Verification of PLC Properties Based on Formal Semantics in Coq

Jan Olaf Blech[1] and Sidi Ould Biha[2]

[1] fortiss GmbH, Munich
joblech@gmail.com
[2] INRIA and Tsinghua University, Beijing
Sidi.Ould_Biha@inria.fr

Abstract. Programmable Logic Controllers (PLC) are widely used in embedded systems for the industrial automation domain. We propose a formal semantics of two languages defined in the IEC 61131-3 standard for PLC programming. The first one is the *Instruction List* (IL) language, an assembly like language. The second one is the *Sequential Function Charts* (SFC) language, a graphical high-level language that allows to describe the main control-flow of the system. A PLC system description may comprise SFC and IL code. We formalized the semantics in the proof assistant Coq. Furthermore, we present an associated tool for automatically generating SFC representations from a graphical description – the text based IL code can be handled in Coq directly – and its usage for verification purposes. We demonstrate our approach to prove safety properties of a PLC in a real industrial demonstrator.

1 Introduction

Discovering and validating properties of Programmable Logic Controllers (PLC), is a prerequisite for the development of safety critical embedded systems. Tools and techniques for different kinds of systems and analysis scenarios have been developed. These comprise techniques aimed for distinct usage scenarios based on model checking and abstract interpretation.

In this work, we describe a general purpose way for the verification of PLC that are modeled using the Instruction List (IL) and Sequential Function Chart (SFC) languages of the IEC 61131–3 [15] standard. The standard is mainly used for modeling PLC functionality in the development of embedded systems for the industrial automation domain. We describe a tool set and method: For a given PLC description given in the graphical SFC language we automatically generate a Coq [9] description and some basic theorems and their proofs. In addition to the SFC language, text based IL programs are used in our PLC descriptions. We have formalized a syntactic representation of IL, thus, IL programs can be imported directly into our Coq environment. We present some standard techniques to reason about our PLC descriptions and verify properties. Furthermore, we present a case study of a PLC used inside a sorting machine.

G. Barthe, A. Pardo, and G. Schneider (Eds.): SEFM 2011, LNCS 7041, pp. 58–73, 2011.

The formalization of the IL and SFC semantics is done in the formal proof system Coq and its extension SSReflect [10]. Choosing Coq enables us to use its extraction mechanisms later and produce a certified compiler or interpreter for PLC based on our semantics. In this development, we also use some SSReflect libraries. In particular we use the libraries on booleans, natural numbers, lists and generic interface for types with decidable equality. The most important contributions of this paper comprise:

- Formal Coq semantics of the IL and SFC languages which are reusable for other projects.
- On overview on a tool to automatically generate SFC representations and some proofs.
- A case study on the verification of PLC properties using an IL and SFC description of a PLC.

Overview

This paper is organized as follows. We give an overview on PLC in Section 2. The IL and SFC language are presented in Section 3 and Section 4. A tool for generating Coq readable SFC representations and related proofs is described in Section 5. A case study is presented in Section 6. Section 7 discusses related work and a conclusion is featured in Section 8.

2 Programmable Logic Controller

A PLC is composed of a microprocessor, a memory, input and output devices where signals can be received from sensors or switches and sent to actuators. Figure 1 shows the architecture of a PLC system. A main characteristic of PLC is their execution mode. A PLC program is typically executed in a permanent loop. PLC program execution can be structured into *scan cycles* which are associated with a cycle time, the inputs are read, the program instructions are executed and the outputs are updated. The cycle time is often fixed or has an upper bound limit. Therefore the instructions which are scheduled to be executed in the cycle should terminate during the cycle time interval.

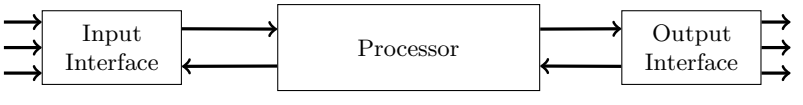

Fig. 1. PLC system

Since the introduction of PLC in the industry, each manufacturer has developed its own PLC programming languages. In 1993, the *International Electrotechnical Committee* (IEC) published the *IEC 1131 International Standard* for PLC. The third volume of this standard defines the programming languages for PLC. It defines 5 languages :

- *Ladder Diagrams* (LD) : graphical language that represent PLC programs as relay logic diagrams.
- *Functional Block Diagrams* (FBD) : graphical language that represent PLC programs as connection of different function blocks.
- *Instruction List* (IL) : an assembly like language.
- *Structured Text* (ST) : a textual (PASCAL like) programming language.
- *Sequential Function Charts* (SFC): a graphical language for describing top-level control-flow and associated data-flow in the PLC.

