *m_Presssure* compared against the setpoint value *k_PressSP*[4]. We present the detailed discussion in Sect. 6.1.

	Result
Condition	c_PressParmTrip
m_Pressure $\leq$ k_PressSP - k_DeadBand	e_not_tripped
k_PressSP - k_DeadBand $<$ m_Pressure $<$ k_PressSP	No Change
k_PressSP $\leq$ m_Pressure	e_tripped

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

Fig. 2. Tabular expression of *c_PressParmTrip*

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, with its latest version being published in 2013 [4]. The DNGS SDS1 trip computer uses built-in IEC 61131-3 FBs as the basis of the formal software design. The methodology outlined in [10,11], 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 PVS. Figure 3 presents an example FBD design (seeded with an error) for the requirement described in Fig. 2, which is further discussed in Sect. 6.2.

2.3 PVS Grammar

The PVS specification language [9] 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. 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.

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

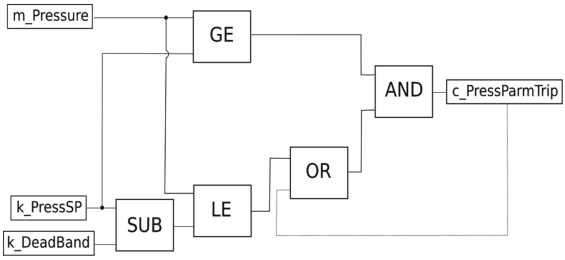

Fig. 3. FBD design for *c_PressParmTrip*[5]

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 dataflow and transitions between variables and internal FBs. We allow for negated statements as well as feedback connections to support 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: "$\times$" for Cartesian product, "$+$" for disjoint union, "$\{\}$" for set, "$\langle\rangle$" for sequence, ":" for type definition and "$\rightarrow$" for function. We begin by defining the following types: I_{ident} is an identifier type that has decidable equality; $K_{conn} : type = \{direct, feedback\}$ is an enumerated type for direct and feedback connections; $C_{class} : type = \{input, output, extern, local, wire\}$ is an enumerated type containing five tokens for FBD variable classification; and $I_{init} : type = I_{ident} + \epsilon$ is an initial value that is either a value represented by an identifier or is empty.

$$F_{fbd} = F_{ident} \times W_{vars} \times \{N_{ntwk}\} \tag{1}$$
$$W_{vars} = \{D_{decl}\} \tag{2}$$
$$D_{decl} = R_{var} + (B_{ident} \times H_{ident} \times \{R_{var}\}) \tag{3}$$
$$R_{var} = V_{ident} \times T_{ident} \times C_{class} \times I_{init} \tag{4}$$
$$N_{ntwk} = N_{ident} \times \{S_{stm}\} \tag{5}$$
$$S_{stm} = U_{velm} \times K_{conn} \times U_{velm} \tag{6}$$
$$U_{velm} = Q_{svar} + Z_{neg} \tag{7}$$
$$Q_{svar} = V_{ident} + (B_{ident} \times P_{ident}) \tag{8}$$
$$Z_{neg} = Q_{svar} \tag{9}$$

[5] There are five internal FBs: subtraction (SUB), less than or equal to (LE), greater than or equal to (GE), logical disjunction (OR) and logical conjunction (AND).

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

The abstract syntax is a recursive data structure, defined by Eqs. (1)–(9), with an entry value of F_{fbd}. A F_{fbd} consists of an identifier accompanied by a variable collection and a set of networks. The variable collection W_{vars} is defined by a set of declarations; D_{decl} is either a variable declaration or a block declaration. A variable declaration R_{var} consists of a variable identifier, a type identifier, and a classification. The second variant of D_{decl} is a block declaration consisting of a block identifier 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 P_{ident}. A network N_{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 U_{velm} consists of two variants, Q_{svar} and Z_{neg}. Z_{neg} is a recursive reference to Q_{svar} and represents a negated interface connection. Q_{svar} 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 graph models connections between FBD variables and FBs. Variable-to-variable statements do not satisfy this condition. Representing block-to-block statements is syntatic sugar. These statements are rewritten as block-to-variable and a variable-to-block statements before producing the graph. The variable introduced is refered to as an interconnector, which is necessary for the PVS formalization. Lastly, the classification property for interface variables are exclusively *input* or *output* values.

