A Tool for the Certification of Sequential Function Chart based System Specifications

Jan Olaf Blech

fortiss GmbH

Abstract. We describe a tool framework for certifying properties of sequential function chart (SFC) based system specifications: CERTPLC. CERTPLC handles programmable logic controller (PLC) descriptions provided in the SFC language of the IEC 61131–3 standard. It provides routines to certify properties of systems by delivering an independently checkable formal system description and proof (called certificate) for the desired properties. We focus on properties that can be described as inductive invariants. System descriptions and certificates are generated and handled using the COQ proof assistant. Our tool framework is used to provide supporting evidence for the safety of embedded systems in the industrial automation domain to third-party authorities. In this paper we focus on the tool's architecture, requirements and implementation aspects.

1 Introduction

Discovering and validating properties of safety critical embedded systems has been a research topic during the last decades. *Automatic verification tools* based on model checking and static analysis techniques are used in various software and hardware development projects. Automatic verification tools are successfully applied to increase confidence in the system design. However, even the verdicts about systems provided by automatic verification tools may be erroneous, since automatic verification tools are likely to contain errors themselves: they use sophisticated algorithms, resulting in complicated implementations. Due to this high level of complexity of their algorithms and the underlying theory, they are hardly ever considered as trustable by certification authorities.

In contrast to general purpose *higher-order theorem provers*, an automatic verification tool possesses a high degree of automation, but it does not achieve the same level of trustability and is usually specialized towards a problem-specific domain. Higher-order theorem provers, like COQ [13], are based on a few deduction rules and come with very small, simple, and trusted proof checkers which are based on type checking algorithms and provide a high level of confidence.

For this reason we provide a verification / certification environment based on higher-order theorem provers. It may be used to re-check properties that have been discovered by automatic verification tools or stated by humans in the first place. If such a check is run successfully in the higher-order theorem prover one lifts these properties to the high level of confidence provided by the higher-order theorem prover.

Based on our ideas on certification of properties for a modeling language [8] and our work on a certificate generating compiler [5] we present a tool framework CERT-PLC which emits certificates and allows reasoning about properties of models for programmable logic controller (PLC) provided in the sequential function chart (SFC) language of the IEC 61131–3 standard [17]. Our work comprises a generation mechanism for COQ representations of our models – a kind of compiler that emits COQ readable files for given models. In addition to this, it comprises other related proof generation mechanisms and a framework for supporting proofs that properties of these models do hold. Our COQ certificates – system description, properties and their proofs – are based on an explicit semantics definition of the SFC language, thereby ensuring that correctness conditions hold for the system described in the certificate.

The COQ environment has been accepted by French governmental authorities in a certification to the highest level of assurance of the Common Criteria for Security [12].

In this paper we focus on the following aspects of the CERTPLC tool framework:
– tool architecture,
– proof generation and the construction of certificates,
– and additional implementation issues.

Furthermore, we give an overview on usage scenarios, the formalized SFC semantics and present and discuss general characteristics of the methodology. A long version of this paper is available as a report [4]. In the current state of implementation the tool framework is applicable for standard PLC described using SFC. An exemplary usage with another language: function block diagrams (FBD) is also described to illustrate the flexibility of the described framework. The support of other languages and a detailed investigation of case studies are not subjects of this paper.

Our certification framework is mostly characterized by:
– The usage of an explicit semantics for properties and systems. This is human readable, an important feature to convince certification authorities.
– The focus on the PLC domain and the integration in an existing tool.
– A high degree of automation – compared to other work using higher-order theorem provers, that still allows human interaction.
– The integration into an existing tool for graphically designing PLC: EasyLab [2].

The high expressiveness of our semantics framework is largely facilitated by the usage of a higher-order theorem prover.

1.1 Certification

In the context of this paper we define

- *certification* as the process of establishing a certificate.
- *automatic certification* is the process of establishing a certificate automatically.
- In our work *certificates* comprise a formal description of a system, a formal description of a desired property and a proof description (a proof script or a proof term) that this property does hold.
- *certificate checking* is the process of checking that the property does indeed hold for the formal system description in the certificate. This checking is done by using the proof description in the certificate.

