Jurnal Teknologi, bil. 23, Jun 1994 hlm. 15 – 26

ON THE SUITABILITY OF SEQUENTIAL PROGRAMMING LANGUAGES

BADRI ABU BAKAR
Dept. of Control Engineering
Faculty of Electrical Engineering
Universiti Teknologi Malaysia
Jalan Semarak, 54100 Kuala Lumpur, Malaysia
E-mail: badri@fkeserv.fke.utm.my

Abstract. In designing sequential control systems, the programming language plays an important role for the success of system implementation. A Programmable Logic Controller (PLC) is normally equipped with several languages like Relay Ladder Logic, Statement List and others. Petri nets, Rule-based and State Transition methods are the language of future PLCs. The paper evaluates the performance of these languages especially for a system which exhibits a pattern in its operation. This is done on a case study based on a trainset problem with multiple-engine on the same track. It will demonstrate the strengths and weaknesses of each technique and it will serve as a basis of favouring one or the other for certain applications in a sequential control system.

Through this paper, we can see various techniques on the problem to show that each technique has its own strengths and weaknesses. In a complex operation with a pattern, the rule-based technique is the well-suited technique. Similarly, if the system uses complementaries and needs exception handling rule-based is the best technique to be employed. Unlike Petri nets, State Transition Matrix (STM), ladder logic and others, the rule-based technique does not relate to dimensionality of the problem and therefore system growth and modification is easily coped with. The Petri net technique is very good at parallel subsequences but falls down when much branching and inverse places of action are needed in the system. STM on the other hand is good at much branching and giving system options clearly and unambiguously so long as the matrix is manageable. The STM technique becomes less helpful when the matrix grows. Although both Petri nets and STM are easily understood and readily communicable, their solutions are implementation specific; system changes and annexation would mean re-programming almost from scratch.

Various functions have certain peculiarities that make them unsuitable to be programmed in a particular method. Therefore, the choice of a technique suitable for a particular problem is still the best method of designing sequential control applications. Failure to program these operations with the most suitable technique will result in a difficult and awkward solution with the consequent penalty associated with undue complexity in terms of error-checking and implementation.

1 INTRODUCTION

In designing sequential control systems, the programming language plays an important role and it is vital to the success of system implementation. In this respect, a PLC – a controller for sequential systems – is normally equipped with several languages for specifying and implementing sequences; the most popular one is the Relay Ladder Logic (RLL) (Balt [2]).

It is said that RLL is a good technique for short and straightforward sequences, and copes very poorly for parallel subsequences. The technique tends to lump all the functions into big and lengthy codes without any proper organizational structure. A technique called Petri net, on the other hand, is good at parallel subsequences. Since the technique supports topological and hierarchical structures, the technique is very useful for partitioning control functions, but the solution is rather clumsy for systems with much branching of states. Another method called State Transition Matrix (STM) however tends to complement the Petri net in this situation, but again the matrix grows very quickly with the number of possible markings in the Petri net (Henry et al [4]). Although both methods are easily understood and readily communicable in many situations, their solutions, including ladder logic, are implementation specific; system changes would mean re-programming almost from scratch. A method called rule-based technique on the other hand can easily be modified to work on a bigger scale and is simple enough to be realized with any high level language on computers (Badri [1]).

This paper is an attempt to ascertain the above notion especially for a system which exhibits a pattern in its operation. Here, a comparison is made in order to assess their performance against a case study based on a trainset problem with multiple-engine on the same track. This, at once, will demonstrate the strengths and weaknesses of each technique on this particular example and it will serve as a basis of favouring one or the other for certain applications in a sequential control system.

2 TWO ENGINE PROBLEM ON THE SAME TRACK

The running of two engines on the same loop of a track such that one does not run into the other is an interesting example (Geur [3]). The ideas used in this example can be extrapolated to include three or more engines. Obviously, without any control one engine will go faster than the other and there will be a collision. To overcome this a technique has to be devised. The track is split into eight sections electrically isolated from each other but still forming a loop. See Figure 1. To detect the passage of an engine over a given point on the track, reed switches are planted between the rails and a small magnet fixed under each engine. The switch will close for a moment as the engine passes over and it is recognized that there is a need to 'de-bounce' contacts. In this example the track sections are left unpowered. A particular track section is powered as needed.