4 Graph Model

In this section we summarize our formalization technique using the abstract syntax, previously defined, as input. We make use of an attribute map, and labelled directed graph to represent interconnections in a FBD network. The labels of the graph contain indices that are used to retrieve properties for blocks, variables and connections from the attribute map. Given the abstract syntax, we use W_{vars} and N_{ntwk} to construct the attribute map and N_{ntwk} to construct the graph. We chose to use variable identifiers I_{ident} to construct the indices.

4.1 Attribute Map

The attribute map is an associative structure that relates indicies 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.

$$M_{map} = \langle (I_{idf} \rightarrow A_{varf}) + (I_{idi} \rightarrow A_{vari}) \rangle \tag{10}$$

$$I_{idf} = V_{ident} \tag{11}$$

$$I_{idi} = B_{ident} \times P_{ident} \tag{12}$$

$$A_{varf} = T_{ident} \times V_{class} \times I_{init} \tag{13}$$

$$A_{vari} = I_{ident} \times T_{ident} \times P_{class} \tag{14}$$

The attribute map, defined by Eqs. (10)–(14), is a sequence of functions from indices to attributes as described by M_{map}. The map has two possible function variants. The first function is the mapping between the index of a FBD variable to its attributes A_{varf}: 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 I_{idi} to the attributes A_{vari}: block name, interface variable type, and interface variable classication. 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 $\mathbb{N}$, 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.

$$G = (\mathbb{N}, E) \tag{15}$$

$$E \subseteq \mathbb{N} \times \mathbb{N} \tag{16}$$

$$L_{node} = V_{ident} + B_{ident} \tag{17}$$

$$L_{edge} = P_{ident} \times \mathbb{B} \times \mathbb{B} \tag{18}$$

A labelled graph consists of a node and edge labelling function (i.e., $l_{node} : \mathbb{N} \rightarrow L_{node}$ 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., I_{ident}) 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 a 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, defined by 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 B_{io} and B_{group} that depend on the secondary structures K_{blk} and I_{arg}. K_{blk} consists of a block identifier and block name. I_{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.

$$K_{blk} = B_{ident} \times H_{ident} \tag{19}$$
$$I_{arg} = V_{ident} \times I_{ident} \times \mathbb{B} \times \mathbb{B} \tag{20}$$
$$B_{io} = K_{blk} \times I_{arg} \tag{21}$$
$$B_{group} = K_{blk} \times \langle I_{arg} \rangle \tag{22}$$
$$f_{io} : M_{map} \to G \to \mathbb{N} \to \{B_{io}\} \tag{23}$$
$$f_{group} : M_{map} \to \{B_{io}\} \to \langle B_{group} \rangle \tag{24}$$

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 B_{io} values from an attribute map, graph and block node. The attribute map is required to retrieve properties for nodes and edges in the graph. Values constructed from variable nodes are not valid. B_{io} consists of granular inputs or outputs for a block. Function f_{group} constructs B_{group} values from a set of B_{io} values by extracting inputs or outputs and grouping the block identifier and block name. The resulting B_{group} set is ordered using M_{map}, as are individual I_{arg} sequences.

5 PVS Translation

We summarize our contributions for translating our mathematical model to PVS expressions. Based on [11], the resulting expression is a predicate with input and output arguments existentially quantified over all its internal FBs.

5.1 Identifying Predicate Arguments

The graph maps interconnections between variables and blocks. From this relationship, we determine whether variables behave as 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 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 sufficent 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.

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

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

An output variable is defined as a terminal node in a dataflow. 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 [11].

5.2 Identifying Existential Variables

The next step of the predicate formalization is the existential quantification 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 precisely described by inference rule (27), which is implemented by our translation process.

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

Using rule (27), we construct the existential quantification 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 interpreted with PVS syntactic types.

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., B_{group}) 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).

[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.

$$f_{expr} : B_{group} \rightarrow Expr \tag{28}$$
$$f_{exprl} = \text{map}(f_{expr}, f_{group}) \tag{29}$$
$$M_{expr} = (Expr, f_{and}) \tag{30}$$
$$f_{and} : Expr \rightarrow Expr \rightarrow Expr \tag{31}$$
$$f_{pexpr} = \text{fold}(f_{and}, f_{exprl}) \tag{32}$$

To specify a binary function for the fold, we define a monoid in Eq. (30), with a signature defined in Eq. (31). The definition of 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 Nuclear Industry Case Study

The DNGS SDS1 TCs monitor a diverse set of nuclear and secondary parameters that cover all critical design basis accident scenarios. In the case of anomalous behavior, 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[8]. 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.