The last language differs from the other. It corresponds to a graphical method for structuring programs and allows to describe the system as a parallel state transition diagram. Each state is associated to some actions. An action is described using one of the other 4 PLC programming languages like LD or IL. SFC are well suited for writing concurrent control programs. In this paper we concentrate on the IL and SFC languages.

3 Instruction Lists

Instruction list (IL) is one of the five programming languages defined in the IEC 61131-3 standard. It is an assembly like language widely used for programming PLC systems. Our IL model is a significant subset of the language defined by the IEC 61131-3 standard. This subset covers assignments instructions and boolean and integer operations. It covers also comparison and branching instructions and *on-delay timers*. We choose to consider only booleans and integers as basic data types. In most of PLC systems, reals are available as basic data types, but rarely used. In practice, real number computation costs much time and may be delegated to an external device that can communicate with the PLC. This is motivated by the need to keep the program scan cycle within a relatively small time upper bound. The IL model we present in the following is an extension of the model defined in a previous work [14].

3.1 Syntax

An IL program comprises declarations of variables followed by a list of instructions. An IL program example is the following:

LABEL	OPERATOR	OPERAND
l1:	LD	x
	AND	y
	LD	z
	ORs	
	JMPC	l1

In the first line of the example above, the value of the variable `x` is loaded on a stack. After the execution of the second line, the stack contains the conjunction of `x` and `y`. In the third line, the value of `z` is put on the top of the stack. The

instruction `ORs` of the example above removes the two previous values loaded on the stack and replace them with $(\mathtt{x} \wedge \mathtt{y}) \vee \mathtt{z}$. The branching instruction `JMPC` is executed if the value at the top of the stack is equal to *true*.

An IL instruction starts with an operator that can be followed by one or more operands: variables or constants. In an instruction, the operator can be preceded by a label.
Instructions:

$i ::=$ `LD` op | `LDN` op	load
| `ST` id | `STN` id | `SR` id | `RS` id	store, set and reset
| `JMP` lb | `JMPC` lb | `JMPC` lb	jump
| `AND` op | `OR` op | `XOR` op	boolean operations
| `ANDN` op | `ORN` op | `XORN` op	
| `ANDs` | `ORs` | `XORs`	
| `ADD` op | `MUL` op | `SUB` op	integer operations
| `ADDs` | `MULs` | `SUBs`	
| `GT` op | `GE` op | `EQ` op	comparison
| `GTs` | `GEs` | `EQs`	
| `TON` id , n	On delay timer
| `RET`	end of program

Operands:

$op ::= id \mid cst$ variable identifier or constant

Constants:

$cst ::= n \in \mathbb{Z} \mid b \in \mathbb{B}$ integer or boolean literal

The data domains of IL constants is the union of integers $\mathbb{Z}$ and booleans $\mathbb{B}$. In practice integers used in PLC are bounded. For simplicity, we restrict ourselves to unbounded integers in the presentation of this work. Adjusting the integer size in Coq – and other higher-order theorem prover based – developments is not a difficult task and has been studied before (e.g., [12]).

We denote the set of IL instructions by *Instr*. For simplicity, we suppose that IL program labels are natural numbers. Since an IL program is a list of instructions, a label indicates the position of the corresponding instruction in the list. For a given program p and an index i, $p(i) \in Instr$ represents the instruction of p at the position i.

3.2 Semantics

We defined a small step operational semantics of IL programs. For the purpose of modeling PLC timers, we suppose having a global discrete time clock and that each program execution cycle has a fixed time duration denoted δ.

Stack: in IL an evaluation *stack* is used for the current result computation. It is also used to store intermediate results that will be pulled back when an instruction like `ADDs` or `ANDs` are executed. A *stack* $V := v_1, \ldots, v_m$ is a finite

$$\text{LD}\ \frac{p(i) = \textbf{LD}\ op}{(s,\sigma,i) \rightarrow (push\ op\ s, \sigma, i+1)} \qquad \frac{p(i) = \textbf{LDN}\ op}{(s,\sigma,i) \rightarrow (push\ \neg op\ s, \sigma, i+1)}\ \text{LDN}$$

$$\text{ST}\ \frac{p(i) = \textbf{ST}\ x \qquad \sigma' = \sigma[x \mapsto top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)} \qquad \frac{p(i) = \textbf{STN}\ x \qquad \sigma' = \sigma[x \mapsto \neg top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)}\ \text{STN}$$

$$\text{SR}\ \frac{p(i) = \textbf{SR}\ x \qquad \sigma' = \sigma[x \mapsto x \vee top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)} \qquad \frac{p(i) = \textbf{RS}\ x \qquad \sigma' = \sigma[x \mapsto x \wedge \neg top\ s]}{(s,\sigma,i) \rightarrow (s,\sigma',i+1)}\ \text{RS}$$

