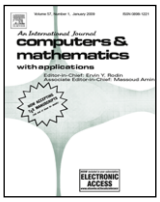
Computers and Mathematics with Applications 60 (2010) 695–705

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa



Specifying and verifying PLC systems with TLA$^+$: A case study[☆]

Hehua Zhang[a,*], Stephan Merz[b], Ming Gu[c]

[a] DCST, Tsinghua University, Beijing, PR China

[b] INRIA Nancy–Grand Est & Loria, Nancy, France

[c] School of Software, KLISS, TNLIST, Tsinghua University, Beijing, PR China

ARTICLE INFO

Article history:
Received 19 October 2009
Received in revised form 14 April 2010
Accepted 14 May 2010

Keywords:
TLA+
PLC
Formal specification
Model checking

ABSTRACT

We report on a method for formally specifying and verifying programmable logic controllers (PLCs) in the specification language TLA$^+$. The specification framework is generic. It separates the description of the environment from that of the controller itself and its structure is consistent with the scan cycle mechanism used by PLCs. Specifications can be parameterized with the number of replicated components. In our experience, the structuring mechanisms of TLA$^+$ help to obtain clear, well-organized, and configurable specifications, finite instances of which are verified by the TLA$^+$ model checker TLC. We have validated our approach on a concrete case study, a controller for fire fighting equipment in a ship dock, and report on the results obtained for this case study.

1. Introduction

Programmable Logic Controllers (PLCs) are widely used in industry [1,2] for embedded systems. A PLC interacts with user commands and with the controlled plant, following a so-called scan cycle mechanism that starts with inputting environment data (including user commands), then performs a local computation, and finally outputs the results to the environment. With their increasing use, PLC systems become more and more complex. Formal methods promise to ensure the correctness of complex systems. Through formal specifications of the designed behaviors of the system, they can help a designer to clarify the design ideas, eliminate ambiguities and find errors with the help of tools.

TLA$^+$ is a formal specification language, which was designed for specifying and reasoning about concurrent and reactive systems [3]. It can formulate both system behaviors and their properties in a single logic formalism. Moreover, it provides supports for parameterization, advanced data structures, etc. Last but not the least, it has powerful tool support, in particular the TLA$^+$ model checker TLC [4]. Theorem-proving support in the form of a proof assistant is also being developed, but we have not used it for our specifications of PLCs.

In this paper, we propose a method for specifying PLC systems using TLA$^+$. In particular, we specify the reaction cycle according to the scan cycle mechanism of PLCs. Exploiting the parameterization and abstraction facilities of TLA$^+$, we obtain a generic specification pattern per module that distinguishes between actions of the user, the controller, and the plant. This pattern is instantiated for a concrete PLC, and different modules are composed to obtain the overall system specification. We exhibit the method on a running example, a controller for fire fighting in a ship dock, and report our experiences on verifying the specifications with the model checker TLC.

The paper is organized as follows. We introduce the specification language TLA$^+$ in Section 2. The dock fire-fighting example is described in Section 3. In Section 4, we illustrate our method of specifying a PLC system with TLA$^+$. The properties

☆ This research is sponsored by NSFC Program (No. 90718039) and 973 Program (No. 2010CB328003) of China.

* Corresponding author.
E-mail address: zhang-hh04@mails.tsinghua.edu.cn (H. Zhang).

0898-1221/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2010.05.017

```
---------------------------- MODULE DigitalClock ----------------------------
EXTENDS Naturals
VARIABLE hr, min
 **Type invariant defintion **
TypeInv ≜  ∧ hr ∈ (0 . . 23)
           ∧ min ∈ (0 . . 59)
 **Initialization **
Init   ≜   ∧ hr = 0
           ∧ min = 0
 **Action definitions **
ClockAction1 ≜  ∧ hr < 23
                ∧ min′ = IF min = 59 THEN 0 ELSE  min + 1
                ∧ hr′ = IF min = 59 THEN hr + 1 ELSE  hr
ClockAction2 ≜  ∧ hr = 23
                ∧ min′ = IF min = 59 THEN 0 ELSE  min + 1
                ∧ hr′ = IF min = 59 THEN 0 ELSE  hr
Next ≜   ∨ ClockAction1
         ∨ ClockAction2
vars ≜ ⟨hr, min⟩
 **The specification **
DigitalClock  ≜   Init ∧ □[Next]_vars ∧ WF_vars(Next)
-----------------------------------------------------------------------------
```

Fig. 1. A digital clock TLA$^+$ specification.

description and the experimental results with the tool TLC are presented in Section 5. In Section 6, we explain the features of our work by comparing it with other model checking languages. Finally, we conclude the paper and present future work.

2. The TLA$^+$ specification language

The specification language TLA$^+$ [3,5] is based on the Temporal Logic of Actions TLA [6], a dialect of linear-time temporal logic, and on Zermelo–Fränkel set theory for representing data structures. Built on ordinary mathematics together with a light-weight temporal logic, TLA$^+$ can be used to write precise, formal specifications of discrete systems.

To get an intuitive understanding of TLA$^+$ specifications, a digital clock example is shown in Fig. 1, with the comment lines shadowed.