6.1 Parameter Trip Setpoint Requirements

In this example, we consider the requirements of a generalized parameter trip. The special safety system is designed to provide coverage of a pressure input *m_Pressure*. The TE (Fig. 2, Sect. 2.1) specifies that *c_PressParmTrip* generates a trip response, if the pressure input (*m_Pressure*) is above or equal to the setpoint (*k_PressSP*). 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 value of the setpoint (or else it affects behaviour rather than simply reducing noise). The value of *c_PressParmTrip* does not change at all if the pressure input is in the deadband region. 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 e_tripped.

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

6.2 Design and Formal Translation

An example design (Fig. 3, Sect. 2.2) uses several built-in IEC 61131-3 FBs to specify the functional behaviour and 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_PressParmTrip = FALSE* is equivalent to *c_PressParmTrip = e_tripped*.

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. 4.

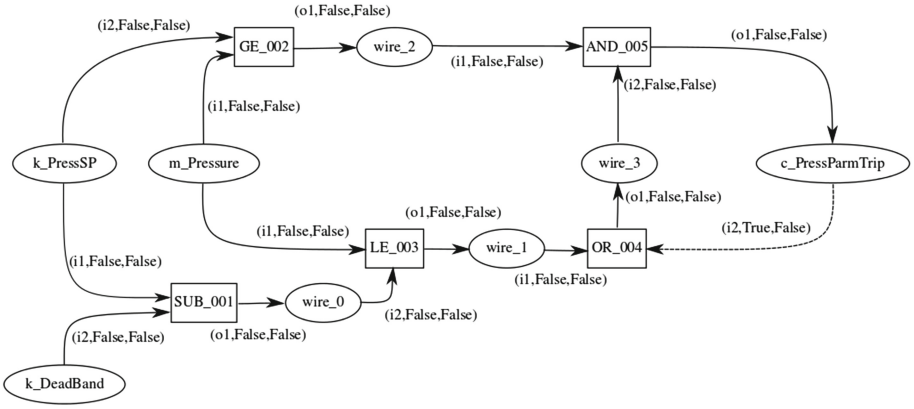

Fig. 4. Labelled directed graph for *c_PressParmTrip*

The translation rules are further applied and the resulting PVS code is illustrated in Fig. 5. Using the input and output identification rules from Eqs. (25) and (26), inputs and outputs of the graph in Fig. 4 are respectively: *k_PressSP*, *k_DeadBand*, *m_Pressure*, and *c_PressParmTrip*. The existential identification rule from Eq. (27) yields the internal variables: *wire_0*, *wire_1*, *wire_2*, and *wire_4*. Lastly, the conjunction of internal blocks SUB, GE, LE, OR[9] and AND completes the expression as shown[10].

6.3 Verification

CE-1001-STD [7] 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

[9] The underscore (...._) is used for generated names that conflict with PVS keywords.
[10] The FBD is formalized over a discrete time series of equally distributed samplings, i.e., ticks. The *pre* operator returns the previous time sample.

```
c_PressParmTrip ( k_PressSP : [ tick → DINT ] ,
                  k_DeadBand : [ tick → DINT ] ,
                  m_Pressure : [ tick → DINT ] ,
                  c_PressParmTrip : [ tick → BOOL ] )
                ( t : tick ) : bool =
  if init ( t ) then
   c_PressParmTrip ( t ) = false
  else
   exists ( wire0 : [ tick → DINT ] ,
            wire1 : [ tick → BOOL ] ,
            wire2 : [ tick → BOOL ] ,
            wire3 : [ tick → BOOL ] ) :
    AND_ ( wire2 ,
           wire3 ,
           c_PressParmTrip )( t ) and
    GE ( m_Pressure ,
         k_PressSP ,
         wire2 )( t ) and
    LE ( m_Pressure ,
         wire0 ,
         wire1 )( t ) and
    OR_ ( wire1 ,
          lambda ( t : noninit_elem ) :
           c_PressParmTrip ( pre ( t ) ) ,
          wire3 )( t ) and
    SUB ( k_PressSP ,
          k_DeadBand ,
          wire0 )( t )
  endif
```