$$\text{JMPC}_\text{T}\ \frac{p(i) = \textbf{JMPC}\ l \qquad top\ s = \textbf{T}}{(s,\sigma,i) \rightarrow (s,\sigma,l)} \qquad \frac{p(i) = \textbf{JMPC}\ l \qquad top\ s = \textbf{F}}{(s,\sigma,i) \rightarrow (s,\sigma,i+1)}\ \text{JMPC}_\text{F}$$

$$\text{JMPCN}_\text{F}\ \frac{p(i) = \textbf{JMPCN}\ l \qquad top\ s = \textbf{F}}{(s,\sigma,i) \rightarrow (s,\sigma,l)} \qquad \frac{p(i) = \textbf{JMPCN}\ l \qquad top\ s = \textbf{T}}{(s,\sigma,i) \rightarrow (s,\sigma,i+1)}\ \text{JMPCN}_\text{T}$$

$$\text{ANDs}\ \frac{p(i) = \textbf{ANDs} \qquad (t,t') = top2\ s}{(s,\sigma,i) \rightarrow (push\ (t \wedge t')\ (pop2\ s), \sigma, i+1)} \qquad \frac{p(i) = \textbf{AND}\ op \qquad t = top\ s \wedge op}{(s,\sigma,i) \rightarrow (push\ t\ (pop\ s), \sigma, i+1)}\ \text{AND}$$

$$\text{ADDs}\ \frac{p(i) = \textbf{ADDs} \qquad (t,t') = top2\ s}{(s,\sigma,i) \rightarrow (push\ (t + t')\ (pop2\ s), \sigma, i+1)} \qquad \frac{p(i) = \textbf{ADD}\ op \qquad t = top\ s + op}{(s,\sigma,i) \rightarrow (push\ t\ (pop\ s), \sigma, i+1)}\ \text{ADD}$$

$$\text{GTs}\ \frac{p(i) = \textbf{GTs} \qquad (t,t') = top2\ s}{(s,\sigma,i) \rightarrow (push\ (t < t')\ (pop2\ s), \sigma, i+1)} \qquad \frac{p(i) = \textbf{GT}\ op \qquad t = top\ s < op}{(s,\sigma,i) \rightarrow (push\ t\ (pop\ s), \sigma, i+1)}\ \text{GT}$$

$$\text{TON-OFF}\ \frac{p(i) = \textbf{TON}\ Tx, Pt \qquad top\ s = \textbf{F} \qquad \sigma' = \sigma[Tx.Q \mapsto \textbf{F}, Tx.ET \mapsto 0]}{p \vdash (s,\sigma,i) \rightarrow (s,\sigma',i+1)}$$

$$\text{TON-ON}\ \frac{p(i) = \textbf{TON}\ Tx, Pt \qquad top\ s = \textbf{T} \qquad Tx.ET < Pt \qquad \sigma' = \sigma[Tx.Q \mapsto \textbf{F}, Tx.ET \mapsto Tx.ET + \delta]}{p \vdash (s,\sigma,i) \rightarrow (s,\sigma',i+1)}$$

$$\text{TON-END}\ \frac{p(i) = \textbf{TON}\ Tx, Pt \qquad top\ s = \textbf{T} \qquad Tx.ET >= Pt \qquad \sigma' = \sigma[Tx.Q \mapsto \textbf{T}, Tx.ET \mapsto Tx.ET + \delta]}{p \vdash (s,\sigma,i) \rightarrow (s,\sigma',i+1)}$$

Fig. 2. IL operational semantics

sequence of data values. In the following we use the standard stack operations *push* (add an element to the stack), *pop* and *pop2* (remove respectively the top and the two top elements of the stack), *top* and *top2* (return respectively the top and the two top elements of the stack).

States: functions from variable identifiers to data values. They represent the program variable states and are denoted σ of the type $\mathcal{S} = Var \rightarrow D$, where D is the union of the IL variables data domains: $\mathbb{Z} \cup \mathbb{B}$.

Configurations: elements of the set $\mathcal{E} = Stack \times \mathcal{S} \times \mathbb{N}$. A configuration (s, σ, i) corresponds to a stack s, a state σ and a position or program location i.

Transitions: relation on configurations $\subseteq \mathcal{E} \times \mathcal{E}$. Figure 2 gives some relevant inference rules of the IL configurations transitions relation. We denote the relation defined by the inference rules of Figure 2 by $\xrightarrow{x}$ where x is an IL instruction. The IL transition system is defined by an initial configuration $(s_0, \sigma_0, 0)$, where s_0 is the empty stack and σ_0 is the initial state that maps all the integer variables to 0 and boolean variables to $false$.

