IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 1, 2021, accepted December 5, 2021, date of publication December 7, 2021,
date of current version December 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3133630

Toward Reliable Programmable Logic Controller Function Block Diagrams

JIANYONG ZHAO AND ZHE TAO
Institute of Intelligent and Software Technology, Hangzhou Dianzi University, Hangzhou 310018, China

Corresponding author: Jianyong Zhao (zjy@hdu.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2020YFB2010901, and in part by the Key Research and Development Program of Zhejiang Province under Grant 2020C01031.

ABSTRACT Programmable logic controllers (PLCs) are widely used in industrial electronic systems. With the augmenting complexity of system, the reliability poses a crucial challenge in safety critical applications. This paper proposes a formal modeling and verification approach for programming function block diagrams. Function block diagrams are formalized in a logic specification system. We consider the equivalence checking problem which occurs frequently between design implementations under different performance constraints. We present a novel method to harness a powerful co-induction proof strategy with bisimulation to establish the equivalence in a higher-order logic theorem proving system. We validate the effectiveness of our approach by a real industry application example with key scenarios. The soundness and the completeness of our approach are substantiated.

INDEX TERMS Bisimulation, function block diagram, programmable logic controller.

I. INTRODUCTION

From nuclear and chemical plants to elevators, programmable logic controllers (PLCs) are primary components in many safety-critical control systems. Thus their failure may lead to unacceptable consequences, such as huge loss of property, life-threatening. Recently, we were asked to fix a PLC elevator control program following a serious accident with casualties. We need to ensure that there are no more discrepancies between the corrected program and its given specification. But this task is nearly impossible to be done using traditional empirical approaches like code review, testing and simulation. Because these approaches are incapable of expressing complex specifications and exhaustively verifying that implementations conform to their specifications. To ensure the correctness of this system, we investigated formal methods and designed a novel and rigorous methodology in Coq [1] (an interactive theorem prover combines higher-order logic and richly-typed functional programming language). The Coq has been widely applied in systems such as industrial control systems [2], [3] and SCADA [4]. With the assist of the Coq, we have successfully verified the correctness of the corrected elevator control program.

The associate editor coordinating the review of this manuscript and approving it for publication was Vivek Kumar Sehgal.

In this paper, we present a novel methodology to verify Function Block Diagram (FBD) programs in real-time control systems conform to given specifications. First, we propose a coinduction execution model to characterize time-dependent behaviors. Then we formalize an FBD program and its given specification into this model. Finally, we prove the equivalence relation using bisimulation. The specific challenges and contributions of this work are the following:

- *How to formalize FBD programs*: FBD [5] is a graphical PLC programming language that derives from signal-flow graph (SFG) [6], but more flexible and operational. While an SFG strictly represents a set of linear algebra equations, FBD is less constrained. For example, FBD consists of a list of networks whose execution order might be non-deterministic. These features prevent us from completely defining the formal semantics of FBD. Thus, our formalization considers a deterministic subset of FBD. We assume that 1) the list of FBD networks is executed in sequential; 2) the feedback paths are sampled at the end of a scan cycle. This paper proposes a novel method to model the deterministic FBD programs into Mealy machines.
- *How to model the execution of PLC programs*: Differing from other controllers, PLC executes programs repeatedly in cycles. The control programs of real-time PLC systems like elevators are typically non-terminating.

They are designed to react to input signals cyclically, continuously and infinitely. We propose a coinduction model to formalize the execution of real-time PLC control programs.

- *How to define and prove the equivalence relation*: The behavior of real-time PLC programs is time-dependent. So, our equivalence relation should express this time-dependent behavior. Because our PLC execution model is cycle-based, we use the cycle-dependent equivalence to express the time-dependent equivalence. We assume each cycle of PLC execution takes a constant unit period. In fact, each cycle takes some time to complete the program in PLCs one time. Then we use bisimulation to define the time-dependent equivalence of two execution models. This paper proposes a method to model the given specification into a Mealy machine and prove the bisimulation of an FBD program and its given specification.

There have been already many valuable attempts [7] to apply formal methods on PLC systems for various purposes, such as testing [8], engineering [9], [10], and validation [11]. The existing works which formalize and verify PLC programs have trade-offs and are not universally applicable. They can be divided into two categories: theorem proving and model checking. Most of them adopt model checking [12]–[16]. They typically transformed a PLC program into a finite-state transition system, which is called state space. This space contains all states that a system can reach. Then they used model checking tools to exhaustively and automatically check if every state satisfies the specification given in temporal logic. Model checking provides a ''push-button'' solution but can easily encounter the combinatorial explosion problem caused by its brute state space exploration [17], because the state space of a Real-time PLC program can be enormous and exponentially grow in the number of state variables. The application of model checking is limited in certain kinds of verification works.

To avoid this problem, we adopt theorem proving to handle complex PLC systems, but related works are relatively few. Wan *et al.* [18] abstracted and verified timer-related behavior of ladder diagram (LD) programs in Coq. They presented a detailed analysis of a quiz machine with timers and proved its time-dependent correctness. Their formalization is program-specific, which means that different programs derive different sets of axioms and propositions. By comparison, our syntax-based modeling of FBD programs is general and can be automated.

In [19], Wan presented an approach to model PLC programming elements using inductive dependent-type in Coq. Specification and verification are modularized and parameterized. We generalize this approach to hierarchically model FBD programs and construct proofs.

Newell *et al.* [20] and Pang *et al.* [21] coped with FBD programs. They formalized FBD and ST programs into the prototype verification system (PVS), defined specifications in tabular expressions and presented a detailed case study. But there are some drawbacks in their formalization of FBD. They defined a delay function to implement function blocks like the flip-flop, timer and counter block rather than support internal variables of FBD. Consequently, the applicability of their FBD formalization is limited. By comparison, our FBD formalization is more practical and applicable. Our method uses dependent types to support FBD programs with internal variables and internal instances of stateful function blocks like timers.

Besides Newell and Pang's work [20], [21], there are other works involving the formalization of FBD. Some of them are lack details. Yoo *et al.* [22] presented a high-level framework and a case study to verify the equivalence between FBD programs using model checking. But they did not describe details of their formalization and verification. Others are not practical enough. Soliman *et al.* [16] defined an abstract syntax of FBD. They disassemble the structure of FBD as function blocks and connections between them. Then they presented a transformation from FBD to a large number of timed automatons for model checking purposes, which simply defines every connection of two function blocks as a timed automaton. Pavlović and Ehrich [15] presented a method to transform FBD programs into a text form for model checking purpose. But they only considered basic functions like arithmetic and Boolean operation. They ignored lots of distinguished FBD features like function blocks and feedback paths. This formalization made FBD just an alternative form of assignment statements and expressions of common imperative languages.

