FORMAL VERIFICATION OF PLC-PROGRAMS GENERATED FROM SIGNAL INTERPRETED PETRI NETS

THOMAS MERTKE and GEORG FREY
Brandenburg University of Technology Cottbus, Germany, th.mertke@gmx.de
University of Kaiserslautern, Germany, frey@eit.uni-kl.de

Abstract

This paper outlines an approach for applying model-checking to logic control algorithms in a way that is easy to understand and to apply by non-specialists. Non-specialists in this case means the designers of PLC programs (mostly control engineers and technicians) because they often have only restricted knowledge in computer science. A graphical design approach (based on Signal Interpreted Petri Nets, SIPN) is used to generate a controller for a benchmark problem. This controller is then checked against a set of semi-verbally formulated properties, and improved to fulfill them. The combination of the graphical SIPN design approach and the semi-verbal specification language results in a very transparent and easy to apply approach to the design and verification of PLC software.

Keywords

Logic Control, PLC, Petri net, Verification, Model Checking.

1. Introduction

In today's flexible, customer oriented, automated production the creation of control software for the machines and plants used is an essential time and cost factor. The increasing complexity of modern software and the rising user-defined safety and functionality requirements cause the urge for new methods to verify software in view of given requirements. This means, not merely to test the software but to prove the correctness and the safety of control programs. One way to achieve this is to use Petri nets, e.g. [DESROCHERS AND AL-JAAR 1994]. There are two Petri net based approaches:

- Petri nets as modeling tool (design) and
- Petri nets as analyzing tool (verification).

In this contribution both approaches are combined. An approach for the design of controllers using a special form of Petri net and its automatic translation into executable IL-Code [Frey 2000] is used to derive the controller and to compile it into IL-

Code for a PLC. There are two ways to apply model-checking to an SIPN controller. First, the direct transformation of the SIPN into a form accepted by the model-checker [WENG AND LITZ, 2000]. Second, the transformation of the PLC-Code generated from the SIPN into the SMV input structure.

In the presented work, the second approach is used. Therefore, the SIPN is automatically translated into an Instruction List (IL) according to IEC61131 standard [IEC 1992], [JOHN AND TIEGELKAMP 2001].

The IL is then translated into another Petri net [HEINER ET AL. 1997] and [HEINER AND MENZEL 1997]. This Petri Net is combined with a formal model (Petri net) of the PLC and its environment (plant-model or stochastic). The overall model is translated into an input-file for the model-checker and used for verification. After this translation, functional and safety requirements can be proven on the generated Petri net.

For the formulation of requirements on a PLC-program many possibilities are available. The most elementary and understandable form is the description of necessary program properties using natural language. Natural language, however, is ambiguous by nature. The safety-oriented specification language (in German: ‘Sicherheitsfachsprache’ - SFS) used in the presented approach is an unambiguously defined subset of natural language with formal syntax and semantics based on temporal logic. It allows a semi-verbal formulation of the requirements and an automatic translation of these requirements into formulae of temporal logic.

Given the formal model of the controller and the requirements in temporal logic, a model-checker can be used to verify the systems properties. The process of model-checking produces a path to the wrong states of the system, if a property is not fulfilled. The one-to-one-to-one correspondence between binary variables in the model-checker input, the IL-program, and the SIPN allows the easy interpretation of the results at the original design.

The transition from SIPN to IL and then back to PN may at first seem cumbersome. However going this way, instead of verifying properties of the SIPN directly, it is guaranteed that the behavior of the ac-
tual control code is studied including the behavior of the PLC. Hence, the actual controller is verified and not its formal representation.

The rest of this paper is organized as follows: in the next section the programming approach using SIPN is shortly introduced. Section 3 provides some details about the verification procedure. Thereafter the air-chamber problem is presented. Section 5 describes how the problem was solved using the presented approach. A short summary concludes the paper.

2. The Programming Approach using SIPN

2.1 SIPN

To allow an easy access to PLC programming the used programming language should
- be capable of graphically describing sequential and concurrent algorithms,
- give visual feedback of the control-flow in these algorithms (current state, following state etc.),
- be easy to apply, and
- be easy to implement i.e. results in fast code.

With Signal Interpreted Petri Nets (SIPN), such a language is at hand. An SIPN is an ordinary Petri net with binary marking and the following extensions for the information flow Every transition is associated with a Boolean function of the input signals, the firing condition. Every place is associated with an output function, that assigns a subset of output signals while it is marked.