$$\textsc{Left}\ \frac{i < \mathit{length}\ p \qquad (s,\sigma,i) \overset{p}{\Longrightarrow} (s'',\sigma'',i'') \qquad (s'',\sigma'',i'') \overset{[p,x]}{\Longrightarrow} (s',\sigma',i')}{(s,\sigma,i) \overset{[p,x]}{\Longrightarrow} (s',\sigma',i')}$$

$$\textsc{Right}\ \frac{i = \mathit{length}\ p \qquad (s,\sigma,i) \overset{x}{\longrightarrow} (s'',\sigma'',i'') \qquad (s'',\sigma'',i'') \overset{[p,x]}{\Longrightarrow} (s',\sigma',i')}{(s,\sigma,i) \overset{[p,x]}{\Longrightarrow} (s',\sigma',i')}$$

$$\textsc{Out}\ \frac{\mathit{length}\ p \leq i}{(s,\sigma,i) \overset{p}{\Longrightarrow} (s,\sigma,i)}$$

Fig. 3. IL natural semantics

The first four transition rules of Figure 2 correspond to the *load* and *store* instructions. In the first case the stack is updated while in the second the variable state is updated. The transitions corresponding to the *set/reset* instructions (rules SR and RS) update the variable state function with the corresponding values for the given operands and the top of the stack. The transition relation for the TON instruction is given by the rules TON-OFF, TON-ON and TON-END of Figure 2. The elapsed time variable ET of the TON timer is incremented by the global constant δ when the timer is activated (the value of top of the stack is *true*). The timer output Q is activated when the elapsed time variable ET is greater or equal to the timer delay parameter PT.

Natural semantics: sometimes in the reasoning about IL programs we need to interpret the execution of the entire program. This can be done using natural semantics or big-step semantics. On top of the small-step semantics presented above, we defined also a big-step semantics of IL programs. The inference rules of this semantics are given in Figure 3. They correspond to the definition of the transitive closure of the small-step semantics relation. As we mentioned before an IL program should terminate during the cycle scan time. This termination property is assured by the rule OUT. A final state is one where the location index is greater than the program instruction list length. By using the rule OUT we can prove that every execution, following the rules of our natural semantics, will reach a final (or stable) configuration.

3.3 Formalization

We formalized the IL semantics defined above in the formal proof system Coq. The Coq system provides a powerful mechanism to define recursive or finite types or sets: *inductive types*. It is especially useful when defining the syntax of a programming language. We define the IL syntax and operational semantics presented above, using the Coq inductive type mechanism. In our formalization, IL instructions are represented by the type `Instr` and an IL program or a list of IL instructions is an object of the type `code := seq Instr`[1].

We also formalized the IL big-step semantics defined in Figure 3 as a Coq inductive relation. The definition is given in the Figure 4. Since it is not always

[1] `seq` is the type of list in SSReflect.

```
Inductive il_exec : code -> ILConf -> ILConf -> Prop :=
| il_exec_consl : forall p x cf cf1 cf2, cf.2 < size p ->
  il_exec p cf cf2 -> il_exec (rcons p x) cf2 cf1 ->
  il_exec (rcons p x) cf cf1
| il_exec_consr : forall p x cf cf1 cf2, cf.2 = size p ->
  il_trans (rcons p x) cf cf2 -> il_exec (rcons p x) cf2 cf1 ->
  il_exec (rcons p x) cf cf1
| il_exec_out : forall p cf, size p <= cf.2 -> il_exec p cf cf.
```

Fig. 4. Coq definition of the IL program execution predicate

possible to know how many transitions are needed to execute an IL program, we define the program execution as a propositional relation rather than a computational function. However it is possible to define a function that returns the configuration corresponding to the result of the execution an IL code after a given number of steps. This function corresponds to the definition given in Figure 5. We also proved that the relational definition and functional one are equivalent. This is given by the lemmas `il_exec_seq_exec` and `il_exec_exec_seq` of Figure 5.

```
Fixpoint il_exec_seq n p cf : ILConf :=
  if n is n'.+1 then il_exec_seq n' p (il_transf p cf) else cf.
Lemma il_exec_seq_exec : forall n p cf cf', cf' = il_exec_seq n p cf ->
  size p <= cf'.2 -> il_exec p cf cf'.
Lemma il_exec_exec_seq : forall p cf cf', il_exec p cf cf' ->
  exists n, il_exec_seq n p cf = cf'.
```

Fig. 5. Coq definition of the IL program execution function and equivalence proofs

4 Sequential Function Charts

The SFC language is a graphical language for modeling PLC. It is part of the IEC 61131–3 standard and frequently used together with IL and other languages of this standard. SFC are used to describe the overall control flow structure of a system. Due to the graphical nature of the language, we have written a tool which generates Coq representations from graphical SFC models.

The parts of the standard describing SFC leave a few semantical aspects open to the implementation of the PLC modeling and code generation tool. In cases where the semantics is not well defined by the standard we have adapted our semantics to be compatible with the EasyLab [1] tool. EasyLab is a tool that allows the graphical modeling of PLC and C code generation. The description given in this work follows the description given in [4].