Overall, the novelties of our work are the following:

- We present a novel formalization of FBD programs with details. Our formalization is more practical and applicable than existing works.
- We propose a novel method with details to verify the equivalence between FBD programs and their given specifications, which have not been explored previously. Our method considers key features of PLC real-time control systems like non-terminating execution and time-dependent behavior. This method can verify complex PLC programs.

Our work fills the vacancy of the effective formalization and formal equivalence verification of real-time FBD programs using theorem proving. It is structured as follows. Section II discusses some key features of PLC. Section III defines the infinite trace, bisimilarity, and execution model in Coq. Section IV illustrates how to formalize standard graphical blocks. Section V demonstrates the formalization of FBD implementations and specifications with an example. Section VI presents the proof script of bisimilarity. Section VII presents our conclusions and directions for future work.

II. PROGRAMMABLE LOGIC CONTROLLERS

A. EXECUTION SCHEMA

A PLC system executes in a periodic schema, which repeats the scan cycle. The scan cycle consists of three steps: input

scan, user program scan, and output scan. In order to verify the equivalence between an FBD program and a given specification, we assume the execution of PLC is non-terminating and use coinduction to model it.

B. FUNCTION BLOCK DIAGRAM

IEC 61131-3 defines five programming languages for various purposes. It includes two textual languages, ST and instruction list (IL); two graphical languages, LD and FBD; and a mixed language, SFC which can be specified in textual or graphical forms [5]. Currently, we only consider PLC programs implemented in FBD because it is widely used and compatible with LD. Additionally, there are many studies applied to other textual languages, but the study in graphical languages is in a relatively early stage.

C. PROGRAM ORGANIZATION UNITS

In PLC programming, programs are implemented as program organization units (POUs), which are reusable structured elements. There are several kinds of POUs defined in IEC 61131-3 for the purpose of modularization. Commonly, a POU has a well-defined purpose and an interface with inputs and outputs. These POUs include Functions (FC), Function Blocks (FB), and Programs [5]. Accordingly, our specifications and verifications are also modularized, parameterized and reusable for further developments. IEC 61131-3 defines a series of standard functions and function blocks as standard programming blocks, and they are used as basic units in this investigation.

III. PRELIMINARIES

For easier understanding and reading, we introduce some preliminaries in this section, where timing behavior, infinite trace, bisimilarity, and execution model in Coq are introduced.

A. TIME

Timing behavior is crucial to real-time systems. Since PLC executes in cycles [5], we model time in a discrete manner. We assume that each scan cycle goes through a unit of time encoded as a natural number, because the exact period of each cycle is implementation-dependent but time increases monotonically. The modeling details of timer behavior are presented in Section IV.

B. INFINITE TRACES

In this work, the input and output sequences of PLC programs during the non-terminating execution are specified as infinite traces.

We adopt the definition of the ω-language and define infinite traces as ω-words (words of infinite length [23]). ω-languages are sets of ω-words. We assume that $\omega = \{0, 1, 2, \cdots\}$ denotes the set of natural numbers, and that $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \cdots\}$ is a finite alphabet. Σ^* is the set of all finite words over Σ, while Σ^ω is the set of all infinite words over Σ. $X_\Sigma \in \Sigma^\omega$ denotes an infinite trace (ω-word) over the alphabet Σ and has length $|X_\Sigma| = \omega$.

A short tutorial to Coq can be found in [24]. we briefly introduce some basic commands in the following descriptions. In Coq, we define Σ^ω as a coinductive type of infinite trace (lines 1-2). The coinductive definition of trace includes all possible infinite traces. Each possible infinite trace of trace can be defined without specifying every term and justified by applying finite or infinite numbers of coinductive rules:

```
1  CoInductive trace (A: Type): Type:=
2  | Cons: A -> trace A -> trace A.
3  Notation ``x ':::' X'':= (Cons x X') (at
       level 20).
4  Definition head (X: trace):=
5  match X with Cons a _ => a end.
6  Definition tail (X: trace):=
7  match X with Cons _ a => a end.
```

We use the command CoInductive to define Σ^ω as a coinductive type trace with a *constructor* Cons. The constructor Cons constructs an infinite trace by concatenating an element with another infinite trace. Additionally, the command Notation defines an infix notation ::: of the constructor Cons (line 3).

We define two functions head and tail (line 4-7) to specified infinite traces, e.g., an infinite trace $X_\Sigma = \sigma_0, \sigma_1, \sigma_2, \cdots$ is specified as the head term $head(X_\Sigma) = \sigma_0$ with the tail trace $tail(X_\Sigma) = \sigma_1, \sigma_2, \cdots$.

C. BISIMULATION

Bisimulation is a rich concept of behavioral equivalence. Originally, it is introduced by Milner [25] as the notion of behavioral equivalence between processes. Today it has been employed in various areas of computer sciences in various forms. We use the bisimulation equity, called bisimilarity, on infinite traces to define the behavioral equivalence between FBD programs and given specifications.

Definition 1 (Bisimilarity on Infinite Traces): A bisimilar on infinite trace is a relation $\sim \subseteq \Sigma^\omega \times \Sigma^\omega$ *that if* $s \sim t$, *then* $head(s) = head(t)$ *and* $tail(s) \sim tail(t)$.

The bisimilarity is defined as a coinductive predicate Bisimilar in Coq.

```
1  CoInductive Bisimilar {T: Type}
2  (X1 X2: trace T)
3  : Prop:=
4  | bisimilar: (head X1) = (head X2) ->
5  Bisimilar (tail X1) (tail X2) ->
6  Bisimilar X1 X2.
```

D. POU EXECUTION MODEL

Definition 2 (Syntax of POU): A POU can be represented as $\mathcal{P} = V \cup Prog$, *where*

- $V = V_{in} \cup V_{out} \cup V_{intl}$ *is the set of variable declarations, which consist of input variables* V_{in}, *output variables* V_{out}, *and internal variables* V_{intl}, *and*
- *Prog is the program body.*

The POU variables V are of various data types, and the program body *Prog* can be implemented in several PLC programming languages. In the following sections, we describe

our methodology to formalize them into an execution model. We formalize the semantics of a POU based on a Mealy machine.