The dynamic behavior of an SIPN is given by the flow of tokens through the net, i.e., the change of its marking. This flow is realized by the firing of transitions. Firing of a transition removes a token from each of its pre-places and puts a token on each of its post-places. For the firing process there are four rules:
1. A transition is enabled if all its pre-places are marked and all its post-places are unmarked.
2. A transition fires immediately if it is enabled and its firing condition is fulfilled.
3. All transitions that can fire and are not in conflict with other transitions fire simultaneously.
4. The firing process is iterated until a stable marking is reached (i.e. until no transition can fire anymore). Iterated firing is interpreted as simultaneous. This also means that a change of input signal values can not occur during the firing process.

After a new stable marking is reached, the output signals are computed by evaluating the output functions of the marked places.

A prototypical tool for editing, visualizing, animating, analyzing, and translating SIPNs has been implemented using DiaGen, an environment for rapidly developing diagram editors from a formal specification of the diagram language based on hypergraph grammars and hypergraph transformation (DiaGen is freely available from http://www2.informatik.uni-erlangen.de/DiaGen). The generated SIPN tool consists of a graphics editor that allows for easily editing SIPNs in a direct manipulation manner and a compiler to produce IL-Code [Frey and Minas 2000].

2.2 Example

Figure 1 shows an example of a controller designed using SIPN. The controller is used to fill a tank with a liquid, heat it until a specified temperature is reached, and then empty it again.

![Figure 1: SIPN Example.](image-url)

In the initial state only P1 is marked and hence the output of the net is (0, 0, 0, 0). If the start-button is pressed (i4 = 1) and the tank is empty (i1 = 0 and i2 = 0), then transition T1 fires. The token is removed from P1 and two tokens on P2 and P5 are generated. The new output of the net results in (1, 0, 1, 0), where o3 = 1 means stirring and o1 = 1 filling. After the filling level is reached (i2 = 1) the further processing depends on the temperature in the tank. If the temperature is already above the desired level (i3 = 1) then T5 fires, removing the token from P3 and putting a token in P4. If otherwise, the temperature is below the level (i3 = 0) then T3 fires, also removing the token from P2 but putting it in P3. There the tank is heated (o4 = 1) until the temperature is OK (i3 = 1) and T4 can fire putting the token from P3 to P4. In P4 the tank is emptied (o2 = 1). When the tank is empty, the firing of T6 finally removes the tokens from P5 and P4 and puts a token in P1, hence resulting in the initial state again.
2.3 Code-Generation

To use SIPN as a PLC language on a large variety of PLC hardware it is best to translate the resulting algorithm into one of the standardized PLC languages as for example Instruction List (IL). The direct implementation of an SIPN compiler would only work for one special PLC, but an IL according to the specifications of IEC61131 can be executed on nearly any PLC. This code-generation is done automatically within the above mentioned tool.

3. The Verification Process

3.1 Overview

In this sub-section an overview of the verification procedure is given (see also [HEINER ET AL. 1997], [MERTKE AND MENZEL 2000]). In the following sub-sections all the steps in the verification procedure are explained in some detail. Figure 2 shows the principle verification cycle. This cycle is divided in two main streams: The generation of a PLC system model in terms of a Petri net, and the description of the requirements in formulae of temporal logic (cf. sub-section 3.6). The generation of the PLC system model as a Petri net can be formulated in five steps:

1. designing a controller and translating it to IL,
2. compiling of the PLC user program in to a Petri net model (sub-section 3.2),
3. modeling the system program (sub-section 3.3),
4. modeling the plant with the help of a suitable library as a Petri net (sub-section 3.4), and
5. composition of these three partial models into a model of the whole system in terms of Petri nets (cf. sub-section 3.5).

After the composition step it is possible to prove a set of requirements in the system model. While step 3 in the generation of the system model has to be done only once during the realization of this approach, and step 4 only once for each plant configuration, steps 1 and 2 have to be repeated for each edited PLC program as long as the given requirements are not satisfied.

3.2 Modeling of PLC-program

The PLC-program is automatically compiled into a Petri net (using a compiler called Petri net generator). Therefore, the PLC-program has to be written in a subset of Instruction List called IL₀ [HEINER AND MENZEL 1997], [HEINER AND MENZEL 1998]. The Petri net generator produces a safe place/transition Petri net. Each variable and value is represented in binary form. Pairs of places represent a Bit in its possible states 0 and 1. The number of such pairs corresponds to the number of bits needed by the variable or value.

