A PROGRAMMABLE LOGIC CONTROLLER BASED ON A HIGH LEVEL SPECIFICATION TOOL

M. COURVOISIER, R. VALETTE, J.M. BIGOU, P. ESTEBAN

Laboratoire d'Automatique et d'Analyse des Systèmes
Université Paul Sabatier TOULOUSE France

ABSTRACT

This paper describes an original programmable logic controller. Applications are described by using a high level description language based on Petri nets. A programming console accepts this input language and translates it into data. A validation package is also provided to the user in order to debug his specification before implementing it in the logic controller. This controller is based on a so called "token player" which emulates the Petri net of the control system. Distributed implementation are also considered in the paper.

1. INTRODUCTION

Implementation of logic control is traditionally performed through many successive steps including the translation of the behaviour from written statements to state machines, design of a set of logic state and output expressions derived from the machines and the programming of these logic expressions in a programmable logic controller (PLC).

At each step, many errors can be introduced so that it is difficult to be confident in the implementation obtained, and what is worth, debugging an application is a very hard task.

These considerations bring to light the need of a formal and powerful specification tool which could also be used to verify the model obtained before its implementation. An other important feature holds in the possibility of obtaining a direct implementation from the specified model, the obvious advantage being to preserve the specification without introducing supplementary bugs. If complex control systems are considered (flexible workshop control systems, for instance), distributed implementations and communication networks must be used. Consequently, at the user level, rules for distributing the specifications on the sites must be defined.

The aim of this paper is threefold:
- to define a PLC which emulates a Petri net based specification; this PLC is made up of:
  - a programming console on which the operator can define the application by means of a Petri net based language and use a verification package in order to analyze it,
  - a logic controller which emulates the specification,
- to extend these ideas to a network of PLCs by giving three distribution rules.

2. THE SPECIFICATION AND VALIDATION OF CONTROL SYSTEMS

2.1. Need of an efficient high level model

The kind of industrial installations considered in this paper is mainly manufacturing plants corresponding to discrete process control systems. Generally controlling these systems requires the cooperation of a lot of concurrent procedures. Therefore it is necessary to use specification and implementation languages that are based on formal models which allow a natural expression of parallelism and synchronization.

Presently, most of the PLCs are based on boolean languages or ladder diagrams which obviously have not been defined in order to describe these concepts at a high level.

Among existing specification tools Petri nets have been proved to be, at the same time, powerful enough and simple to use.

2.2. Background

A Petri net \( \langle P,T,A,M_0 \rangle \) is a four-tuple where:
\( P \) is a finite set of places,
\( T \) is a finite set of transitions,
\( A \) is a set of arcs connecting transitions to places and places to transitions,
\( M_0 \) is the initial marking of the net, represented by tokens contained in places at the initial state.

In order to use Petri nets for the modelling of logic control systems, interaction with the external world (the industrial process) must be possible. This can be done by introducing input
and output variables. Input variables are involved in extra firing conditions associated with transitions and output variables represent control signals sent to the process.

2.3. Example

Let us consider the simplified example of a control system represented in figure 1. Two wagons are shuttling between two loading stations and a blast in order to feed its loading skip.

When wagon A is not in operation, it is located in its loading station. As soon as an operation cycle is started, wagon A is loaded, then it moves right until waiting position A is reached. When the unloading section is free, it moves right again and is unloaded. Finally it moves left towards loading station A. Wagon B is controlled the same way.

The specification of the control is given by the Petri net in figure 2. The left (right) part of the net represents the control of wagon A (resp. B). The two wagons can move concurrently until the switch is reached. The condition "unloading section free" is represented by a token in place P9.

2.4. Analysis of a Petri net specification

The advantages of a Petri net specification result from the fact that a detailed analysis is possible, allowing to detect a large number of design errors before any implementation.

Firstly a Petri net representing the control part of a controller has necessarily some "good" properties (bounded or safe, live etc....). Some tools allowing a computer aided analysis of uninterpreted Petri nets exist, for example Givie/Ovide '72 which is now maintained and developed by the firm SYSEDA. An error detected at this level means that the structure of the control part is not consistent. For example deadlocks are possible or some control sequences exist that can never be executed. It must be pointed out that this analysis does not imply the enumeration of all the reachable states (reachable markings) when the structure of the net is not too much complex. It is the case of the Petri net of Figure 2 which can be proved safe and live by a mere automatic reduction procedure.