TLA$^+$ specifications are composed by modules. With the EXTENDS statement, a module can extend standard TLA$^+$ modules or the modules already defined in another specification. The variables should be declared before use by the VARIABLES statement. A type invariant definition is not mandatory but often adopted to present the type constraint that the specification should satisfy. A TLA$^+$ specification of a transition system is defined by a formula, with the form

$$Init \wedge \Box[Next]_v \wedge L$$

where v is a tuple containing all state variables of the system. The first conjunct *Init* describes the possible initial states of the system. The second conjunct of the specification asserts that every step (i.e., every pair of successive states in a system run) either satisfies *Next* or leaves the term v (and therefore all state variables) unchanged. Allowing for such stuttering steps is a key ingredient to obtain compositionality of specifications, however, it means that executions that stutter indefinitely are allowed by the specification. The third conjunct *L* is a temporal formula stating the liveness conditions of the specification, and in particular can be used to rule out infinite stuttering.

As to the digital clock specification in Fig. 1, we take zero hour and zero minute as the initial state of the system. The next-state action can be either *ClockAction1* or *ClockAction2*. The former defines how the minute and the hour change when the hour is less than 23, while the latter defines the case when the hour equals 23. The liveness constraint is given by the form $WF_v(A)$, to represent the weak-fairness constraint of action $\langle A\rangle_v$. It requires that the action $\langle A\rangle_v$ (defined as $A \wedge v' \neq v$) holds infinitely often provided it is persistently enabled. Since the action $\langle Next\rangle_{vars}$ is always enabled, it ensures the movement of the clock, instead of it stopping forever.

TLC is a model checker designed and implemented for TLA$^+$ specifications. It analyzes a finite instance of a TLA$^+$ specification and checks whether desired properties are true in every execution of that instance. It is an explicit-state, on-the-fly model checker written in Java. TLC keeps data on disk and makes efficient use of the disk by a sophisticated algorithm for not limiting the size of the specification to be handled. TLC is also multi-threaded and can take advantage of multiprocessors [4].

3. The dock fire-fighting system description

Our running example is a system used to fight fires that may happen at ship docks. It operates the fire-fighting equipment under the control of a user and displays information about the current operating state. The physical configuration of the dock

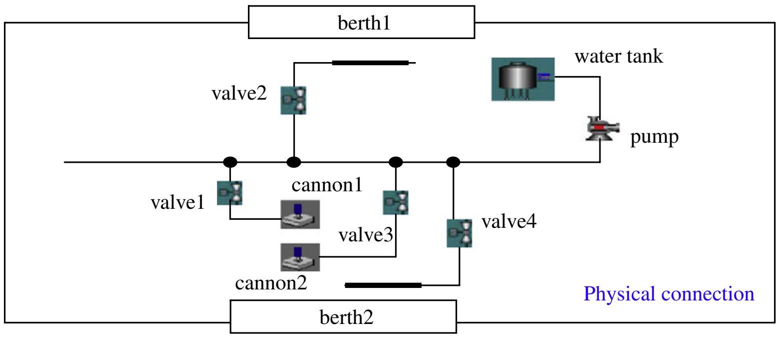


Fig. 2. The physical configuration of the system.

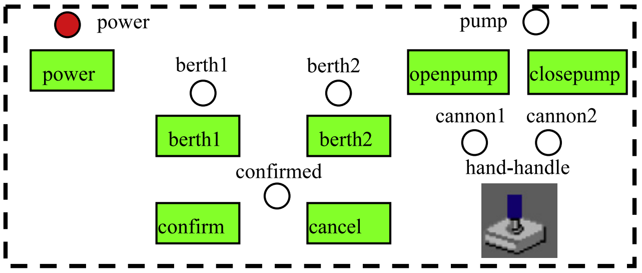


Fig. 3. The architecture of the system.

is illustrated in Fig. 2. The dock has two berths for boats. There are two water cannons that can be used (exclusively) for fire fighting. Water is supplied to the cannons by a pump that draws water from a water tank. Several valves connect the different components. For example, *cannon1* can be used for fire fighting only if both *valve1* and *valve2* are opened.

The control panel contains buttons and status indicators for the different components of the dock, as shown in Fig. 3. There is a self-locking button for power, six non self-locking buttons for starting and stopping the pump, selecting the place (*berth1* or *berth2*) on fire, confirming and canceling the indicated settings. A hand-handle is used to control the direction of the cannon in use. There are seven lights to indicate the status of the system. For example, the light *pump* is on when the pump is opened and is off when the pump is closed. Cannons are selected implicitly by the choice of the berth; the lights *cannon1* and *cannon2* reflect which cannon is in use.

When there is a fire-fighting request, the user can control the equipment using the panel. The preparatory steps of the operations are as follows: (1) power up the system (by pressing down the *power* button); (2) open the pump (by pressing the *openpump* button; (3) according to the place on fire, select *berth1* or *berth2*; (4) confirm the selection; (5) control the direction of the cannon by the hand-handle. To shut down the system, the user proceeds as follows: (6) close the pump first (by pressing the *closepump* button); then (7) press the *cancel* button to finish the procedure. Finally, one can restart by going back to step (2) or finish by uplifting the *power* button.

The PLC system responds according to the user operations, its computations and the feedback of the controlled plants. Specifically, it opens or closes the pump, the corresponding valves, sends the directions to the cannon in use and shows the system status by the lights on the control panel.

4. Specifying the PLC system with TLA$^+$

In this section, we introduce how we specify a PLC system with TLA$^+$. Our presentation of the general specification pattern is explained using the dock fire-fighting example.

4.1. Specifying system parameters

Just as with real systems, specifications should be written with regard to extensibility. For example, we can specify a concrete dock fire-fighting system consisting of two fire cases and a particular physical routing. However, as a high-level specification, it is better to be flexible and parameterize the system instead of restricting it to a fixed configuration. Of course, what and how much to generalize is an important decision the system designer has to make, which depends on the practical situation.

