A Petri Net based Approach to the Development of correct Logic Controllers

Design, Verification, Validation, Evaluation, and Implementation

Stéphane Klein*,**, Georg Frey* and Lothar Litz*

* Institute of Automatic Control
University of Kaiserslautern, Germany
email: {sklein, frey, litz}@eit.uni-kl.de

** University Laboratory in Automated Production Research
Ecole Normale Supérieure de Cachan, France
email : stephane.klein@lurpa.ens-cachan.fr

Abstract: An overview on the different steps involved in the development of a logic control algorithm from the informal specification to the final implementation on a programmable logic controller (PLC) is given. Based on this overview the steps in the development process are presented in detail. An example is used throughout the paper to illustrate the methods. The approach uses Signal Interpreted Petri Nets for the formal description of control algorithms, symbolic model checking for Verification and Validation, and automatic code generation in Instruction List according to IEC 61131-3 for implementation.

Keywords: IEC 61131-3, Petri Net, V&V, PLC, Implementation, Evaluation
1 Introduction

The main purpose of applying formal methods to logic controller design is to derive a correct control algorithm prior to implementation (cf. Figure 1). If this goal is achieved, the realization of the algorithm on a PLC should contain neither formal nor functional errors. The only remaining problems that could arise during the installation of such a controller are due to implementational aspects not considered in the formal process, as for example things like timing problems in the PLC hardware. All other errors should have been found in an earlier stage during the formal design. This is important because the earlier a problem is found, the easier, faster, and therefore cheaper it is to solve.

![Design Process Diagram]

Figure 1. Design Process

2 Formal Methods in Controller Design

Figure 2 shows different parts of the informal specification, how they are formalized, and how the control algorithm can be checked against the other parts of the formal specification. In Figure 1 as well as in Figure 2 informal parts are shaded gray.

2.1 Informal Specification

First of all we have to realize that an informal specification contains different types of properties. There are functional properties (standard and problem specific) and non-functional properties like for example quality demands on the controller.

Problem specific functional properties. This category contains everything in the informal specification that describes the expected behavior of the controller to be designed. For example that a motor should be turned on after a certain button is pressed.

Standard functional properties. Standard functional properties are independent of the task the controller is intended to solve. An example is that the controller should be deterministic or that it should never get stuck in an infinite loop. While they are independent of the specific control problem, the formalization of standard functional properties often depends on the formalism used to describe the controller.
Non-functional properties. Non-functional properties are all properties that do not influence the behavior of the controller. They include everything that is related to software quality, for example demands on the readability of the code. The formalization of non-functional properties can depend on the formalism used to describe the controller, in the sense that for example the definition of readability for a graphical description differs from that for a textual one.

Figure 2. Formal Methods in the Design Process

2.2 Formalization

To be used in a formal design approach all these informally given properties have to be formalized.

**Formalization of problem specific functional properties.** This formalization consists of two parts: First, the formalization of the control algorithm (design) and second, the formal description of problem specific properties that have to be fulfilled by the algorithm.

In most approaches these two formalization steps use two different formal methods introducing the possibility to express properties in different ways. This is important because not every property is easily formulated in every method. The control algorithm is often specified using graphical descriptions like Petri nets or finite automata. For the properties, temporal logic is widely used. However there are also descriptions that are specific for a given method used for the design of the
controller. For example, if the algorithm is designed using finite automata, a specification could be stated in terms of forbidden states.

The formalization of problem specific properties can not be done automatically but only by man. It is the main engineering effort during the controller development.

**Formalization of standard functional properties.** Standard properties include first and foremost that the algorithm respects the precepts of his formalism. Furthermore properties like liveness or determinism can be required. Standard properties can often be generated automatically by the controller design tool.

**Formalization of non-functional properties.** Not all non-functional properties can be strictly formalized. However often it is possible to connect a non-functional property to some measurable properties of the algorithm and to define a corresponding metric. For example the readability of an algorithm can be connected to the number of comments.