This first analysis is complemented by another one based on the notion of linear invariant '73,4'. Linear relations involving the marking of set of places can be obtained automatically. These relations allow to detect inconsistencies at the level of the interpreted net. For example from the net of the figure 2 the following property can be derived: only one wagon will be unloaded at a time. As a matter of fact the following invariant: M(p6)+M(p9)+M(p15)<1

is such that it is not possible that more than one of the two places p6 and p15 contains a token at any time. Consequently, the unloading operation will be safe.

Linear invariants are derived automatically by the computer without enumeration of all the states and induction is only necessary to prove that some required specification constraints are implied by them. This way a large number of constraints can be proved to be verified, including the absence of contradictory commands or of dangerous control sequences. When the proof is not possible it can be deduced that some design error is present or that the correctness relies on some temporal relations (and not purely logical ones). When flexibility and reliability are required this second case is generally considered as a design error also.

These two analysis procedures allow to detect the major part of the design errors and enable to reduce the simulation step. As a matter of fact only temporal relations remain to be verified.

3. THE PROGRAMMABLE CONTROLLER

The principles of the programmable controller under elaboration '5' are presented first. Then the programmation console and the controller are described.

3.1. Principles

A programmable controller is generally made up of two parts. One is concerned by the programmation i.e. the description of the required operation. The other one is concerned by the online control of the process. Frequently these two parts are split on two different devices: the programmation console and the controller itself.

A traditional programmation console generally translates a program written in a relatively high level language into code that can be executed by the controller. Our approach is a little more sophisticated because it is based on the utilization of a formal model. Consequently the programmation console of the prototype which is being built includes an editor, a syntactic and semantic analyser, a Petri net analyser and a simulation tool.

When the designer is satisfied with his specification, the controller is loaded by tables describing the interpreted net. The controller then emulates this net taking into account the real values of the inputs (sensors) and actually controlling the actuators.

3.2. The programmation console

The programmation console is a mere personal 8-bit microcomputer with the operating system CP/M. Required software has been exclusively developed in PASCAL in order to remain relatively machine independent.

The editor is any one able to produce a standard CP/M file on a floppy disk. Consequently edition and syntactic analysis are split into two different programs, each one capable of been run
on the 8-bit microcomputer.

The language used to describe the required operation of the controller is structured and non-procedural. A program is then a mere sequence of declarations depicting the interpreted Petri net corresponding to the control structure and giving details of the operations and timers utilized in the data part.

Each identifier involved in the specification has an associated type. The following types and sub-types are defined:
- integer and boolean with sub-types input, output, timer (only for boolean), internal,
- place,
- transition,
- predicate (or boolean expression).

The program control is structured by means of composed instructions. The four possible composed instructions are the following ones:
- NODES that allows the declaration of place and transition identifiers,
- ARCS that allows the description of the graph corresponding to the Petri net by means of three instructions corresponding to the declarations of input or output place list for transitions and the description of graph paths,
- VAR allowing variables declarations including input, output and timer identifiers,
- INTERPRETATION specifying the boolean conditions attached to the transitions and the operations that have to be executed.

Table 1 presents illustrative extracts from the program describing the interpreted Petri net of figure 1.

The syntactical and semantical analyser checks the syntax of a given program, verifies that the identifiers are used correctly (type verification) and translates the interpreted Petri net into tables. These tables are exactly the ones the controller needs for the emulation of the net. At the present point nets of 100 places, 100 transitions and 300 arcs can be translated. The interpretation is limited to 100 input/output variables.

The analysis of the Petri net ("good" properties and invariants) can be done off-line on a large computer. All the analysis procedures are not implementable on a 8-bit computer. Nevertheless the fact that frequently the structure of Petri nets specifying control are relatively simple have encouraged us to design a mini-analysér. It includes marking enumeration with an algorithm allowing the verification of the "good" properties without a systematical storage of all the markings. It also includes reduction (with simple rules), invariant search 'b' and simulation.