Definition 3 (Mealy Machine): A Mealy machine [26] is a 5-tuple $\mathcal{M} = (I, O, S, s_0, trans)$, where

- *I is the finite set of inputs,*
- *O is the finite set of outputs,*
- *S is the finite set of states,*
- *$s_0 \in S$ is the initial state, and*
- *$trans : I \times S \to O \times S$ is the state transition function.*

In this paper we use i, o, and s to denote the elements of I, O, and S, respectively.

According to the periodic and non-terminating execution schema of the PLC system, we define a coinductive execution model to characterize the POU's execution behavior. The observable input and output sequences of the infinite execution cycles are defined as infinite input and output traces.

Definition 4 (Execution Model): The execution model of a POU is a 6-tuple $\mathcal{E} = (I, O, S, s_0, trans, exec)$ emulates the infinite execution of PLC programs. It extends the Mealy machine with an execution function $exec : I^\omega \times S \to O^\omega$. This execution function corecursively yields an infinite output trace with an infinite input trace and an initial state. Given an input trace $X_I \in I^\omega$ and an output trace $X_O \in O^\omega$, exec is defined with the transition function trans as

$$X_O = exec(X_I, s_0),$$

where

$$\begin{aligned}(head(X_O), s') &= trans(head(X_I), s_0),\\ tail(X_O) &= exec(tail(X_I), s').\end{aligned}$$

This execution model is defined in Coq as

```
Module Type ProgramOrganizationUnit.
Parameters I O S: Type.
Parameter s0: S.
Parameter trans: I -> S -> O * S.
Parameter exec: trace I -> S -> trace O
    .
Parameter o_default: O.
End ProgramOrganizationUnit.
```

o_default is an assistant definition denoting the default output value. It will be used in the formalization of FBD programs in Section V-A.

In Coq, the command Module Type defines a list of parameters and axioms, which is called the signature of a module. A module of a module type is an implementation of the signature of this module type. POUs are implemented as modules in Coq, which have ProgramOrganizationUnit as their signatures, as shown in this example:

```
Module ExamplePOU <: ProgramOrganizationUnit
    .
(*definitions*)
End ExamplePOU.
```

Additionally, an assistant function build_exec is defined to build the execution function exec with the transition function trans :

```
CoFixpoint build_exec {I O S: Type}
(trans: I -> S -> O * S)
(XI: trace I) (s: S)
: trace O:=
match XI with
| i::: XI' =>
let (o, s'):= trans i s in
let XO':= build_exec trans XI' s' in
o::: XO'
end.
```

IV. STANDARD GRAPHICAL BLOCKS

In this section, we formalize standard graphical blocks in Coq, including standard functions and standard function blocks. These blocks are reusable POUs of FBD programs.

A. STANDARD FUNCTION

Definition 5 (FC): As a type of POU, the syntax of a PLC function can be represented as $\mathcal{P} = V \cup Prog$.

IEC 61131-3 specifies a series of standard functions including numerical and arithmetic functions, bit Boolean functions, and selection and comparison functions. Take the standard function AND (Boolean *and* operator) as an example, we formalize $\mathcal{P}_{AND} = V \cup Prog$ into its execution model $\mathcal{E}_{AND} = (I, O, S, s_0, trans, exec)$, as shown in Fig. 1.

```
FUNCTION AND
VAR_INPUT
    IN1 : BOOL;
    IN2 : BOOL;
END_VAR
VAR_OUTPUT
    OUT : BOOL;
END_VAR
```

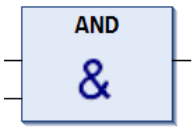


FIGURE 1. Standard Boolean *and* function AND.

1) FORMALIZE VARIABLES V

We use the input variables V_{in} to define the input set I and the output variables V_{out} to define the output set O. For $V_{in} = \{in_1, in_2, \cdots, in_n\}$, then $I = \prod_{i=1}^{n} dom(in_i)$, where $dom(in_i)$ is the domain of the variable in_i.

In most cases, the internal variables will be retained with the POU instance, with the state set S defined by the internal and output variables V_{intl} and V_{output} (i.e., $S = \prod_{i=1}^{m} dom(intl_i) \times \prod_{i=1}^{n} dom(out_i)$). However, the internal variables of PLC functions are temporal. Therefore, we only use V_{out} to define the state set S of PLC functions (i.e., $S = \prod_{i=1}^{n} dom(out_i)$).

The AND function presented previously has $V_{in} = \{IN1, IN2\}$, $V_{out} = \{OUT\}$, and $V_{intl} = \emptyset$, where IN1, IN2, and OUT are Boolean values. Therefore, $I = \mathbb{B} \times \mathbb{B}$, $O = \mathbb{B}$, and $S = \mathbb{B}$, where $\mathbb{B}$ is the Boolean domain.

Although PLC program languages support many data types, we consider only Booleans, natural numbers, and durations (time) here. In Coq, Booleans are defined values of type

bool, and natural numbers and durations are defined as values of type nat. We define sets, such as I, O, and S, using record types. We define I, O, S, s_0 and the default output o_default of the AND function as follows:

```
Record I_proto: Type:= mk_i{IN1:bool; IN2:
    bool}.
Record O_proto: Type:= mk_o{OUT:bool}.
Record S_proto: Type:= mk_s{_OUT:bool}.
Definition I:= I_proto.
Definition O:= O_proto.
Definition S:= S_proto.
Definition s0:= {|_OUT:= false|}.
Definition o_default:= {|OUT:= false|}.
```

The variables are defined as fields (e.g., IN1, IN2) of record types (e.g., I_proto). We define a set by specifying values of each field and obtain the value of s0's field _OUT with (_OUT s0). Additionally, we define a series of helper notations for each field that allows us to modify the value of s0's _OUT field using the simple assignment (s0&{|_OUT:= true|}). Using record types to define sets of variables reduces confusion when defining, accessing, and modifying them.

2) FORMALIZE PROGRAM BODY *Prog*

The behavior of standard functions are defined in IEC 61131-3, but their implementations (i.e., program body *Prog*, which defines the transition function *trans* and the execution function *exec* of execution model) are vendor-specific. Therefore, we implement the behavior of standard functions in Coq according to IEC 61131-3. As an example, AND accepts two Booleans as input and returns their Boolean *and* result. We use the Boolean *and* function andb in the Coq standard library to implement it. The transition function *trans* of AND is defined as a function trans in Coq. The definition detail is omitted and represented instead by ... to save space.

```
Definition trans (i: I) (s: S): O * S:= \
    ldots
```

The execution function *exec* is defined using the assistant function build_exec with trans as a corecursive function exec.

```
Definition exec:= build_exec trans.
```

B. STANDARD FUNCTION BLOCK

Definition 6 (FB): As a type of POU, a FB can be represented as $\mathcal{P} = V \cup Prog$.