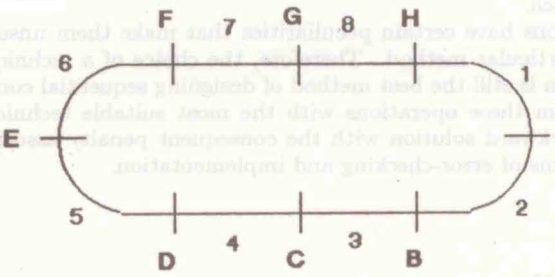

Fig. 1 8 track sections with 8 switches

For the proper running of the two engines, there must be a length of unpowered track between each of the engines at least one track section long. At the end of each section an

engine will be able to proceed providing the next section is available to be powered i.e. not immediately behind the other engine.

For only one engine on the track, no waiting is ever required. But for the two engine problem, it calls for a consideration. When an engine passes over a switch, there will be two possibilities :

i) the next track section is available, the power is switched on to this section and the train moves smoothly from one track section to the next.

ii) the next section is not available, power will not be switched to the next track section. The engine will run from the powered track section to the unpowered track section and stop.

This means that there is a need to 'remember' that the engine is waiting for power to be put on the next track section i.e. 'remembering' that the engine has passed the detector at the end of the previous track section. This calls for an extra state which is switched on when the engine goes over a reed switch and reset when power is switched unto the next track section. Now, we want to see how all the techniques solve this problem for us.

3 LADDER LOGIC IMPLEMENTATION

For the above trainset operation, the ladder logic solution for one engine problem is as shown in Figure 2. Note that the INIT is necessary for the system to start. For the two engine problem, we need to remember that the engine has passed the detector at the end of the previous track section is implemented using an extra rung. This is as depicted in Figure 3. To power the track section 2, switch A sets state AA; when track section 4 is clear, AA powers track section 2 and resets AA. Consequently, it leaves two clear track sections between the engines. The complete solution to the two engine problem is given in Figure 4. Note that, to start with, two pairs of track sections are energized i.e. 1 & 8 and 4 & 5.

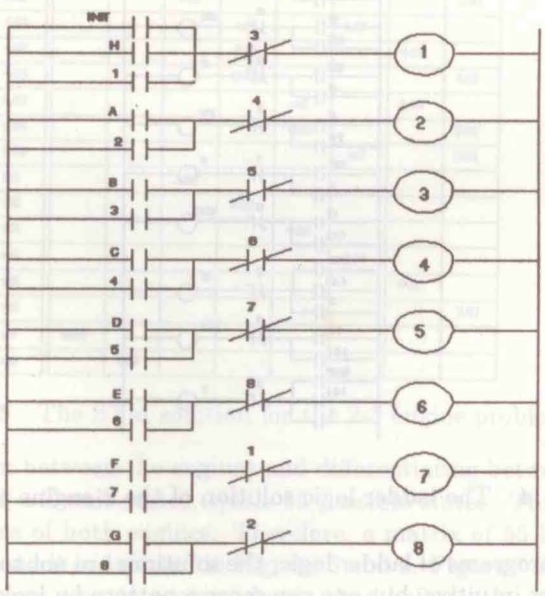

Fig. 2 The ladder logic solution of 1-engine problem

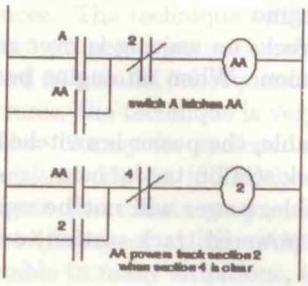

Fig. 3 Using an extra rung to implement a wait state

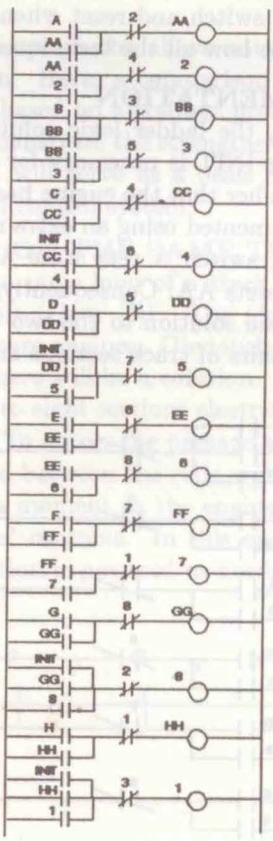

Fig. 4 The ladder logic solution of the 2-engine problem

Looking at these programs of ladder logic, the solutions are not too difficult. The method of realization is rather intuitive, but one can sense a pattern by looking at the solution. We would say the ladder logic solution works well for this example and therefore is not ill-suited to the problem.