3.3. Controller

3.3.1. Outlines of the controller

i) operation cycle

We have chosen a synchronous operation cycle because of its regularity. This cycle is defined by a periodic pulse of the real-time clock. The execution of the programs contained in the controller have to be punctuated by this cycle. After each pulse the input values are modified and the interpreted Petri net have to evaluate as far as a stable marking i.e. a marking for which no transition is fireable (enabled with the attached condition true). The program controlling the evolutions of the interpreted net is called the token player. The execution of the token player as far as a stable marking defines the execution cycle that has to be included in the operation cycle in absence of malfunction. Consequently at each cycle there is an idle time.

These idle times can be utilized to execute test sequences and so to improve the security of the controller. As a matter of fact security is an important requirement when industrial process control is concerned.

ii) emulation principles

The execution of the token player can be decomposed into the three following steps:

Step 1 : Establish a list of the transitions enabled by the current marking, go to step 2.
Step 2 : Consider the next transition of the list, if the list is ended (a stable state has been reached) execute the actions attached to the places, wait for the next real-time clock pulse and then go to step 1, else go to step 3.
Step 3 : Check the condition associated with the transition, if it is false go to step 2, else fire the fireable transition (evolution of the current marking), execute the actions associated with it (evolution of the data part of the controller) and go to step 1.

3.3.2. Software structure

i) Data structure

The Petri net structure is stored by means of lists in order to save memory space. In order to obtain the list of the transitions enabled by the current marking without scanning all the transitions of the net (a very slow procedure for large system) a list of output transitions is attached to each place. In this way a list of transitions that have at least one input place containing a token can be directly obtained from the list describing the marking (list of the marked places with their token loads). In order to complement the description of the Petri net graph a list of input places and a list of output places are associated with each transition. An arc of the graph can appear in more than one list. However it has been shown that an optimization algorithm can be developed in order to avoid any redundancy in the graph storage.

The interpretation of the Petri net is made up of the boolean conditions attached to the transitions and of the actions to be executed.
Boolean conditions are transformed into trees and actions are stored in tables.

It is the programmable console that translates the high-level description of the interpreted net into the adequate lists, trees and tables and provides them to the controller.

ii) Execution

After each real-time clock pulse an operation cycle is initiated. This cycle consists in emulating the interpreted Petri net until a stable state is reached, taking into account the new input values. This emulation has to be terminated before the occurrence of the next real-time clock pulse. It is therefore necessary to avoid the evaluation of every firing condition and of every boolean expression. The data structure we have chosen allow to consider only the transitions that have at least one input place containing a token. It is only for these transitions that the firing conditions are evaluated (step 1 of the emulation). It must be pointed out that neither the places nor the transitions have to be enumerated during this step i.e. the execution time of this step depends only of the number of places that are marked concurrently and does not depend of the size of the interpreted Petri net specifying the control algorithm.

* During step 3 only boolean conditions attached to enabled transitions (i.e. transitions likely to be fired) are evaluated. Consequently the execution time of the emulation algorithm does not depend of the size of the net at all. In fact it depends of the degree of concurrency and of the complexity of the boolean conditions. Moreover the boolean conditions associated with the transitions are evaluated optimally: each variable is evaluated only once and all the variables are only evaluated in the worst case.

3.1.3. Hardware of the controller

The programmable logic controller prototype has the following hardware structure:

- A processor board based upon a Z8008A microprocessor with a 4MHz clock. The emulator program is stored in a 32K ROM on this board. The real-time clock is programmed by means of an INTEL 8253/5 counter/timer.
- A 32K dynamical RAM board where the data utilized by the emulator are stored.
- A serial RS232 interface board is necessary to load the interpreted Petri net description from the programming console. This interface is also used for the interaction with the human operator, if desired (current marking and input/output variable display, for example).
- A board allowing the acquisition of 64 TTL inputs from the industrial installation (an interface is required between this board and the actual sensors). These TTL inputs are updated at each real-time clock pulse.
- A board allowing 64 TTL outputs. An interface is necessary between this board and the actuators.

