Binary Petri-nets and PROLOG for Intelligent Robots*

Z. Cai*, A. Farnham*, P. Gomez^, M. Hermida^,
R. Newcomb*^, V. Rodellar^, and J. Tian*
*Microsystems Laboratory
Electrical Engineering Department
University of Maryland
College Park, Maryland 20742 USA
Phone: (301) 454-6869
*Computer Science Department
University of Maryland
College Park, Maryland 20740 USA
^Groupo de Sistemas PARCOR
Facultad de Informatica
Universidad Politecnica de Madrid
Ctra. de Valancia, Km 7,00
28031 Madrid Spain

Abstract:

This paper presents two algorithms for implementing binary Petri-nets using the logical language PROLOG. The first algorithm is derived directly from the characteristics of the Petri-net graph. The second algorithm is an implementation of the equations governing binary Petri-nets. We use the implementation of the equations to verify the results of the first algorithm.

We use PROLOG to analyze the implementation of a binary Petri-net graph. PROLOG and binary Petri-nets are reviewed, and the program structure for the two algorithms is given. The implementation of the binary Petri-net graph is identical to the implementation of the binary Petri-net. Consequently, if a line of robots is based on the binary Petri-net, the behavior of the line can be analyzed. When used with direct executing computer architectures, the binary Petri-net becomes an extremely efficient method of modelling intelligent production lines.

I. Introduction

Binary Petri-nets can be used to model intelligent robots and related systems, such as flexible production lines [1]. In such an application, the implementation of the Petri-net graph is identical to the implementation of the Petri-net mathematical model. Consequently, the qualities of the production line can be analyzed prior to building the line. The equations describing the Petri-net provide for a high level logical language that, when executed, exactly represents the binary Petri-nets logical equations. Therefore, implementing the binary Petri-net equations with the high level logical language PROLOG gives us an opportunity to investigate the behavior of an intelligent robot system.

The advantages of using a computer to analyze the model are: (1) The computer can implement a logical program based on the Petri-net model; (2) After the CPU reads the system's states, the computer databases can be altered to represent the implementation of the binary Petri-net system. The continuous monitoring of the databases allows for deadlock and conflict actions to be solved immediately; (3) The high-level PROLOG language can be associated with high level computer architectures, thus providing an efficient means to implement the binary Petri-net system [2][3].

In section II, we review the equations that govern the behavior of binary Petri-nets. In section III, we present an algorithm for a PROLOG program derived directly from the Petri-net graph. Additionally, we give a brief discussion on the behavior of PROLOG. In section IV, we present our algorithm for producing a PROLOG program from the equations. Both of these programs implement a binary Petri-net model of a system. A final discussion is included that gives the relation between the execution of the binary Petri-net model and the execution of PROLOG.

II. The Equations Describing the Binary Petri-net

Binary Petri-nets have been represented in terms of integer algebra [4]. The full set of equations describing binary Petri-nets allows for the determination of the firing vector in terms of the marking function and the input. During system execution, there are logical relationships among the finite set of initial states, the final states, the marking function, the external inputs, and the timing sequence. In binary Petri-nets, actions are modelled by transitions, and occurrences are modelled by the firing of a

* This research was supported in part by the US-SPAIN Joint Commission under Grant CCB-84-02-002.

transition at time λ, where λ is an element of the time sequence 0,1,2,...,N. It is important to analyze the sequence of actions in the system, because it exactly describes the execution behavior of Petri-nets. If we use the set of possible sequences of actions of the system, then we can implement the system by executing these sequences. Consequently, we use the transition as a basis for our first algorithm using PROLOG.

In a binary Petri-net, there is a strong logical relation between system states and actions associated with a firing time sequence [5]. Therefore, we use the logical language PROLOG to describe these logical relations; and we implement the system as a logical program in a computer. The system state equations represent simple logical relations between the system firing function, marking function, input function, and initial states. In contrast to Petri-nets without timing sequence restrictions, binary Petri-nets execute with clocked timing sequences. These timing sequences trigger the system implementation and maintain the system state consistency. Since most robot systems are real time systems, binary Petri-nets should suitably model them for real time performance, especially in the case of flexible robot production lines [1].

The clocked processors of an assembly line of robots can be modelled using binary Petri-nets. The actions of the robot line might depend upon the absence or presence of just one item (or token like signal). The processor can be thought of as an array of gates. The output of any one gate will be transmitted to all the gates connected to it. For timed binary Petri-nets, the number of tokens in a place is a binary number and each place connected to a fired transition receives a token.