Fig. 5. Generated PVS for *c_PressParmTrip*

corresponding functions in the requirements using mathematical techniques or rigorous argument. SDV uses a specialization of the four variable model [12] to confirm the satisfaction of Eq. (33).

$$OUT \circ SOF \circ IN \vdash REQ \tag{33}$$

For the purposes of our example, *REQ* is the TE from Fig. 2 plus other supporting information (not shown) that defines the monitored and controlled variables, the constants, and the enumerated types. *SOF* is the FBD from Fig. 3 plus 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_PressParmTrip* is shown in Sect. 6.2). Our verification was performed in PVS using *cond* expressions to specify the requirements [18]. We then created a PVS specification containing a theorem in the form of Eq. (33). By running PVS, we discovered an unprovable

sequent that prevented us from discharging the proof. Upon investigation, we recognize the design failed to add a negation to the first input of the AND block. 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[11] for SDV on the DNGS SDS1 TC replacement project helped identify design pattern inconsistencies 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 user interaction with PVS to discharge the proof.

7 Related Work

IEC 61131-3 provides definitions for five PLC languages[12] and various research work has produced formalization and verification of PLC programs. In terms of the formal verification 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.

In the case of model checking, [8] 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. [15] transforms FBD descriptions to its logically equivalent Uppaal models that perform the verification of safety applications in the industrial automation domain. [5] translates ST and FBD into a synchronized data-flow language SIGNAL to compile and reason about the verification of specifications. In the case of theorem proving, [3] uses Coq to check the correctness of SFC programs, which is automatically generated from a graphical front-end. [16] formalizes PLC programs using higher-order logic and uses HOL to discharge safety properties. Also, [14] presents an algebraic approach to verify PLC programs.

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

[11] The approach was qualified using a combination of trial use, inspection and acceptance testing.

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

novel in that: (1) we translate a FBD design to a formal PVS model; and (2) the resulting PVS model can be verified against TE-based requirements input to PVS.

8 Conclusion and Future Work

In this paper, we have extended the work presented in [10] 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 requirements documented using TEs, and non-conformance between a FBD design and its requirements.

As on-going and future work, we first aim to improve our translation rules using PVS to provide more precision for potential tool designers. Secondly, we are currently formalizing proof scripts to increase the level of automation, which has potential application in other industrial domains, e.g., aerospace. Lastly, we plan to extend our formalization technique to other IEC 61131-3 compliant programming 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. We would also like to thank Lucian Patcas for his thorough review.

References

1. IEEE 7–4.3.2: Standard for Digital Computers in Safety Systems of Nuclear Power Generating Stations (Revision of IEEE Std 7–4.3.2-2003). The Institute of Electrical and Electronics Engineers (IEEE) (2010)
2. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Special Committee 205 of RTCA (2011)
3. Blech, J.O., Biha, S.O.: On formal reasoning on the semantics of PLC using Coq. CoRR abs/1301.3047 (2013)
4. IEC: 61131–3 Ed. 3.0 en: 2013: Programmable Controllers – Part 3: Programming Languages. International Electrotechnical Commission (2013)
5. 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)
6. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Sci. Comput. Program. **75**(11), 980–1000 (2010)

7. Joannou, P., Harauz, J., Viola, M., Cirjanic, R., Chan, D., Whittall, R., Tremaine, D., Moum, G.: Standard for Software Engineering of Safety Critical Software. CANDU Computer Systems Engineering Centre of Excellence Standard CE-1001-STD Rev. 3 (2014)
8. Németh, E., Bartha, T.: Formal verification of safety functions by reinterpretation of functional block based specifications. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 199–214. Springer, Heidelberg (2009)
9. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
10. 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)
11. 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)
12. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
13. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Trans. Software Eng. **20**, 948–976 (1994)
14. Roussel, J.M., Faure, J.: An algebraic approach for PLC programs verification. In: 6th International Workshop on Discrete Event Systems, pp. 303–308 (2002)
15. 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)
16. Völker, N., Krämer, B.J.: Automated verification of function block-based industrial control systems. Sci. Comput. Program. **42**(1), 101–113 (2002)
17. 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)
18. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)