1.2 The Trusted Computing Base in Certification

Apart from components like operating system and hardware, in our certification approach, the trusted computing base (TCB) comprises the certificate checker (the core of the COQ theorem prover) and the program that generates formal PLC descriptions for COQ automatically. The check that these descriptions indeed represent the original PLC can be done manually. One goal for the generation is human readability to make such a check feasible at least for experienced users. Not part of the TCB are the proof description and its generation mechanism. The proof description only provides hints to the certificate checker. In case of faulty proof descriptions a valid property might not be accepted by a certificate checker. It can never occur that a faulty property is accepted even if wrong proof descriptions are used. Thus, our approach is sound, but not necessarily complete.

1.3 Related Work

Notable milestones on frameworks to certify properties of systems comprise proof carrying code [16]. Proofs for program-specific properties are generated during the compilation of these programs. These are used to certify that these properties do indeed hold for the generated code. Thus, users can execute the certified code and have, e.g., some safety guarantees. At least two problems have been identified:

1. Properties have to be formalized with respect to some kind of semantics. This is sometimes just implicitly defined.
2. Proof checkers can grow to a large size. Nevertheless, they have to be trusted.

The problem of trustable proof checkers is addressed in foundational proof carrying code [1, 22]. Here the trusted computing base is reduced by using relatively small proof checkers. The problem of providing a proof carrying code approach with respect to a mathematically founded semantics is addressed in [20]. In previous work we have also addressed the problem of establishing a formal semantics for related scenarios [5, 8].

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 [10, 11]. This work was a starting point for our formalization of SFC semantics.

The paper [3] considers the SFC language, too. Untimed SFC models are transformed into the input language of the Cadence SMV tool. Timed SFC models are transformed into timed automata. These can be analyzed by the Uppaal tool.

Another language of the IEC 61131–3 standard used for specifying PLC are function block diagrams (FBD). Work in the formal treatment of FBD can be found in [23]. The FBD programs are checked using a model-checking approach. A COQ formalization of instruction lists (IL) – also part of the IEC 61131–3 standard – is presented in [18].

The approach presented in [21] 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.

1.4 Overview

We present the IEC 61131–3 standard, including the SFC language and its semantics as formalized in COQ in Section 2. The tool environment in which our CERTPLC tool framework is supposed to be used and an overview about the tool's architecture is described in Section 3. The CERTPLC ingredients and their interactions are described in some detail in Section 4. Typical proofs that can either be generated or hand-written by using our semantics are discussed in Section 5. Finally, an implementation overview and a short evaluation is given in Section 6. A conclusion is featured in Section 7.

2 IEC 61131–3, SFC, Semantics and Certification

In this section we sketch the semantics of sequential function charts (SFC). The description in this section is based on our earlier work [6] which is influenced by the descriptions in [10, 11]. Furthermore, we present some work on the integration of function block diagrams (FBD) into our tool framework.

2.1 The SFC Language

Our tool framework works with PLC described in the SFC language. The SFC language is a graphical language for modeling PLC. It is part of the IEC 61131–3 standard and frequently used together with other languages of this standard. In such a case, SFC are used to describe the overall control flow structure of a system. The standard is mainly used in the development of embedded systems in the industrial automation domain.

The standard leaves 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 tool to be compatible with the EasyLab [2] tool.

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. The mapping of steps to sets of action blocks is done by the function F. Memory is represented by a function from a set of variables V to a set of their possible values Val_V. $S_0 \subseteq S$ is the set of initially active steps.

A transition is a tuple (S_{in}, g, S_{out}). It features a set of states 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. 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.

Figure 1 shows an example 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.
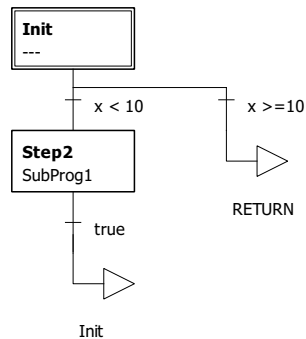
Fig. 1: A loop in the SFC language

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

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

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

1. A rule for processing of 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 customary 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.

2.2 The FBD Language

Function block diagrams are a language from the IEC 61131–3 standard used to model the behavior of action blocks in SFC. Other languages that may be used for this purpose comprise instruction lists (IL) and ladder diagrams (LD).