For binary Petri-nets with p places and t transitions, $M(\lambda)$ is the t-vector containing the firings at time λ. An entry of 1 denotes a firing at time λ, while an entry of 0 denotes no firing at time λ. The external inputs (into the places) are contained in the p-vector $I(\lambda)$, with entries restricted to 1 or 0. Matrix $C=C^{+}-C^{-}$ is the incidence matrix and $M_o(0)$ is the initial net marking matrix. The entries C_{ij}^{+} of C^{+} are 1 or 0 depending on whether transition i is incident or not incident upon place j. Likewise, the entries C_{ij}^{-} of C^{-} are 1 or 0 depending upon whether place j is incident or not incident upon transition i. With these preliminaries, we can give the binary Petri-net transition functions.

$$M(\lambda) \doteq M_o(\lambda)+I(\lambda) \tag{1a}$$
$$F(\lambda+1)=C^{-}\Box M(\lambda) \tag{1b}$$
$$M_o(\lambda+1) \doteq M(\lambda)+C^{T}F(\lambda+1) \tag{1c}$$

In equations 1, "$\doteq$", is the dot equality which means that we carry out the operations on the right using normal integer arithmetic. Then, in the final result, we replace every positive number on the right by 1 and every other number by 0. This forces the Petri-net into a state restricted to binary numbers. The "$\Box$" operation is defined as

$$a\Box b=(a_1'\vee b_1)\wedge(a_1'\vee b_2)\wedge\ldots\wedge(a_p'\vee b_p)\wedge [a_1\wedge b_1\vee a_2\wedge b_2\vee\ldots\vee a_p\wedge b_p]$$

where a is a row p-vector of zeros or ones, b is a column p-vector of zeros or ones, i=0,1,...p, and $\vee$, $\wedge$, and ' are logical boolean AND, OR, and complement operations, respectively.

III. Using the Graph as a Basis for the PROLOG Program

In this section, an implementation of binary Petri-nets is derived from the Petri-net graph, rather than from the logical equations. The program takes advantage of the characteristics of the transition in a Petri-net. A transition will fire only when each and every one of the input places has a token available. The PROLOG program similarly checks for tokens in every input place of every transition. Once a transition is enabled, the program transfers the tokens from the input places to the output places. An advantage of PROLOG is that it continually searches for new transitions to fire. The appropriate application of this quality to Petri-nets is for deadlock. Once deadlock occurs, no more transitions can fire. This algorithm allows the user to be able to produce a binary Petri-net model without having to go through the trouble of producing numbers for the equations. The program allows for an arbitrary number of places, transitions, and connections between the two.

The program acts as a system simulation tool, allowing the user to change the program input if the behavior of the net is not satisfactory. This eliminates any costly hardware implementation before the behavior of the net is actually known.

PROGRAM STRUCTURE

Step 1. Initialization
- a. initialize counters to zero
- b. ask user to input the total number of transitions and places in the net
- c. assert the number of transitions and places into the database.
- d. ask the user to input the places that enable the transitions.
- e. ask the user to input the places that the transitions enable.
- f. assert Step 1d and 1e into the database.

g. repeat Step 1d, 1e, and 1f for each transition in the net.

Step 2. External input to the net
  a. ask the user if any input is needed for the net. (note that this is the only way to get out of deadlock) b. assert the new marking into the database and proceed to Step 3.

Step 3. Determining which transitions are enabled
  a. remove all items from the database that are from a previous firing sequence.
  b. clear all counters to zero.
  c. assert into the database that all transitions are ready to be checked for whether or not they are enabled.
  d. check for each transition whether the enabling places have tokens in them.
  e. for each transition, keep an account of the number of enabling places that have tokens.
  f. for each transition, check for the total number of places that had tokens as found by Step 3e.
  g. if the total number of enabled places for the transition matches the total number of input places, then go to Step 4.
  h. repeat Step 3g for all the transitions in the net.
  i. start at Step 2 again. (this is an infinite loop)

Step 4. Firing the transitions
  a. change the account of enabled places found in Step 3e to zero enabled places.
  b. for each input place to the transition, remove the token that enabled the transition.
  c. for each output place connected to the transition, add a token to it.
  d. return to Step 3h.