4.1 Syntax

Syntactically we represent an SFC as a tuple $(S, S_0, T, A, F, V, Val_V)$. It comprises a set of steps S and a set of transitions T between them. A step is a system location which may either be active or inactive in an actual system state, it can be associated with SFC action blocks from a set A. These perform sets of operations and can be regarded as functors that update functions representing memory. Memory is represented by a function from a set of variables V to a set of their possible values Val_V. The mapping of steps to sets of action blocks is done by the function F.

In our SFC framework, action blocks are described using the IL semantics defined in the previous section. We have established functions that allow conversion of SFC states into IL state and vice versa. Thus, the execution of an action block comprises the following steps:

- Conversion of the SFC state into an IL state
- Execution of the IL program associated with the action block using the semantics from Section 3.
- Update of the SFC state by using the final IL state.

A transition is a tuple (S_{in}, g, S_{out}). It features a set of steps that have to be enabled $S_{in} \subseteq S$ in order to take the transition. It features a guard g that has to be evaluated to true for the given system state. The guard g is a function from system memory to a truth value – in Coq we formalize this as a function to the *Prop* datatype. A transition may have multiple successor steps $S_{out} \subseteq S$. The types Val_V that are formalized in our SFC language comprise different integer types and boolean values. The set of SFC steps includes also a set $S_0 \subseteq S$ representing the initially active steps.

Figure 6 shows an example of an SFC structure realizing a loop with a conditional branch. The execution starts with an initialization step *init*. After it has been processed control may pass to either *Step2* or to a step *Return*. Once *Step2* has been processed control is passed to *init* again.

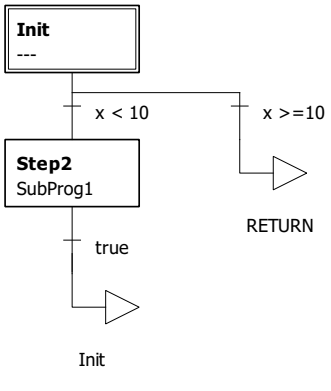

Fig. 6. A loop in the SFC language

Please note that in addition to loops and branches, SFC allows also the definition of parallel processing and synchronization of control. This is due to the multiple successor and predecessor steps in a transition.

The Coq realization of the SFC syntax follows the presented description. For compatibility with the EasyLab tool and to ease generation we distinguish between steps and step identifiers in our Coq files, thereby introducing some level of indirection.

4.2 Semantics

Semantically the execution of an SFC encounters states, which are (m, s, a) tuples. They are characterized by a memory state m, the function from variables to their values, a set of active steps s and a set of active action blocks a that need to be processed.

The semantics is defined by a state transition system which comprises two kinds of rules:

1. A rule for processing an action block from the set of active action blocks a. This corresponds to updating the memory state and removing the processed action block from a.
2. A rule for performing a state transition. The effect on the system state is that some steps are deactivated, others are activated. Each transition needs a guard that can be evaluated to true and a set of active steps. Furthermore, we require that all pending action blocks of a step that is to be deactivated have been executed.

It is custom to specify the timing behavior of a step by time slices: a (maximal) execution time associated with it. In our work, this is realized using special variables that represent time.

5 Tool Support for PLC Verification and Proving Principles

For the generation of graphical SFC representations and reasoning about them we have created a tool (CertPLC, described in a report [3]). It is implemented in Java and uses SFC files built with a graphical PLC configuration environment: EasyLab [1]. The text-based IL code can be imported directly into the generated file. In this section we describe our tool's architecture, usage scenarios and frequently used principles for proving properties.

5.1 The CertPLC Tool

Figure 7 shows the CertPLC ingredients and their interconnections. In an invocation of the tool framework an SFC model is given to a **representation generator** which generates a Coq representation out of it. This is included in one or several files containing the model specific parts of the semantics of the

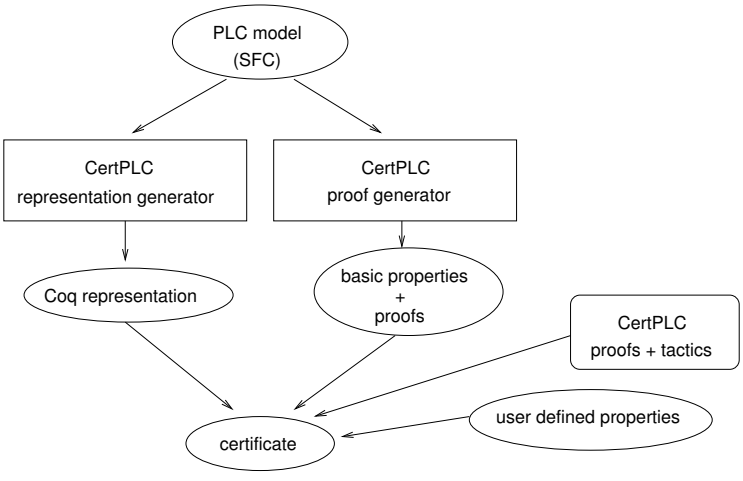

Fig. 7. CertPLC overview

SFC model. The Coq representation is human readable and can be validated against the original graphical SFC specification by experienced users. No representation generation is required for IL, since IL is already a textual format which can be used directly within the Coq proof assistant.