If it is not possible to strictly define a computable metric at least the method for the evaluation of a property should be described in detail in a semi-formal way.

### 2.3 Verification and Validation

These two notions are often confused, therefore let us remind the definition of these two terms as given in [Boehm 1979]:

- « Verification: Am I building the product right? »
- « Validation: Am I building the right product? »

**Verification.** We check if the designed control algorithm fulfils the standard functional properties. This step is the first that has to be done after all the formalizations and if the result is *False*, the control algorithm has to be re-designed until a *True* result occurs. Furthermore, this step has to be done after each modification of the control algorithm.

**Validation.** We check whether the designed control algorithm fulfills the problem specific functional properties or not. If the result of the validation is *False*, we first have to check the control algorithm to see if it has a major failure. If there is no failure in the algorithm, a second way is to look whether the informal property has been correctly formalized because this operation is often the most complicated of the design process. A third possible reason for a non-validated property can be incomplete, unclear, or even contradictory parts of the informal specification. In this case the informal specification has to be improved and formalized anew. Another solution could be the introduction of additional assumptions about the behavior of the plant and its environment. If, during the validation, the control algorithm has to be modified, a step of verification must be done before a new validation.
2.4 Evaluation
In the evaluation procedure non-functional properties of the controller are checked or measured. Depending on the result, parts of the algorithm may have to be improved. A change in the algorithm that goes beyond commenting or graphical beautification makes a new round of V&V necessary.

2.5 Implementation
In order to get a correct realization from the correct formal description of the control algorithm an automatic code-generator should be used for implementation

3 Drill Simulator Example
The presented development approach was used to develop a controller for a flexible manufacturing line [Maas and Frey 2001], [Klein et al. 2002]. However this example is too complex to be used as illustration for all steps in the controller development as it is intended in this paper. Therefore we use the Drill Simulator example. This Drill Simulator (cf. Figure 3) was built at the Logic Control Lab of the University of Kaiserslautern [Klein 2001] to have a minimum working plant for the test of new approaches. It is connected to a Siemens SMP16 IPC running a Soft-PLC of infoteam Software GmbH.

![Drill Simulator](image)

Figure 3. Drill Simulator

The purpose of this simulator is to reproduce the behavior of an automated drilling machine. Since no moving parts were wanted on the example, sensors have been replaced by push buttons and switches, and actuators by LED’s. The final system uses 4 input signals and 4 output signals (cf. Table 1).
Table 1. I/O Variables of the Drill Simulator

<table>
<thead>
<tr>
<th>Addr</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0.0</td>
<td>start_button</td>
<td>Start of the drilling cycle</td>
</tr>
<tr>
<td>I0.1</td>
<td>part_in_position</td>
<td>A part is at the drilling position (switch)</td>
</tr>
<tr>
<td>I0.2</td>
<td>lower_position</td>
<td>Drill head is at the lower position</td>
</tr>
<tr>
<td>I0.3</td>
<td>upper_position</td>
<td>Drill head is at the upper position</td>
</tr>
<tr>
<td>Q0.0</td>
<td>active</td>
<td>The drill is working</td>
</tr>
<tr>
<td>Q0.1</td>
<td>motor</td>
<td>Drilling motor is ON</td>
</tr>
<tr>
<td>Q0.2</td>
<td>move_down</td>
<td>Drilling head moves down</td>
</tr>
<tr>
<td>Q0.3</td>
<td>move_up</td>
<td>Drilling head moves up</td>
</tr>
</tbody>
</table>

3.1 Informal Specification

Problem specific functional properties. In order to reproduce the behavior of an automated drilling machine, the control algorithm has been designed using the following informal specification:

- when there is a part in position and the Start button is pressed, the drill operates,
- a drilling operation consists of moving the drill down until the lower limit is reached and then moving the drill up,
- the drilling motor is ON during moving down and up,
- a light (active) blinks during the drilling cycle,
- the part has to be removed before a new drilling cycle is enabled.

