Converting IEC 61499 Function Blocks to PRISM

Zeeshan Ejaz Bhatti

August 4, 2016

Technical Report No. 689
Department of Electrical and Computer Engineering
The University of Auckland
Auckland, New Zealand

Supervisor:	Dr. Partha S. Roop
Advisor:	Dr. Roopak Sinha

Last Updated: August 4, 2016

1 Introduction

IEC 61499 [1] is a standard for designing industrial automation systems, proposed as a possible successor of IEC 61131-3. It adopts a component-oriented approach for designing systems using reusable software modules, named *function blocks* (FBs). *Basic* FBs model execution behaviours using Moore-like state machines called *execution control charts* (ECCs), whereas interconnected function block networks (FBNs) are used to model complex behaviours. IEC 61499 enables *model-driven design* [2] as closed-loop systems, where abstract models of a *plant*, modelling the physical processes being controlled, and a *controller* that controls the plant, are iteratively refined until an acceptable level of detail is captured. Models and structures defined by IEC 61499 must be accompanied with compatible semantics to enable the model executability. However, several execution semantics are available for IEC 61499 [3]. BlokIDE and its companion tool-chain adopts the *synchronous execution semantics* [4], which give IEC 61499 an unambiguous, deterministic, and dead-lock free execution behaviour.

In this report, we present a formal structure for IEC 61499 [1], which is adopted from [5]. This formalism is used for devising a set of sound transformation rules that convert function blocks into Markov decision processes [6] in a semantics-preserving manner. The motivation behind converting function blocks to Prism language is to develop a foundation for model-based safety assessment framework (MBSA) of systems designed using IEC 61499 e.g., for quantitative risk assessment.

We chose the Prism language [6] for representing the generated Markov decision processes, which is a probabilistic model checker for quantitative analysis of stochastic systems. It supports the Prism language [6] that is used for system and model specification in the form of variables and probabilistic commands. Using these transformation rules, a given FBN can be automatically converted into a behaviourally equivalent Prism model. This generated model can then be subjected to automated verification of probabilistic CTL (PCTL) properties [7]. The soundness of the transformation rules ensure that the result of this verification is also sound for the given FBN.

2 The Boiler Control System

We present a boiler controller implemented as an embedded control system with hardware components and software control. This system can be understood from its *piping and instrumentation diagram* (P&ID) presented in Figure 1. It consists of a cylinder mounted with a pressure sensor. The contents of the cylinder are heated using a heat exchanger (e.g., a gas burner). The pneumatic control of the heat-exchanger allows altering the amount of heat being transferred to the cylinder. A pressure relief control valve allows reducing the pressure of the boiler in case the pressure rises above the desired value. Flow indicators (FI) are mounted in 1-out-of-2 (1*oo*2) redundant fashion to confirm the pressure relief. The desired behaviour for this system is to boil the contents of the cylinder.

For the purpose of simplicity, we restrict our discussion around the specific scenario where

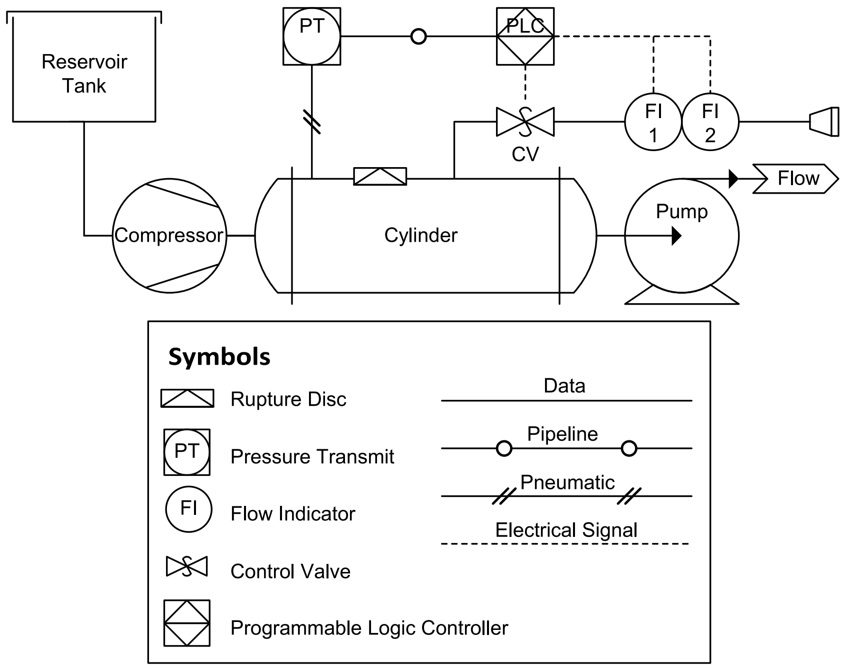


Figure 1: Piping and instrumentation diagram (P&ID) of a boiler

pressure increases beyond the desired value and the pressure relief valve is to be opened by the controller. This scenario imposes a safety requirement and hence, a corresponding safety function is to be implemented. From the perspective of functional safety, two main aspects have to be considered in this specific scenario namely, the reliability of physical components (pressure transmitter (PT), programmable logic controller (PLC), flow-indicators (FI)) and correctness of the safety function implemented as a software component. Here, achieving safety involves addressing the reliability of the physical components of the system and correctness of the corresponding software. In the following sections, we use this system as an illustrative example to present the transformation rules.

The function block network implementation of the boiler system presented in Figure 2 uses the *model-view-controller* (MVC) design pattern [8] consisting of a plant-model comprising models of the valve, pressure transmitter and flow indicators, as well as the controller. Optionally, a *view* can be connected to the network for visual monitoring and simulation. The controller reads inputs from the pressure transmitter and based on a threshold value decides whether to open a control valve for pressure relief. In order to avoid the pressure over-run hazard, the controller must also monitor the flow indicators (relief input variable) and sound an alarm (warning output variable) if the flow indicators do not report expected inputs.

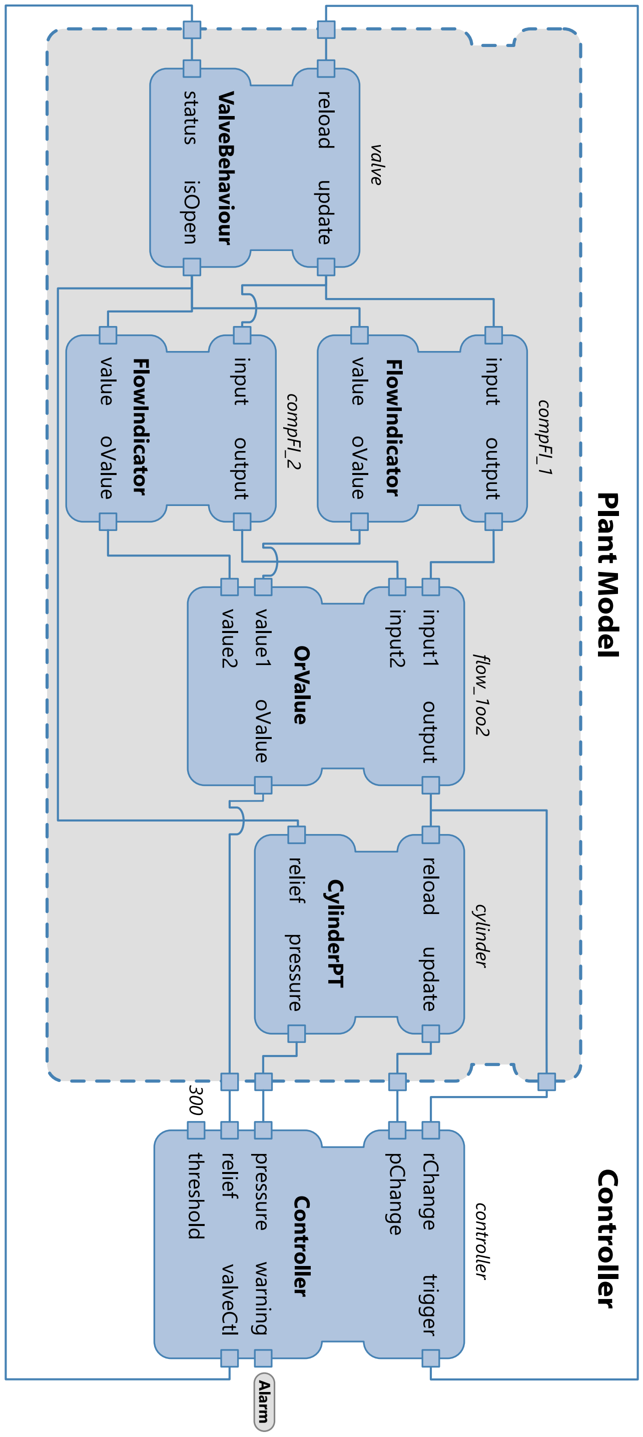


Figure 2: IEC 61499 implementation of the boiler control system

3 Formalisation of Function Blocks

IEC 61499 provides two types of *function blocks* (FBs) for developing complex control software. *Basic FBs* are the smallest units of execution, whereas *Composite FBs* encapsulate networks of function blocks. All FBs have *interfaces* that expose their respective inputs and outputs as defined below.

Definition 3.1 (Function Block Interface [5])**.** *A function block interface $\mathcal{I}$ is a tuple such that, $\mathcal{I} = \langle E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}, V_O^{\mathcal{I}} \rangle$ where, $E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}$ and $V_O^{\mathcal{I}}$ are finite sets of input events, input variables, output events and output variables respectively. Figure 2 shows interface of several function blocks, for example, the* Controller *function block, with a set of inputs and outputs:*

- $E_I^{\mathcal{I}} = \{\textsf{rChange}, \textsf{pChange}\}$
- $V_I^{\mathcal{I}} = \{\textsf{pressure}, \textsf{relief}, \textsf{threshold}\}$
- $E_O^{\mathcal{I}} = \{\textsf{trigger}\}$
- $V_O^{\mathcal{I}} = \{\textsf{warning}, \textsf{valveCtl}\}$

3.1 Formalisation of Basic Function Blocks

Basic function blocks (BFBs) implement their behaviour using Moore-type finite state machines called execution control charts (ECCs). The ECC of a BFB accepts inputs and emits outputs through the encapsulating FB interface. The accepted input variable values can be processed using textual blocks of code called *algorithms*, which may be executed upon entering a state and new values may be generated for the output variable values. For the purpose of internal computations and hidden data, *local variables* may be used. Algorithms and local variables are formally defined as the local declaration of a BFB. These definitions are adopted from [5] and are presented as following.

Definition 3.2 (Local Declaration)**.** *Local declaration of a BFB over an interface $\mathcal{I}$ is a tuple $L^{\mathcal{I}} = \langle V_L^{\mathcal{I}}, A_L^{\mathcal{I}} \rangle$ where, $V_L^{\mathcal{I}}$ is the set of internal variables and $A_L^{\mathcal{I}}$ is the set of algorithms operating over $V_L^{\mathcal{I}}$, $V_I^{\mathcal{I}}$ and $V_O^{\mathcal{I}}$.*

Definition 3.3 (Algorithm)**.** *An algorithm in a local declaration is a finite sequence of statements (ρ_0, ρ_1, $\dots$, ρ_n) that operate over available variables $V_L^{\mathcal{I}}, V_I^{\mathcal{I}}, V_O^{\mathcal{I}}$ i.e., local variables as well as input and output variables.*

Algorithms in IEC 61499 are implemented in external host languages, such as the “C” language. The structure of algorithms is simplified by abstracting away all branching statements and modelling them as transitions. This leaves the algorithms to contain only increment/decrement and assignment statements that are built using Boolean/numeric literals and expressions. Therefore, the grammar for an algorithm is given in Backus-Naur form as following.