The same SFC model is given to a **proof generator** which generates Coq proof scripts that contain lemmas and their proofs for some basic properties that state important facts needed for machine handling of the proofs of more advanced properties. For example a proof script is generated for a fact that the set of active action blocks in all reachable states of the PLC system does contain only action blocks specified by the syntactic PLC descriptions. The PLC shows only behavior achieved by combining these action blocks.

One goal of CertPLC is the generation of Coq files – a certificate – that certifies a property of a PLC. For this, one needs to formalize the desired property. The property is proved in Coq by using a provided tactic or a hand written proof script. We provide a collection of some **proofs and tactics**. This is a kind of library to be used in our proofs. The Coq system description, used lemmas and their proofs, and the property and its proof form a certificate.

5.2 Proof Structure for Inductive Properties

As stated above, some inductive properties are already generated together with the Coq representation generation. Others can also be proven by using the following scheme: We start with an inductive invariant property I and an SFC description of a PLC SFC. Following the ideas presented in [5] the structure of a proof contained in our certificates is realized by generated proof scripts, generic lemmas and tactics. They establish a proof principle that proves the following goal:

$$\forall\ s\ .\ s \in \mathit{Reachable}_{SFC} \Longrightarrow I(s)$$

$\mathit{Reachable}_{SFC}$ is the inductively defined set of reachable states, $[\![SFC]\!]$ specifies the state transition relation (cf. Section 5). First we perform an induction using the induction rule of the set of reachable states. This rule is automatically established by Coq when defining inductive sets. After the application the following subgoals are left open:

1. $I(s_0)$ for initial states s_0,
2. $I(s) \wedge (s, s') \in [\![SFC]\!] \Longrightarrow I(s')$

The first goal can be solved by some relatively simple tactic which just checks that all conditions derived from I are fulfilled in the initial states.

For the second goal the certificate realizes a proof script which – in order to allow efficient certificate checking – performs most importantly the following operations:

- Splitting of conjunctions in invariants into independently verifiable invariants.
- Splitting of disjunctions in invariants into two independently verifiable subgoals.
- Normalizing arithmetic expressions and expressions that make distinctions on active steps in the SFC.
- Exhaustive case distinctions on possible transitions. Each case distinction corresponds to one transition in the control flow graph of the SFC. A typical case can have the following form:

  Precondition on states associated with a case distinction
  Transition condition associated with a case distinction
  Conditions on possible reachable states after one transition
  $\Longrightarrow$
  Property holds for succeeding states

  The elements in such a goal can feature arithmetic constraints, which can be split into further cases.

Some of the cases that occur can have contradictions in the hypothesis. For example one can imagine an arithmetic constraint for a variable from a precondition of a state contradicting with a condition on a transition. These contradictions result from the fine granularity of our case distinctions. Some effort can be spent to eliminate contradicting cases as soon as possible (cf. [5]) which can speed up the checking process.

6 Case Study

Figure 8 shows an overview of the SFC structure of a PLC program that controls a sorting station on the left side and a picture of the sorting station itself on

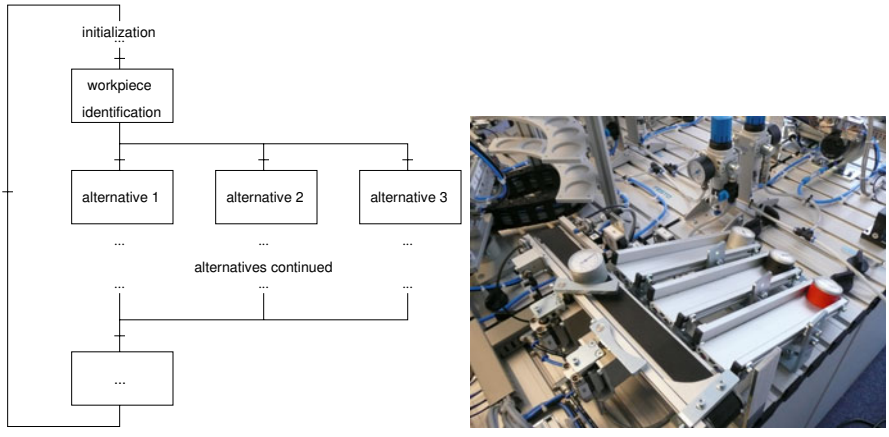

Fig. 8. Sorting machine overview

the right side. Work pieces are transported to two sensors. Based on the values observed by these sensors, a work piece is handled in a different way. The sensor observation is done in the step *workpiece identification*. The handling is done by choosing one of the three alternatives. We have modeled this system in EasyLab and generated the Coq representation of the SFC structure for this case study using CertPLC. We have imported the IL programs describing the actions which are taken at the different SFC steps.