From this informal specification of the behavior, a set of informal functional properties can be derived. These are within others:

P1: Moving the drill down is always followed by moving the drill up.

P2: After the start button has been pressed, the part is drilled, i.e. the drilling motor is on and the drill moves down.

P3: The drill never moves up and down simultaneously.

Standard functional properties. Since the controller will be designed using Signal Interpreted Petri Nets (SIPN), we demand that the controller should be formally correct according to the criteria defined for SIPN in [Frey and Litz 2000 (a)]. This means:

- It should be deterministic, i.e. there should be no conflicts in the SIPN.
- It should terminate, i.e. there is no loop without a stable marking in the SIPN.
- The output signals should be correct in the sense that in each state of the controller an unambiguous value for every output is specified.

Furthermore it should be possible that the algorithm is executed over and over again. Therefore we demand that:

- the SIPN should be live and reversible.
Non-functional properties. As a measure for the quality of the design of an SIPN algorithm the concept of transparency was introduced [Frey and Litz 2000 (b)]. An algorithm is said to be transparent if it is ‘easy and clear to see’ what the controller does at the moment and what it will do in the next steps. A number of criteria for transparency are defined. These criteria cover different aspects such as number of comments, directionality, and I/O-behavior. They are combined in a weighted sum to an automatically computable metric. For the drill controller we specify:

- The SIPN should be well structured
- The SIPN should be easy to read

3.2 Formalization

As already mentioned there are several formalization steps involved in the development of a correct controller.

Formalization of the Algorithm. For the formalization of the control algorithm, we use a special type of Petri Net, the Signal Interpreted Petri Net (SIPN). Following [Frey 2001], 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 without a change of the input signal values). 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.

For real-world programs, SIPN controllers tend to be large and difficult to handle like for most visual languages. However in most cases, certain subnets can be identified which realize certain subtasks and are only active in specific situations. The complete SIPN can, therefore, be replaced by an abstract SIPN where single
places are used instead of these subnets. The subnets are subordinated to these hierarchical places and can in turn contain hierarchical places and so on. Hence a hierarchical SIPN so called SIPN\textsuperscript{H} results. Hierarchy does not only enhance the readability of SIPNs. It is also a valuable means for top-down design of SIPN controllers: Instead of trying to create a single, large net, a programmer can start with an abstract SIPN which is then refined step by step.

The subnets in an SIPN\textsuperscript{H} contain an input and an output place. The input place is given a token at the same time as the corresponding hierarchical place is marked. The token from the hierarchical place can only be removed together with the token on the output place. With this construction and an additional passivity constraint, it is assured, that the subnet is only influencing the process while it is activated by the hierarchical place. For a formal definition see [Frey 2001].

A prototypical tool for editing, visualizing, animating, and translating SIPNs has been implemented using DiaGen (Diagram Editor Generator), an environment for rapidly developing diagram editors from a formal specification of the diagram language based on hypergraph grammars and hypergraph transformation. The main component of the generated SIPN tool is a graphics editor that allows easily editing SIPNs in a direct manipulation manner [Frey and Minas 2000]. The SIPN in the following figures have been developed with this editor.

For the drill simulator an SIPN\textsuperscript{H} as shown in Figure 4 (main net) and Figure 5 (subnet corresponding to the hierarchical place P4 in the main net) is designed.

![Figure 4. Main SIPN](image)

The main SIPN is used to make the active light blink and to call the drilling cycle. In this SIPN we also check that a drilled part is removed (place P5) before a new drilling cycle is enabled.

The drilling cycle is described in Figure 5. The drilling motor is set (place P42) in parallel to the movements down (place P43) and up (place P44) of the drilling
head. When the drilling cycle is completed (place P45), the subnet can be deactivated via transition T4 (only in the phases where the blinker is OFF).

