DESIGNING FAULT-TOLERANT CONTROLLERS USING SIPN AND MODEL-CHECKING

Stéphane Klein¹,², Georg Frey¹, Lothar Litz¹

¹University of Kaiserslautern, Institute of Automatic Control, PO Box 3049, D-67653 Kaiserslautern, Germany
e-mail: {sklein, frey, litz}@eit.uni-kl.de, Phone.: +49 631 205 4310, Fax: +49 631 205 4462

²University Laboratory in Automated Production Research (LURPA), Ecole Normale Supérieure de Cachan, 61, avenue du Président Wilson, F-94235 Cachan Cedex, France
e-mail: stephane.klein@lurpa.ens-cachan.fr

Abstract: Formal modeling and verification of a control algorithm allow to derive a correct and fault tolerant controller. In this contribution the combination of Signal Interpreted Petri Nets (SIPN) as formal model and symbolic model checking as verification method is proposed. To apply model checking, the SIPN is translated into a form accepted by the model checker and the properties to be verified are specified in temporal logic. A method for the automation of these tasks is presented. The algorithm generates an SMV input file containing a description of the SIPN and temporal logic formulae for standard properties of the controller. Copyright © 2002 IFAC

Keywords: Petri-nets, Verification, Validation, Programmable Logic Controllers, Modeling.

1 INTRODUCTION

The problems solved by Programmable Logic Controllers (PLCs) get more and more complex. This is partially due to the growing power of these special computers and partially due to the high complexity of today’s flexible manufacturing systems. Growing demands on safety and fault-tolerance of the controllers further increase the complexity. Because of this growing complexity, classical methods of controller design like the direct implementation of the control algorithm using some low-level PLC programming language are no longer feasible.

The key to handle the complexity lies in the application of formal methods in the controller design process. Formal modeling of a control algorithm allows the application of formal verification and validation (V&V) methods. These methods can guarantee that a controller fulfills certain properties. Given a formal model of a controller and some formally specified properties, three different approaches in V&V can be distinguished depending on the amount of information used about the controlled process: Model-based, non-model-based, and constraint-based (Frey, 2002).

In model-based approaches a model of the process under control is included in the analysis. The properties checked are statements on the controlled system. Constraint-based approaches are typically non-model-based with the inclusion of some very restricted knowledge about the process. In the presented work the use of a non-model-based approach to verify standard functional properties like determinism of the control algorithm is proposed. The advantage of a non-model-based approach is that it does not include any knowledge about the process under control; the reaction of the controller on all possible failures is included in the calculation. If the algorithm is proved to be defined deterministically in the presented verification, then the controller will behave deterministically whatever happens in the process it is connected to. In this sense a highly fault-tolerant or robust controller is derived.

For the formal modeling of the controller Signal Interpreted Petri Nets (SIPN) are used in the presented approach, this formalism is described in the following section. The verification is done using the model-checker SMV. Therefore, the SIPN has to be translated into SMV input code (Section 3) and the properties to be checked have to be formalized using temporal logic (Section 4). The approach is illustrated using a small example throughout the paper.
2 SIGNAL INTERPRETED PETRI NETS (SIPN)

2.1 Overview

In the approach developed at the Institute of Automatic Control, controllers are designed using Signal Interpreted Petri Nets (Frey, 2002). This class of Petri nets is an extension of binary marked Petri nets. In the following, the formal definition of this class, its evolution rules and its extension to timed and hierarchical SIPN are presented. The Section closes with the presentation of a tool developed for controller design using SIPN.

2.2 Formal Definition

An SIPN is given as a 9-tuple SIPN = (P, T, F, m₀, I, O, φ, ω, Ω) with:

(P, T, F, m₀) an ordinary PN with places P, transitions T, arcs F, and binary initial marking m₀, with |P|, |T|, |F| > 0,
I a set of input signals with |I| > 0,
O a set of output signals with I ∩ O = ∅ and |O| > 0,
φ a mapping associating every transition ti ∈ T with a firing condition φ(ti) = Boolean function in I,
ω a mapping associating every place pi ∈ P with an output ω(pi) ∈ {0, 1, -, C}, where “-” means “don’t care” or undefined output,
Ω the output function combines the output ω of all marked places Ω: {0, 1}^P → {1, 0, -, C}^O, where “C” means “Contradiction”, i.e. the variable is set to 0 in one place and set to 1 in an other one simultaneously.