4 STATE TRANSITION MATRIX SOLUTION

The STM solution calls for a new thought on the problem. There are 8 track sections where the first engine can be. The second engine cannot be on the same section, nor on the two sections immediately in front or behind. That leaves only 3 possible track sections for the second engine. However, there is a need to latch reed switches when the engine passes over and waits for the next track section to be available. If we do not distinguish between the first and the second engines, there will be $(8 \times 3/2) + 8 = 20$ possible states in the matrix.

A variation to the above solution would be to have only one section between the engines which leaves 5 possible track sections for the second engine. This gives rise to $(8 \times 5/2) + 8 = 28$ possible states in the matrix. The solution is as depicted in Figure 5. On close observation, we can detect some form of a pattern in the solution, but it is not clear.

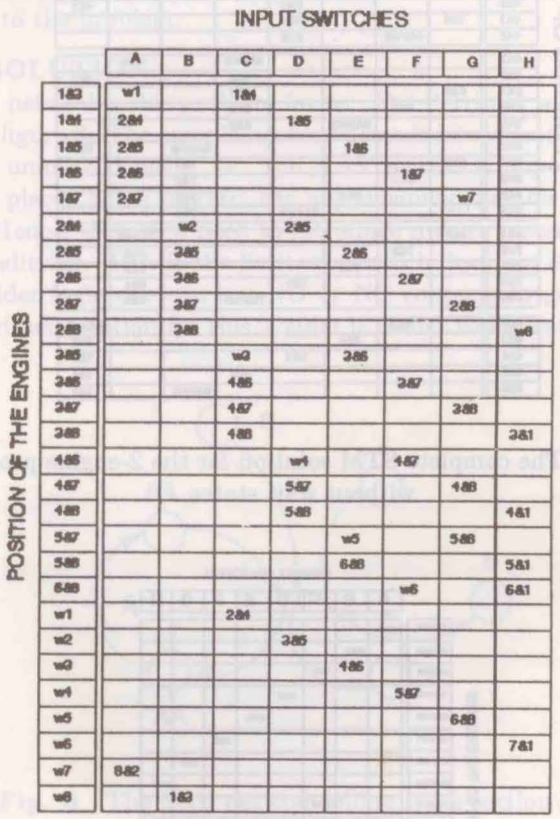

Fig. 5 The STM solution for the 2-3 engine problem

With one clear section between the engines and differentiating between the two engines, there will be $(8 \times 5) + 8 \times 2$ wait states equals 56 possible states. This result includes all the possible combinations of both engines. Therefore, a matrix of 56 by 8 is the complete solution to the problem and it is depicted in Figures 6 and 7. Figure 7 shows the wait states solution. This matrix shows a pattern more clearly than before.

Close observation reveals that both solutions show a pattern, but the pattern of the 56 state version is easier to follow and check as compared with the 28 state version. Although