For this case, users and engineers told us that the mode of operation is fixed, but that it is desirable to adapt the controller to different numbers of fire cases to fit further fire-fighting requests. Moreover, the correspondences between cannons and fire-fighting cases should be parameterizable, although the relation is always one-to-one. Moreover, the physical route is also susceptible to change when new equipment is added. Hence, we take the fire cases, the cannons, the correspondence between fire cases and cannons, the valves and the physical connections between valves and cannons as parameters. The parameters are declared by the CONSTANTS statements in TLA$^+$:

$$\begin{array}{ll} \text{CONSTANTS} & \mathit{FireCase}, \\ & \mathit{Cannon}, \\ & \mathit{Valve}, \\ & \mathit{CannonInCase}(\_), \\ & \mathit{BelongTo}(\_) \end{array}$$

where *FireCase*, *Cannon*, *Valve* denotes the set of all the fire cases, cannons, and valves, respectively. *CannonInCase*(_) and *BelongTo*(_) are constant operators. $\mathit{CannonInCase}(c) = i$ represents that the cannon c is used in the fire case i. $\mathit{BelongTo}(v) = c$ denotes that the valve v belongs to the cannon c. To get the concrete specification of the current system, what we need is just to replace each parameter with its actual value. For our example, the correspondence is specified as follows.

$$\mathit{FireCase} = \{\text{“berth1”}, \text{“berth2”}\}$$
$$\mathit{Cannon} = \{\text{“cannon1”}, \text{“cannon2”}\}$$
$$\mathit{Valve} = \{\text{“valve1”}, \text{“valve2”}, \text{“valve3”}, \text{“valve4”}\}$$
$$\mathit{CannonInCase}(\text{“cannon1”}) = \text{“berth1”}$$
$$\mathit{CannonInCase}(\text{“cannon2”}) = \text{“berth2”}$$
$$\mathit{BelongTo}(\text{“valve1”}) = \text{“cannon1”}$$
$$\mathit{BelongTo}(\text{“valve2”}) = \text{“cannon1”}$$
$$\mathit{BelongTo}(\text{“valve3”}) = \text{“cannon2”}$$
$$\mathit{BelongTo}(\text{“valve4”}) = \text{“cannon2”}.$$

4.2. Variables for representing the system state

The state variables *var* of a PLC specification can be grouped into four classes: user variables (UV), system variables (SV), plant variables (PV) and auxiliary variables (AV). A group of variables is expressed as a tuple in TLA$^+$, so we have $\mathit{vars} \triangleq \text{UV} \circ \text{SV} \circ \text{PV} \circ \text{AV}$. The four classes are self-explanatory by their names. For the fire-fighting case, UV are those representing the buttons and the hand-handle. SV are those representing the lights, the valves, the pump command, the cannon command and the system state. The system responses also depend on the real status of the plant. In our case, the group PV consists of exactly one variable *realPump* representing the pump status. Variables in group AV are not related to the semantics of the system, but make it easier to write the formal specification in TLA$^+$.

A straightforward choice for representing the system state would be to use BOOLEAN for each button and light and use the variables *u_handle*, *s_handle* taking values in the set $\mathit{Direction} \triangleq \{\text{“up”}, \text{“down”}, \text{“left”}, \text{“right”}, \text{“none”}\}$ for the user hand-handle and the cannon commands. However, we would obtain at least 17 variables (7 for buttons, 7 for lights, one for user cannon, one for pump command and one for cannon command) in this case. With the system scale increasing, this representation will result in many state variables and make the specification hard to manage. Moreover, only a few variables typically change during a system step, leading to long UNCHANGED formulas (with UNCHANGED V denoting the tuple V doesn't change in the action). We therefore aim at obtaining a better state representation that leads to a concise and readable specification. Here we use three arrays for the buttons, the button lights and the cannon lights. Arrays are represented by functions in TLA$^+$. For example, the function *s_buttonLight* denoting the array of button lights is specified with the domain *ButtonLight* and the range Boolean as follows.

$$\mathit{s\_buttonLight} \in [\mathit{ButtonLight} \rightarrow \text{BOOLEAN}]$$

where $\mathit{ButtonLight} \triangleq \{\text{“power”}, \text{“pump”}, \text{“confirmed”}, \text{“berth1”}, \text{“berth2”}\}$ is the set of all the lights related to the buttons. In this way, for example, the system action that lights the power light can be written in TLA$^+$ as

$$\mathit{s\_buttonLight}' = [\mathit{s\_buttonLight} \text{ EXCEPT } ![\text{“power”}] = \text{TRUE}].$$

This formula should be compared to the equivalent action specification for the direct state representation, which reads

$$\wedge\ \mathit{s\_powerLight}' = \text{TRUE}$$
$$\wedge\ \text{UNCHANGED } \langle \mathit{s\_pump}, \mathit{s\_comfirmed}, \mathit{s\_berth}1, \mathit{s\_berth}2 \rangle.$$

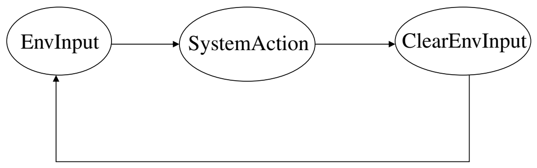


Fig. 4. The data transmission among the components.