2.3 Dynamic Behavior

The dynamic behavior of an SIPN is given by the token flow through the net i.e. the change of its marking. This flow is realized by the firing of transitions. The firing of a transition ti removes a token from each of its pre-places (places pj with (pj, ti) ∈ F) and puts a token on each of its post-places (places pj with (ti, pj) ∈ F). 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. An enabled transition fires immediately, when its firing condition is fulfilled: φ(ti) = True.
3. All transitions that can fire and that are not in conflict with other transitions fire simultaneously. (Note: since conflicts are treated as design errors in SIPN, there is no rule for conflict resolution).
4. The firing process is iterated until a stable marking is reached (i.e. until under the current setting of input signals no transition can fire anymore). Since firing of a transition is supposed to take no time, iterated firing is interpreted as simultaneous.

After the firing of transitions, the output signals are recalculated by applying Ω to the marking.

The algorithm that describes the evolution of an SIPN is illustrated by the flow chart in Fig. 1. The algorithm starts with reading the input signals. Then for all transitions it checks whether they can fire at the current marking and under the current input signal setting. A stable marking is reached when no transition can fire (stability check). If one or more transitions can fire (non stable marking), these transitions are fired and the new marking is set. With this new marking, the transitions are checked again. This process is repeated until no more transition can fire under this input configuration (stable marking). Thereafter, the new output signals are computed. Now, the process starts anew with the scanning of the input signals.

Fig. 1. Evolution algorithm of an SIPN.

2.4 Timed SIPN

The presented SIPN is not sufficient to model real systems because there is no formalism for the representation of time. Therefore, the model has been extended to timed SIPN (Frey, 2001).

In timed SIPN (tSIPN) delay times associated with the input arcs of transitions are considered. A tSIPN is given by a 10-tuple tSIPN = (SIPN, τ) with an SIPN as described in Section 2.1 and a mapping τ, associating every arc fi that is an input arc to a transition fi ∈ (P × T) ∩ F with a delay fi ∈ R^+ with fi = (p_i, t_i), τ_i is the time that a token has at least to stay in p_i before it can be removed by the firing of t_i. Since in timed nets the marking is no longer sufficient to describe the state of the system a clock vector δ is used that associates each place p_i with a clock δ_i. The clock’s position shows the time elapsed since a token was created on the place. If the place is empty, then the corresponding clock reads zero. To see if a transition is enabled, i.e. the token has spent enough time in the place, the value of the corresponding clock δ_i is scanned during the phase of input reading.

2.5 Example

In Fig. 2 a controller for a heating tank is given as an SIPN. The informal specification is:
After pressing the start button (i4 = 1) the empty tank (i1 = 0) is filled by opening valve1 (o1 = 1). The filled tank (i2 = 1) is heated (o4 = 1) until the temperature limit is reached (i3 = 1). The heated contents remain stirred during 10s before the tank is emptied below the minimal level (i1 = 0) by opening valve2 (o2 = 1). During the whole process the contents are stirred (o3 = 1).

While the plant is working, the Run Time system gives a feedback of variables associated to the places so that the actual situation can be visualized in the SIPN Editor. The editor has been developed as a Java applet, available under (Frey, SIPN Editor) so that it can be run on nearly every PC.

The code generation for the model checker as presented in the following section has been integrated into the SIPN Editor.

3 CODE GENERATION

3.1 Structure of the SMV Code

The SMV code generation handles timed and hierarchical SIPN. However, hierarchical SIPN are unfolded to a flat SIPN according to the definition given in (Frey, 2002) prior to the code generation. Throughout the presentation of the code generation the example described in section 2.5 is used as an illustration.

First, modules have to be defined. The SIPN is described as the main module and timers are described using a second module definition. According to the SMV reference language defined in (McMillan, 2000), the codes starts with:

```
MODULE main -- description of the SIPN
```

Where `--` is used to insert a comment in SMV.

The net description in the SIPN Editor, contains the structure of the net, i.e. the sets of places, transitions and arcs. But this is not sufficient to generate a code for the model checker. Additionally, the SIPN dynamics (cf. Fig. 1) has to be described. To describe this cyclic behavior, a stability variable `eoc` (End of Cycle) is introduced (Canet, et al., 2000). To decide whether `eoc` is reached for each transition a variable `ti_fires` is implemented, that gives true when the transition can fire under the current conditions.