3.3 Modeling of the PLC (System Program)

Each PLC program is embedded in a PLC system. Such a controller has a system program which transforms the physical data from the plant into data which can be used by the program. For that purpose, the system program maps external values on internal variables values. With a special place named ‘block’ in the environment model a general requirement of a PLC is guaranteed: during the processing of the PLC user program, no changes in the input and output values are possible.

3.4 Modeling of the Environment (Plant)

For the verification of a control system also a description of the behavior of the plant is needed. This behavior is also modeled as Petri net in the so called environment model. In general each state which is possible in the plant (switch is on or off) is modeled as a place in the environment model. Each action (switch on and switch off) which changes the state in the plant is modeled as transition with a place associated with the action as side-requirement. In this model it is possible to model fault states which represent mechanical failure. Finally as state-driven-model it gives the possibility to build a library with standard components, which make the process of modeling further plants easier.
3.5 Combination to the system model

For the verification of a set of the given requirements a model of the whole control system is needed. To this end, the three sub-models (user program, system program, environment) are combined via the fusion of common places. The result of this combination is an ordinary 1-bounded place-transition net, describing the whole control system.

3.6 The safety-oriented specification language

The requirements are formulated in a semi-verbal safety-oriented specification language (SFS) which is defined in [MERTKE AND DEUSSEN 2001]. This specification language, used in this project, is modeled after typical formulation styles of control engineering. It is defined by a formal syntax and a formal semantics based on temporal logic. Hence it allows a semi-verbal formulation of typical requirements. An automatic translation of the safety-oriented specification language into formulas of temporal logic is possible. CTL and LTL are used as formal basis, and the tools PROD, PEP and a self developed tool based on DBB (see [SPRANGER 1998]) are used for model checking. All these formalisms are hidden in the background, a great amount of work was put into an uncomplicated, user-friendly interface to ensure a high user acceptance.

The main principle of SFS is to formulate the requirements in a conditional, i.e. in a sentence like “If a condition is fulfilled, then a reaction has to happen”. A special adding is given by the use of time-related attributes like “immediately”, “simultaneous”, “sometime” or “never”.

The specification language is defined as a subset of German language. But it’s also possible to transfer the structures into other languages like English or French. To do so, the special keywords and the typical word order corresponding to the grammar of the new language have to be adapted.

4. Air chamber example

The following example originates from industrial applications, but is simplified to serve as a benchmark problem and to be used for education, cf. [LITZ AND FREY 1998]. Figure 3 shows an air chamber (compressed air accumulator). Consumers can draw off compressed air via a valve from the chamber. Two binary sensors PS1- and PS2- are used to monitor the pressure in the chamber. Via two compressors (A and B) air is fed into the chamber. The compressors give a signal when they are disturbed. The controller has to meet the following specification:

1. If the pressure is less than 6.1 bar (PS1 switches on), one compressor should run.
2. If the pressure is greater than 6.1 bar (PS1 switches off), no compressor should run.
3. The two compressors should run alternately.
1. If one compressor is disturbed, the other one should substitute it.
2. If the pressure is less than 5.9 bar (PS2 switches on), both compressors should run.

To implement the controller the I/O signals are coded as shown in Table 1 and Table 2.

Table 1: Input signals

<table>
<thead>
<tr>
<th>i</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>i1</td>
<td>Pressure &lt; 6.1 bar</td>
</tr>
<tr>
<td>i2</td>
<td>Pressure &lt; 5.9 bar</td>
</tr>
<tr>
<td>i3</td>
<td>Compressor A disturbed</td>
</tr>
<tr>
<td>i4</td>
<td>Compressor B disturbed</td>
</tr>
</tbody>
</table>

Table 2: Output signals

<table>
<thead>
<tr>
<th>o</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>Compressor A running</td>
</tr>
<tr>
<td>o2</td>
<td>Compressor B running</td>
</tr>
</tbody>
</table>

Figure 4 shows a possible solution by an SIPN with five places. There is no unique solution of the problem, solutions with as few as two places exist.
and used for model-checking in [WENG AND LITZ, 2000] was used as a starting-point. Another controller with 6 places was designed using the SFS and then automatically translated into the functional requirements have been written directly in temporal logic in the form required by the model-checker (SMV in this case). The functional requirements have been formulated in 

After the first verification cycles the controller with six places was chosen for further development because due to its symmetrical structure it is easier to understand, and to analyze (If there is a design-error in the controller, it always shows up at two symmetrical states. If the verification finds a single bad state, there is most certainly an error in the input of the SIPN (something like a typo in the firing conditions).