![Sub-SIPN](image)

**Figure 5.** Sub-SIPN

**Formalization of Specific Properties.** In our approach we use model checking [Berard et al. 2001] to perform verification and validation. Therefore we have to translate the properties into Temporal Logic formulae. For the properties P1 to P3 we derive the following formal description:

P1: SPEC AG ((move_down=1) → EF (move_up=1))

This property can be reinterpreted as: It is always true (AG) that: The drill moves down (move_down=1) implies (→) that in the future (EF) the drill will move up (move_up=1). The next property has the same structure.

P2: SPEC AG (start_button → EF (motor=1))

P3: SPEC AG ~((move_up=1) & (move_down=1))

Property 3 means: moving up and down is not (~) possible at the same time.

**Formalization of Standard Properties.** As mentioned above, verification is also done using model checking, therefore the standard properties are also needed as TL formulae.

Standard properties of SIPN like the correct definition of output signals, i.e. an output signal has to be defined in every stable marking, are now automatically written in Temporal Logic formulas by our SIPN-Editor during the formalization of the control algorithm.

As an example the formulae generated for P4 and P5 are given below:

P4: The controller should never run in an infinite loop, i.e. it always reaches a stable marking. To check this property, an End of Cycle (eoc) variable [Canet et al. 2000] which tells us when a stable marking is reached is defined in the tool.
P4: SPEC AG (EF eoc)

P5: the output signals are defined in every stable marking reached, i.e. they are never specified as Non defined (N) or in Conflict (C). For the drilling motor, this would be:

P5: SPEC AG \sim (eoc \& ((\text{motor}=N) \text{ OR } (\text{motor}=C)))

**Formalization of Non-functional Properties.** For SIPN, several criteria for quality have been defined and combined in a weighted sum to a transparency metric [Frey and Litz 2000 (b)]. To formalize our non-functional properties, we only have to set the corresponding weighting factors in this metric. The metrics for comments, directionality, and crossings of arcs in the SIPN are evaluated (weighting factor 1) whereas the other transparency criteria - concerning dynamics and I/O-Behavior - are ignored (weighing factor0). The evaluated properties are defined as follows (Note: in the formulae \# means ‘number of’):

- **Comments:** $t_1 = \frac{\# \text{Comments}}{\# \text{Places} + \# \text{Transition}s}$
- **Directionality:** $t_2 = \frac{\# \text{Arcs in Preferred Direction}}{\# \text{Arcs}}$
- **Crossing of arcs:** $t_3 = 1 - \frac{\# \text{Intersected Arcs in SIPN}}{\# \text{Arcs in SIPN}}$

### 3.3 Verification and Validation

We use the same method to perform Validation and Verification: Symbolic model checking. This is a technique in which a finite model of the system is built and the expected properties of the system are checked on this model. The system is modeled as a finite state transition system and the properties are expressed in a temporal logic. A search procedure (exhaustive state space search or reachability analysis) is then used to check whether the expected properties hold on the finite state transition system or not.

In symbolic model-checking, the state space of the finite state transition system is not explicitly built and we use Binary Decision Diagrams (BDD) to represent the system’s states.

The tool we use, Cadence SMV [SMV Website], requires on the one hand side a description of the control algorithm given in a text file and on the other hand a set of properties written in Temporal Logic. As a result, the model checker gives us a verdict (True/False) and a diagnosis which is a counter example given as a trace. So in order to use SMV, we have to translate the SIPN describing the control algorithm into SMV input code.

**Coding of the control algorithm.** To build the input code for SMV, we have to describe the dynamics of the SIPN. The behavior of the SIPN we use lays on the search of a stable marking before outputs can be affected. This algorithm of
stability search is shown in Figure 6 and the translation into SMV input code is presented in [Weng and Litz 2001].

Figure 6. Coding of the SIPN dynamics

The stability check is made on the firable transitions as given in the formal SIPN definition. The eoc variable is set when the output signals have been computed, i.e. when the control variable equals input_reading.