Table 1: BNF notation for algorithms

```
<algorithm> := { <statement> }

<statement> := <ident> = <expr> ;
             | <ident> = ( <bexpr> ) ? <expr> : <expr> ;
             | <ident> ( += | -= | *= ) <expr> ;
```

Definition 3.4 (Basic Function Block)**.** *A basic function block is a tuple* $\texttt{BFB} = \langle \mathcal{I}, L^{\mathcal{I}}, \texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}} \rangle$ *where,* $L^{\mathcal{I}} = \langle V_L^{\mathcal{I}}, A_L^{\mathcal{I}} \rangle$ *is a local declaration over interface* $\mathcal{I} = \langle E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}, V_O^{\mathcal{I}} \rangle$ *and,* $\texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}} = \langle Q, q_0, X, T \rangle$ *is the execution control chart where:*

- Q *is a finite set of execution control states, with* $q_0 \in Q$ *as the initial state.*
- $X : Q \to (A_L^{\mathcal{I}} \cup E_O^{\mathcal{I}})^*$ *is the action function that assigns a finite sequence of algorithms and output events to a given state* $q \in Q$.
- $T : Q \to 2^{(E_I^{\mathcal{I}} \,\cup\, \{\texttt{true}\}) \,\times\, \mathcal{B}(\hat{V}) \,\times\, Q}$ *is the transition function where* $\hat{V} = (V_I^{\mathcal{I}} \cup V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}})$ *is the set of input, output and internal variables. Here, the notation* $(E_I^{\mathcal{I}} \cup \{\texttt{true}\})$ *denotes set of all input events including the always present* `true` *event, and* $\mathcal{B}(\hat{V})$ *is the set of all Boolean expressions over all variables. Furthermore, the inputs are read from the previous tick, which results in the flow of events and variables between function blocks in a unit-delayed fashion as proposed by the synchronous execution semantics [9]. The notation* $t = (q, e, b, q')$ *represents individual transitions* $t \in T(q)$. *For every* $q \in Q$, $T(q)$ *is always an ordered set, i.e., for any two elements* $t_1 = (q, e_j, b_j, q_j)$, $t_2 = (q, e_k, b_k, q_k) \in T(q)$ *we have* $\forall q \in Q : (t_1 > t_2) \vee (t_1 < t_2)$. *We capture this order of transitions graphically using the notation* **<n>***, where is* **n** *is the index of an element in the order set.*

Figure 3 presents the execution control chart of the Controller function block. Using this example for illustration, we have the following elements.

- $A_L^{\mathcal{I}} = \{$incCounter, openValve, resetCounter, setAlarm, closeValve$\}$
- $Q = \{$DO_OPEN, OPENED, DO_CLOSE, CLOSED, ALARM$\}$
- $q_0 =$ CLOSED
- X is the action function that maps states to respective action sets. For example, it maps the DO_CLOSE state to $\{$closeValve, incCounter, trigger$\}$. Where closeValve and incCounter are the algorithms to be invoked, and trigger is the output event to be emitted.
- T is the transition function that maps states to ordered set of respective egress tran-

sitions. For example it maps the DO_CLOSE state to (t_0, t_1, t_2) such that:

$$t_0 = (\mathsf{DO\_CLOSE}, \mathsf{pChange}, \mathsf{!relief}, \mathsf{CLOSED})$$
$$t_1 = (\mathsf{DO\_CLOSE}, \emptyset, \mathsf{counter} \ < 10, \mathsf{DO\_CLOSE})$$
$$t_2 = (\mathsf{DO\_CLOSE}, \mathtt{true}, \emptyset, \mathsf{ALARM})$$

3.2 Synchronous Execution of BFBs

Execution of a BFB under the adopted synchronous execution semantics [9] is performed in a step-by-step manner, where each step executes in logical time called a *tick*. During each tick, a given BFB reads inputs, updates its current state and execute state entry actions as depicted in Algorithm 1. Here, lines $2-4$ ensure that execution begins from the initial state q_0. Line 6 loads inputs from the encapsulating function block interface, such that the values are loaded from the previous tick as indicated by the keyword pre. Lines $8-17$ begin to iterate over a sorted set of all transitions from the current state, however, this iteration stops as soon as an enabled transition is located and processed. The result of this pre-emptive iteration over an ordered set induces a *priority* such that a higher order transition has a higher priority of getting selected. This processing of the enabled transition entails three steps. Firstly, the current state is updated to point towards the successor state of the transition (line 10). Secondly, algorithms in the state-entry actions of the new current state are executed (line 12). Thirdly, any output events in the state-entry actions are emitted through the function block interface (line 14). This emission also updates the value of the output variables reflect any change that may have been caused by the algorithm executions. Finally, the loop is aborted to make sure that at most one transition is taken (line 15). This concludes one tick process of a BFB.

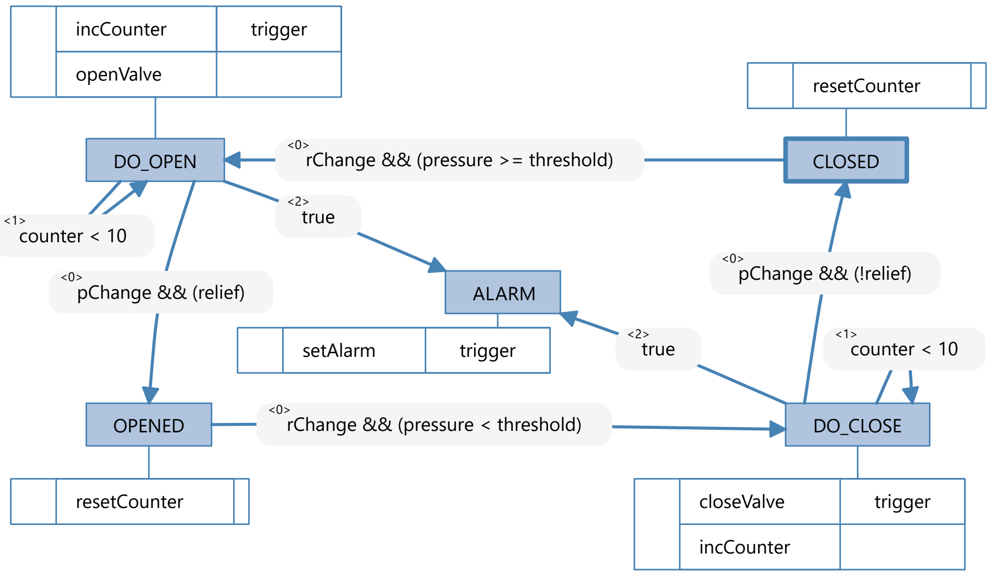


Figure 3: Execution Control Chart of the Controller function block

3.3 Formalisation of Composite Function Blocks

Composite function blocks (CFB) are similar to BFBs except that their behaviour is implemented by an encapsulated network of function blocks. This function block network (FBN) may contain several instances of various function blocks including both BFBs as well as CFB. The exposed IOs of these instances is connected through virtual *wire connections* indicating the flow of events and variables from outputs to inputs of the respective blocks. Figure 2 presents the FBN of the boiler control system, which contains several function block instances including two instances (compFI_1 and compFI_2) of the FlowIndicator function block type. We formally define FBN as following.

Definition 3.5 (Function Block Network)**.** *A function block network is a tuple,* $\texttt{FBNetwork} = \langle \texttt{FBs}, C_{\texttt{e}}, C_{\texttt{v}} \rangle$ *where,*

- $\texttt{FBs} = \{\texttt{FB}_1, \texttt{FB}_2, \cdots, \texttt{FB}_n\}$ *is a finite set of function block instances. A function block instance is a pair* $\texttt{FB}_i = \langle name_i, \texttt{FBT}_i \rangle$ *where* $\texttt{FBT}_i$ *is an FB type with an interface* $\mathcal{I}_i = \langle E_I^{\mathcal{I}_i}, V_I^{\mathcal{I}_i}, E_O^{\mathcal{I}_i}, V_O^{\mathcal{I}_i} \rangle$ *and* $name_i$ *is a unique identifier within the scope of* $\texttt{FBNetwork}$ *and is called instance name.*

- $C_{\texttt{e}} \subseteq (\bigcup_{i=1}^{n} \texttt{FB}_i.E_O^{\mathcal{I}_i}) \times (\bigcup_{i=1}^{n} \texttt{FB}_i.E_I^{\mathcal{I}_i})$ *is the set of event connections between the instances of the network.*

- $C_{\texttt{v}} \subseteq (\bigcup_{i=1}^{n} \texttt{FB}_i.V_O^{\mathcal{I}_i}) \times (\bigcup_{i=1}^{n} \texttt{FB}_i.V_I^{\mathcal{I}_i})$ *is the set of variable connections between the instances of the network. Variable connections are restricted such that for any two distinct variable connections* $C_{v1} = (src_1, dest_1), C_{v2} = (src_2, dest_2) \in C_{\texttt{v}}$ *,* $dest_1 \neq dest_2$ *i.e., an input variable cannot read from multiple sources. Furthermore, any event or*

Algorithm 1: Execution of a single *tick* of a Basic Function Block

```
Function FBTick(BFB):
    if cState = $null$ then
        cState ← BFB.$q_0$
    /* Load input events and variables from previous tick */
    loadInput(pre $E_I^{\mathcal{I}}$, pre $V_I^{\mathcal{I}}$);
    /* Iterate over the ordered set of transitions */
    for t ∈ getTransitions(cState) do
        if isEnabled(t) then
            cState ← getSuccessor(t);
            /* Execute algorithms of the successor state */
            executeAlgos (cState);
            /* Emit output events and variables */
            emitOutput($E_O^{\mathcal{I}}$, $V_O^{\mathcal{I}}$);
            return;
    return;
```

variable generated by an instance $\mathtt{FB}_i$ *is only available for reading in the next tick. This imposes a unit-delayed communication between FB instances, thus making the order of* FBs *irrelevant as the order of execution of FB instances has no effect on their behaviour. This concept is well-known in synchronous programming languages such as* Esterel *[4].*

The boiler system FBN in Fig. 2 contains a set of function block instances and a set of wire connections i.e., the following.

- FBs = {valve, compFI_1, compFI_2, flow_1oo2, cylinder, controller}
- C_{e} has several pairs representing event-connections for example (cylinder::update, controller::pChange) and (comFI_1::output, flow_1oo2::input1) etc.
- C_{v} has several pairs representing variable-connections for example (cylinder::pressure, controller::pressure) and (comFI_1::oValue, flow_1oo2::value1) etc.

A given FBN can be encapsulated by a function block interface thus forming a composite function block (CFB). The inputs and outputs of the encapsulated FBN are exposed through proxy connections. This encapsulation results in a provision for hierarchy in the IEC 61499 system model such that, a composite function block may become part of a higher level function block network by exposing its encapsulated IO. Figure 4 shows an instance of BFB named bfb1 encapsulated inside a composite function block cfb. The inputs and outputs of the encapsulated block are exposed through proxy connection and connected to in a higher level FBN for example, bfb1::oVar is connected to bfb2::iVar through a proxy output cfb::oVar.