FBD comprise two basic kinds of elements: function blocks and connections between them. Each function block represents an instruction. There are special instructions for reading and writing global variables. Edges between function blocks are used to model dataflow. Thus, FBD are used to describe functions.

Apart from the basic functionality, FBD may contain cycles in their dataflow description. Semantically such a cycle must feature a delay element. Variable values associated with such an FBD are computed in an iterative process.

In the case of cyclic dependencies an FBD has to be associated with a time slice, a maximal time – number of iterations – for the execution of the FBD. Thus, on an abstract level, FBD may still be regarded as functions and as SFC action blocks.

We have formalized an FBD syntax and semantics framework in COQ that follows the description above. Most parts of this, however, are only to be used manually by users who manually change system descriptions and corresponding proofs.

3   The Tool Setting

In this section we describe our CERTPLC tool's architecture and usage scenarios. Figure 2 shows the CERTPLC ingredients and their interconnections. In an invocation of
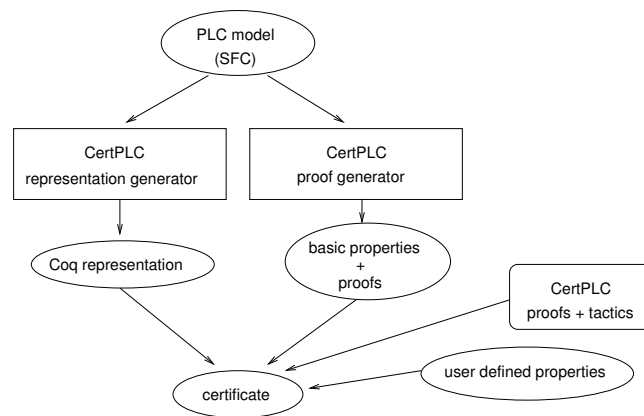

Fig. 2: CERTPLC overview

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 SFC model. The COQ representation is human readable and can be validated against the original graphical SFC specification by experienced users.

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.

In order to achieve a certificate one needs a property that the certificate shall ensure. One needs to formalize this desired property. The property is proved in COQ by using either a provided tactic or a hand written proof script. Our provided tactics use the generated properties and their proofs – provided by the proof generator – and a collection of

- **proofs and tactics**, a kind of library. It contains additional preproved facts and tactics which may be used to automatically prove a class of properties.

System description, used lemmas and their proofs, and the property and its proof form a certificate.

Furthermore, our tool framework comprises a COQ library that can be used by generated and non-generated COQ files. It allows storage of often used definitions in addition to the elements described so far. We have formalized some behavioral definitions of PLC blocks which are typically modeled in other languages than SFC.

Usage Scenario

CERTPLC is developed to support the following standard usage scenario:

- A PLC is developed using the following work-flow:
  1. Establishing requirements,
  2. and derive some early formal specification.
  3. Based on this specification the overall structure – e.g., the control flow – is specified using the SFC language. More fine-grained behavioral aspects are textually specified, e.g., by annotating the SFC structure.
  4. Taking the requirements and this specification, developers potentially using the help of automatic verification tools derive and specify consistency conditions and properties that must hold. Some consistency conditions may directly correspond to a subset of the requirements.
- Regarding 3) the SFC structure is modeled in the graphical EasyLab tool or imported into EasyLab.
- Regarding 4) properties and SFC action blocks are specified using the COQ syntax by trained developers. It is not required to do any proofs in COQ for this.
- CERTPLC generates representations for the PLC specification. Together with the properties a certificate is established automatically or with user interaction: the choice of tactics and in some-cases hand-written proof script code.
- The PLC development is further refined and fine grained parts may be implemented using other languages from the IEC 61131–3 standard.
- Certificates may be either regenerated – if possible – or manually adapted – in case of unsupported language elements that may occur during the refinement – to cope with possible changes.

The certificate can be distributed and analyzed independently by third parties. One overall goal is to convince certification authorities and potential customers of the correctness of PLC with the help of certificates. Since the certificates are independent of the original development and its tools some confidential data (e.g., the certificate generation mechanism and the analysis algorithms used to discover properties of the system) does

not have to be revealed during the process of convincing customers or certification authorities.

The described usage scenario can be adapted. It is, e.g., possible to integrate hand written specifications and proofs.