4. PLC NETWORK

4.1. PLC interaction specification

With the recent development of flexible manufacturing systems, isolated PLCs are no longer sufficient. In fact, PLC networks are required and the major problem is a correct programming of each PLC taken into account their mutual interaction. This interaction has to be structured and verified so that hazards are avoided during the control system operation. For this kind of application a high-level well-structured specification language is still more necessary.

In order to avoid hazards, the use of common variables has to be discarded. Interaction is then expressed by asynchronous messages or by "rendez-vous" (handshakes). When a Petri net specification is used, these two mechanisms can be depicted by merging of places or merging of transitions respectively. The global synchronisation produces then a global net that can be analysed and checked.

4.2. Rules for a correct distribution

It is supposed that the programming of all the PLCs is done through a common programming console. It is then possible to check that the distribution of the global mechanism among the various PLCs is consistent. Three rules have to be applied in order to have a correct behaviour, they are the following ones.

Label rule:

Any variable (sensors or internal counters) involved in the extra firing conditions attached to the transitions has to be private. In particular, a sensor cannot be used by two different PLCs.

Place rule:

All the output transitions of a place have to belong to a unique PLC. In particular, a choice or a conflict has to be solved within a given PLC.

Transition rule

When a transition represents a "rendez-vous" between two PLCs, at most one of its input place can represent a choice (have more than one output transition). This rule derives from the fact that "rendez-vous" implementation implies a dissymmetry. When a PLC initiates a "rendez-vous", it cannot solve a conflict at the same time.

4.3. Example

Let us consider the Petri net in figure 2. The place rule implies that transitions t4 and t13 cannot be distributed in two different PLCs. However the transition rule allows splitting transitions t16 and t13 (p13 does not represent a choice) and thus control of wagon B could be implemented in a separate PLC provided that the
communication is "rendez-vous" based.

In fact this solution violates the label rule because the sensor "unloading position reached", for instance, would be utilized in the two PLCs. Consequently, no repartition is possible. As a matter of fact, a more detailed description of the control mechanism would make appear an unloading sequence for each wagon corresponding to the refinements of places p6 and p15. As it is shown in figure 3 these sequences can be merged and distributed in a different PLC by splitting places p20 and p21. Inter-PLC communication will therefore be asynchronous. It must be pointed out that the two output transitions of place p20 are in a unique PLC and that the place rule is satisfied.

5. CONCLUSION

Presently a programming console and a logic controller have been implemented and tested.

The next step of the project consists in interconnecting such PLCs by using a standard local area network. Because of the communication processor, each PLC becomes a multiprocessor and the IEEE standard 796 (multibus) is planned to be used.

The programming console will be common to all the PLCs and therefore the Petri net based specification language has to be extended in order to allow the description of the distribution over the PLCs and to depict their interactions.

REFERENCES


---

\[
\begin{array}{l}
\text{NODES} \\
P_1, P_9, P_{10}: \text{PLACE} 1; \\
P_2, P_3, P_4, P_5, P_6: \text{PLACE}; \\
\hline
\text{ARCS} \\
\text{PATH} = (P_1, T_1, P_2, T_2, P_3, T_3, P_4, T_4); \\
\{\text{wagon A on section A}\} \\
\hline
\text{VAR} \\
\text{STARTA, STARTB: BOOLEAN INPUT;} \\
\text{LOA, LOB: BOOLEAN OUTPUT;} \\
\text{EOLA, EOLB: BOOLEAN INPUT;} \\
\text{MFA, MFb: BOOLEAN OUTPUT;} \\
\{\text{loading}\} \\
\{\text{end of loading}\} \\
\{\text{move forward}\} \\
\hline
\text{INTERPRETATION} \\
\text{COND T1 = STARTA;} \\
\text{ACTION P2 = LOA;} \\
\text{COND T2 = EOLA;} \\
\text{ACTION P3 = MFA;} \\
\{\text{move forward}\}
\end{array}
\]

\text{TABLE 1: Example of program}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{system.png}
\caption{System to be controlled}
\end{figure}
**FIGURE 2.** Specification of the control mechanism

**FIGURE 3.** Distribution of the unloading control mechanism