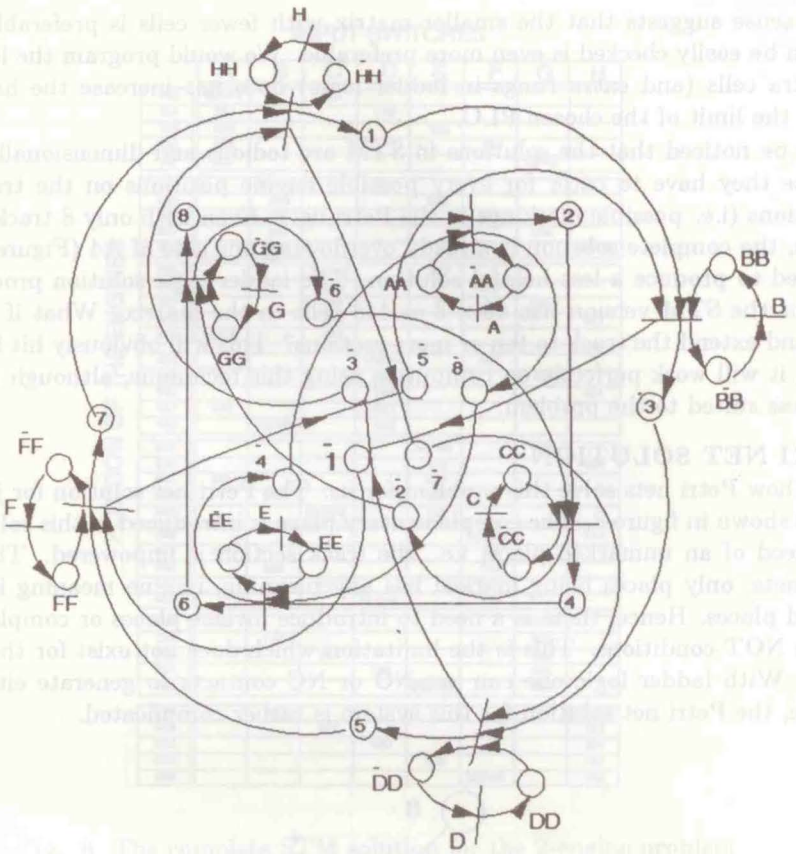

Fig. 9 The complete petri net solution of the 2-engine problem

addition or modification, means re-programming. No two similar applications produce alike solutions that can be re-usable.

However, the idea of repeated patterns in all the above solutions, including the STM versions, could be programmed on the computer or PLC itself. Once a pattern is created and loaded into the computer, it should be possible to generate the rest automatically. Surely, this way is more reliable than doing it manually. For instance, the Petri net in Figure 9 above could be generated automatically from Figure 8 by the computer.

6 RULE-BASED SOLUTION

We now examine how the rule-based technique works on this example of two engines on the loop track. Generally, in using rule-based technique one has to formulate the rules needed in an operation. More frequently than not, these rules come through the deliberations of questions and answers as to how the system operates. In this particular example, when the engine arrives at the end of any track section, say i, only one question is asked. Can it proceed to the next section, i+1? The answer to this is that it can only proceed if the track in front, i+2, is unpowered, otherwise the engine has to wait until the track is free. This rule is applied for every track section again and again for the system to operate successfully. This rule governs any number of track sections used in the system and works for any

number of engines on the track. Thus, system expansion has no effect to the implementation of this rule and only the subscripts of the track sections have to be taken care of during operation.

For clarity of presentation, we shall assume:

(1) (i) a main program to supply the switch number to the interrupt subroutine handling the rule
(2) (ii) infinite track length. The problem of 'wrap-around' will be dealt later.

On encountering the switch at the end of any track section, the rule is : IF (the track section after the next section ahead is not powered) THEN (power the next section and unpower the section behind) ELSE (wait for input switch at the end of the third section ahead). IF (this switch is activated) THEN (power the next section and unpower the section behind).

The implementation of the first part of the above rule is straightforward; the second part is better coped with by leaving a message, i.e. setting a flag to wait for an event. On collecting the message, the control proceeds with powering and unpowering appropriate sections.

```
Interrupt subroutine        RAIL1(N)
LOGICAL TRACK(M),WAIT(M),ON,OFF
rem M= number of track sections - infinite
ON=.TRUE.:OFF=.FALSE.
IF .NOT.TRACK(N+2)          THEN        TRACK(N+1)=ON:
                                        TRACK(N-1)=OFF:
                            ELSE        WAIT(N+3)=ON:
ENDIF
IF WAIT(N)                  THEN        WAIT(N)=OFF:
                                        TRACK(N-2)=ON:
                                        TRACK(N-4)=OFF:
ENDIF:RETI
```

N.B. : Track section ON means it is powered and OFF unpowered.

However, the above routine is built on the assumption that there are just two engines. The routine could be made more general by removing this assumption to give a routine which could easily cope with any number of engines.