To describe the evolution algorithm, two types of variables must be distinguished. First, there are variables that are evaluated in the current computation cycle (the cycle in Fig. 1). That is the case for the output signals (only when a stable marking is reached), the stability variable and the firing of transitions. These variables have to be defined. Second, there are variables that depend on the current value of other variables but only evolve in the next cycle. These are the input signals and the place variables and the timers. These variables have to be declared first and then assigned a value.

3.2 Variable Declaration

The variable declaration begins with the SMV statement `VAR`.

**Input Signals.** Using the SIPN Editor, all the variables used in the controller design are declared as described in the IEC 61131-3 standard (IEC, 2001), (John and Tiegelkamp, 2001). Hence for each external signal an address and a type is given. The decla-

---

Fig. 2. SIPN for the heating tank

2.6 Further Extension: Hierarchy

For real-world programs, SIPN controllers tend to be large and difficult to handle like for most visual languages. But the analysis of these nets shows that independent subnets can be identified. Therefore, the concept of hierarchy that allows that the complete SIPN can be replaced by an abstract SIPN where single places instead of subnets are used has been introduced in (Frey, 2001).

2.7 Tool Support: SIPN Editor

In order to edit, visualize and implement SIPN, a web-based tool (Frey and Minas, 2001) has been implemented. This tool has been developed using DiaGen, available under (Minas, Diagen), which is an environment for rapidly developing diagram editors from a formal specification of the diagram language based on hypergraph grammars and hypergraph transformations (Minas, 2000).

The SIPN-Editor allows not only a smart drawing of hierarchical SIPN but also the direct generation of Instruction List code (Frey, 2000) according to the IEC 61131-3 Standard (IEC, 2001), (John and Tiegelkamp, 2001). This code can be downloaded on a Run Time system and used to control a real plant.
ration allows to distinguish input and output signals by testing if the address contains an I or a Q character. The Type of all the signals has to be Boolean to be in accordance with the SIPN definition. Hence each input signal is declared like i1 in the following:

\[ i1 : \text{boolean}; \]

**Places.** Each place \( p_i \) of the SIPN is represented by a Boolean variable named \( p_i_{\text{active}} \). This variable indicates whether the place is marked or not.

\[ p_1_{\text{active}} : \text{boolean}; \]

**Timers.** Timers are defined as a module \( \text{TON} \) (see section 3.5 for the module definition). In the main module for every timed arc, a variable \( \text{TON}_{p_i} \) is defined where \( p_i \) is the pre-place of the arc. \( \text{TON}_{p_i} \) is the result of the computation of the timer with \( p_i_{\text{active}} \) and \( eoc \) as inputs. In the computation of the firing conditions, the variable \( \text{TON}_{p_i.o} \) (0 for output, type Boolean) is used to check whether the time has elapsed. The heating tank contains only one timed arc, origination from p4. This results in the following declaration:

\[ \text{TON}_{p4} : \text{TON}(p4_{\text{active}}, eoc); \]

### 3.3 Variable Definition

The definition part starts with the statement `DEFINE`. The type of every variable is given by the type of the signal used in its computation. Variables for Transitions, End of Cycle, Outputs, and Places are defined.

**Transitions.** A variable \( t_i_{\text{fires}} \) is defined for each transition \( t_i \) of the net. For each \( t_i_{\text{fires}} \), it is checked that every pre-place of the transition is marked, that no post-place is marked, that the firing condition is true and the times associated to the pre-arcs have elapsed. As an example, the definition of \( t1_{\text{fires}} \) (firing condition associated to the transition) and \( t4_{\text{fires}} \) (timer associated to the entering arc) are given:

\[ t1_{\text{fires}} := p1_{\text{active}} \& !p6_{\text{active}} \& !p2_{\text{active}} \& !i1 \& !i2; \]

\[ t4_{\text{fires}} := p4_{\text{active}} \& !p5_{\text{active}} \& \text{TON}_{p4.o} \& 1; \]

**End of Cycle – eoc.** This variable indicates when a stable marking is reached. According to the formal definition, it is set to 1 when none of the transitions can fire without a change of the input signals.

\[ eoc := !(t1_{\text{fires}} \| t2_{\text{fires}} \| .. \| t6_{\text{fires}}); \]

**Output Signals.** Output signals can have more than two values. It could happen that in a marking an output signal is not defined (N) or that is simultaneously set to 0 and to 1 (C, contradiction).