IEC 61131-3 also specifies a series of standard function blocks including bistable elements, edge detection routines, counters, and timers. Taking the TON (on-delay timer) FB as an example, we use the approach presented in the preceding subsection to formalize $\mathcal{P}_{TON} = V \cup Prog$ into its execution model $\mathcal{E}_{TON} = (I, O, S, s_0, trans, exec)$, as shown in Fig. 2.

1) FORMALIZE VARIABLES *V*

Similar to standard functions, the input set I, the output set O, and the state set S of execution models are defined using the variables V in FBs.

```
FUNCTION_BLOCK TON
VAR_INPUT
    IN : BOOL;
    PT : TIME;
END_VAR
VAR_OUTPUT
    Q  : BOOL;
    ET : TIME;
END_VAR
```

th

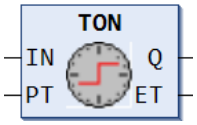


FIGURE 2. On-delay timer TON.

Taking the TON (on-delay timer) FB as an example, the function requires an enable signal IN and a preset time PT as input variables $V_{in} = \{IN, PT\}$ and yields a timeout signal Q and an elapsed time ET as output $V_{out} = \{Q, ET\}$, where IN and Q are Boolean values and PT and ET are durations. It has no internal variables, so $V_{intl} = \emptyset$. Therefore, the input set $I = \mathbb{B} \times \mathbb{N}$, the output set $O = \mathbb{B} \times \mathbb{N}$, and the state set $S = \mathbb{B} \times \mathbb{N}$, where $\mathbb{B}$ and $\mathbb{N}$ are Boolean and natural number domains. We define I, O, S, s_0, and the default output o_default of TON as follows:

```
Record I_proto: Type:= mk_i{IN:bool; PT:nat
    }.
Record O_proto: Type:= mk_o{Q:bool; ET:nat}.
Record S_proto: Type:= mk_s{_Q:bool; _ET:nat
    }.
Definition I:= I_proto.
Definition O:= O_proto.
Definition S:= S_proto.
Definition s0:= {|_Q:= false; _ET:= 0|}.
Definition o_default:= {|Q:= false; ET:=
    0|}.
```

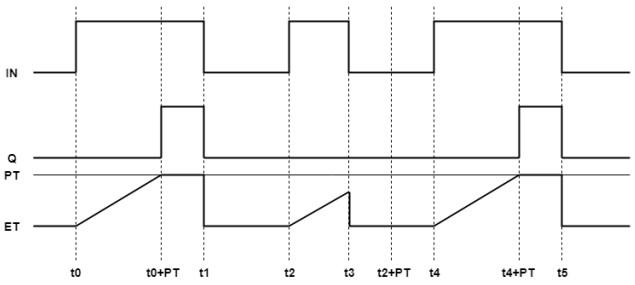


FIGURE 3. Timing diagram of TON function.

2) FORMALIZE PROGRAM BODY *Prog*

Informally, the behavior of the TON function is described in a timing diagram (Fig. 3). Once IN rises and holds true, ET increases from 0. If IN becomes false, ET is reset to 0. Only if ET $\geq$ PT is the timeout signal Q true. We assume that the outputs of timers are sampled at the ending of each scan cycle and affect the next scan cycle, and formally define the transition function *trans* of TON as a function trans in Coq.

```
Definition trans (i: I) (s: S): (O * S):= \
    ldots
```

Its execution function *exec* is defined using the helper function build_exec with trans as a corecursive function exec.

```
Definition exec:= build_exec trans.
```

V. FBD IMPLEMENTATIONS AND SPECIFICATIONS

In this section, we describe in detail the method of formalizing an FBD implementation and its specification into the POU execution model.

In practice, we implemented a syntax-based translator to parse the XML file of FBD programs and translate them to Coq definitions, where the XML file is the storage form of FBD program which is standardized by IEC 61131-3, and FBD components and Coq definitions have a one-to-one correspondence. We used it to formalize our elevator control system and chose a door control FB block fb_close_delay (Fig. 4) as an example.

Informally, this block provides a function to determine when to send a signal to close the elevator door after it has opened. Once the block receives the enable signal that the door has been opened, the first delay duration (*force-open*) begins. During this initial *force-open* period, 1) the block will not send the close signal; and 2) if it receives a signal to open the door, the elapsed time of this duration will reset. Once the *force-open* duration ends, the second duration (*keep-open*) begins. In this *keep-open* duration, 1) if the block receives a signal to close the door, or the *keep-open* duration ends, it will send the close signal; and 2) if it receives a signal to open the door, the elapsed time of this duration will reset. These two durations run only when the enable signal holds true; otherwise, the program will be reset to the initial (disabled) state.

Although it is relatively small, its behavior involving timing is well-defined, and its complexity is suitable for us to describe our formalism to FBD programs. Therefore, we choose it to demonstrate how we formalize a graphical program and encode the function it expresses.

A. FBD IMPLEMENTATIONS

Definition 7 (Syntax of FBD-Implemented POU): A POU implemented in FBD can be represented as $\mathcal{P} = V \cup Prog$, *where*

- $V = V_{in} \cup V_{out} \cup V_{intl}$ *is the declaration of variables, and*
- $Prog = \cup_{i=1}^{n} Ntwk_i$ *is the program body which consists of FBD networks.*
- *Ntwk is an FBD network composed of graphical blocks, including standard functions, standard function blocks, and user-defined blocks.*

Additionally, we assume that the FBD networks $Ntwk_i$ *in Prog are sorted by their execution order.*

Using fb_close_delay as an example, we now illustrate how to formalize an FBD-implemented POU $\mathcal{P}_{close\_delay} = V \cup Prog$ into the POU execution model and $\mathcal{E}_{close\_delay} = (I, O, S, s_0, trans, exec)$ in two steps.

1) FORMALIZE VARIABLES *V*

We formalize fb_close_delay 's variables V in a manner similar to the preceding section. fb_close_delay takes three input variables $V_{in} = \{opened, open\_request, close\_request\}$, where we define the following:

- opened: a Boolean value denoting the elevator door has been opened.
- open_request: a Boolean value denoting the request to open the elevator door.
- close_request : a Boolean value denotes the request to close the elevator door.