4.3. Specifying the reaction pattern of the PLC

The fundamental purpose of our specification is to capture the interaction between the PLC controller and its environment. Here the environment refers to both user operations and feedback from the plant. There are several possible interaction modes. For example, one environment action can occur in the midst of a series of system actions, or on the contrary, the system may react only after several environment actions. However, to specify a proper interaction with users of our system, we expect that each environment action should be captured and processed by the system, even if the system does not actually need to react given its current state. This is ensured by the inherent scanning cycle running pattern of PLC controllers. It consists in receiving the input signals at the beginning of a cycle, then processing these inputs, and finally generating the outputs at the end of the cycle, after which a new cycle is started.

Moreover, the input values can be viewed as being discarded at the end of each cycle, because PLCs will recheck environment inputs at the beginning of the next cycle. In summary, the pattern on PLC reaction is illustrated in Fig. 4.

To implement this interaction in TLA$^+$, we use an auxiliary variable named *aux* to enforce the PLC scanning sequence, as shown in Fig. 4. The TLA$^+$ template for this reaction mode is written is as follows.

$$
\begin{aligned}
Next \triangleq\ &\lor \land\ aux = 0\\
& \land\ EnvInput\\
& \land\ \text{UNCHANGED SV}\\
& \land\ aux' = 1\\
&\lor \land\ aux = 1\\
& \land\ SystemAction\\
& \land\ \text{UNCHANGED UV}\\
& \land\ aux' = 2\\
&\lor \land\ aux = 2\\
& \land\ ClearEnvInput\\
& \land\ \text{UNCHANGED SV}\\
& \land\ aux' = 0.
\end{aligned}
$$

The next action *Next* can be either a environment action *EnvInput*, a system action *SystemAction* or the input clearing action *ClearEnvInput*. We will indicate the specification of these actions in the following sections. The auxiliary variable *aux* ensures proper sequencing of the individual actions.

4.4. Specifying the environment action

The environment includes both user input via the control panel and the feedback of the controlled plant (fire-fighting equipment). Both provide the inputs to the PLC. We introduce first the specification of user operations, then that of plant feedback, and finally their combination.

Specifying user actions. This part is related with the state variable tuple UV $\triangleq \langle u\_button, u\_handle\rangle$. Users can operate the buttons or the cannon handle on the control panel. In practice, users can operate the buttons and the cannon handle in an arbitrary manner, not necessarily according to the description of proper operation given at the end of Section 3. The PLC should cope with arbitrary user action; for example, it should not crash. Therefore, we define the action formula *UserAction*, which describes all possible user actions.

$$
\begin{aligned}
UserAction \triangleq\ &\land\ u\_button' \in [Button \rightarrow \text{BOOLEAN}]\\
&\land\ u\_handle' \in Direction.
\end{aligned}
$$

This formula only asserts "type correctness"; in particular, the user may press any number of buttons and/or operate the cannon to move in any direction.

Specifying feedback of the plant. As described previously, we only consider reactions of the pump and therefore have PV $= \langle realPump\rangle$ consisting of a single variable. Again, the pump need not always operate according to our expectations. For example, it may fail unexpectedly. However, we assume that the sensor data represented by variable *realPump* reflects the actual pump status. We obtain the TLA$^+$ action formula

$$PlantAction \triangleq realPump' \in \text{BOOLEAN}.$$

Deriving the environment specification. User actions and feedback of the controlled plant make up the environment of the PLC controller. We have mentioned the scan cycle mechanism of PLCs and presented the corresponding TLA$^+$ description in Section 4.3. However, the user operations and the plant feedback are independent and without any sequential constraints. We only wish to express that the variables in UV change only according to formula *UserAction* and that variables in PV change according to formula *PlantAction* (if at all). Using the square bracket notation of TLA$^+$, this can be written as

$$EnvInput \triangleq [UserAction]_{\mathrm{UV}} \wedge [PlantAction]_{\mathrm{PV}}$$

where for any action formula A and state function v, the formula $[A]_v$ is defined as $A \vee v' = v$.

4.5. Specifying system actions

The PLC system should react to any meaningful input according to its functional specification. When no reaction is defined or necessary, it should simply maintain its current state. (In more elaborate cases, one could for example react to erroneous input with an error message.) We therefore obtain the following pattern for the specification of the system action:

$$\begin{aligned} SystemAction \triangleq\ & \vee\ React \\ & \vee\ NoReact \end{aligned}$$

where *React* (specified below) describes the expected system reactions and *NoReact*, defined as

$$\begin{aligned} NoReact \triangleq\ & \wedge\ \neg\mathrm{ENABLED}\ React \\ & \wedge\ \mathrm{UNCHANGED\ SV}, \end{aligned}$$

covers the remaining cases. ENABLED *A* gives the enabling condition of the action *A*.

The action formula *React* is a disjunction of several action formulas, each describing one possible response of the system:

$$\begin{aligned} React \triangleq\ & \vee\ PowerUp \\ & \vee\ OpenPump \\ & \vee\ PumpLightOn \\ & \vee\ \exists i \in FireCase : SelectCase(i) \\ & \vee\ Confirm \\ & \vee\ ResponseHandle \\ & \vee\ ClosePump \\ & \vee\ PumpLightOff \\ & \vee\ Cancel \\ & \vee\ PowerDown. \end{aligned}$$

The meaning of the different actions are also obvious by their names. Here, we only present the detailed definition of the *OpenPump* action as an example.