Definition 3.6 (Composite Function Block)**.** *A composite function block is a tuple,* $\mathtt{CFB} = \langle \mathcal{I}, \mathtt{FBNetwork}, P, \rangle$ *where,* $\mathcal{I} = \langle E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}, V_O^{\mathcal{I}} \rangle$ *is the interface that encapsulates the function block network* $\mathtt{FBNetwork} = \langle \mathtt{FBs}, C_{\mathsf{e}}, C_{\mathsf{v}} \rangle$*. Also, a set of proxy connections exist between the interface and the encapsulated FBN, namely* $P = P_{E_I^{\mathcal{I}}} \bigcup P_{E_O^{\mathcal{I}}} \bigcup P_{V_I^{\mathcal{I}}} \bigcup P_{V_O^{\mathcal{I}}}$ *such that,*

- $P_{E_I^{\mathcal{I}}} \subseteq (E_I^{\mathcal{I}}) \times (\bigcup_{i=1}^{n} \mathtt{FB}_i.E_I^{\mathcal{I}_i})$ *is the set of input event proxy connections.*

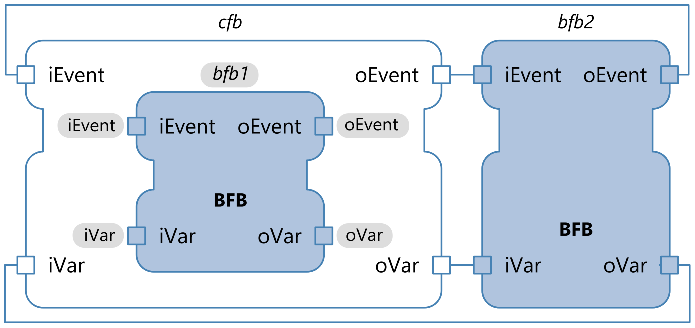


Figure 4: A composite function block

- $P_{E_O^{\mathcal{I}}} \subseteq (\bigcup_{i=1}^{n} \mathtt{FB}_i.E_O^{\mathcal{I}_i}) \times (E_O^{\mathcal{I}})$ *is the set of output event proxy connections.*
- $P_{V_I^{\mathcal{I}}} \subseteq (V_I^{\mathcal{I}}) \times (\bigcup_{i=1}^{n} \mathtt{FB}_i.V_I^{\mathcal{I}_i})$ *is the set of input variable proxy connections.*
- $P_{V_O^{\mathcal{I}}} \subseteq (\bigcup_{i=1}^{n} \mathtt{FB}_i.V_O^{\mathcal{I}_i}) \times (V_O^{\mathcal{I}})$ *is the set of output variable proxy connections. Variable proxy connections are restricted such that for any two distinct variable connections* $p_1 = (src_1, dest_1), p_2 = (src_2, dest_2) \in P_{V_O^{\mathcal{I}}}$, $dest_1 \neq dest_2$ *i.e., an output variable cannot proxy multiple output variable sources.*

3.4 Synchronous Execution of CFBs

Under the *synchronous execution semantics* [9], execution of a given function block network `FBNetwork` is performed in logical time intervals called *ticks*. During each tick, each FB in the given FBN is executed as described in Algorithm 2. Similar to the execution to BFBs, CFBs load inputs from their encapsulating interface from the previous tick (see line 3). Lines 5 – 14 iterate over all instances in the function block network and execute a corresponding tick one by one. Here, if the instance is of a BFB type, the execution is performed using Algorithm 1 (line 9) whereas, if the instance is of a CFB type, recursion of Algorithm 2 is performed (line 12). Please note that the depth of recursion tree is same as the levels of hierarchy in the given CFB.

Observation 3.1 (Flattening a Composite Function Block)**.** *An instance of CFB can be replaced with an instantiation of its encapsulated FBN such that, all proxy connections are replaced by directly connecting the source and target events and variables i.e., the following.*

Algorithm 2: Execution of a single *tick* of a Composite Function Block

```
1 Function CFBTick(CFB):
      /* Load input events and variables from interface              */
2     loadInput(pre $E_I^{\mathcal{I}}$, pre $V_I^{\mathcal{I}}$);
      /* Iterate over the set of encapsulated BFBs and CFBs          */
3     for FB $\in$ FBN do
          /* Execute one tick on each FB                             */
4         if FB is Basic then
              /* Execute BFBTick function defined previously         */
5             BFBTick(FB);
6         else
              /* Recursively execute CFBTick function                */
7             CFBTick(FB);
      /* Emit output events and variables                            */
8     emitOutput($E_O^{\mathcal{I}}$, $V_O^{\mathcal{I}}$);
9     return;
```

- $\forall p_{E_I^{\mathcal{I}}} \in P_{E_I^{\mathcal{I}}} : p_{E_I^{\mathcal{I}}}.\texttt{src} \rightarrow \texttt{FB}_i.E_I^{\mathcal{I}_i}$, *replaces input event proxy connections.*
- $\forall p_{E_O^{\mathcal{I}}} \in P_{E_O^{\mathcal{I}}} : \texttt{FB}_i.E_O^{\mathcal{I}_i} \rightarrow p_{E_I^{\mathcal{I}}}.\texttt{dest}$, *replaces output event proxy connections.*
- $\forall p_{V_I^{\mathcal{I}}} \in P_{V_I^{\mathcal{I}}} : p_{V_I^{\mathcal{I}}}.\texttt{src} \rightarrow \texttt{FB}_i.V_I^{\mathcal{I}_i}$, *replaces input variable proxy connections.*
- $\forall p_{V_O^{\mathcal{I}}} \in P_{V_O^{\mathcal{I}}} : \texttt{FB}_i.V_O^{\mathcal{I}_i} \rightarrow p_{V_O^{\mathcal{I}}}.\texttt{dest}$, *replaces output variable proxy connections.*

Recursively applying this flattening scheme to a nested hierarchy of CFBs results in a behaviourally equivalent FBN that only consists of instances of BFBs. In fact, any given hierarchy of CFBs can be automatically flattened into an FBN that only consists BFBs. Therefore, without loss of generality we can assume that FBNs only contain instances of BFBs.

4 Formalisation of Prism Models

The Prism language provides syntax for modelling of various type of stochastic systems such as *discrete-time Markov chains* (`dtmc`), *continuous-time Markov chains* (`ctmc`), *Markov decision processes* (`mdp`), *probabilistic automata* (`pa`), and *probabilistic timed automata* (`pta`) [6]. This wide range of semantics allow modelling various types of problems, however, in the context of our work we only discuss discrete models e.g., Markov decision processes (MDP). This selection is compatible with the synchronous execution of function blocks in logical time units i.e., *ticks*. In this section, we present a basic introduction to Prism language and Prism models under MDP semantics. Furthermore, we devise a formalisation for its representation, composition, and execution. Later in this text, we use this formalisation for the purpose of automatic transformation of function blocks into Prism models.

A Prism module with two variables, `status` and `count`, is presented in Table 2. This module has a Boolean variable named status with an initial value `false`. An integer variable named counter has an initial value of 0, and is bounded by a range between [0, 100]. It contains two commands that operate over these variables to implement an input counter. The first command waits for an action `i` to read an external Boolean variable named input, upon which it increments the counter. The second command executes as soon as the value

Table 2: An example of a Prism module

```
module input_counter
        //variables
        counter : [0..100] init 0;
        status : bool init false;

        //commands
        [i] (input) -> (counter' = (counter + 1)) & (status' = false);
        [ ] (counter >= 99) -> (counter' = 0) & (status' = true);
endmodule
```

of the counter becomes 99, upon which it resets the counter and set the value of the `status` flag indicating an overflow.

In the following text, we present a basic introduction to PRISM models and their underlying formalism i.e., the structure and semantics of the PRISM language [6]. A PRISM model (see Definition 4.1) consists of modules (see Definition 4.2), which comprises of variables and commands (see Definition 4.3). We define these constructs as following.

Definition 4.1 (PRISM Model)**.** *A* Prism *model is tuple,* $\mathcal{M} = \langle \text{type}, G, M, A\rangle$*, where* type *is an element from the set of all* Prism *types* $\{\texttt{dtmc}, \texttt{ctmc}, \texttt{mdp}, \texttt{pa}, \texttt{pta}\}$*.* G *is the set of global constructs (constants* G_c *and variables* G_v*) i.e.,* $G = \{G_c, G_v\}$*).* M *is the composition of the finite set of* Prism *modules* $\{M_0, M_1, ..., M_k\}$ *that synchronise using elements from set of all action labels* A*.*

Due to the scope of our work, we only require the type `mdp` (Markov Decision Processes) that allows non-deterministic and probabilistic transitions. Furthermore, the proposed approach does not use global variables and only utilises global constants to support the concept of parameters in function block, such as the pressure threshold in the boiler control system. Also, we utilise action labels only for lock-step transitions to mimic the synchronous execution of function blocks. Thus, in our context a PRISM model can be presented as following.

$$\mathcal{M} = \langle \texttt{mdp}, G_c, M, \{\texttt{tick}\}\rangle \tag{1}$$

The PRISM module presented in Table 2 comprises of various variables and commands e.g., it contains a variable named `counter`, and command is presented on Line 7. We formalise the structure pf a PRISM module as following.

Definition 4.2 (PRISM Module)**.** *A* Prism *module is a tuple* $M_i = \langle V_i, C_i\rangle$*, where* V_i *is the set of local variables of the module* M_i *and* C_i *is the set of commands in* M_i*. A command* $c \in C_i$ *may read from the set of all variables in a the parent* Prism *model* $\left(i.e., \mathcal{V} = \bigcup_{i=0}^{n} V_i\right)$*, but can only write to local variables* $v \in V_i$*.*

The limitations of the PRISM model checker puts some restrictions on the use of variables and possible operations on the values. Primarily, these limitations attempt to avoid state-space explosion problem during the verification process. The limitations are listed as following.

- A variable can either be a `Boolean` type or `Integer` type. These values map to IEC 61499 types `BOOL`, and any of the integer types e.g., `INT`, `DINT`, `USINT`, or `BYTE`.
- Integer type variables must define an initial, a minimum and a maximum value. The execution of PRISM model begins with initial variable values and during all possible executions, the value of the variable must not exceed the said range.
- Arithmetic operations are allowed on integer type variables except for division. This restriction is often imposed by many model checkers to simplify the verification process.

Definition 4.3 (PRISM Command)**.** *A* Prism *command is a tuple $C = \langle a, g, U \rangle$, where:*

- *$a \in A$ is the action label of the command used for synchronisation with other modules, namely the* `tick`.
- *$g \in \mathcal{B}(\mathcal{V})$ is an element from the set of all Boolean expressions over all variables in the parent* Prism *model.*
- *$U = \{(\lambda_0, u_0), ..., (\lambda_n, u_n)\}$ is the set of probabilistic updates with λ probability values, such that for all $\lambda_i \in U, \lambda_i \geq 0$ and $\sum_{i=0}^{n} \lambda_i = 1$. For a given probabilistic update $(\lambda, u) \in U$, u is and update sequence and is of the form $u = (v'_i = expr_i) \wedge (v'_j = expr_j) \wedge \cdots \wedge (v'_n = expr_n)$. In an update sequence u, each element has two components, namely:*
  - *v_i : a variable to be updated $v_i \in V_k$, v_i being a local variable of* Prism *module M_k is writeable. Furthermore, in a given command $C \in M_k$, any variable $v \in V_k$ can be updated only once in the scope of u.*
  - *$expr_i \in \mathcal{E}(\mathcal{V})$, is an element from the set of all Boolean and arithmetic expressions over all variables in the parent* Prism *model.* Prism *uses the notion of delayed updates i.e., any expression evaluated in an i^{th} execution cycle will use values from updates made in the $(i-1)^{th}$ execution cycle.*

Definition 4.4 (PRISM Module Composition)**.** *The composition of a set of* Prism *modules $\{M_0, M_1, ..., M_k\}$ is also a* Prism *module i.e., $M = \langle V, C \rangle$, where V is the set of all variables of the* Prism *modules M_0, M_1, ..., M_k, and C is the set of commands created as the*