4 The CertPLC Tool Environment and Coq

In this section we describe COQ specific parts of the CERTPLC tool. We present some static COQ code that is generic to our framework. Furthermore, we present some PLC specific example COQ code – definitions and proofs – to demonstrate aspects of its generation.

Taking the semantics sketch of SFC in Section 2 the semantic representation of the SFC structure is encoded in COQ as a transition system. For each given SFC SFC we generate a COQ representation. It specifies a set of reachable states and a transition relation.

4.1 Realization Using Generic and Generated Files

In order to certify properties of PLC we need files that contain semantics of systems, interesting properties and proofs of these properties. Some of these files are generic, i.e., they can be used for a large class of PLC, properties, and proofs. CERTPLC provides a library of static files that contain generic aspects. Other files are highly specific to distinct PLC. For each PLC CERTPLC generates files that are just needed for this particular PLC, properties formulated on it, and proofs that can be conducted on it.

In particular the following aspects are generic, thus, stored in static files:

- Generic definitions and templates for SFC:
  - Datatype definitions and derived properties of these datatypes.
  - Definitions for building blocks: SFC action blocks, FBD blocks, and common combinations of these blocks.
  - Generic semantics framework comprising an instantiable state transition relation and a generic definition for a set of reachable states.
- CERTPLC is designed to support tactics for solving certain proof aspects. In particular we distinguish:
  - Tactics that contain an overall proof structure, deal with certain system structures and property structures.
  - Tactics that solve arithmetic constraints.

The following aspects are individual for each PLC, thus, they are generated:

- A state transition like representation of SFC formalized using generic SFC definitions and a concrete definition of reachable states instantiating a generic SFC definition.
- Lemmas containing system-specific facts on the PLC and their proofs.

Furthermore, the properties that shall hold are of course specific to each PLC. Their verification is done by either using a tactic that assembles the generic and non-generic parts of the proof or by some hand-written proof script adaptations.

4.2 Generic / Static Parts of the Coq Infrastructure

Here we describe generic parts of the COQ parts in our CERTPLC tool framework. These are realized as static COQ files and can used by the dynamically created files.

Datatypes Datatypes which we have formalized for SFC comprise integers of different length (8,16,32 bit, unbounded) and bools. In COQ they are stored using the datatype *nat* of natural numbers plus a flag that tags them as being members of an integer type. Operators working on these integers perform operations compliant with the type. An easy integration of other bounded integer formalizations (as used e.g., in [15]) is also possible.

Other datatypes like floating point are seldomly used in PLC applications. They are not yet supported, although they could be integrated relatively easy: The basic semantics definitions in our framework are able to support a much richer type system, even dependent datatypes.

Building Blocks Building blocks define common elements for the construction of PLC. Two levels of building blocks can be distinguished:

– Function blocks that are intended to become part of FBD.
– Predefined action blocks. These may be, but do not have to modeled using FBD.

As mentioned in Section 3 we have formalized some of these blocks. Further formalization of blocks should be done together with new case studies since different application domains have different sets of FBD and SFC elements. FBD elements that are highly specific to a single application or an application domain are highly common in PLC. For FBD we have experienced even vendor specific elements for the basic arithmetic operations.

Generic Semantics Framework The COQ realization of the SFC syntax follows the description presented in Section 2. 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. Most importantly, our semantics framework comprises a template for a state transition relation of PLC systems and a template for defining the set of reachable states. In order to realize this, we first define generic instantiable predicates that formalize a state transition relation. We provide a predicate *executeAction* defined in Figure 3 to give a look and feel. It formalizes the effect of the execution of an action block: The predicate takes two states (sometimes called configurations c and c') and returns a value of type *Prop*. We require four conditions to hold in order to take a state transition:

1. An action block a needs to be active.
2. The memory mapping after the transition is the application of a to the previous memory mapping. This is the updating of the memory by executing the action block.
3. The action block a is removed from the set of active action blocks during the transition.

```
Definition executeAction:
   fun c c' =>
      let '(m,activeA,activeS) := c in
      let '(m',activeA',activeS') := c' in
         (exists a, In a activeA /\ m' = a m /\
          activeA' = remove Action_eq_dec a activeA) /\
          activeS = activeS'.
```

Fig. 3: The *executeAction* predicate

4. The rest of the state does not change.

Another predicate *stepTransition* formalizes the effect of a transition from a set of SFC steps to another. Here we require the following items:

1. The validity of the transition (guard expression).
2. The memory state may not change.
3. The activation of steps is conform to the semantics.
4. The activation and requirements of action blocks is semantics conform.