Based on this, we have verified that consistency conditions hold. These comprise:

- The verification of inductive invariant based properties. This is described in Sections 5.1 and 5.2.
- The verification of non-inductive properties. During the conduction of the case study it turned out that non-inductive properties like: Identification of a certain work piece implies treatment in a work piece specific way and this occurs within a fixed amount of execution steps, are also of relevance. Mutual exclusion properties of work piece treatment can be proved by doing these work piece specific proofs for all kinds of possible work pieces, first, and using these results for proving the mutual exclusion property.

Proofs for are done in a modular fashion: we verify the effect of IL parts in the PLC execution and use these proofs to derive facts on the execution of several SFC steps.

Figure 9 shows an example of Coq code + pseudo code to give a look and feel on the nature of our proof goals. Given a concrete workpiece and conditions on a state x which corresponds to a state just before the workpiece identification. A succeeding state x' in the SFC language has to fulfill requirements on variable values m' the set of currently active steps S' and currently active actions A' after a certain execution time. In our example it involves several single SFC

```
conditions on workpiece
->
let '(m,S,A):= x in
  (conditions on m /\ S = SWorkPieceId::nil /\ A = AWorkPieceId::nil
      )
->
...
transition conditions between x x'
...
->
let '(m',S',A'):= x' in
  (some conditions on m' /\ S' = S13::nil /\ A' = nil))
```

Fig. 9. Constraint in Coq

state transitions (state transition rule applications, cf. Section 4) to get from x to x'. This is given in the transition conditions between x and x'. The set of currently active steps in the resulting state x' comprises one step *S13* which corresponds to a step in the first alternative for handling our workpiece.

Evaluation Aspects: Coq representation generation for SFC programs and the import of IL code is feasible for IEC 61131–3 based PLC descriptions that are solely described with these languages. Extending the semantics definition for additional commands which may appear in some PLC descriptions is relatively easy, due to the modularity of our semantics framework.

The inductive proof techniques used in the properties generated by the CertPLC tool and the non-inductive proof techniques used manually in the case study have been successfully applied in previous work which did not deal with PLC (e.g., our own work [5]). Here we have demonstrated their applicability for a realistic PLC. Using our Coq semantics and CertPLC, basic properties of a PLC can be verified by experienced Coq users within several hours. This may result in up to a few hundred lines of proof code for an example as in Figure 9. Common tactic applications are encapsulated into user defined tactics and libraries to further speed this process up, make the scripts smaller, and especially make the approach usable for people who have some knowledge in formal methods but are not Coq experts.

7 Related Work

Formal treatment of PLC and the IEC 61131–3 standard has been discussed by a larger number of authors before. Formalization work on the semantics of the Sequential Function Charts is given in [6,7]. This work was a starting point for our formalization of SFC semantics.

Work on the formal treatment of the FBD language – which is also a part of IEC 61131–3 – can be found in [20,19]. The FBD programs are checked using a model-checking approach.

The approach presented in [16] regards a translation from the IL language to an intermediate representation (SystemC). A SAT instance is generated out of this representation. The correctness of an implementation is guaranteed by equivalence checking with the specification model.

There are plenty of examples of the use of *model checking* for the verification of PLC programs. The paper [2] considers the SFC language. Untimed SFC models are transformed in to the input language of the Cadence SMV tool. Timed SFC models are transformed into timed automata. These can be analyzed by the Uppaal tool. In [13] a semantics of IL is defined using timed automata. The language sub-set contains TON timers but data types are limited to booleans. The formal analysis is performed by the model checker Uppaal.

In [8] an operational semantics of IL is defined. A significant sub-set of IL is supported by this semantics, but it does not include timer instructions. The semantics is encoded in the input language of the model checker Cadence SMV and linear temporal logic (LTL) is used to specify properties of PLC programs.

In contrast to the model checking work, we are using a higher-order theorem prover for our work. In general higher-order theorem provers require a higher level of interaction (we are aiming at overcoming this drawback by generating proof scripts and providing automatic tactics). On the plus side they allow in general richer specifications, abstractions and proofs. In the theorem proving community, there has been some work on the formal analysis of PLC programs. In [17] the theorem prover HOL is used to verify PLC programs written in FBD, SFC and ST languages. In this work, modular verification is used for compositional correctness and safety proofs of programs. For the Coq system, an example of verification of a PLC program with timers is presented in [18]. A quiz machine program is used as an example in this work, but no generic model of PLC programs is formalized. There is also a formalization of a semantics[2] of the LD languages in Coq. This semantics support a sub-set of LD that contains branching instructions. This work is a component of a development environment for PLC.