Table 3: Composition of two PRISM modules [6]

$$[P1]\ \frac{(\emptyset, g, U) \in M_i}{(\emptyset, g, U) \in M} \qquad [P2]\ \frac{(\emptyset, g', U') \in M_j}{(\emptyset, g', U') \in M}$$

$$[P3]\ \frac{(a, g, U) \in M_i}{(a, g, U) \in M}\ (a \notin A) \qquad [P4]\ \frac{(a', g', U') \in M_j}{(a', g', U') \in M}\ (a' \notin A)$$

$$[P5]\ \frac{(a, g, U) \in M_i \quad \textit{and} \quad (a, g', U') \in M_j}{(a, (g \wedge g'), (U \odot U')) \in M} \qquad (a \in A)$$

Where, $\odot$ is defined as:

$$\begin{aligned}
U &= \{(\lambda_0, u_0), (\lambda_1, u_1), ..., (\lambda_n, u_n)\} \\
U' &= \{(\lambda'_0, u'_0), (\lambda'_1, u'_1), ..., (\lambda'_{n'}, u'_{n'})\} \\
U \odot U' &= \{(\lambda_0 * \lambda'_0), (u_0 + u'_0)), ..., (\lambda_n * \lambda'_0), (u_n + u'_0)), \\
&\quad (\lambda_0 * \lambda'_1), (u_0 + u'_1)), ..., (\lambda_n * \lambda'_1), (u_n + u'_1)), \\
&\quad \vdots \qquad \vdots \qquad \vdots \\
&\quad (\lambda_0 * \lambda'_{n'}), (u_0 + u'_{n'})), ..., (\lambda_n * \lambda'_{n'}), (u_n + u'_{n'}))\}
\end{aligned}$$

result of rule-based composition. The composition of commands of two Prism *modules M_i and M_j is created by the following rules presented in Table 3. Here, rules $P1$ to $P4$ generate interleaved composition of the commands by individually adding them to the resultant module. In contrast, rule $P5$ handles the commands with matching action labels, which are composed in parallel by taking the conjunction ('$\wedge$') of the condition guards and concatenation ('$\odot$') of the probabilistic update sequences. An illustration of rule $P5$ is presented in Table 4 where two probabilistic commands from two different modules are combined together using concatenations of the probabilistic updates to form a new probabilistic command in the composition.*

Table 4: Composing probabilistic commands in PRISM modules

```
    [a] (m1A) -> 0.3 : (m1B'=false) + 0.7 : (m1B'=true);
```

```
    [a] (m2A) -> 0.7 : (m2B'=false) + 0.3 : (m2B'=true);
```

```
    [a] (m1A && m2A) -> 0.21 : (m1B'=false) & (m2B'=false)
                      + 0.49 : (m1B'=true)  & (m2B'=false)
                      + 0.21 : (m1B'=true)  & (m2B'=true)
                      + 0.09 : (m1B'=false) & (m2B'=true);
```

4.1 Execution of a Prism Module

The execution of a PRISM module is cyclic and in every cycle a PRISM module performs the steps depicted in Algorithm 3. The first step in this execution is to compute a set of enabled commands. For this purpose, line $3-8$ iterate over all commands of the PRISM module and evaluate its Boolean guard. This evaluation uses variables values from the previous cycle. Resultantly, a set of all enabled commands K is computed. If no command can be enabled, the cycle is completed as instructed by lines $9 - 11$. Otherwise, an enabled command is selected for execution (line 13) from the computed set. This selection is performed non-deterministically e.g., using a random number generator. Next, an update-pair is selected from the command (line 15) e.g., using a probabilistic simulation that respects the associated probability values. This update-pair consists of a set of update statements, which are then executed (line 17) to update the values of the variables for the next cycle. This concludes one cycle of execution of a PRISM module.

Algorithm 3: Execution of a single *cycle* of a Prism module

```
1  Function PrismCycle(M):
       /* iterate over all commands */
2      for $c_i \in M.C$ do
           /* Evaluate all enabled commands */
3          if evaluate($c_i$.g) then
4              $K \leftarrow (K \cup \{c_i\})$;
5      if isEmpty(K) then
6          return;
       /* Non-deterministically select an enabled command */
7      $c_k \leftarrow$ nSelect(K);
       /* Probabilistically select an update-pair from $c_k$ */
8      $u_k \leftarrow$ pSelect($c_k.U_k$);
       /* Execute update statements in $u_k$ */
9      execute($u_k$);
10     return;
```

5 Converting Function Blocks to Prism

The goal of this conversion is to create a Prism model which is semantically equivalent to the given FBN. The synchronous execution of FBs in an FBN has some similarities to the MDP semantics of Prism language. For example, the step-by-step synchronous execution can be emulated using the cyclic execution of Prism modules. Similarly, the unit-delayed transfer of events and variable values over wire connections of FBs may be emulated in Prism model where values of variables are read from the updates in the previous cycle. Because of Observation 3.1, we assume that every function block in the given FBN is a BFB. Therefore, the transformation maps every BFB instance to a corresponding Prism module as follows.

$$\texttt{BFB}_i \in \texttt{FBNetwork} \Longleftrightarrow M_i \in \mathcal{M} \tag{2}$$

We start with presenting an intuitive illustration of the Controller function block is presented in Figure 3. The corresponding generated Prism module is presented in Table 5. Here, the 8 transitions of the controller BFB are mapped to 8 respective Prism commands i.e., Lines 16 – 45. Additionally, we have an initialisation command on Line 14, and three generated self-loop commands on Lines 48 – 54.

Table 5: Generated PRISM module from the Controller function block, which was previously presented in Figure 3. Note that "_controller" postfix is omitted from variables names for readability.

```
module controller
    s : [-1..4] init -1;
    //s = {0, 1, 2, 3, 4} : {CLOSED, DO_OPEN, OPENED, DO_CLOSE, ALARM}

    //Generated from output events
    trigger : bool init false;

    //Generated from internal and output variables
    //Omitting '_controller' postix for readability
    warning : bool init false;
    valveCtl : bool init false;
    counter : [0..10] init 0;

    [t] (s=-1) -> (s'=0) & (trigger'=false) & (counter' = 0);

    [t] (s=0) & output_cylinder & (status_cylinder >= 300) ->
            (s'=1) & (trigger'=true) & (valveCtl' = true) &
            (counter' = (counter < 10) ? (counter + 1) : counter);

    [t] (s=1) & output_orValue & oValue_orValue ->
            (s'=2) & (trigger'=false) & (counter' = 0);

    [t] (s=2) & output_cylinder & (status_cylinder < 300) ->
            (s'=3) & (trigger'=true) & (valveCtl' = false) &
            (counter' = (counter < 10) ? (counter + 1) : counter);

    [t] (s=3) & output_orValue & (oValue_orValue = false) ->
            (s'=0) & (trigger'=false) & (counter' = 0);

    [t] (s=1) & (output_orValue = false) & (counter >= 10) ->
            (s'=4) & (trigger'=true);

    [t] (s=3) & ((output_orValue = false) | (oValue_orValue)) &
        (counter >= 10) ->
            (s'=4) & (trigger'=true);

    [t] (s=1) & ((output_orValue = false) | (oValue_orValue = false)) &
        (counter < 10) ->
            (s'=1) & (trigger'=true) & (valveCtl' = true) &
            (counter' = (counter < 10) ? (counter + 1) : counter);

    [t] (s=3) & ((output_orValue = false) | (oValue_orValue)) &
        (counter < 10) ->
            (s'=3) & (trigger'=true) & (valveCtl' = false) &
            (counter' = (counter < 10) ? (counter + 1) : counter);

    //Generated self-loops for emulating synchronous execution semantics
    [t] (s=0) & ((output_cylinder = false) | (status_cylinder < 300)) ->
            (s'=0) & (trigger'=false);

    [t] (s=2) & ((output_cylinder = false) | (status_cylinder >= 300)) ->
            (s'=2) & (trigger'=false);

    [t] (s=4) -> (s'=4) & (trigger'=false);
endmodule
```

A Prism module consists of a set of variables and commands i.e., $M_i = \langle V_i, C_i \rangle$ (see Definition 4.2). Thus, the conversion of each BFB of the given FBN entails systematic construction of variables and commands in the corresponding Prism module such that, the overall behaviour of the Prism model is equivalent to the FBN. The overall process of conversion is presented as a flow chart in Figure 5. The process begins with iterating over all BFBs and iteratively constructing Prism modules that contain variables and commands. In the following sections, we present the various concepts employed in this construction.

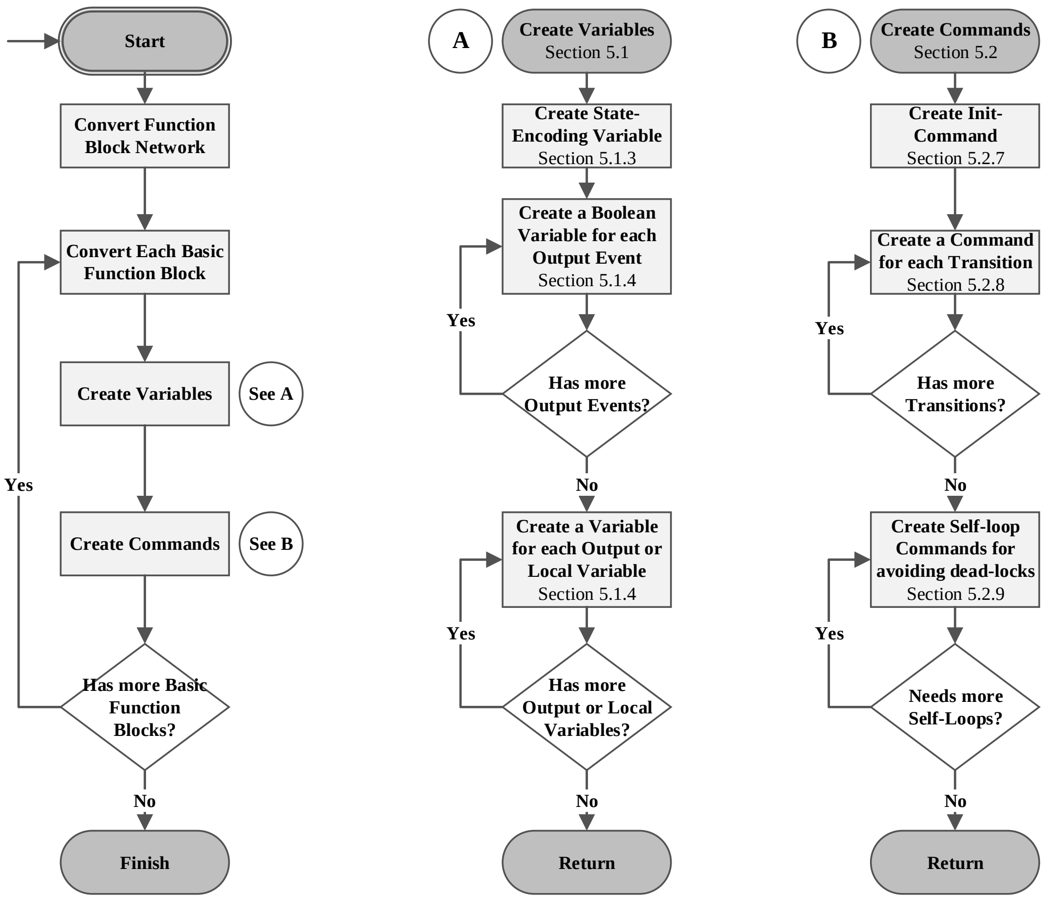

Start
Convert Function Block Network
Convert Each Basic Function Block
Create Variables
See A
Create Commands
See B
Has more Basic Function Blocks?
Yes
No
Finish
A
Create Variables Section 5.1
Create State-Encoding Variable Section 5.1.3
Create a Boolean Variable for each Output Event Section 5.1.4
Has more Output Events?
Yes
No
Create a Variable for each Output or Local Variable Section 5.1.4
Has more Output or Local Variables?
Yes
No
Return
B
Create Commands Section 5.2
Create Init-Command Section 5.2.7
Create a Command for each Transition Section 5.2.8
Has more Transitions?
Yes
No
Create Self-loop Commands for avoiding dead-locks Section 5.2.9
Needs more Self-Loops?
Yes
No
Return

Figure 5: The process of converting a FBN to a Prism model presented as a flow chart

5.1 Generating Variables

In this section we present discussion on generating variables in the Prism module. These variables follow a specific naming convention, which is ensured by the `rename` macro. This section also presents the boilerplate used for variable definitions in Prism syntax, which is used by the transformation rules $T1 - T3$ to generate state-encoding variable as well as other variables that correspond to output events, output / local variables of a given BFB.