Using these predicates we define inductively the set of reachable states as a predicate. It depends on an initial state (comprising a list of initially active steps), and a transition relation. It is defined following the description in Section 2.

Structural Tactics CERTPLC supports structural tactics that perform the most basic operations for proofs of properties. They work with semantics definitions based on our generic semantics framework. Depending on the property such a tactic is selected by the user and applied as the first step in order to prove the desired property. Different tactics have to be selected by the user: Selection depends on whether the property is some kind of inductive invariant – the default case mostly addressed in this paper – or another class of properties. We have identified several other classes that are relevant for different application domains. Such a tactic is applied as the first step in order to prove the desired property. These tactics already perform most operations concerning the system structure. Especially for the non-standard cases, tactics applications may leave several subgoals open. These may be handled with more specialized tactics tailored for the corresponding characteristics of these proof-goals.

Arithmetic tactics Arithmetics tactics solve subgoals that appear at later stages in the proof. They may be called by structural tactics or work on open subgoals that are left open by these tactics. They comprise classical decision procedures like (e.g., Omega [19] – its implementation in COQ).

Up till now, we are only using existing tactics designed by others. However, we are also working on tactics that combine arithmetic aspects with other system state dependent information.

4.3 Semantics Definitions as State Transition Systems

As seen in Section 4.2 we only need to instantiate a template in order to create a system definition that captures the semantics of our PLC. We need to provide at least a set of initially active steps, a transition relation, and action block definitions.

For the initial step, we provide an initial memory state, where all values are set to a default value and a single active entry step.

The transition relation is generated by translating the SFC transition conditions into COQ. The generated COQ elements of the transition relation for the SFC depicted in Figure 1 are shown in Figure 4. Three tuples are shown, each one comprises a set of activated source steps, a condition and a set of target steps activated after the transition. It can be seen that the condition maps a variable value mapping – part of the SFC state – to a truth condition – returning the type *Prop*. The types used in this expression are 16-bit integer types.

```
( Init::nil ,
  fun m => ((fun  (x : int16) => x <int16 10 )  (m VAR_x) ),
  Step2::nil )

( Step2::nil ,
  fun m => ((fun  (x : int16) => 1 )  (m VAR_x) ),
  Init::nil )

( Init::nil ,
  fun m => ((fun  (x : int16) => x >=int16 10 )  (m VAR_x) ),
  Return::nil )
```

Fig. 4: Generated transition rules in COQ

Appropriate action blocks are selected by their names. In addition, to this, we generate several abstract datatype definitions for identifying steps with names and identifiers and function blocks and action blocks.

4.4 Automatically Generated Proofs for System-specific Facts

CERTPLC is designed to automatically generate for each system basic properties and proofs. These prove some system-specific facts of the system. These proofs are used automatically by tactics, but can also be used manually to prove additional user defined properties of systems.

One important fact that needs to be proven is that only those action blocks may appear in the set of currently active action blocks that do belong to the actual system definition. Our proof generator generates an individual lemma and its proof for each PLC. Figure 5 shows such a lemma for an SFC that comprises just two possible action blocks: *action_Init* and *action_Step2*. The predicate *SFCreachable_states* is created by instantiating a template definition from the generic semantics framework for a concrete

```
Lemma aux_1:
   forall s, SFCreachable_states s -> (forall a, In a (snd s) ->
         (  a = action_Init \/  a = action_Step2 )).
```

Fig. 5: An automatically generated basic property

PLC. *In* and *snd* (second) are COQ functions to denote membership in a set and select
an element of a tuple, respectively. In the case at hand *snd* selects the set of active action
blocks from a state. The proof script itself is also generated. It comprises an induction
on reachable states of the concrete system. Depending on the number of action blocks
in the PLC it can typically comprise several hundred applications of elementary COQ
tactics.

The certification of properties is the key feature of CERTPLC. Users write their
desired properties in COQ syntax. This does not require as much understanding of the
COQ environment as one could think at a first glance. All that is required is writing a
logical formula that captures the desired property.