The current state of the system according to its expected operating cycle is represented by the variable *s_sysState*. In our case study, this variable takes values in the set

$$\begin{aligned} StateSet \triangleq\ & \{\text{“init”}, \text{“power”}, \text{“openpumpSent”}, \text{“pumpopened”}, \text{“selected”}, \\ & \text{“cannonOnUse”}, \text{“closepumpSent”}, \text{“pumpclosed”}\}. \end{aligned}$$

Opening the pump is possible when the power is switched on and the user presses the *openpump* button, but no other button on the control panel. The action is specified by the formula

$$OpenPump \triangleq$$

$$\wedge\ s\_sysState = \text{“power”} \quad (1)$$

$$\wedge\ u\_button[\text{“power”}] \quad (2)$$

$$\wedge\ u\_button[\text{“openpump”}] \quad (3)$$

$$\wedge\ \forall i \in UnlockedButton : i \neq \text{“openpump”} \Rightarrow \neg u\_button[i] \quad (4)$$

$$\wedge\ pumpCtl' \quad (5)$$

$$\wedge\ s\_sysState' = \text{“openpumpSent”} \quad (6)$$

$$\wedge\ \mathrm{UNCHANGED}\ \langle s\_buttonLight, s\_cannonLight, s\_handle, valve\rangle. \quad (7)$$

The conjuncts (1)–(4) represent the enabling condition of the action *OpenPump*; lines (5)–(7) describe the effects of executing the action. In more detail, the action denotes that when the system is at the state *power* (line (1)), the self-locked button of “power” is not lifted up (line (2)), the *openpump* button was pressed by the user (line (3)) and no other button is pressed down at the same time (line (4)), the action *OpenPump* is enabled. When it is executed, it will send the pump opening command (line (5)), set the system state to be *openpumpSent* (line (6)), and leave the other variables unchanged (line (7)).

The specification of the other actions is very similar. Overall, the representation of transition systems in the action logic of TLA$^+$ leads to system specifications that are easy to understand and clean.

4.6. Composing the whole specification

As indicated in Section 2, the overall system specification is composed of the description of the initial states *Init*, the action formula *Next* representing the next-state relation and the liveness constraints *L*.

The initial state is easy to define by considering the tuples of variables SV, UV, PV, and AV. For the system variables SV, all the lights are off, the pump command has not been issued, and the system state is “init”. As to the user variables UV, the buttons are not pressed down initially, and we assume that the directions of the user cannon and the cannon commands are initially “none”. Similarly, the plant variable *realPump* can be assumed to be false initially. Finally, the auxiliary variable *aux* takes initial value 0. Therefore, the initial condition is defined by the following TLA$^+$ formulas:

$$\begin{aligned}
InitUser \triangleq\ & \wedge\ u\_button = [i \in Button \mapsto \text{FALSE}] \\
& \wedge\ u\_handle = \text{“none”}
\end{aligned}$$

$$InitPlant \triangleq realPump = \text{FALSE}$$

$$\begin{aligned}
InitSystem \triangleq\ & \wedge\ s\_buttonLight = [i \in ButtonLight \mapsto \text{FALSE}] \\
& \wedge\ s\_cannonLight = [i \in Cannon \mapsto \text{FALSE}] \\
& \wedge\ s\_handle = [i \in Cannon \mapsto \text{“none”}] \\
& \wedge\ s\_sysState = \text{“init”} \\
& \wedge\ valve = [i \in Valve \mapsto \text{FALSE}] \\
& \wedge\ pumpCtl = \text{FALSE}
\end{aligned}$$

$$\begin{aligned}
Init \triangleq\ & \wedge\ InitUser \\
& \wedge\ InitPlant \\
& \wedge\ InitSystem \\
& \wedge\ aux = 0.
\end{aligned}$$

The action *Next* representing the next-state relation was already defined in Section 4.3. As for the liveness constraint, we will only rule out infinite stuttering of the system and therefore define

$$L \triangleq \text{WF}_{vars}(Next)$$

to ensure that the PLC system never stops. The overall system specification is given by the TLA$^+$ formula

$$Control \triangleq Init \wedge \Box[Next]_{vars} \wedge L.$$

5. Verifying the specification

TLC is a model checker developed for TLA$^+$ specifications. Properties are again expressed as a temporal formula in TLA$^+$. Which properties should be verified, and at what point a system is considered correct, depends on the system at hand, and we cannot give a general answer. However, several generic kinds of properties arise naturally for reactive systems. We present these properties first, and then show how TLC is able to check them. We limit attention to safety properties because the liveness condition of our example is very basic.

5.1. Properties classification

Limited response properties. The system should only execute the specified actions when users operate on the control panel according to the intended interaction sequence. In other words, a system action can only be enabled (and then executed) given certain well-defined conditions. As an example, the pump should not close if the system is not at the state that the pump can be closed. This is represented by the following temporal formula in TLA$^+$:

$$\begin{aligned}
& ClosePumpNotResponse \triangleq \\
& \quad \Box(u\_button[\text{“closepump”}] \wedge s\_sysState \neq \text{“cannonOnUse”} \Rightarrow \neg(\text{ENABLED } ClosePump)).
\end{aligned}$$

This formula asserts that the action *ClosePump* is disabled unless both the user pressed the button *closepump* and the system state is “cannonOnUse”.