5.1.1 Mapping Variable Names

The rename macro maps a given function block variable name to its equivalent in the generated PRISM module. This is performed by a rename mechanism, which ensures that the identifier names are unique by using the name of the corresponding PRISM module as a postfix. Using the example shown in Figure 6 for illustration, we have:

- rename(cfb.bfb1.oVar1) $=$ `oVar1_bfb1_cfb1`
- rename(bfb2.oEvent) $=$ `oEvent_bfb2`

5.1.2 The variable Boilerplate

generates a variable definition in the current PRISM module with a specified *name*, *type*, *range*, and *initial* value. The context of invocation determines the current PRISM module. We use the boilerplate notation:

$$\mathsf{variable}(\mathit{name}, \mathit{type}, \mathit{range}, \mathit{init}) \quad \textit{where,} \tag{B1}$$

- *name:* a valid identifier for PRISM variable names. The uniqueness of this variable is ensured by using the rename(*name*) macro.
- *type:* a valid PRISM variable type i.e., either integer (INT) or Boolean (BOOL).
- *range:* an ordered set of valid values for the given PRISM variable type i.e., an upper and lower bound for integer type variables $[u, l]$ where $u, l \in \mathcal{Z}$ and $u \geq l$. Whereas, the range for a Boolean variable is the set of Boolean constants $[\texttt{false}, \texttt{true}]$. Since the range of Boolean variables is statically defined, it may be omitted by using the *don't care* symbol $\top$.
- *init:* a valid initial value for the generated PRISM variable from the ordered set *range* i.e., $\mathit{init} \in \mathit{range}$.

We use the controller BFB (see Figure 3) to extract and present examples generating by the variable boilerplate as follows:

- variable(counter, INT, [0, 5], 1) $=$ `counter_controller : [0..5] init 1;`
- variable(trigger, BOOL, $\top$, `true`) $=$ `trigger_controller : bool init true;`

5.1.3 Encoding the States

The states of a given BFB are encoded using distinct integer values of a state-encoding variable s (see Table 5 Line 2).

$$\mathsf{variable}(s, \mathrm{INT}, [-1, \mathsf{len}(Q) - 1], -1) \tag{T1}$$

The state-encoding variable is defined as an integer type variable with a specified range of $[-1, \mathsf{len}(Q) - 1]$ where, $\mathsf{len}(Q)$ is the number of states in the given BFB. It uses -1 as its initial value, representing that the pre-initialisation state of a BFB. The purpose of adding this value is to enable semantically correct translation of Moore-type BFBs to Mealy-type PRISM modules. The macro valueOf manages the subsequent value mapping of the state-encoding variable using the declarative index (indexOf) of a given state in the corresponding ECC e.g., a value between 0 and $\mathsf{len}(Q) - 1$. Whereas, the value -1 is assigned for a special don't care element is used i.e., $q = \top$.

$$\mathsf{valueOf}(q) \quad = \quad \begin{cases} \mathsf{indexOf}(q) & q \in Q \\ -1 & q = \top \end{cases} \qquad s.t.,\ q \in \top \cup Q$$

Boolean guards over s induce *locations* in the generated PRISM module such that, the initialisation value -1 induces the *init-location* to perform the module initialisation. We use the valueOf macro for generating Boolean guards that induce locations in the generated PRISM module. Similarly, we use the valueOf macro for updating values of the state-encoding variable against a given successor ECState i.e., the following.

$$\begin{aligned} \mathsf{checkState(q)} &= (\text{“}\mathsf{s{=}{=}valueOf(q)}\text{”}) \\ \mathsf{updateState(q)} &= (\text{“}\mathsf{s{=}valueOf(q)}\text{”}) \end{aligned}$$

Consider the following examples taken from the controller BFB in Figure 3.

- (s==-1) \\ *for init-command*
- (s==2) \\ *induce location against OPENED ECState*
- (s=3) \\ *update location to DO_CLOSE ECState*

5.1.4 Encoding the Events and the Variables

Function blocks consist of two types of signals i.e., valued signals and pure events, PRISM language on the other hand, only contains valued signals. For the purpose of model transformation, events are encoded as Boolean variables as shown in Table 5 Line 6. The status of an event (*present* or *absent*) is mapped to the Boolean value of the corresponding variable (`true` = *present* and `false` = *absent*). The initial value of events under synchronous execution semantics is absent, which is depicted in the rule $T2$ as well.

$$e \in E_O^{\mathcal{I}} \implies \mathsf{variable}(e, \mathrm{BOOL}, \top, \mathtt{false}) \qquad (T2)$$

Similarly, rule *T3*-*T4* encode output and local variables respectively with the corresponding initial values of their BFB counterparts (see Table 5 Lines $9 - 11$).

$$v \in V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}} \implies \mathsf{variable}\big(v, \mathsf{typeOf}(v), \mathsf{rangeOf}(v), \mathsf{initOf}(v)\big) \quad where, \qquad (T3)$$

- **typeOf**: determines the type of a given IEC 61499 variable and maps it to either integer (INT) or Boolean (BOOL).

- **rangeOf**: returns the range of a given IEC 61499 variable, which is statically defined for Boolean variables as $[\texttt{false}, \texttt{true}]$. The integer type variables can define their own lower and upper bound values, however in case of missing information, we can use the values for short-integer namely, `SINT_MIN` and `SINT_MAX` constants as defined by ISO-C standard.
- **initOf**: determines the initial value of a given IEC 61499 variable that belongs the range specified above. However, in case of missing values, we assign 0 to be the initial value of an integer type variable, and `false` to be the initial value of a Boolean variable.

In addition to creating the state-encoding variable suing rule $T1$, we thus create variables corresponding to output events, output variables and local variables using rules $T1 - T3$.

5.2 Generating Commands

In this section, we present our approach to bridge the gap between syntax and semantics of IEC 61499 and PRISM language. In a PRISM module, commands are akin to its execution instructions. In the context of model to model transformation from function blocks to PRISM , these commands must emulate the synchronous execution semantics of the source structure. For this purpose, we define a set of macros that facilitate the generation of commands in the generated PRISM modules, and present the algorithms for generating the PRISM modules.

5.2.1 Encoding Wire-Connections

In FBNs, events and data flow from the outputs of FB instances to the inputs using explicit wire connections (Definition 3.5]). Because of the delayed-communication, the output emitted from a source FB instance is read by the destination FB instance in the next tick. The new values loaded by these wire connections is used for the purpose of (i) evaluating Boolean guards for enabling/disabling transitions and (ii) using values in execution of algorithms.

Emulation the reading new values from the source FB instances is straight forward in PRISM because any PRISM modules can read any variable directly from any other PRISM modules. Therefore, wire-connections in a given FBN can be encoded using direct read access on source PRISM module. The **sourceOf** exploits this global read-access to eliminate the redundant copies of output variables, which would have otherwise been required to emulate the wire-connections between of source and target function blocks. Consider the example presented in Figure 6. Here, we can use the sourceOf macro to maps a given input signal to a set of source signals using its wire-connections. This set of source signals may contain multiple elements for event-wire connections that allow multiple sources, or a single element in case of a variable-wire connection. The extracted source set is then used to construct an expression in PRISM syntax, for example multiple events in the source set can be arrange in a Boolean disjunction. Whereas, source sets for variable connections only contain a single

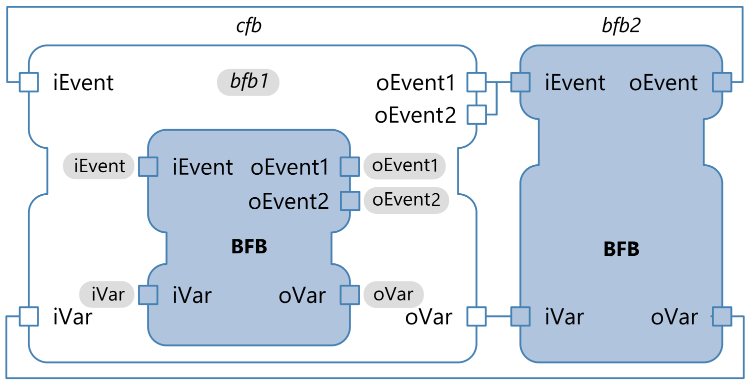


Figure 6: Lookup source of wire-connections for PRISM syntax generation

element i.e., name of the source variables, which can be used as a valid expression. Consider the example presented in Figure 6.

- sourceOf(cfb.bfb1.iEvent) = `(oEvent_bfb2)`
- sourceOf(cfb.bfb1.iVar) = `(oVar_bfb2)`
- sourceOf(bfb2.iEvent) = `(oEvent1_bfb1_cfb | oEvent2_bfb1_cfb)`
- sourceOf(bfb2.iVar) = `(oVar_bfb1_cfb)`

In the first two examples presented above, the source of the input was traced back to output signal of bfb2, which were then renamed using the rename macro. The third example presents a case where multiple sources are available for an input event. In this case, the sources are traced and renamed, and then combined to form a Boolean disjunction. Since input variables cannot have multiple sources, a combinator function (e.g., Boolean disjunction) is not required for them.

5.2.2 Ensuring Life-time of an Event

Synchronous execution semantics define the life-time of an event as 1-tick. In the generated PRISM module, we ensure this by explicit setting values of the corresponding Boolean variables in each execution cycle i.e., *present* $=$ `true` and *absent* $=$ `false`. The **setStatus** macro is used for this explicit encoding of the current status of output events, which takes two disjoint sets of output events to map their statuses either as present or absent.

$$\text{setStatus}(\mathcal{E}, \overline{\mathcal{E}}): \quad \mathcal{E} \rightarrow \texttt{true}, \ \overline{\mathcal{E}} \rightarrow \texttt{false} \quad s.t., \mathcal{E} \uplus \overline{\mathcal{E}} = E_O^{\mathcal{I}}$$

5.2.3 Encoding the State-Entry Actions

The state-entry actions in BFBs comprise a sequence of algorithms and output events (see Definition 3.4). These actions can be encoded by generating a set of assignment statements

PRISM syntax that emulated the said state-entry actions. For this we define a macro named stateActions as follows.

$$\mathsf{stateActions}(q) \;=\; \mathsf{setStatus}(\,\mathcal{E}_q, \overline{\mathcal{E}_q}) \cup \Gamma \;\; \textit{where,}$$

- $q \in Q$ is an ECState of a given BFB
- $\varepsilon_q = E_O^{\mathcal{I}} \cap X(q)$ is the set of output events emitted in the context of state-entry action for the ECState q.
- $\overline{\varepsilon_q} = E_O^{\mathcal{I}} / X(q)$ is the set of output events that are *not* emitted in the context of state-entry actions for the ECState q.
- $\Gamma q = \{\rho \mid \rho \in A_L^{\mathcal{I}} \cap X(q)\}$ is the flattened set of statements from the algorithms invoked from the state-entry action of the ECState q.