IV. Using the Equations as a Basis for the PROLOG Program

To verify that PROLOG behaves in a matter appropriate for binary Petri-nets, we can produce a PROLOG program with the equations as a basis, rather than the graph itself.

PROGRAM STRUCTURE

Step 1. Initialize the Petri-net
  a. assert the external input vector into the database
  b. assert the initial marking into the database
  c. assert the incidence matrix into the database

Step 2. Solve for marking function
  a. call Step 4 and solve for M_0
  b. search database for current input
  c. add the two, and take the dot equality
  d. go to Step 3

Step 3. Solve for firing function
  a. search database for C^-(row-by-row)
  b. use the box equality with the Marking matrix found in Step 2.
  c. repeat a. and b. for every row of C^-
  d. go to Step 4

Step 4. Solve for the intermediate marking function
  a. calculate the transpose of the incidence matrix
  b. multiply this with the firing vector found in Step 3
  c. add this with the current marking found in Step 2
  d. take the dot equality of this
  e. begin again with Step 1

V. Discussion

In real applications, such as flexible robot production lines, conflict actions can happen. We can classify these conflict actions into two classes. There are (1) logic decision conflicts and (2) information requirement conflicts. For the first class of conflicts, we can use the state equations of the binary Petri-net to set up correct logical relations. Since the system states are always consistent when using the binary Petri-net model, a conflict decision error can not be allowed during the system execution. For the second class of conflicts, we can borrow techniques from computer information theory and distributed database theory to solve them. Although system states change during program execution, we can still use the algorithms that these theories provide to prevent conflict actions in the computer database. Also, we can treat the central control computer and any computers in the robot itself as a distributed system; thus, we can still use distributed database techniques to prevent information requirement contradictions, such as deadlock.

To test the two programs, we used Example 1, of Reference [4]. The implementation of the equations provided a check for the implementation directly from the graph; however, the implementation of the graph was more versatile. First, no knowledge of the mathematics was needed to implement the first algorithm. Moreover, the graph implementation was able to handle net deadlock more effectively. External input could be provided for any robot at any time; consequently, no input vector had to be derived to get out of deadlock.

We have presented two algorithms which implement binary Petri-nets using PROLOG. The qualities of PROLOG allow for binary Petri-nets to be modelled in a detrministic manner. PROLOG continually searches for solutions, allowing all firings of the net to take place before deadlock is reached. This method of examining binary Petri-nets can be extended to the less deterministic classical Petri-net.

References

[1]. Z. Cai, A. Farnham, Z. Ghalwash, P. Gomez, V. Rodellar, and R. Newcomb, "Petri-nets for Robot Lattices," Proceedings of the 1987 IEEE International Conference on Robotics and Automation, Vol. 2, March 1987, pp. 199-204.

[2]. K. Hwang, J. Ghosh, and R. Chowkwanyun, "Computer Architectures for Artificial Intelligence Processing," <u>IEEE Computer Magazine</u>, Vol. 20, No. 1, January 1987, pp. 19-27.

[3]. Y. Chu and M. Abrams, "Programming Languages and Direct Execution Computers," <u>IEEE Computer Magazine</u>, Vol. 13, No. 7, July 1981, pp. 22-32.

[4]. H. Alayan and R. Newcomb, "Binary Petri-net Relationships," to appear in the <u>IEEE Transactions on Circuits and Systems</u>, 1987.

[5]. J. L. Peterson, "Petri Net Theory and the Modeling of Systems," Prentice-Hall, Englewood, NJ, 1981.

309

1987 IEEE WORKSHOP ON LANGUAGES FOR AUTOMATION

THE TECHNICAL UNIVERSITY OF VIENNA
VIENNA, AUSTRIA AUGUST 26-27, 1987

LANGUAGES LANGUAGES LANGUAGES LA
G PROGRAMMING PROGRA
N DESIGN DESIGN DESIGN
HIGH-LEVEL COMMUNICATION LA
GES ROBOTIC CONTROL LANGUAGES ROB

Sponsors:
Austrian Computer Society
University of Pittsburgh

In cooperation with:
Computer Society of the IEEE
Technical Committee on Factory Communications
IEEE Industrial Electronics Society

Computer Society Order Number 797
Library of Congress Number 87-45381
IEEE Catalog Number TH0199-0
ISBN 0-8186-0797-1
SAN 264-620X

THE COMPUTER SOCIETY OF THE IEEE

IEEE

COMPUTER SOCIETY