Another temporal formula asserts that a fire case selecting will not be responded if it’s not at the right state:

$$\begin{aligned}
& SelectCaseNotRespond \triangleq \\
& \quad \Box(\forall i \in FireCase : u\_button[i] \wedge s\_sysState \notin \{\text{“pumpopend”}, \text{“selected”}\} \Rightarrow \neg(\text{ENABLED } SelectCase(i))).
\end{aligned}$$

Competition properties. This class of properties expresses competition relationships among several objects, such that no more than one object can be selected as a result. For example, limited by the physical resources, we request that at any time, only one fire case can be selected and processed. This is described by the formula *CaseSelectOnlyOne*:

$$\begin{aligned}
& CaseSelectOnlyOne \triangleq \\
& \quad \Box(\forall i, j \in FireCase : s\_buttonLight[i] \wedge s\_buttonLight[j] \Rightarrow i = j).
\end{aligned}$$

The mutex exclusion property of cannon using can be described by the following temporal formula:

$$\begin{aligned}&\mathit{CannonUsedOnlyByOne} \triangleq\\ &\quad \Box(\forall i, j \in \mathit{Cannon} : \mathit{s\_cannonLight}[i] \wedge \mathit{s\_cannonLight}[j] \Rightarrow i = j).\end{aligned}$$

Another property we'd like to ensure is the mutex exclusion among the valves for different cannons. That is to say, at any time, only the valves belonging to the same cannon can be opened at the same time. This is described by the formula *ValveMutex*.

$$\begin{aligned}&\mathit{ValveMutex} \triangleq\\ &\quad \Box(\forall i, j \in \mathit{Valve} : \mathit{valve}[i] \wedge \mathit{valve}[j] \Rightarrow \mathit{BelongTo}(i) = \mathit{BelongTo}(j)).\end{aligned}$$

Sequencing properties. Often, certain actions of the system should only be executed in a fixed order. The informal description in Section 3 presented the expected sequencing of the interactions with the system. For example, the fire case selections can only be executed after the pump is opened. The following formula expresses this property:

$$\begin{aligned}&\mathit{SelectAfterOpenPump} \triangleq\\ &\quad \Box[\forall i \in \mathit{FireCase} : (\mathit{SelectCase}(i) \Rightarrow \mathit{s\_buttonLight}[\text{“pump”}])]_{\mathit{vars}}.\end{aligned}$$

In other words, the fire case i can be selected only if the pump light is on, and therefore the pump has been opened. Note that in TLA$^+$, the property can be specified not only on the state variables but also on the actions.

Another sequencing property requests that the pump opening can only occur after the power is on. It is denoted by the following formula:

$$\begin{aligned}&\mathit{OpenPumpAfterPower} \triangleq\\ &\quad \Box[\mathit{OpenPump} \Rightarrow \mathit{s\_buttonLight}[\text{“power”}]]_{\mathit{vars}}.\end{aligned}$$

Priority properties. Certain system actions take priority over others. In our example, the button *power* has highest priority, which means that whatever the system is doing, once the user uplifts the self-locked power button, the power will be off and the system stops immediately. This property is described by the following formula:

$$\begin{aligned}&\mathit{ClosePowerAlwaysResponse} \triangleq\\ &\quad \Box(\neg \mathit{u\_button}[\text{“power”}] \wedge \mathit{s\_buttonLight}[\text{“power”}]\\ &\quad \Rightarrow \text{ENABLED } \mathit{PowerDown} \wedge \neg\text{ENABLED } (\mathit{SystemAction} \wedge \neg \mathit{PowerDown})).\end{aligned}$$

The formula asserts that if the user uplifts the power button while power is on then only the action *PowerDown* is enabled and all other system actions are disabled. As a result, the power will be turned off and the system will be stopped according to the action definition of *PowerDown*.

The *FCFSPriority* formula defines another priority property, which denotes the first-come-first-serve characteristic of cannon using.

$$\begin{aligned}&\mathit{FCFSPriority} \triangleq\\ &\quad \forall i, j \in \mathit{Cannon} : (\mathit{s\_cannonLight}[i] \wedge i \neq j \wedge \mathit{u\_button}[\mathit{CannonInCase}(j)] \Rightarrow \Box(\neg \mathit{s\_cannonLight}[j])).\end{aligned}$$

Error checking properties. Besides the properties which are related to the application domain, we can define some general properties to check errors in specifications writing. The mainly used one in the case of our practice is to check whether the actions defined in the TLA$^+$ specification can happen at some point. The basic idea is that each action defined in the specification has its reason to be defined. If a defined action can never be enabled, there were probably some errors when writing it. These kinds of properties are described by the formulas with the pattern:

$$\mathit{CheckAction} \triangleq \Box[\neg \mathit{Action}]_{\mathit{vars}}$$

where *Action* is a general name for the action to be checked. The formula *CheckAction* is an invariant declaring that *Action* is never executed. For example, the formula $\mathit{CheckOpenPump} \triangleq \Box[\neg \mathit{OpenPump}]_{\mathit{vars}}$ denotes that the *OpenPump* action never happens. As to this kind of property, we expect the TLC model checker returns the result FALSE, and provides a counter example showing a path that makes *Action* occur. If TLC returns the result TRUE, there are probably some errors in the specification of *Action* and a further examination is needed.

5.2. Model checking with TLC

We checked the *Control* specification by the TLA$^+$ model checker TLC (10 April 2008 Release), on a computer with an Intel® Core™ 2 duo, T8100 2.10 GHz CPU and 3 GB memory.

Checking the properties. Regarding the error checking properties which are expected to be FALSE, we checked 12 properties, each of which corresponds to a meaningful action in the specification. These error checking properties are defined in Table 1. For all the 12 properties, TLC succeeded in returning the result FALSE and reporting a path to reach the corresponding action as expected.