5.2.4 Preserving Transition Priority

Transitions from a given state $q \in Q$ of a BFB is an ordered set namely, $T(q)$ (see Definition 3.4). However, commands in PRISM modules are unordered (Definition 4.2). To preserve the order of transitions, we use negations of Boolean guards of the higher priority transitions for ensuring that a lower priority PRISM command can only be enabled when higher priority commands are disabled. We define a macro named ensureOrder to preserve order of a given transition using the said Boolean negation mechanism as follows.

$$\mathsf{ensureOrder}(t_j) \;=\; \widehat{\mathcal{B}}(e_j, b_j) \bigwedge_{t_i \in T(q)}^{t_i > t_j} \neg\big(\widehat{\mathcal{B}}(e_i, b_i)\big) \;\; \textit{where},$$

- $t_j = (q, e_j, b_j, q'_j) \in T(q)$ is a given transition to preserve its order.
- $t_i = (q, e_i, b_i, q'_i) \in T(q) \;\; s.t., t_i > t_j$ is the set of transitions that contain zero or more elements such that, any given element from this set t_i is of higher priority than the given transition t_j. Recall that t_0 is the highest priority transition, we use the declarative index of the transitions in a manner where $i < j \implies t_i > t_j$.
- $\widehat{\mathcal{B}}(e, b)$ is the Boolean guard of a given transition $t = (q, e, b, q')$, which comprises the status checking of the event e, and evaluation the Boolean expression b over the set of local and input variables of the given BFB.

Here, we take the first i elements from the ordered set $T(q)$, where $i < j$. This selection results in an ordered subset of all transitions from q that are of higher priority than the given transition t_j. Then, a new Boolean guard is computed through conjunction of negations of selected Boolean guards of all selected transitions $\forall t_i$ with the Boolean guard of the given transition t_j. In case where the given the transition is the highest priority transition, i.e. $t_j = t_0$, it results in an empty set of higher priority transitions. Thus, in this case the resultant Boolean guard comprises only of the Boolean guard of the given transition t_j.

Consider the example of DO_CLOSE ECState in Figure 3. Here, we have the following transitions.

$$T(\mathsf{DO\_CLOSE}) = \{ \; t_0 = \mathsf{DO\_CLOSE} \xrightarrow{<0> \;\; \mathsf{pChange\&\&(!relief)}} \mathsf{CLOSED},$$
$$t_1 = \mathsf{DO\_CLOSE} \xrightarrow{<1> \;\; \mathsf{counter<10}} \mathsf{DO\_CLOSE},$$
$$t_2 = \mathsf{DO\_CLOSE} \xrightarrow{<2> \;\; \mathsf{true}} \mathsf{ALARM} \;\; \}$$

Illustrating the ensureOrder macro, we get the following.

- ensureOrder(t_0) = pChange&&(!relief)
- ensureOrder(t_1) = (counter<10) && (!pChange || relief)
- ensureOrder(t_2) = (counter>=10) && (!pChange || relief)

5.2.5 Converting Expressions and Statements

The **convExpr** macro is used for converting expressions and statements from IEC 61499 syntax into equivalent Prism syntax. This macro is useful for translating algorithms and condition guards for a given BFB. This translation entails processing *identifiers* and *operators* in the given expression or statement.

$$\mathsf{convIdent(identifier)} = \begin{cases} \mathsf{rename(identifier)} & \backslash\backslash \text{ output and local events and variables} \\ \mathsf{sourceOf(identifier)} & \backslash\backslash \text{ input events and variables} \end{cases}$$

$$\mathsf{convOp(operator)} = \text{Look up equivalent operator from Table 6}$$

Using Figure 3 for illustration, we present the following examples of the convExpr macro.

- convExpr({“counter == 10”}) = `(counter_controller = 10)`
- convExpr({“valveCtl = true”}) = `(valveCtl_controller' = true)`
- convExpr({“pChange && (!relief)”})
  = `(update_cylinder) & (!oValue_flow__1oo2)`
- convExpr({“rChange && (pressure >= threshold)”})
  = `(output_flow__1oo2) & (pressure_cylinder >= 300)`

In the first example, the local variables counter does not require a source lookup and is simply renamed using the rename macro. In the second example, we used the assignment operator (see Table 6) to assign a Boolean *true* value to a variable. Note that for IEC 61499 we assume ‘C’-like syntax that uses ‘=’ and ‘==’ operators to differentiate between Boolean equality and assignment operations. In the third example, the input event pChange was looked up using the sourceOf macro to find the source output event i.e., cylinder.update, which was then renamed using the rename macro. Next, the input variable relief was replaced with `oValue_flow__1oo2`, the source output variable renamed to match the generated Prism syntax. Note that the separator underscore was encoded using *byte-stuffing* to avoid ambiguities and name conflicts. Similarly, the last example was processed using the sourceOf macro, except that the source of the input threshold was an integer constant value 300. In all of the above examples, operators were translated using the Table 6.

Table 6: Lookup table for translating syntax from IEC 61499 algorithms to PRISM update statements

Operator Name	**IEC 61499 Operator**	**PRISM Operator**
disjunction	|| or |	|
conjunction	&& or &	&
equality	== or =	=
inequality	! = or <>	! =
negation	!	!
addition	$+$	$+$
subtraction	$-$	$-$
multiplication	$*$	$*$
division	/	not supported
assignment	$\mathtt{a} = \mathtt{b}$	$\mathtt{a}' = \mathtt{b}$
increment	$\mathtt{a}++$	$\mathtt{a}' = (\mathtt{a}+\mathtt{1})$
decrement	$\mathtt{a}--$	$\mathtt{a}' = (\mathtt{a}-\mathtt{1})$
addition w/ assignment	$\mathtt{a}+=\mathtt{b}$	$\mathtt{a}' = (\mathtt{a}+\mathtt{b})$
subtraction w/ assignment	$\mathtt{a}-=\mathtt{b}$	$\mathtt{a}' = (\mathtt{a}-\mathtt{b})$
multiplication w/ assignment	$\mathtt{a}*=\mathtt{b}$	$\mathtt{a}' = (\mathtt{a}*\mathtt{b})$
division w/ assignment	$\mathtt{a}/=\mathtt{b}$	not supported
conditional assignment	$\mathtt{a} = (\mathtt{b})?\mathtt{c}:\mathtt{d}$	$\mathtt{a}' = (\mathtt{b})?\mathtt{c}:\mathtt{d}$

5.2.6 The command Boilerplate

Transitions are encoded in the generated PRISM module as commands using the **command** boilerplate. Given a set of Boolean expressions, and a set of assignment statements and event statuses, the command boilerplate produces a PRISM command.

$$\mathsf{command}(G,S) \quad = \text{`}\mathtt{[t]}\text{'} \oplus \mathop{\&}_{g\in G} \mathsf{convExpr}(g) \oplus \text{`}\mathtt{->}\text{'} \oplus \mathop{\&}_{s\in S} \mathsf{convExpr}(s) \oplus \text{`}\mathtt{;}\text{'} \qquad (B2)$$

The definition presented above uses the string concatenation operator $\oplus$ to combine the various fragments and form the PRISM command. Here, the term `'[t]'` is the synchronization label of PRISM which is enforced on all generated PRISM commands to perform execution in a lock-step. Next, we take all Boolean expressions from the given set G and convert them to their PRISM equivalent syntax using the convExpr macro. The converted expressions are used to construct a Boolean conjunction using the '&' operator (see Table 6). Lastly, we take all assignment statements from the given set S and construct a sequence of equivalent assignment statements in PRISM syntax using the '&' operator. This is followed by the statement terminator `';'`. Consider the following examples derived from the ECC of the controller BFB presented in Figure 3.

- command({"(counter < 10)"}, {"(counter = (counter<10)?(counter+1):counter)"})

```
      = [t] (counter_controller < 10) ->
            (counter_controller' = (counter_controller < 10) ?
               (counter_controller + 1) :  counter_controller);
```

- command({"rChange && (pressure >= threshold)"}, {"(valveCtl = true)"})

```
      = [t] (output_flow__1oo2) & (pressure_cylinder>=300) ->
            (valveCtl_controller'=true)
```

The examples presented above rely on the convExpr macro to convert the Boolean expressions and assignment statements, which are then concatenated with the PRISM '&' operator. Together with literals '`[t]`', '`->`' and '`;`', the resultant PRISM command is complete and syntactically correct.

5.2.7 Encoding the Moore-type Initialisation

The ECCs of BFBs are Moore-like state machines, which may have initial state actions. In synchronous execution semantics, these actions are invoked in an initialisation tick. Unlike other ticks, in the initialisation inputs are ignored and the current state is set to the initial ECState $q_0 \in Q$. These initialisation can be encoded in the Mealy-like PRISM command structure by generating an *init-command* in generated PRISM modules as follows.

$$\text{command}(\{\text{s==-1}\}, \{\text{s=valueOf}(q_0)\} \cup \text{stateActions}(q_0)) \qquad (T4)$$

Using the controller function block presented in Figure 3, we generate the following PRISM command i.e., Line 14 of Table 5.

```
[t] (s_contoller=-1) -> (s_contoller'=0)
      & (trigger_controller'=false) & (counter_controller'=0);
```

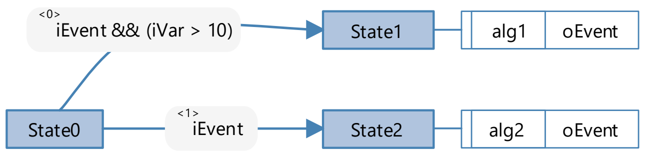
In this generated command, `[t]` is the action label for the lock-step execution of this command as suggested by the synchronous execution semantics. Secondly, the special *init-value* (-1) is used to induce the *init-location*. This is followed by a set of update statements generated from the action set associated with the initial state q_0. Together, these three components make the *init-command*.

5.2.8 Encoding Transitions

Given a transition $t = (q, e, b, q') \in T(q)$ of a given BFB, we can use the command boilerplate to generate behaviorally equivalent PRISM commands in the corresponding PRISM module i.e., as follows.

$$\text{command}\big(\ \text{checkState}(q) \cup \text{ensureOrder}(t),\ \text{updateState}(q') \cup \text{stateActions}(q')\ \big) \quad (T5)$$

The rule $T5$ uses the checkState macro to induce a location corresponding to the predecessor state q. Here, the ensureOrder macro is used to generate a condition guard, which ensures that the transition order is preserved. Lastly, the action set of the successor state q' i.e., the sequence of algorithm invocations and emissions of output events is converted using the stateActions macro. Figure 7 illustrates the conversion of two transitions into corresponding PRISM commands. Here, the wire-connection encoding was performed by the sourceOf macro using the function block network presented in Figure 6.



```
    [t] (s\_bfb2 = 0) & (oEvent1_bfb1 | oEvent1_bfb1) & (oVar_bfb1 > 10) ->
            (s' = 1) & (oEvent_bfb2' = true) & (oVar_bfb2' = false)

    [t] (s\_bfb2 = 0) & (oEvent1_bfb1 | oEvent1_bfb1) & (oVar_bfb1 <= 10) ->
            (s' = 2) & (oEvent_bfb2' = true) & (oVar_bfb2' = true)
```

Figure 7: Illustration of rule $T5$ for encoding deterministic transitions. The source state-transition structure is shown above and the generated PRISM commands are shown below.