Two models of the PLCs environment were built: ‘Environment model 1’ is a completely stochastic model, allowing all logically possible sequences and combinations of input signals. ‘Environment model 2’ is a more concrete model of the plants behavior. It includes the simulation of the pressure in the chamber and the stimulation of the sensors. The consummation of air and the failure massages of the motors are of course still stochastic. In this model there are no sensor failures.

The requirements formulated can be grouped in four categories. The simple requirements (categories one and two) have been written directly in temporal logic formulae in the form required by the model-checker (SMV in this case). The functional requirements (categories three and four) have been written using the SFS and then automatically translated into temporal logic.

1. Simple requirements that are independent of the realization (Code): Considering the system. All actuators and all sensors must reach all their possible states (liveness of the system). An error in this category shows a design error in the controller or the environment model. As an example: The input \( i_1 \) should take both values (TRUE and FALSE) at some time and when the input is read by the PLC (rdy_in) results in:
   
   - SPEC EF(rdy_in \& i_1)
   - SPEC EF(rdy_in \& \neg i_1)

2. Simple requirements that depend on the realization (Code): Considering the program. All internal variables must reach different states (if not, they are constants). An error in this category shows a design error in the controller. As an example: The internal variable \( PV_1 \) should take both values (TRUE and FALSE) at some time at the end of a PLC calculation cycle (rdy_plc) results in:
   
   - SPEC EF(rdy_plc \& PV_1);
   - SPEC EF(rdy_plc \& \neg PV_1)

3. Functional Requirements that are independent of the realization: Here the requirements on the system as given in the specification are implemented. This is the primary set of requirements as considered in most works. An error in this category is interesting one, because it may be due to an error in the controller or in the specification. As an example: ‘While the pressure is above 6.1 bar, motor 1 should not be turned on and motor 2 should not be turned on.’ results in:
   
   - SPEC AG !( rdy_plc \& (!i_1) \& \{ o_1 | o_2 \})

4. Functional requirements that depend on the implementation: This forth set of requirements basically checks if the functional behavior of the controller is like it should be according to the SIPN. For every transition in the SIPN: Does it work as expected (if pre-places marked and post-places unmarked, and firing condition fulfilled, does it fire unmarking the pre-places and marking the post places). An error in this category would mean, that the translation from SIPN to the implemented controller is not correct. In a second step it is tested if the respective conditions for the firing of a transition are reachable at all. An error here would mean that there is a dead transition: A clear design error

Figure 5: Final, correct controller.

After the controller fulfilled the simple requirements. The functional requirements have been formulated and proved. Here the problem arises that an error is not always an error in the controller but often the specification is not clear enough. Then the supposedly wrong state gives an hint on what special situation was not considered (for example sensor failures). Naturally, Environment Model 1 (full stochastic) produces more errors in this category. Dur-
ing the verification process nearly all firing conditions of the initial net had to be extended to take care for special situations.

For the complete list of requirements, the description of the environment models, the detailed results and conclusions see [MERTKE AND FREY 2001].

6. Conclusions

With the presented work, it was shown that a Petri net based approach is feasible in deriving correct PLC programs. The combination of graphical controller design (using SIPN) and a semi-verbal specification language (SFS), frees the user from the effort to learn and understand most of the complicated formalisms used for verification. The one-to-one correspondence between places in the SIPN, variables in the IL-Code and variables in the formal model of the PLC-program used for verification allow the easy re-interpretation of verification results at the original design. And therefore also the easy adjustment of this design.

The verification of a controller for a benchmark problem showed, that the initial design was done without enough consideration of all the possible failures that may arise during the operation of the plant (especially present in the stochastic environment model). An example of such a situation is the simultaneous failure of both compressors. Furthermore, it showed, that the given specification is not complete in the sense that it covers all possible states of the system. For example, it is not specified how the controller should react in case of sensor failures (Should the compressors run or not, if the sensors imply that the pressure is above 6.1 bar AND below 5.9 bar at the same time?).

Hence, during the development of the final correct controller the specification also evolved. This is a common fact in verification and may be the main reason to use an approach like the one presented here consisting of iterated design and verification steps in contrast to formal controller synthesis. The specification derived during the presented design approach is finally consistent and complete enough to be used for formal synthesis, the original specification was not.

References


Acknowledgement: The presented work was supported by the Stiftung Rheinland-Pfalz für Innovation under project No 435 (Georg Frey).