The translation of the control algorithm in order to use Cadence SMV is now also done automatically with our SIPN Editor. The hierarchical structure can also be handled but the SIPN\textsuperscript{H} is virtually flattened before its translation. Figure 7 shows the flat SIPN of the drill, according to the definition given in [Frey 2001].

Figure 7. Flat SIPN

**Verification.** We check if the designed control algorithm has the earlier defined SIPN standard properties.
P4: SPEC AG (EF eoc)
This property is verified by the control algorithm.

P5: SPEC AG ~(eoc & ((motor=N) | (motor=C)))
This property is also supported by our control algorithm.

Validation. Validation lets us show if the designed control algorithm has the expected properties.

P1: SPEC AG ((move_down=1) -> EF (move_up=1))
The result of this validation is False. SMV gives us the following counter example.

Figure 8. SMV trace for property P1
The analysis of this trace shows that the drill doesn’t move up if it has moved down and the upper position sensor is already ON. Since this case is physically not possible, we can assert that upper_position and lower_position never have the value 1 simultaneously. This can be written in Temporal Logic using following statement:

A1: assert G ~(upper_position & lower_position)
Using this assertion, the property is supported by the model.

P2: SPEC AG (start_button -> EF (motor=1))
The validation result is False because if the variable part_in_position never gets the value 1, the drilling cycle never begins. Here the problem is that the informal specification was not clear enough. Therefore we reformulate P2 as P2a: when the start button is pressed and a part is at the drilling position the part is being drilled, i.e. the drilling motor is on and the drill moves down.
P2a: SPEC AG (start_button & part_in_position -> EF (motor=1))
This new property is validated as true.

P3: SPEC AG ~(move_up=1) & (move_down=1))
This property is supported by our control algorithm.

3.4 Evaluation
For the three aspects of transparency that are evaluated in our example we get optimal values for directionality (all arcs are oriented top-down) and crossings (there are no crossings at all). The third aspect (comments) gives a medium value: only 12 out of 18 net elements have a comment attached to them. Overall, this results in a value for transparency of 0.89 which is quite near to the maximum of 1. Therefore we conclude that we have a good design that could be slightly improved by adding more comments in the control algorithm.

3.5 Implementation
As the SIPN tool has been prototyped as a programming tool, created SIPNs are translated by the tool into equivalent IL programs which implement the SIPNs on PLCs [Frey 2000]. These IL programs are in accordance with the IEC 61131-3 standard for PLC programming languages [John and Tiegelkamp 2001]. Generated code therefore is simply transferred into standard IL compilers which finally produce code for PLCs. Furthermore, the IL-code-generator of the SIPN tool creates code which reflects the hierarchical structure in SIPNH: Instead of creating a “flat” IL program, each subordinated SIPN is translated into a function-block, a kind of PLC subroutine. The IL code, therefore, remains more readable than an unstructured one. The generated code for the drill simulator example is available in [Klein 2001].

4 Conclusions and Outlook
In this contribution, we gave an overview on the different steps of the development of a logic control algorithm. We also defined the method we use for each step and we illustrated it on a small example. The main advantage of our approach is that we developed an SIPN tool that allows us to make the verification and validation easy by the automatic generation of SMV input code and standard functional properties. With the concept of transparency we also have a method for the evaluation of non-functional properties of the controller. Finally, our tool allows us to automatically generate IL code that can be implemented on a PLC.

Our future works will still concern our SIPN tool in order to visualize SMV counterexamples directly on the SIPN. This will make the verification and validation steps more user friendly in the sense that the model checker will be
totally transparent for the designer. Furthermore, we intend to develop a Temporal Logic Editor which should make easier the formalization of problem specific functional properties.

5 References


Note: The Reports of the Institute of Automatic Control, University of Kaiserslautern, cited above are available as pdf-files via the web under http://www.eit.uni-kl.de/litz/