Therefore, the input set is $I = \mathbb{B} \times \mathbb{B} \times \mathbb{B}$, which produces an output variable $V_{out} = \{close\}$, where close is a Boolean value indicating a signal to close the elevator door. Thus, the output set $O = \mathbb{B}$.

fb_close_delay has four internal variables $V_{intl} = \{timeout\_force, timeout\_keep, force, keep\}$, where we define the following:

- force and keep: TON instances, which count the *force-open* and *keep-open* durations.
- timeout_force and timeout_keep: Boolean variables indicating when the *force-open* and *keep-open* durations have ended.

Therefore, the state set is $S = \mathbb{B} \times \mathbb{B} \times S_{TON} \times S_{TON} \times \mathbb{B}$, where S_{TON} is the state set of TON's execution model $\mathcal{E}_{TON}$.

We predefine I_common and O_common in Coq from module FB_CLOSE_DELAY, which will be used to define I and O of the FBD implementations and specifications. We define the constants pt_force and pt_keep, which are the preset timeout of two timers:

```
Record I_common: Type:=
mk_in{OPENED: bool;
      OPEN_REQUEST: bool;
      CLOSE_REQUEST: bool}.
Record O_common: Type:=
mk_out{CLOSE: bool}.
Definition pt_force: TIME:= 10.
Definition pt_keep: TIME:= 5.
```

We define I, O, S, s_0, and the default output o_default with I_common and O_common in Coq as follows:

```
Record S_proto: Type:=
mk_st{TIMEOUT_FORCE: bool;
TIMEOUT_KEEP: bool;
      TON_FORCE: TON.S;
      TON_KEEP: TON.S;
      _CLOSE: bool}.
Definition I:= I_common.
Definition O:= O_common.
Definition S:= S_proto.
Definition s0: S:= \ldots
Definition o_default:= \ldots
```

2) FORMALIZE PROGRAM BODY *Prog*

The implementation of fb_close_delay has two FBD networks; that is, $Prog = Ntwk_1 \cup Ntwk_2$. We define an evaluation function $eval_i : I \times O \times S \rightarrow O \times S$ for each FBD network $Ntwk_i$. Each network takes the input i of this cycle, the output o and the state s updated by last network (or the default output $o_{default}$ and the initial state s of this cycle, in terms of the first network), updates o and s during its execution, and returns the new output o' and the new state s'. The transition function *trans* will be defined with these evaluation functions.

```
FUNCTION_BLOCK fb_close_delay
VAR_INPUT
    opened: BOOL;
    open_request: BOOL;
    close_request: BOOL;
END_VAR
VAR_OUTPUT
    close: BOOL;
END_VAR
VAR
    timeout_force: BOOL;
    timeout_keep: BOOL;
    force: TON;
    keep: TON;
END_VAR
```

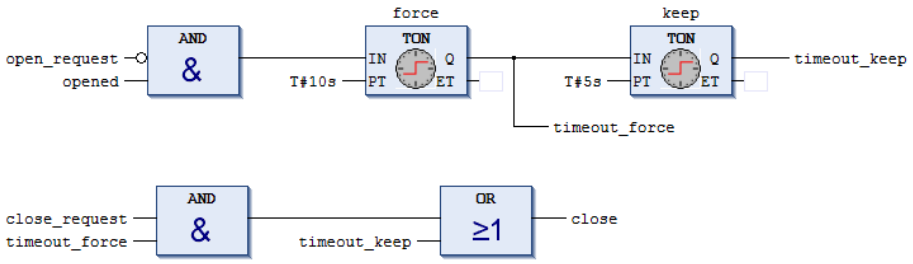


FIGURE 4. FBD implementation of FB fb_close_delay.

Every FBD network is composed of interconnected graphical blocks. According to the evaluation rules for FBD networks, a network or block in a network will be evaluated once all its inputs are produced by previous blocks or received from the periphery. We assume that the FBD networks $Ntwk_i$ in *Prog* are sorted by their execution order. In terms of fb_close_delay, its first network $Ntwk_1$ has four blocks in which the negative symbol of the input variable open_request denotes a NOT block. They will be evaluated in the following order: NOT, AND, TON (named force), TON (named keep). The evaluation function $eval_1$ of $Ntwk_1$ is defined as follows in Coq:

```
1  Definition eval1 (i: I) (cont: O * S)
2  : O * S:= let (o, s):= cont in
3  let (o_NOT, _):=
4  NOT.trans {|NOT.IN:=OPEN_REQUEST i|} NOT.s0
       in
5  let (o_AND, _):=
6  AND.trans {|AND.IN1:=NOT.OUT o_NOT;
7     AND.IN2:=OPENED i|} AND.s0 in
8  let (o_TON_FORCE, s_TON_FORCE):=
9  TON.trans {|TON.IN:=AND.OUT o_AND;
10           TON.PT:=pt_force|}(TON_FORCE s) in
11 let (o_TON_KEEP, s_TON_KEEP):=
12 TON.trans {|TON.IN:=TON.Q o_TON_FORCE;
13           TON.PT:=pt_keep|} (TON_KEEP s) in
14 (o, s & {|TIMEOUT_FORCE:= TON.Q o_TON_FORCE
       |}
15 & {|TIMEOUT_KEEP:= TON.Q o_TON_KEEP|}
16 & {|TON_FORCE:= s_TON_FORCE|}
17 & {|TON_KEEP:= s_TON_KEEP|}).
```

where these FBs are encoded in line 3-4, 5-7, 8-10, 11-13, and the output and state are updated in line 14-17. The evaluation functions of the other networks are defined in the same manner. These evaluation functions are defined as a list evals in Coq:

```
1  Definition eval2 (i:I)(cont:O * S):O * S:= \
       ldots
2  Definition evals:= (eval1::eval2::nil)%list.
```

After defining the evaluation functions of each network, we define an assistant function build_trans to construct the transition function *trans* as a function trans in Coq with the list of evaluation functions evals and the default output o_default:

```
1  Definition build_trans {I O S: Type}
2  (evals: list (I -> O * S -> O * S))
3   (o_default: O) (i: I) (s: S)
4  : O * S:=
5   (fix f (evals: list (I -> O * S -> O * S))
6   (i: I) (cont: O * S) {struct evals}
7  : O * S:=
8   match evals with
9   | (eval:: evals')
10  let cont':= eval i cont in
11  f evals' i cont'
12  | nil => cont
13  end) evals i (o_default,s).
```

Finally, the transition function *trans* and the execution function *exec* of fb_close_delay are defined as follows:

```
1  Definition trans:= build_trans evals
       o_default.
2  Definition exec:= build_exec trans.
```

B. FBD SPECIFICATIONS

Definition 8 (Specification of POU): The specification of a POU is modeled as an execution model $\mathcal{E}_{spec} = (I, O, S, s_0, trans, exec)$.