Prior to the definition of the value of the output signal, two more variables are introduced for each output signal. These variables \( \text{set}_{o1} \) and \( \text{reset}_{o1} \) represent the places that set \( o1 \) to 1 or to 0. Using these two variables, the four combinations can be computed to find the value of the output signal. Since output signals are only computed when a stable marking is reached (\( eoc = 1 \)), a fifth state (NC, not computed) is defined to give the output signals a value between two stable markings. As an example the code for the output signal \( o1 \) of the heating tank is presented below. The output signal \( o1 \) is set to one in place \( p2 \) and set to 0 in places \( p1, p3, p4 \) and \( p5 \):

\[
\begin{align*}
\text{set}_{o1} & := p2_{\text{active}}; \\
\text{reset}_{o1} & := \text{p1}_{\text{active}} \| \text{p3}_{\text{active}} \| \text{p4}_{\text{active}} \| \text{p5}_{\text{active}}; \\
\text{o1} & := \text{case} \\
& \quad \text{eoc} \& \text{set}_{o1} \& \text{reset}_{o1} : 0; \\
& \quad \text{eoc} \& \text{set}_{o1} \& \text{reset}_{o1} \& 1; \\
& \quad \text{eoc} \& \text{set}_{o1} \& \text{reset}_{o1} \& 2; -- \text{N} \\
& \quad \text{eoc} \& \text{set}_{o1} \& \text{reset}_{o1} \& 3; -- \text{C} \\
& \quad 1 : 4; -- \text{NC} \\
& \quad \text{esac};
\end{align*}
\]

### 3.4 Variable assignment

Variable assignment is used to define the evolution of the variables defined in section 3.2. This part begins with the SMV statement `ASSIGN`.

**Input signals.** According to the algorithm described in Section 2.2, input signals can evolve only after a stable marking has been reached. Their next or initial value is chosen arbitrary between 0 and 1. It may not be conform to the initial physical state of the controlled plant but the advantage is that it models possible sensor failures. Between two stable markings, their value remains fixed.

\[
\begin{align*}
\text{init} (i1) & := \{0, 1\}; \\
\text{next} (i1) & := \text{case} \\
& \quad \text{eoc} : \{0, 1\}; \\
& \quad 1 : i1; \\
& \quad \text{esac};
\end{align*}
\]

**Places.** The marking of the places evolves as long as no stable marking has been reached. A place is marked (variable set to 1) if one of its pre-transitions fires and unmarked if one of its post-transitions fires. Simultaneous setting and resetting a place is impossible because an active place disables its pre-transition. The initial marking of a place is indicated by the token placed in the place during the design.

\[
\begin{align*}
\text{init} (p1_{\text{active}}) & := 1; \\
\text{next} (p1_{\text{active}}) & := \text{case} \\
& \quad \text{eoc} \& t5_{\text{fires}} : 1; \\
& \quad \text{eoc} \& t1_{\text{fires}} : 0; \\
& \quad 1 : p1_{\text{active}}; \\
& \quad \text{esac};
\end{align*}
\]

### 3.5 Timers

Timers are defined in a special module. The advantage is that the timer module has to be defined only once and can be used for all timed arcs. The module timer has two input variables. The variable \( p_{\text{active}} \) represents the activity of the timed place and \( \text{stable} \) is set to \( eoc \), to indicate a stable marking because only then timers are evaluated:

\[
\text{MODULE TON(place, stable)}
\]
To model the timer a stochastic automaton (Fig. 3) representation is used (Rossi, 2000).

![Stochastic Automaton Diagram]

**Fig. 3.** Automaton representation of timers

Using this abstraction, a timer has three possible states: inactive (1), running (2) and triggered (3). As long as the considered place is not active, the timer is inactive. When a token has been generated in the place, the timer starts running. As soon as the place is set inactive, for example when another transition fires, the timer is set inactive again. While the timer is running, it can remain in this state or go stochastically in the triggered state. When this state is reached, the output signal of the module is set to 1.

As for the module main, variables must be defined and assigned.

**Variable declaration**

**VAR**

\[o : boolean ; \]
\[state : 1..3 ; \]

Two variables are declared. \(o\) is the Boolean output signal that is set to 1 when the time has elapsed (automaton in state 3). \(state\) is an integer variable used to represent the three possible states of the automaton.