5   Automatic Certification of Invariant Properties

In this section we describe the principles of automatically proving properties correct.
Proof scripts encapsulating these principles are generated by the CERTPLC framework
components as described in Section 4. We focus on inductive invariants.

5.1   Proof Structure for Inductive Properties

We start with an inductive invariant property I and an SFC description of a PLC SFC.
Following the ideas presented in [8] 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:

$$\forall s . s \in Reachable_{SFC} \Longrightarrow I(s)$$

The set of reachable states for SFC is denoted $Reachable_{SFC}$. $[\![SFC]\!]$ specifies
the state transition relation (cf. Section 4). 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:

$$I(s_0) \text{ for initial states } s_0 \qquad I(s) \wedge (s, s') \in [\![SFC]\!] \Longrightarrow I(s')$$

The first goal can be solved in the standard case by a simple tactic which 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 on a transition from a partially specified state s to a partially specified succeeding state s' can have the following form:

$\forall\, s\, s'$.
$\quad\quad I(s)$ and case distinction specific conditions on $s\ \wedge$
$\quad\quad$ case specific transition conditions that need to be true to go from s to $s'\ \wedge$
$\quad\quad$ case distinction specific definition of $s'\quad \Longrightarrow\quad I(s')$

  The case distinction specific parts in such a goal can, e.g., feature arithmetic constraints, which can be split into further cases.
  Some of the cases that occur can have contradictions in the hypothesis. Consider for example 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. [8]) which can speed up the checking process. Unlike in classical model-checking we get the abstraction from (possibly infinite) concrete states to (finite) arcs in the control flow graph almost for free. Thus, in our case distinctions, we do not have to regard every possible state, we rather partition states into classes of states and reason about these classes symbolically.
- The final step comprises the derivation of the fact that the invariant holds after the transition from the transition conditions and the decision of possible arithmetic constraints.

[8] features a completeness result for a class of inductive invariants for a similar problem.

5.2 Proving Non Inductive Invariants

The main focus of CERTPLC is on inductive invariants, However, additionally we have established a collection of preproved lemmas useful for proving the (un-)reachability of certain states. In particular the following cases turned out to be necessary in our case studies:

- State s can only be reached via a transition where a condition e must be enabled, s is not initial, e can never be true in the system, this implies s can not be reached.
- Under system specific preconditions: Given an expression over states e, if e becomes true the succeeding state will always be s. This is one of the few non-inductive properties. However, the proof of this benefits from a proof that e can only become true in an explicitly classified set of states. This can be provided by one of the techniques above.

Additional consistency properties may be certified by hand-written proof scripts. This, however, requires some level of expertise in COQ.

6 Additional Implementation Aspects and Evaluation

Here we describe additional implementation aspects that are not covered in the previous sections and provide a short evaluation.

The COQ representation generator is implemented as an Eclipse plug-in in Java using the IEC 61131–3 meta model of EasyLab and the Eclipse Modeling Framework (EMF) [14]. Representations and lemmas + proofs for basic properties are generated for COQ 8.3. Likewise our libraries for tactics, lemmas and SFC action blocks are formalized using this version. The realization of this representation generator can be regarded as a simple compiler or model to model transformation. A kind of visitor pattern is used to pass through the model representation in EMF format and emit corresponding COQ code. The generation of PLC specific lemmata and their proofs is similar to code generation. A visitor picks all necessary information and generates the lemma text and its proof script. Some storage of intermediate information is needed. The setup is similar to the techniques used in [8] and [9].

Likewise our work builds upon the PLC semantics of EasyLab which we have formally described [6] and realized in COQ. A combination of our SFC semantics with a semantics of the instruction list (IL) language and an associated case study can be found in [7].

7 Conclusion and Future Work

In this paper we have presented the CERTPLC environment for certification of PLC We described the architecture of the tool framework, possible usage scenarios, the technical realization, and parts of the COQ semantics. CERTPLC is aimed at the formal certification of PLC descriptions in the SFC language. Nevertheless, some features of FBD are integrated. Future work shall extend this support and aims at integrating other languages from the IEC 61131–3 standard. At the current state, the implementation of the tool is sufficient to handle SFC comprising standard elements and smaller invariants efficiently. We believe that the generic parts common to most SFC verification work are realized in CERTPLC. The tool framework is designed such that it is easily extendable, e.g., with additional tactics, arithmetic decision procedures and building blocks for SFC and FBD elements. Such additions – which might be used only in certain problem and application domains – are subject to future work.