We continue with the fb_close_delay example. Using the informal description of its behavior from the beginning of the section, we now model its specification formally in the execution model $\mathcal{E}_{spec}$. We use $\mathcal{E}_{impl}$ to denote the execution model we formalized in the preceding subsection.

The input set I_{spec} and output set O_{spec} of $\mathcal{E}_{spec}$ are identical to $\mathcal{E}_{impl}$. Given a natural number $et \in \mathbb{N}$, the state set $S_{spec} = (unopened, force\_open(et), keep\_open(et))$ contains three states, where we define the following:

- $unopened \in S_{spec}$ denotes that the elevator door is not yet completely open.
- $force\_open(et)$ denotes that the elevator door has been in the *force-open* duration for an elapsed time et, where $force\_open : \mathbb{N} \to S_{spec}$.
- $keep\_open(et)$ denotes that the elevator door has been in the *keep-open* duration for an elapsed time et, where $keep\_open : \mathbb{N} \to S_{spec}$.
- $closing \in S_{spec}$ denotes the intermediate state before a door closing signal takes effort. This state characterizes the one-cycle delay behavior of explicit/implicit feedback paths.

We define I_{spec}, O_{spec}, and S_{spec} in Coq as follows:

```
1  Definition I:= I_common.
2  Definition O:= O_common.
3  Definition o_default:= o_default.
4  Inductive S_proto: Type:=
5  | FORCE_OPEN: TIME -> S_proto
6  | KEEP_OPEN: TIME -> S_proto
7  | UNOPENED: S_proto
8  | CLOSING: S_proto.
9  Definition S:= S_proto.
10 Definition s0: S:= UNOPENED.
```

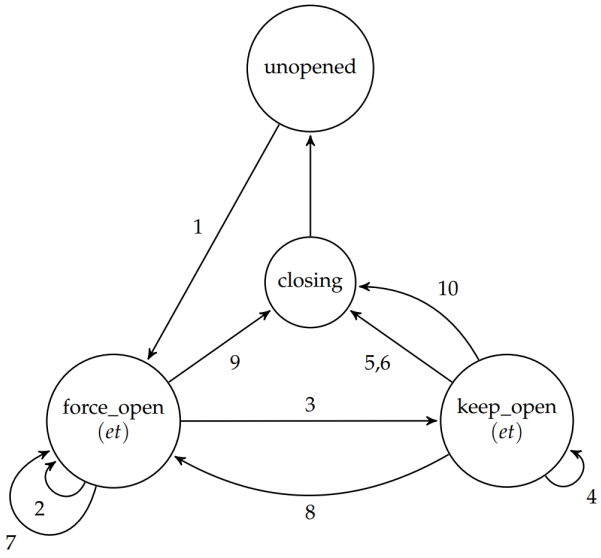


FIGURE 5. Transition diagram of fb_close_delay's specification.

We formalize the informal behavior described at the beginning of this section as transition rules 1 through 10 in the transition diagram (Fig. 5), where nodes denote states and arrows denote transition rules. In Table 1, we describe these transition rules in detail. The $\rightarrow$ column denotes transition rules. Columns s and s' denote the states that a transition rule transfers from and to, respectively. The column *input*/*output* denotes the input and output set of this transition rule, where the terms of input set are ordered by *opened*, *open_request*, *close_request*, *closing*. We use 1 and 0 to denote *true* and *false* of Boolean variables. The notation _ means the value of this input variable is omitted in this transition rule. pt_{force} and pt_{keep} are the preset timeouts of *force-open* and *keep-open* durations.

Taking transition rule 2 as an example, given an elapsed time $et \in \mathbb{N}$, a transition from $force\_open(et)$ to $force\_open(et+1)$ with output $\{0\}$ happens only when $et < pt_{force}$ and input is $\{1, 0, \_\}$. The transition function $trans_{spec}$ is encoded in Coq as follows according to these transition rules:

```
1  Definition trans (i: I) (s: S): (O * S):=
2  if negb (OPENED i) then
3  ({|CLOSE:= false|}, UNOPENED)
4  else match s with
5  | FORCE_OPEN et =>
6  if OPEN_REQUEST i then
7  ({|CLOSE:= false|}, FORCE_OPEN 1)
8  else if ge_dec et pt_force then
9  ({|CLOSE:= false|}, KEEP_OPEN 1)
10 else
11 ({|CLOSE:= false|}, FORCE_OPEN (et+1))
12 | KEEP_OPEN et => \ldots
13 | UNOPENED => \ldots
14 end.
```

where lines 6-7, 8-9, 10-11 encodes transition rules 3, 2, 9. Other codes are omitted to save space.

The execution function $exec_{spec}$ is defined using the assistant function build_exec with trans as a corecursive function exec.

```
1  Definition exec:= build_exec trans.
```

VI. EQUIVALENCE VERIFICATION

In this section, we present formal equivalence verification between FBD programs and given specifications. Because we assume that each scan cycle goes through a constant unit period, we use cycle-dependent behavior to express time-dependent behavior. The time-dependent equivalence is defined as bisimilarity (defined in Section III-C) on infinite traces. The outline of equivalence proof is the following.

A. CORRESPONDING STATES

Before proving the equivalence between FBD implementation $\mathcal{E}_{impl}$ and specification $\mathcal{E}_{spec}$, we need to relate their states. We define a relation *Corr* to specify that when two states are corresponding.

Definition 9 (Corresponding Relation Between States): Given two state sets S_1 and S_2, the corresponding relation $Corr \subseteq (S_1 \times S_2)$ means that when $s_1 \in S_1, s_2 \in S_2$, $(s_1, s_2) \in Corr$ implies that s_1 and s_2 are corresponding.