5.2.9 Avoiding Deadlocks

ECCs of BFBs have an implicit stay operation when none of the egress transitions are enabled. In order to avoid deadlocks in the corresponding generated PRISM module, self-loop commands are added to mimic stay operations using the rule $T6$. This rule ensures that every module has a reaction in every execution cycle, thus ensuring that the composition of PRISM modules remains reactive. However, if a state already has an unconditional self-loop, this rule is not needed.

$$\mathsf{command}\big(\,\mathsf{checkState}(q) \cup \mathsf{negateAll}(q),\ \mathsf{updateState}(q) \cup \mathsf{setStatus}(\emptyset,\ E_O^{\mathcal{I}})\,\big) \qquad (T6)$$

- Here, the invocation of macros checkState(q) and updateState(q) induce a self-loop from the given state q to itself.
- The negateAll(q) macro is used to ensure that the generated self-loop bears the lowest priority among all egress transitions from the given ECState q. This is implemented by taking all Boolean guards from all egress transitions and constructing a Boolean conjunction over their negations. Thus, the generated self-loop can only execute when all other egress transitions are disabled. The implementation of negateAll(q) is as follows.

$$\mathsf{negateAll}(q) = \bigwedge_{t_i \in T(q)} \neg(\widehat{\mathcal{B}}(e_i, b_i)) \quad \textit{where, } t_i = (q, e_i, b_i, q_i)$$

- Lastly, the macro invocation setStatus($\emptyset$, $E_O^{\mathcal{I}}$) generates Boolean assignment statements that set status of all output events to *absent*. This ensures that the state-entry actions are not invoked but the life-time of the event (e.g., *1-tick*) is enforced.

Using the controller ECC for illustration (see Figure 3), we see that three ECStates CLOSED, OPENED, and ALARM require self-loops. Whereas two ECStates DO_OPEN and DO_CLOSE do not require self-loops because they already have a un-conditional egress transitions predicated on the Boolean expression `true`. For the said three ECStates, the generated self-loops command are visible as lines $48 - 54$ of Table 5.

5.3 Boilerplate for Generating the Prism Model

This section presents the proposed boilerplate for converting a given FBN to a PRISM model. It uses the transformation rules $T1 - T6$ presented in the previous sections. The representation used for this boilerplate is a string template inspired from *Microsoft T4 Templates* [10], which permit an embedded algorithm in the following syntax.

- A *string literal* is presented as mono-space font with single quotes around e.g., `'token'`
- A code block is presented inside delimiters <# and #>. A string yielding macro can be invoked to return its value when presented inside delimiters <#= and #>
- The symbol $\oplus$ represents string concatenation
- Single-line comments start with two backslash symbols i.e., \\ This is a comment
- The ForEach iterator takes each element from a given set and executes its nested block
- The If block is a conditional code block that execute only if the given condition holds
- Macros used in the template have been defined in the previous sections and are summarized as follows.

Name	**Purpose**
nameOf	Returns the fully qualified name of the given BFB
variable	A boilerplate for definition of a variable in PRISM syntax
len	Returns the number of items in a given set
typeOf	Maps the type of a variable in BFB to its PRISM equivalent type
rangeOf	Determines the range of a given variable in BFB
initOf	Determines the initial value of a given variable in BFB
command	A boilerplate for declaration of a PRISM command

continued on the next page

Name	Purpose
valueOf	Assigns a numeric value to a given element e.g., based on its declaration index
stateActions	Returns assignment statements generated from state-entry actions of a given ECState
checkStatus	Generates an expression to evaluate the state-encoding variable against a given ECState
ensureOrder	Takes a transition and preserves its order by appending its Boolean condition guard with negated condition guards of higher order transitions
updateState	Generates an assignment statement to update the value of the state-encoding variable to represent a given successor ECState
negateAll	Constructs a Boolean expression that comprises negation of condition guards of all egress transitions from the given ECState. This is required to create a lowest-order self-loop command that only executes when all egress transitions are disabled
setStatus	Sets the *present* / *absent* status of events by generating assignment statements for the corresponding Boolean variables

Table 7: Boilerplate for generating PRISM model from a given FBN.

```
‘mdp’

<# ForEach $\mathtt{BFB}_i \in \mathtt{FBNetwork}$ #>
‘module’ $\oplus$ <#= nameOf ($\mathtt{BFB}_i$) #>

    \\generting state-encoding variable (see rule T1)
    <#= $\mathsf{variable}\big(s, \mathtt{INT}, [-1, \mathsf{len}(Q)-1], -1\big)$ #>

    \\generting variables against output events of $\mathtt{BFB}_i$ (see rule T2)
    <# ForEach $e \in E_O^{\mathcal{I}}$ #>
        <#= $\mathsf{variable}\big(e, \mathrm{BOOL}, \top, \mathtt{false}\big)$ #>
    <# EndFor #>

    \\generting variables against local and output variables of $\mathtt{BFB}_i$ (see rule T3)
    <# ForEach $v \in V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}}$ #>
        <#= $\mathsf{variable}\big(v, \mathsf{typeOf}(v), \mathsf{rangeOf}(v), \mathsf{initOf}(v)\big)$ #>
    <# EndFor #>

    \\generting init-command for initial state actions of $\mathtt{BFB}_i$ (see rule T4)
    <#= $\mathsf{command}\big(\{\mathsf{s==\text{-}1}\}, \{\mathsf{s{=}valueOf}(q_0)\} \cup \mathsf{stateActions}(q_0)\big)$ #>

    <# ForEach $q \in Q$ #>
        \\generting commands for all transitions in $\mathtt{BFB}_i$ (see rule T5)
        <# ForEach $t = (q, e, b, q') \in T(q)$ #>
            <#= $\mathsf{command}\big(\ \mathsf{checkState}(q) \cup \mathsf{ensureOrder}(t), \mathsf{updateState}(q') \cup \mathsf{stateActions}(q')\ \big)$ #>
        <# EndFor #>

        \\generting self-loop commands (see rule T6)
        <# If $(q, \mathtt{true}, \emptyset, q') \notin T(q)$ for all $q, q' \in Q$ #>
            <#= $\mathsf{command}\big(\ \mathsf{checkState}(q) \cup \mathsf{negateAll}(q), \mathsf{updateState}(q) \cup \mathsf{setStatus}(\emptyset, E_O^{\mathcal{I}})\ \big)$ #>
        <# EndIf #>
    <# EndFor #>

‘endmodule’
<# EndFor #>
```

6 Preserving Execution Semantics

The application of the transformation rules generates a Prism model that preserves the synchronous execution semantics of the source function block network. As a result, all possible execution behaviours of one model will be matched in the other model, which is also known as *trace equivalence* [11]. We use the notion of execution steps i.e., *ticks* in BFBs and *cycles* in Prism modules to present proofs of step-wise equivalence.

Lemma 6.1 (Equivalence of Initialization Step)**.** *The execution behaviours of a given FBN* `FBNetwork` *and the generated* Prism *model $\mathcal{M}$ are equivalent in the initialization step. We use the following notation to represent this equivalence.*

$$\texttt{FBNetwork} \overset{step=0}{\rightsquigarrow} \mathcal{M} \tag{3}$$

Proof. The initialization step of a given `FBNetwork` and the generated $\mathcal{M}$ is equivalent due to the following reasoning.

1. The pre-initialization state of both models is equivalent thanks to the following observations on the the boilerplate presented in Table 7.
   (a) Lines 9 – 12 ensure that every output event ($e \in E_O^{\mathcal{I}}$) of each BFB in the given FBN ($\texttt{BFB}_i \in \texttt{FBNetwork}$) has a corresponding Boolean variable in the respective Prism module $M_i \in \mathcal{M}$. The initial status of the output event is absent, which is matched with the Boolean `false` value.
   (b) Lines 14 – 17 ensure that every output and local variable ($v \in V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}}$) of each BFB in the given FBN ($\texttt{BFB}_i \in \texttt{FBNetwork}$) has a corresponding variable in the respective Prism module $M_i \in \mathcal{M}$. The type of BFB variables, its range, and its initial value matches with that of the corresponding Prism variable, which is ensured by the macros typeOf, rangeOf, initOf respectively.
   (c) The state-label on both models is unassigned in the pre-initialization state.
2. During initialization, every BFB in the given FBN performs its initial state actions i.e., invoke algorithms and emit output events. Lines 19 – 20 generate an init-command that matches this initialization in the following manner.
   (a) The initial value of the state-encoding variable (s==-1) ensures that init-command is the first and the only command that executes (also see Lines 6 – 7).
   (b) The set of assignments generated by the macro stateActions(q_0) encode the state-entry actions of the initial ECState q_0, which include algorithms invocations and setting status of output events. Thus, the post-initialization value of output events, and output / local variables is equivalence.
   (c) The assignment statement (s=valuesOf(q_0) ensures that state label of both models in post-initialization state is equivalent i.e., the state label of the initial state q_0.

The pre-initialization and post-initialization states of every BFB in the given FBN matches with that of all corresponding Prism modules in the generated Prism model. Thus, we proved that the execution behaviour of both models in the initialization step is equivalent.
□

We use the Lemma 6.1 to develop a proof by induction. In an arbitrary execution of the source FBN, we start with demonstrating the equivalence of the execution of the first step as the base case of induction. Given this base case we assume the behavioural equivalence at an arbitrary k^{th} step (for $k \geq 1$) of execution, which is then used for establishing the equivalence of the $(k+1)^{th}$ step.

Theorem 6.2. *The execution behaviour of a given FBN* `FBNetwork` *is matched by the generated* Prism *model $\mathcal{M}$. We use the following notation to represent this equivalence.*

$$\texttt{FBNetwork} \rightsquigarrow \mathcal{M}$$

Base Case:
The base case of this theorem is presented as a lemma as following.

Lemma 6.3 (Equivalence of Initialization Tick)**.** *The execution behaviours of a given FBN* `FBNetwork` *and the generated* Prism *model $\mathcal{M}$ are equivalent in the first step i.e., the following.*

$$\texttt{FBNetwork} \overset{step=1}{\rightsquigarrow} \mathcal{M}$$

Proof. The first execution step of `FBNetwork` begins from the initial ECStates of all contained BFBs i.e., $q_{0i} \in \texttt{FBNetwork.BFB}_i$. Under synchronous execution semantics, every execution step comprises three sub-steps: input sampling, taking a transition, and execution state-entry actions. These sub-steps are emulated in the generated PRISM model $\mathcal{M}$ to ensure equivalence. Thus, we can demonstrate that the first execution step of the `FBNetwork` is equivalent with that of the generated $\mathcal{M}$ using the following reasoning.

1. **Input sampling:** Every BFB in the given FBN $\texttt{BFB}_i \in \texttt{FBNetwork}$ samples its inputs from its input wire connections, which is not need in the respective PRISM module $M_i \in \mathcal{M}$ since it reads its inputs directly from the variables of the source modules (see Section 5.2.1). In both structures, all components (BFBs and PRISM modules) read inputs from the previous (initialisation) step, which is known to be equivalent for the two structures (see Lemma 6.1).

2. **Transitions:** In the first tick, any given BFB $\texttt{BFB}_i$ takes at most one transition from the initial state q_{0i}. There are two possible cases in this step.

   (a) **Case I**: Based on inputs and transition guard conditions, no transition can be enabled from the current state of $\texttt{BFB}_i$. In this case, the current state will not be changed and consequently, no state actions shall be performed. This case will be matched by the respective PRISM module M_i due to rules $T6$ (see Lines $28-31$). Macros $\mathsf{checkState}(q_{0i})$ and $\mathsf{updateState}(q_{0i})$ emulate the stay operation by taking a self-loop on the initial ECState q_{0i}. The macro $\mathsf{negateAll}(q_{0i})$ ensures that this self-loop is only taken when no-other transition can be taken. Lastly, the macro $\mathsf{setStatus}(\emptyset, E_O^{\mathcal{I}})$ sets all output events to the *absent* status, thus emulating the no-action behaviour of a given $\texttt{BFB}_i$.