Table 1
The 12 error checking properties.

No.	Property definitions
1	$CheckPowerUp \triangleq \Box[\neg PowerUp]_{vars}$
2	$CheckOpenPump \triangleq \Box[\neg OpenPump]_{vars}$
3	$CheckPumpLightOn \triangleq \Box[\neg PumpLightOn]_{vars}$
4	$CheckSelectCases \triangleq \Box[\forall i \in FireCase : \neg SelectCase(i)]_{vars}$
5	$CheckConfirm \triangleq \Box[\neg Confirm]_{vars}$
6	$CheckResponseHandle \triangleq \Box[\neg ResponseHandle]_{vars}$
7	$CheckClosePump \triangleq \Box[\neg ClosePump]_{vars}$
8	$CheckPumpLightOff \triangleq \Box[\neg PumpLightOff]_{vars}$
9	$CheckPowerDown \triangleq \Box[\neg PowerDown]_{vars}$
10	$CheckNoReact \triangleq \Box[\neg NoReact]_{vars}$
11	$CheckUser \triangleq \Box[\neg UserAction]_{vars}$
12	$CheckPlant \triangleq \Box[\neg PlantAction]_{vars}$

Consider the other four kinds of properties, which are expected to return TRUE by TLC on the TLA$^+$ model. We checked the 9 properties defined in Section 5.1. All 9 properties were successfully verified by TLC. It generated 203,010 states, of which 99,275 were distinct, and verified all the 9 properties in 23.8 s.

Guiding the practice. A main task of a PLC program is to correctly represent the complicated logics among different actions. In this sense, designing PLC systems with the method introduced in this paper is quite helpful. The first reason is the clear representation of the enabling condition and the execution of an action in TLA$^+$, see the example *OpenPump* in Section 4.5. The second reason is the procedure to get a correct design with the help of TLC.

Take the *OpenPump* action as an example. With the informal description of the dock fire-fighting system, it is possible for an intuitive description of *OpenPump* to be:

$$\begin{aligned}
&OpenPump \triangleq\\
&\wedge\ s\_sysState = \text{“power”}\\
&\wedge\ u\_button[\text{“openpump”}]\\
&\wedge\ pumpCtl'\\
&\wedge\ s\_sysState' = \text{“openpumpSent”}\\
&\wedge\ \text{UNCHANGED}\langle s\_buttonLight, s\_cannonLight, s\_handle, valve\rangle.
\end{aligned}$$

That is to say, *OpenPump* is enabled when the system is at the “power” state, and the user presses the “openpump” button. When we try the property *CheckOpenPump* in TLC, TLC can successfully return the result FALSE and provide a path to make *OpenPump* occur. However, when inspecting the path, it shows that when the user presses both the “openpump” button and the “closepump” button, the action *OpenPump* also happens. This phenomenon pushes us to question on what we expect if more than one button is pressed down simultaneously? As one solution, we choose to accept only the single press of the button “openpump” and add a condition $\forall i \in UnlockedButton : i \neq \text{“openpump”} \Rightarrow \neg u\_button[i]$ for the *OpenPump* action. Besides, TLC also shows that the property *ClosePowerAlwaysResponse* doesn't hold. It contradicts with our design to make *PowerDown* the highest priority compared with other actions. As a result, we add another condition $u\_button[\text{“power”}]$ and get the final description of the *OpenPump* action. These subtle problems will be hard to check without the support of a clear and formal specification and then formal verification.

Further experiments. We also compare the *Control* specification with its two variations to justify our choices in designing the specification framework. The compared results are shown in Table 2. Towards the first column in Table 2, *Control* denotes the specification generated by the method presented in this paper. *CtlTestUser* is a variation which gives some constraints on user actions: users can operate the buttons or the cannon handle, but not both. This variation is obtained by taking the following formula for user actions:

$$\begin{aligned}
UserAction \triangleq\ &\vee\ \wedge\ u\_button' \in [Button \rightarrow \text{BOOLEAN}]\\
&\quad\ \wedge\ \text{UNCHANGED}\ u\_handle\\
&\vee\ \wedge\ u\_handle' \in Direction\\
&\quad\ \wedge\ \text{UNCHANGED}\ u\_button.
\end{aligned}$$

CtlTestEnv is the same as *Control* except that it discards the environment cleaning phrase when specifying the PLC running pattern. The space and time costs for checking the 9 properties on the three specifications are shown from the second column to the last column in the table. The 9 properties hold for all the three cases, while the costs are quite different. The case *CtlTestEnv* needs remarkably greater time and space costs, which certifies the effectiveness of introducing *ClearEnvInput* in our framework to simulate of PLC scan cycle. On the other hand, The case *CtlTestUser* needs less time and space costs than our design *Control*, but pays the price of a weaker conclusion. Since *CtlTestUser* puts constraints on user actions, we can't verify the system responses beside the constraints. For example, what will happen when the user is pressing down some button and operating the handle at the same time? On the other hand, *Control* doesn't put any constraint on user actions,

Table 2
Time and space costs comparisons.

Case	Nodes	Distinct nodes	Run time (s)
Control	203,010	99,275	23.8
CtlTestUser	42,260	20,659	7.2
CtlTestEnv	57,923,243	99,221	1226.9

and thus avoids the over-specification of the environment, but with the price of more time and space cost. It's up to the designer to decide whether or how many constraints to put on the environment.