Acknowledgments

References

1. A. W. Appel. Foundational proof-carrying code *Logic in Computer Science*. IEEE Computer Society, 2001. (LICS'01).

2. S. Barner, M. Geisinger, Ch. Buckl, and A. Knoll. EasyLab: Model-based development of software for mechatronic systems. Mechatronic and Embedded Systems and Applications, IEEE/ASME, October 2008.

3. N. Bauer, S. Engell, R. Huuck, B. Lukoschus, and O. Stursberg. Verification of plc programs given as sequential function charts. In *In: Integration of Software Specification Techniques for Applications in Eng.*, pages 517–540, Springer-Verlag, 2004.

4. J. O. Blech. A Tool for the Certification of PLCs based on a Coq Semantics for Sequential Function Charts. http://arxiv.org/abs/1102.3529, 2011.

5. J. O. Blech and B. Grégoire. Certifying Compilers Using Higher Order Theorem Provers as Certificate Checkers. Formal Methods in System Design, Springer-Verlag, 2010.

6. J. O. Blech, A. Hattendorf, J. Huang. An Invariant Preserving Transformation for PLC Models. Model-Based Engineering for Real-Time Embedded Systems Design, IEEE, 2011.

7. J. O. Blech and S. Ould Biha. Verification of PLC Properties Based on Formal Semantics in Coq. Software Engineering and Formal Methods, 2011. (SEFM'11) *accepted*

8. J. O. Blech and M. Périn. Generating Invariant-based Certificates for Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS). *accepted*

9. J. O. Blech, I. Schaefer, and A. Poetzsch-Heffter. Translation validation for system abstractions. In *Runtime Verification* , vol. 4839 of *LNCS*. Springer-Verlag, March 2007. (RV'07)

10. S. Bornot, R. Huuck, Y. Lakhnech, B. Lukoschus. An Abstract Model for Sequential Function Charts. Discrete Event Systems: Analysis and Control, Workshop on Discrete Event Systems, 2000.

11. S. Bornot, R. Huuck, Y. Lakhnech, B. Lukoschus. Verification of Sequential Function Charts using SMV. Parallel and Distributed Processing Techniques and Applications. CSREA Press, June 2000. (PDPTA 2000)

12. B. Chetali and Q. H. Nguyen. Industrial Use of Formal Methods for a High-Level Security Evaluation. *Formal Methods in the Development of Computing Systems*, vol. 5014 of *LNCS*. Springer-Verlag, 2008.

13. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.3*, 2010. http://coq.inria.fr.

14. The Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/.

15. X. Leroy. A formally verified compiler back-end. In *Journal of Automated Reasoning*, Vol.43, No.4, pp.363-446, 2009.

16. G. C. Necula. Proof-carrying code. *Principles of Programming Languages*. ACM Press, 1997. (POPL'97).

17. Programmable controllers - Part 3: Programming languages, IEC 61131-3: 1993, International Electrotechnical Commission, 1993.

18. S. Ould Biha. A formal semantics of PLC programs in Coq. IEEE Computer Software and Applications Conference, 2011.

19. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *ACM/IEEE Conference on Supercomputing*, 1991. (SC'91).

20. R. R. Schneck and G. C. Necula. A Gradual Approach to a More Trustworthy, Yet Scalable, Proof-Carrying Code. *Conference on Automated Deduction*, vol. 2392 of *LNCS*, Springer-Verlag, 2002. (CADE'02).

21. A. Sülflow and R. Drechsler. Verification of plc programs using formal proof techniques. In *Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008)*, Budapest, 2008.

22. D. Wu, A. W. Appel, and Aaron Stump. Foundational proof checkers with small witnesses. *ACM Conference on Principles and Practice of Declarative Programming*. ACM Press, 2003. (PPDP'03).

23. J. Yoo, S. Cha, and E. Jee. Verification of plc programs written in fbd with vis. In *Nuclear Engineering and Technology*, Vol.41, No.1, pp.79-90, February 2009.