8 Conclusions and Future Works

Programmable Logic Controller applications can be critical in a safety or economical cost sense. Therefore formal verification of PLC programs does increase the confidence in such applications. In this paper we presented a formal framework for the verification of PLC programs written in the languages IL and SFC. We defined a formal semantics of these two languages in the formal proof system Coq. These semantics are used by the CertPLC tool that automatically generates an SFC formal representation from a graphical representation. Using our

[2] Research report in Korean available at: `http://pllab.kut.ac.kr/tr/2009/ldsemantics.pdf`

formal semantics, we proved safety properties for a PLC based real industrial demonstrator.

Future Work

The study of other languages from the IEC 61131–3 standard is an interesting subject for future work. Furthermore, we are interested in extending the tool support for verification of properties based on these semantics.

Another perspective of this work is the development of a certified compiler front-end for PLC. This is an ongoing work and we plan to formalize and certify a transformation of PLC programs written in the graphical language LD to IL. This will open the way to the development of a certified compilation chain for PLC. This chain can be build on top of the CompCert C certified compiler [12]. An integration of our formal semantics of PLC and the certified compiler to the EasyLab framework is also an interesting perspective. This can lead to a complete environment for the development of certified PLC programs.

References

1. Barner, S., Geisinger, M., Buckl, C., Knoll, A.: EasyLab: Model-based development of software for mechatronic systems. In: Mechatronic and Embedded Systems and Applications, IEEE/ASME (October 2008)
2. Bauer, N., Engell, S., Huuck, R., Lohmann, S., Lukoschus, B., Remelhe, M., Stursberg, O.: Verification of PLC programs given as sequential function charts. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 517–540. Springer, Heidelberg (2004)
3. Blech, J.O.: A Tool for the Certification of PLCs based on a Coq Semantics for Sequential Function Charts (2011), `http://arxiv.org/abs/1102.3529`
4. Blech, J.O., Hattendorf, A., Huang, J.: An Invariant Preserving Transformation for PLC Models. In: IEEE International Workshop on Model-Based Engineering for Real-Time Embedded Systems Design (2011)
5. Blech, J.O., Périn, M.: Generating Invariant-based Certificates for Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS) (accepted)
6. Bornot, S., Huuck, R., Lakhnech, Y., Lukoschus, B.: An Abstract Model for Sequential Function Charts. In: Discrete Event Systems: Analysis and Control, Workshop on Discrete Event Systems (2000)
7. Bornot, S., Huuck, R., Lakhnech, Y., Lukoschus, B.: Verification of Sequential Function Charts using SMV. In: Parallel and Distributed Processing Techniques and Applications (PDPTA 2000). CSREA Press (June 2000)
8. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnoebelen, P.: Towards the automatic verification of PLC programs written in Instruction List. In: IEEE International Conference on Systems, Man, and Cybernetics (2000)
9. The Coq Development Team. The Coq System, `http://coq.inria.fr`
10. Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. INRIA Technical report, `http://hal.inria.fr/inria-00258384`
11. Huuck, R.: Semantics and Analysis of Instruction List Programs. Electr. Notes Theor. Comput. Sci. (2005)

12. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
13. Mader, A., Wupper, H.: Timed Automaton Models for Simple Programmable Logic Controllers. In: Euromicro Conference on Real-Time Systems (1999)
14. Ould Biha, S.: A formal semantics of PLC programs in Coq. In: 35th IEEE Computer Software and Applications Conference, COMPSAC 2011, Munich (2011)
15. Programmable controllers - Part 3: Programming languages, IEC 61131-3: 1993, International Electrotechnical Commission (1993)
16. Sülflow, A., Drechsler, R.: Verification of plc programs using formal proof techniques. In: Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008), Budapest, pp. 43–50 (2008)
17. Volker, N., Kramer, B.J.: Automated verification of function block-based industrial control systems. Science of Computer Programming 42, 101–113 (2002)
18. Wan, H., Chen, G., Song, X., Gu, M.: Formalization and Verification of PLC Timers in Coq. In: 33rd IEEE Computer Software and Applications Conference, COMPSAC (2009)
19. Yoo, J., Cha, S., Jee, E.: A verification framework for fbd based software in nuclear power plants. In: 15th Asia Pacific Software Engineering Conference (APSEC), Beijing, China, December 3-5 (2008)
20. Yoo, J., Cha, S., Jee, E.: Verification of plc programs written in fbd with vis. Nuclear Engineering and Technology 41(1), 79–90 (2009)