**Variable evolution**

**ASSIGN**

\[init (state) := 1 ; \]
\[next (state) := case \]
\[!stable : state ; \]
\[!place : 1 ; \]
\{(state = 1) & place : 2 ; \}
\{(state = 2) & place : \{2, 3\} ; \}
\{(state = 3) & place : 3 ; \}
\[esac ; \]

\[o := case \]
\[\{state = 1\} | \{state = 2\} : 0 ; \]
\[\{state = 3\} : 1 ; \]
\[esac ; \]

The evolution of the variable state represents the automaton. When the current state is 2 (running) and the place is active (place = 1) SMV decides arbitrary (b) whether the next state is 2 or 3.

### 4 GENERATION OF PROPERTIES

#### 4.1 Standard Functional properties

In this paper the verification of standard functional properties is considered. For the use of the method with problem specific functional properties see (Klein, et al., 2002). First, an informal specification of standard functional properties for control algorithms is given. Standard functional properties are essential for the correct functioning of an implemented controller and independent of the control problem. In (Frey, 2002) four properties for the formal correctness of an algorithm are defined. In the following only the first of them, the deterministic behavior of the controller is studied:

*Every control algorithm must have deterministically defined dynamics and I/O-behavior. If it had not, its behavior in a given situation would depend on implementation aspects. This of course cannot be the aim of a correct design. In detail, this means that in every state of the controller the reaction on possible input signals is defined and a non-contradictory value for each output signal is specified.*

Properties formulated similar to this are widely accepted as mandatory and can be found in many works on formal methods in control.

There are four possible causes for non-determinism in SIPN. Firstly, two transitions that are in conflict in an SIPN lead to a behavior that is not deterministic. Of course, an implemented algorithm will solve this non-determinism but it cannot be the aim of a correct controller design, to leave the decision of how to react on an input in a given situation to implementation aspects. Another source of non-deterministic behavior is the existence of endless loops in an SIPN, i.e. an SIPN that does not terminate. An endless loop may even lead to the hang-up of the implemented controller. Further problems arise due to the distributed assignment of values to the output signals in SIPN. On the one side, it is possible that an output signal is assigned no value at all (not specified output). On the other side, different values can be assigned in different places of the SIPN in the same reachable marking (contradictory output).

Therefore, the three properties Conflict-Freeness, Termination, and Output-Correctness have to be using temporal logic formulae (Bérard, et al., 2001) in order to verify the deterministic behavior. These formulae are generated automatically during the SMV code generation. For each of the three properties the formalization is described in the following sub-sections.

#### 4.2 Conflict-freeness

For every place having two or more transitions starting at it or leading to it, it must be checked that they never can fire simultaneously. In our example, two places present such a characteristic: p2 with two arcs starting from it and p4 with two arcs leading to it.

Two properties are defined:

**SPEC -- conflict_p2**

\[AG !(t2_fires & t6_fires)\]

**SPEC -- conflict_p4**

\[AG !(t3_fires & t6_fires)\]
4.3 Termination

An SIPN terminates if after a finite number of steps always a stable marking is reached. With the presented translation, this property can be formalized using the eoc variable. The SIPN terminates if always eoc is eventually set to true.

SPEC == stable_marking
   AG EF eoc

4.4 Output-correctness

All output signals must be set to 0 or to 1 in every stable marking reached. Hence, for each output signal a property like the following for o1 has to be verified:

SPEC == o1_defined
   AG (eoc -> ((o1 = 0) | (o1 = 1)))

4.5 Results for the example

The verification of formal correctness for the presented example reveals an error in the definition of the output signal o3 if places p3 and p6 are marked simultaneously. Indeed, o3 is set to 1 in p6 and to 0 in p3. In Table 1 the evolution of the different variables is given and it can be noticed that o3 has the value 3 (Conflict) in the fourth computation.

To solve the problem, o(p3) = (0, 0, 0, 1) is changed into o(p3) = (0, 0, -, 1). Performing verification on the corrected SIPN does reveal no further errors.

5 CONCLUSION AND OUTLOOK

In this contribution, an automatic code generation for the model checker SMV has been detailed. The advantages of this automatic translation of SIPN into input code for SMV are that the presented algorithm is part of a user-friendly graphical tool that allows not only the design of SIPN but also direct implementation and that changes made on the designed SIPN do not need a long and fastidious phase of transcription into the model checker code. Further-more standard properties of the SIPN are generated automatically during the translation of the SIPN.

REFERENCES


Frey, G. Homepage of the SIPN Editor at the University of Kaiserslautern (Germany), http://www-2.cs.cmu.edu/~modelcheck/smv.html#nt