For example, if $S_{spec} = keep\_open\ t$, then a corresponding S_{impl} should satisfy:

- $S_{impl}.timeout\_force = true$
- $S_{impl}.timeout\_keep = false$
- $S_{impl}.ton\_keep.et = t$
- $S_{impl}.\_close = false$

We define the corresponding relation of S_{impl} and S_{spec} as a predicate *isCorr* in Coq.:

```
1  Inductive isCorr: CLOSE_DELAY_impl.S ->
       CLOSE_DELAY_spec.S -> Prop:=
2  | IsCorr: forall (s_impl: CLOSE_DELAY_impl.S
     ) (s_spec: CLOSE_DELAY_spec.S),
3  \ldots -> isCorr s_impl s_spec.
```

B. PROOF OUTLINE

Take the FBD implementation $\mathcal{E}_{impl}$ and the given specification $\mathcal{E}_{spec}$ from Section V as example. They are formalized in our coinduction execution model $\mathcal{E} = (I, O, S, s_0, trans, exec)$ (defined in Section III-D). We use the bisimilarity on their output traces to characterize their equivalence. Assume that:

- They have the same input trace I whose head element is i.

TABLE 1. Transition rules of fb_close_delay's specification.

$\rightarrow$	s	s'	$input/output$
1	unopened	force_open(1)	{1,_,_}/{0}
2	force_open(et) ($et < pt_{force}$)	force_open($et+1$)	{1,0,_}/{0}
3	force_open(et) ($et \geq pt_{force}$)	keep_open(1)	{1,0,_}/{0}
4	keep_open(et) ($et < pt_{keep}$)	keep_open($et+1$)	{1,0,_}/{0}
5	keep_open(et) ($et \geq pt_{keep}$)	closing	{1,0,0}/{1}
6	keep_open(et)	closing	{1,0,1}/{1}
7	force_open(et)	force_open(1)	{1,1,_}/{0}
8	keep_open(et)	force_open(1)	{1,1,_}/{0}
9	force_open(et)	closing	{0,_,_}/{0}
10	keep_open(et)	closing	{0,_,_}/{0}
11	closing	unopened	{0,_,_}/{0}

- Their initial states are corresponding: $(s\_0_{impl}, s\_0_{spec}) \in Corr$.

The equivalence between $\mathcal{E}_{spec}$ and $\mathcal{E}_{impl}$ is formally defined as the theorem CLOSE_DELAY_eq:

```
Theorem CLOSE_DELAY_eq:
forall (XI: trace I_common)
(s_impl: CLOSE_DELAY_impl.S)
(s_spec: CLOSE_DELAY_spec.S),
isCorr s_impl s_spec ->
Bisimilar (CLOSE_DELAY_impl.exec XI s_impl)
(CLOSE_DELAY_spec.exec XI s_spec).
```

This theorem is true if and only if the following two lemmas are true:

- head_equivalent : The head element of two output traces are equal:

```
Lemma head_equivalent:
forall (XI: trace I_common)
(s_impl: CLOSE_DELAY_impl.S)
(s_spec:
CLOSE_DELAY_spec.S),
isCorr s_impl s_spec ->
eq (head
(CLOSE_DELAY_impl.exec XI s_impl))
(head
(CLOSE_DELAY_spec.exec XI s_spec)).
```

- state_invariant : For arbitrary corresponding state pair $(s_{impl}, s_{spec}) \in Corr$, the next state pair (s'_{impl}, s'_{spec}) still satisfies $(s'_{impl}, s'_{spec}) \in Corr$:

```
Lemma state_invariant:
forall (i: I_common)
(s_impl: CLOSE_DELAY_impl.S)
(s_spec: CLOSE_DELAY_spec.S),
isCorr s_impl s_spec ->
isCorr
(snd (CLOSE_DELAY_impl.trans i s_impl))
(snd (CLOSE_DELAY_spec.trans i s_spec)).
```

The proof of the theorem CLOSE_DELAY_eq starts with the tactic cofix coH, which introduces the goal as an assumption in the current context. The use of this assumption should satisfy the guardedness condition, which means the result type of proof must be of the coinductive type and all recursive calls must occur inside one of the arguments of a constructor of the coinductive type.

We then introduce the local variables and premises of the implications and apply the constructor bisimilar to destruct the current goal into two sub-goals. The first sub-goal is the equivalence between the head terms of two output traces:

```
\ldots
============================
eq (head (CLOSE_DELAY_impl.exec XI s_impl))
(head (CLOSE_DELAY_spec.exec XI s_spec))
```

This can be solved by applying the lemma head_equivalent.

The second sub-goal is the bisimulation between the tail traces of the output traces:

```
\ldots
============================
Bisimilar (tail (CLOSE_DELAY_impl.exec XI
    s_impl))
(tail (CLOSE_DELAY_spec.exec XI s_spec))
```

We destruct XI as i0 ::: XI' and transform the sub-goal to another form:

```
\ldots
coH: forall (XI: trace I_common)
(s_impl: CLOSE_DELAY_impl.S)
(s_spec: CLOSE_DELAY_spec.S),
isCorr s_impl s_spec ->
Bisimilar (CLOSE_DELAY_impl.exec XI s_impl)
(CLOSE_DELAY_spec.exec XI s_spec)
============================
Bisimilar eq
(CLOSE_DELAY_impl.exec XI'
(snd (CLOSE_DELAY_impl.trans i0 s_impl)))
(CLOSE_DELAY_spec.exec XI'
(snd (CLOSE_DELAY_spec.trans i0 s_spec)))
```

(snd (CLOSE_DELAY_impl.trans i0 s_impl)) and (snd (CLOSE_DELAY_spec.trans i0 s_spec)) are the next pair of states. Next, the form of goal can be unified with the coinductive assumption coH. Applying coH to the subgoal obtains the last goal:

```
\ldots
H_corr: isCorr s_impl s_spec
============================
isCorr
(snd (CLOSE_DELAY_impl.trans i0 s_impl))
(snd (CLOSE_DELAY_spec.trans i0 s_spec))
```

By applying the lemma state_invariant with the premise H_corr, we finish the proof of the theorem CLOSE_DELAY_eq.

VII. CONCLUSION

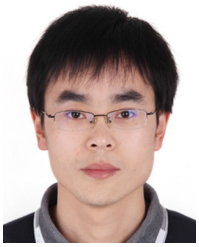
This paper fills the vacancy of the effective formalization and formal equivalence verification of real-time FBD programs

using theorem proving. We firstly propose a coinduction PLC execution model to characterize the time-dependent behavior of non-terminating PLC programs. Then we present a method to formalize FBD programs and given specifications into this model. A deterministic subset of FBD is considered. This method is more practical than existing works. Finally, we prove the equivalence relation using bisimulation. In practice, we have implemented a translator to parse XML files of FBD programs and translate them into Coq definitions. This methodology has been applied to verify that our elevator control program conforms to the given specification.