   In this case, the execution step concludes here and no more actions are performed in either of the structures in the current execution step. ■

(b) **Case II**: At least one transition is enabled in $\texttt{BFB}_i$. In this case, the highest priority transition shall be taken. This case will be matched by the respective Prism module by executing an enabled command (see Lines $23-26$). Rule $T5$ matches the initial ECState q_{0i} using the macro $\mathsf{checkState}(q_{0i})$. Furthermore, it uses explicit negations in the $\mathsf{ensureOrder}$ macro to ensure that **(i)** at most one command is enabled in an iteration, and **(ii)** the transition order is preserved. Therefore, the command executed by M_i will necessarily match the highest priority enabled transition as taken by $\texttt{BFB}_i$.

3. **Successor State Actions:** If a transition was taken (Case II), then all BFBs $\texttt{BFB}_i \in$ `FBNetwork` execute their successor state-entry actions. This step is matched by $\mathcal{M}$ because of the macro $\mathsf{stateActions}$ in rule $T5$ (see Lines $23-26$). This ensures that **(i)** algorithms are translated into update statements, **(ii)** the Boolean encoded events are set with an appropriate Boolean `true` (*present*) or `false` (*absent*) value. Furthermore, the macro $\mathsf{updateState}$ ensures that the state-encoding variable is updated to represent the respective successor state. ■

Conclusion: The equivalence of the three phases of the first execution step namely, input sampling, taking of an enabled transition, and performing successor state actions, yields behavioural equivalence of the first step for any given BFB $\texttt{BFB}_i \in \texttt{FBNetwork}$ with the corresponding Prism module $M_i \in \mathcal{M}$. Thus, we conclude the following.

$$\texttt{BFB}_i \in \texttt{FBNetwork} \overset{step=1}{\rightsquigarrow} M_i \in \mathcal{M} \quad \Longrightarrow \quad \texttt{FBNetwork} \overset{step=1}{\rightsquigarrow} \mathcal{M}$$

□

Hypothesis:

We assume that the execution of an arbitrary k^{th} step of the given `FBNetwork` is equivalent to the execution of k^{th} step of the generated Prism model $\mathcal{M}$.

Assume that for all $(k \geq 1)$ *we have,* $\texttt{FBNetwork} \overset{step=k}{\rightsquigarrow} \mathcal{M}$

Induction:

Given the assumption from hypothesis, we establish the equivalence of the $(k+1)^{th}$ execution step between the given `FBNetwork` and the generated Prism model $\mathcal{M}$ through the following lemma.

Lemma 6.4. *The execution behaviour of a given FBN* `FBNetwork` *and the generated* Prism *model* M *are equivalent in the* $(k+1)^{th}$ *step i.e., the following.*

$$\texttt{FBNetwork} \overset{step=k+1}{\rightsquigarrow} \mathcal{M}$$

Proof. Given the assumption that the execution of an arbitrary k^{th} step of $\mathcal{M}$ is equivalent to the execution of k^{th} tick of `FBNetwork` (for $k \geq 1$). The $(k+1)^{th}$ execution iteration will also be equivalent. This can be established by examining the $(k+1)^{th}$ execution step of any BFB in the given FBN $\texttt{BFB}_i \in \texttt{FBNetwork}$ against that of the corresponding Prism module $M_i \in \mathcal{M}$.

1. **Input sampling:** Both BFBs and PRISM modules rely on unit-delayed updates. Consequently, both $\texttt{BFB}_i$ and M_i sample their inputs from updates computed in the k^{th} step. Due to the assumed equivalence of the k^{th} step, both structures read equivalent inputs.
2. **Transitions:** Based on inputs and the respective guard conditions on egress transitions of the current state (from the k^{th} iteration), $\texttt{BFB}_i$ takes at most one transition into the successor state. This step is matched by the respective PRISM module M_i for similar reasoning as presented in base-case for the *transitions* sub-step in Lemma 6.3. Again, we will two cases:
   (a) **Case I:** $\texttt{BFB}_i$ performs an implicit stay operation, which is matched by a self-loop command generated in Lines $28 - 31$ (see rule $T6$). The justification for this is same as presented in Case I Lemma 6.3. In this case, the execution of $(k+1)^{th}$ step of both structures stops here. ■
   (b) **Case II:** $\texttt{BFB}_i$ takes a transition, which is deterministically matched by a corresponding command in M_i. The justification for this is same as presented in Case II Lemma 6.3.
3. **Successor State Actions:** If a selected enabled transition is taken by $\texttt{BFB}_i$, it results in the invocation of state actions of the respective successor state. This step is matched by the respective PRISM module M_i for similar reasoning as presented in the base-case for successor state actions in Lemma 6.3. ■

,

Conclusion: The equivalence of the three phases of the $(k+1)^{th}$ execution step namely, input sampling, taking of an enabled transition, and performing successor state actions, yields behavioural equivalence of the $(k+1)^{th}$ step i.e., the following is true.

$$\texttt{BFB}_i \in \texttt{FBNetwork} \xrightarrow{step=k+1} M_i \in \mathcal{M} \quad \Longrightarrow \quad \texttt{FBNetwork} \xrightarrow{step=k+1} \mathcal{M}$$

□

Corollary:
Starting from initialisation (Lemma 6.1), and establishing the base-case (Lemma 6.3), we assumed that at an arbitrary k^{th} the execution behaviour of the given FBN is equivalent with that of the generated PRISM model. In the inductive step, we established that based on the base-case and the hypothesis, the execution behaviour of the two structures is equivalent in the $k+1^{th}$ step (Lemmas 6.4). Thus, we can perform a mathematical induction that the execution behaviour of the structures is equivalent in any n^{th} step such that, $n \geq 0$. This proves the Theorem 6.2, which states that the execution behaviour of a given FBN $\texttt{FBNetwork}$ is equivalent to that of the generated PRISM model $\mathcal{M}$ i.e., the following.

$$\texttt{FBNetwork} \rightsquigarrow \mathcal{M}$$

This equivelence in the execution behaviour at every step induces a trace containment relationship [11]. However, under synchronous execution semantics, FBNs are deterministic. This deterministic choice was also enforced at each step of execution in the generated

Prism model. Thus, a trace equivalence relationship can be claimed. Under this equivalence, the soundness of linear-time (LTL) properties is preserved [11].

6.1 Model Verification

Formal verification of models often involves processing an abstraction of the original system model. However, this verification is only valid if the abstraction holds a sound equivalence with the system model. In our case, we generate Prism models from the given function block networks through a semantics-preserving transformation. This equivalence relationship allows for the use of the generated model for the purpose of formal verification such that it provides the following guarantees.

$$\varphi \models \mathcal{M} \iff \varphi \models \texttt{FBNetwork} \quad s.t., \ \varphi \in \texttt{LTL}$$

The equation presented above states that a given property φ is either satisfied by both models or is satisfied by neither. Such verification guarantees are significantly important for model verification where verifying a property on one model gives us information about the other. Specifically, a given property needs only to be verified on the generated Prism model $\mathcal{M}$. A probabilistic extension of linear temporal logic (LTL), called pLTL is used for property specification in Prism model checker [12]. This notation allows quantitative reasoning over branching and non-branching behaviours of the system execution tree. For example, we can specify the query to compute the probability of reaching a failure state during the system execution. However, in our case the generated Prism models do not contain any probabilistic behaviours, therefore the result of the result of quantitative analysis would be either 0 or 1 i.e., the following.

$$P(\boldsymbol{Safe}) := P_{max}{=}0 \ F[\boldsymbol{Alarm}]$$

Here, we are evaluating if the Alarm state is reachable through its maximum probability. This naive quantitative analysis is actually equivalent to performing a qualitative analysis of the corresponding system, where the result is always in the form of a Boolean proposition i.e., whether the given property is satisfied.

7 Discussion

In this report, we presented formal structure and semantics for IEC 61499 function blocks and Prism models. This formalism was later used to present a set of rules that allowed automatic transformation of a given function block network into a behaviourally equivalent Prism model. We presented the arguments and proofs of this equivalence. We further presented the usefulness of this equivalence for the purpose of formal verification of system properties. This method of verification is on a par with the existing qualitative verification approaches such as [13, 14], which offer verification of system properties in the presence of events and data (Boolean and integer variables). However these approaches do not use the synchronous execution semantics. On the other hand, the verification techniques that support synchronous execution semantics are either equally good (i.e., [15]), or are worse at supporting data (i.e., [16]).

References

[1] *IEC 61499-1:2012 Function blocks - Part 1: Architecture.* Number IEC 61499. International Electrotechnical Commission, Geneva, Switzerland, 2012.

[2] L.H. Yoong, P.S. Roop, Z.E. Bhatti, and M.M.Y. Kuo. *Model-Driven Design Using IEC 61499 : A Synchronous Approach for Embedded and Automation Systems.* Springer, 2015.

[3] Cheng Pang, S. Patil, Chen-Wei Yang, V. Vyatkin, and A. Shalyto. A portability study of IEC 61499: Semantics and tools. In *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*, pages 440–445, July 2014.

[4] Li Hsien Yoong, P.S. Roop, V. Vyatkin, and Z. Salcic. Synchronous execution of IEC 61499 function blocks using Esterel. In *Proceedings of 5th IEEE International Conference on Industrial Informatics (INDIN'07)*, volume 2, pages 1189 –1194, June 2007.

[5] R. Sinha, P.S. Roop, G. Shaw, Z. Salcic, and M.M.Y. Kuo. Hierarchical and concurrent ECCs for IEC 61499 function blocks. *Industrial Informatics, IEEE Transactions on*, PP(99):1–1, 2015.

[6] Department of Computer Science, University of Oxford. The PRISM Language - Semantics. http://www.prismmodelchecker.org/doc/semantics.pdf, march 2015.

[7] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[8] V. Vyatkin. IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics*, 7(4):768–781, Nov 2011.

[9] L.H. Yoong, P.S. Roop, Z.E. Bhatti, and M.M.Y. Kuo. *Model-Driven Design Using IEC 61499 : A Synchronous Approach for Embedded and Automation Systems*, chapter 4, pages 65–89. Springer, 2015.

[10] Code Generation and T4 Templates. `https://msdn.microsoft.com/en-us/library/bb126445.aspx`, 2016. Accessed 3 August 2016.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, chapter 3,7, pages 104–120,496–500. The MIT Press, Cambridge, Massachusetts, USA., 2008.

[12] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proceeding of 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *Lecture Notes on Computer Science*, pages 585 –591. Springer, 2011.

[13] Christoph Sünder, Valeriy Vyatkin, and Alois Zoitl. Formal verification of downtimeless system evolution in embedded automation controllers. *ACM Trans. Embed. Comput. Syst.*, 12(1):17:1–17:17, January 2013.

[14] Sandeep Patil, Sayantan Bhadra, and Valeriy Vyatkin. Closed-loop formal verification framework with non-determinism, configurable by meta-modelling. In *Proceedings of 37th Annual Conference on IEEE Industrial Electronics Society (IECON)*, pages 3770–3775. IEEE, 2011.

[15] Li Hsien Yoong and P.S. Roop. Verifying IEC 61499 function blocks using Esterel. *IEEE Embedded Systems Letters*, 2(1):1 –4, March 2010.

[16] Z.E. Bhatti, R. Sinha, and P.S. Roop. Observer based verification of IEC 61499 function blocks. In *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, pages 609 –614, july 2011.