5.3. *Processing larger problems*

The specification framework presented in this paper is generic and can be applied to large-scale systems. The parameterized TLA$^+$ specification for the fire fighting case is still adaptable when devices or physical configurations are changed, as explained in Section 4.1. However, to model check the TLA$^+$ specification for large-scale systems, getting a more abstract version is often necessary. There are several ways to achieve this.

First, the communication mode between PLC and its environment can use a synchronization pattern, instead of the asynchronous one adopted in this paper, which considers the PLC scan cycle mechanism. The abstracted model would have less state space for model checking. However, it would also keep farther to the later implementation.

Second, the environment model can be simplified. User actions or plant feedback are not random, but follow fixed sequences usually. For large-scale systems, it's often a good choice to consider the expected environment changes only, to get a smaller state space for model checking. However, as we explained in Section 5.2, it is a tradeoff.

Third, an abstracted model can be obtained by simplifying the information of variables or abstracting some of them away. For example, we have abstracted the direction of the cannon, which can be represented by a real number, to be an enumerable variable with five values: {"up", "down", "left", "right", "none"}. We can further abstract it as a Boolean variable denoting only moving or not. Further on, When the information of cannons is not much concerned, they can be even abstracted away, by omitting the declarations and deleting all their appearances from the defined actions.

6. Related work

We take the specification language TLA$^+$ to formally describe PLC systems. Undoubtedly, there are several other choices besides TLA$^+$. Among them, the languages based on state-transition system semantics are usually viewed suitable to describe the behaviors of PLC systems. With model checking techniques and tools like NuSMV [7], Spin [8] being widely used in engineering practice, their input languages are often taken as a specification language. In this section, we briefly introduce some of them and comparisons with TLA$^+$.

The input language of NuSMV is often used to model embedded systems. It is a language dedicated to model checking, so it permits only finite states when describing system behaviors, while TLA$^+$ can describe infinite states by parameterizing the number of buttons, devices and so on. The input language of NuSMV is more primitive, and suitable for low-level specifications. It provides array data structures, but the given operations are limited. Furthermore, TLA$^+$ is designed for formal reasoning, so it is possible to prove the parameterized TLA$^+$ specifications directly with theorem prover support. The input languages of other model checkers, like Promela for SPIN, have similar characteristics with the one for NuSMV.

There are some other powerful specification languages, for example the OBJ language Maude [9]. Maude is a high-level declarative language based on both equational logic and rewriting logic. It is expressive, supporting the characteristics of modularization and parameterization. Compared with TLA$^+$, it is more convenient for Maude to define complicated data structures. However, the semantics of the logics behind Maude is not trivial for engineers. On the other hand, TLA$^+$ is based on "ordinary logics", which are easier to learn and understand. Maude also provides object-oriented modules, making it convenient for many applications. However, as far as we know, object-oriented programming languages have not been widely used for PLC programming yet. The popular ones are assembly-like languages like ladder graph, Instruction language [2], etc. There is still a big gap between object-oriented specifications and the procedure-oriented implementations for PLC applications. Finally, similar to TLA$^+$, Maude also has an on-the-fly explicit state model checker, which can verify LTL formulas.

7. Conclusion

In this paper, we developed a format for the specification of PLC systems using the specification language TLA$^+$, and successfully verified a number of correctness properties using the TLC model checker. The specification format clearly distinguishes between user actions, system actions, and plant feedback. The different categories of actions are specified separately by TLA$^+$ action formulas, which are then composed to form the overall specification. This separation makes us confident that we avoided over-specification, in particular of the environment. Working in a high-level language such as TLA$^+$ allows a designer to focus on the essential features of a system specification, avoiding low-level encodings, and leading to configurable and concise specifications. The resulting specifications can nevertheless be analyzed by the TLC model checker in a reasonable amount of time. We identified different kinds of properties that arise naturally in PLC systems.

In future work, we will consider the real-time specification of PLC systems by TLA$^+$. Lamport [3,10] discusses formats for specifying real-time system specifications in TLA$^+$ and techniques for real-time model checking, but their applicability to PLC specifications remains to be validated.

References

[1] R.W. Lewis, Programming Industrial Control Systems Using IEC 1131-3, in: Control Engineering Series, vol. 50, The Institution of Electrical Engineers, Stevenage, United Kingdom, 1998.
[2] F. Bonfatti, P.D. Monari, U. Sampieri, IEC 1131-3 Programming Methodology, CJ International, Fontaine, France, 1999.
[3] Leslie Lamport, Specifying Systems, Addison-Wesley, 2002, See also: http://research.microsoft.com/users/lamport/tla/tla.html.
[4] Yuan Yu, Panagiotis Manolios, Leslie Lamport, Model checking TLA+ specifications, in: Correct Hardware Design and Verification Methods, CHARME'99, 1999, pp. 54–66.
[5] Stephan Merz, The specification language TLA+, in: Logics of Specification Languages, 2008, pp. 401–451.
[6] Leslie Lamport, The temporal logic of actions, ACM Transactions on Programming Languages and Systems 16 (3) (1994) 872–923.
[7] The NuSMV homepage: http://nusmv.irst.itc.it/.
[8] The Spin homepage: http://spinroot.com/spin/whatispin.html.
[9] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, J.F. Quesada, A maude tutorial, Technical report, 2000.
[10] Leslie Lamport, Real-time model checking is really simple, in: Dominique Borrione, Wolfgang J. Paul (Eds.), Correct Hardware Design and Verification Methods, CHARME 2005, in: Lecture Notes in Computer Science, vol. 3725, Springer, Saarbrücken, Germany, 2005, pp. 162–175.