In the future, we will formalize specific domain FBs, where motion control will be our most important focus since PLCopen has already standardized its FBs. Textual languages like ST will be added to model user-defined FBs and safety and liveness properties will be proved in this coinduction model.

REFERENCES

[1] Coq. (2021) *Coq Documentation*. Accessed: Oct. 26, 2021. [Online]. Available: https://coq.inria.fr/documentation
[2] H. Janicke, A. Nicholson, S. Webber, and A. Cau, "Runtime-monitoring for industrial control systems," *Electronics*, vol. 4, no. 4, pp. 995–1017, Dec. 2015.
[3] M. Caselli, E. Zambon, and F. Kargl, "Sequence-aware intrusion detection in industrial control systems," in *Proc. 1st ACM Workshop Cyber-Phys. Syst. Secur.*, Apr. 2015, pp. 13–24.
[4] F. Mercaldo, F. Martinelli, and A. Santone, "Real-time SCADA attack detection by means of formal methods," in *Proc. IEEE 28th Int. Conf. Enabling Technol., Infrastruct. Collaborative Enterprises (WETICE)*, Jun. 2019, pp. 231–236.
[5] *Programmable Controllers—Part 3: Programming Languages*, document IEC 61131-3, IE Commission, 1993.
[6] S. J. Mason, "Feedback theory-some properties of signal flow graphs," *Proc. IRE*, vol. 41, no. 9, pp. 1144–1156, Sep. 1953.
[7] V. Vyatkin, "Software engineering in industrial automation: State-of-the-art review," *IEEE Trans. Ind. Informat.*, vol. 9, no. 3, pp. 1234–1249, Aug. 2013.
[8] A. Guignard, J.-M. Faure, and G. Faraut, "Model-based testing of PLC programs with appropriate conformance relations," *IEEE Trans. Ind. Informat.*, vol. 14, no. 1, pp. 350–359, Jan. 2018.
[9] M. Obermeier, S. Braun, and B. Vogel-Heuser, "A model-driven approach on object-oriented PLC programming for manufacturing systems with regard to usability," *IEEE Trans. Ind. Informat.*, vol. 11, no. 3, pp. 790–800, Jun. 2015.
[10] F. Basile, P. Chiacchio, and D. Gerbasio, "On the implementation of industrial automation systems based on PLC," *IEEE Trans. Autom. Sci. Eng.*, vol. 10, no. 4, pp. 990–1003, Oct. 2013.
[11] E. Estevez and M. Marcos, "Model-based validation of industrial control systems," *IEEE Trans. Ind. Informat.*, vol. 8, no. 2, pp. 302–310, May 2012.
[12] B. F. Adiego, D. Darvas, E. B. Viñuela, J. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Trans. Ind. Informat.*, vol. 11, no. 6, pp. 1400–1410, Dec. 2015.
[13] D. Darvas *et al.*, "Formal verification of complex properties on PLC programs," in *Proc. Int. Conf. Formal Techn. Distrib. Objects, Compon., Syst.* Berlin, Germany: Springer, 2014, pp. 284–299.
[14] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in instruction list," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, vol. 4, Oct. 2000, pp. 2449–2454. [Online]. Available: http://ieeexplore.ieee.org/document/884359/
[15] O. Pavlovic and H.-D. Ehrich, "Model checking PLC software written in function block diagram," in *Proc. 3rd Int. Conf. Softw. Test., Verification Validation*, 2010, pp. 439–448.
[16] D. Soliman, K. Thramboulidis, and G. Frey, "Transformation of function block diagrams to UPPAAL timed automata for the verification of safety applications," *Annu. Rev. Control*, vol. 36, no. 2, pp. 338–345, 2012.
[17] E. M. Clarke *et al.*, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Berlin, Germany: Springer, 2011, pp. 1–30.
[18] H. Wan, G. Chen, X. Song, and M. Gu, "Formalization and verification of PLC timers in coq," in *Proc. 33rd Annu. IEEE Int. Comput. Softw. Appl. Conf.*, Dec. 2009, pp. 315–323. [Online]. Available: http://ieeexplore.ieee.org/document/5254244/
[19] H. Wan, X. Song, and M. Gu, "Parameterized specification and verification of PLC systems in coq," in *Proc. 4th IEEE Int. Symp. Theor. Aspects Softw. Eng.*, Aug. 2010, pp. 179–182. [Online]. Available: http://ieeexplore.ieee.org/document/5587715/
[20] J. Newell, L. Pang, D. Tremaine, A. Wassyng, and M. Lawford, *Formal Translation of IEC 61131-3 Function Block Diagrams to PVS with Nuclear Application*. Cham, Switzerland: Springer, 2016, pp. 206–220.
[21] L. Pang, C.-W. Wang, M. Lawford, and A. Wassyng, "Formal verification of function blocks applied to iec 61131-3," *Sci. Comput. Program.*, vol. 113, pp. 149–190, Dec. 2015.
[22] J. Yoo, S. Cha, and E. Jee, "A verification framework for fbd based software in nuclear power plants," in *Proc. 15th Asia–Pacific Softw. Eng. Conf.*, 2008, pp. 385–392.
[23] D. Park, "Concurrency and automata on infinite sequences," in *Theoretical Computer Science*, P. Deussen, Ed. Berlin, Germany: Springer, 1981, pp. 167–183.
[24] Y. Bertot, "A short presentation of Coq," in *Proc. Int. Conf. Theorem Proving Higher Order Logics*. Berlin, Germany: Springer, 2008, pp. 12–16.
[25] R. Milner, *Communication Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, 1989.
[26] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell Syst. Tech. J.*, vol. 34, no. 5, pp. 1045–1079, Sep. 1995.



JIANYONG ZHAO received the B.S. and M.S. degrees in computer science and technology from Hangzhou Dianzi University, Hangzhou, China, in 2002 and 2005, respectively. He is currently a Senior Experimentalist with the Institute of Intelligent and Software Technology, Hangzhou Dianzi University. His research interests include embedded systems, software development methods and tools, and intelligent control and automation.

ZHE TAO is currently pursuing the bachelor's degree with the Institute of Intelligent and Software Technology, Hangzhou Dianzi University, Hangzhou, China. His research interests include formal verification, logic, and programming languages.