warning doc title undefined

WARNING DOCUMENT TITLE UNDEFINED

By
FanFan Huang, B.Eng

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M.A.Sc
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2003) McMaster University
Hamilton, Ontario

TITLE: warning document title undefined

AUTHOR: FanFan Huang, B.Eng(McMaster University)

SUPERVISOR: Dr. Martin von Mohrenschildt

NUMBER OF PAGES: v, 6

Contents

List of Tables

List of Figures

Chapter 1

Introduction to Ladder Logic

1.1 Background of Ladder Logic

Ladder logic was originally developed to replace physical relays in PLC's. As a result the "language" resembles a circuit diagram. The left most and right most "rung" represent power rails analogous to GND and VCC what's placed in between those rungs is the load components/citeebookmorris. In the case of programming the entire logic is created from loads you place inside these power rails.

Several other conventions are also observed, power always flows from left to right along each rung. Power also flows from top to bottom along the rails. This is counter intuitive since ladder logic is suppose to be analogous to a circuit schematic and there is no implicit ordering in circuits. In addition each run must start with inputs and end with at least one output. Any device that is on a rung is shown in its initial position.

Modern PLC's operate more like a traditional micro controller and thus the original schematic based language can prove to be awkward to work with.

The inputs in ladder are referred to as a load and represented by the symbol $-|\ |-$. Loads are boolean values and can be a negated input using the $-|/|-$ symbol. In addition an address is usually assigned to each input referring to which port on the physical PLC the input is connected to. Logical and can be formed by having two

logical loads on one rung[**?**]. Similarily logical or can be formed by creating a branch along one rung as shown in figure.

We can define the language of Ladder Logic as follows $Q = \langle M, S, C, F, R, P \rangle$

- M: set of monitored variables.
- S: set of state variables.
- C: set of controlled outputs.i
- R: set of rungs.
- P: set of power rails.

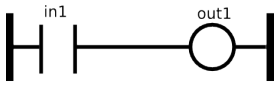

Figure 1.1: Basic Ladder Logic Diagram

The most basic structure of ladder logic is shown in figure 1.1. We have

$$M = \{in_1\}, S = \{out_1\}, C = \{out_1\}, R = \{rung_1\}, P = \{L, R\}.$$

Note that $rung_1$ is not explicitly labeled in ladder logic but we give it a name here so we may demonstrate our language framework.

The symantics of figure 1.1 is then:

Action	Result
$@T(in_1 = true)$	$out_1 := true$
$@T(in_1 = false)$	$out_1 := false$

Table 1.1: Semantics for Fig 1.1

Where $@T(< condition >)$ is used to denote the positive edge of a condition becoming true. We also assume negligable delay between the action occuring and the result being asserted. It is important to note that our function table must be complete that is have an entry for all possible combinations in the input domain.

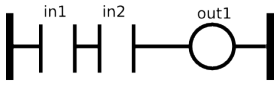

Figure 1.2: Simple AND Logic Diagram

When multiple actions are connected on the same rung it is interpreted as a logical AND expression. In figure 1.2 we can expand our model to:

$$M = \{in_1, in_2\}, S = \{out_1\}, C = \{out_1\}, R = \{rung_1\}, P = \{L, R\}.$$

We can see that both in_1 and in_2 are on $rung_1$. This is interpreted as follows:

Action	Result
$@T(in_1 = true \wedge in_2 = true)$	$out_1 := true$
$@T(in_1 = false \vee in_2 = false)$	$out_1 := false$

Table 1.2: Semantics for Fig 1.2

The conditions in_1 and in_2 are combined to form our composed action $@T(in_1 = true \wedge in_2 = true)$ as seen in table 1.2. We also note that there is no action for each individual condition becoming true nor do we need to individually calculate the timing on in_1 or in_2 individually.

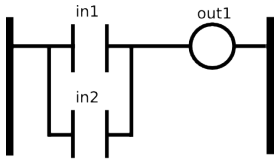

Figure 1.3: Branching Rungs

In addition to mutiple actions connected to the same rung, actions can also be branched. A branched rung as show in figure 1.3 behaves like a logical OR. In addition two or more rungs can be joined as show in figure 1.4. The semantics are equivalent in both figure 1.3 and figure 1.4.

Since the semantics are the same for both figure 1.3 and figure 1.4 we can express both outcomes with function table 1.3. As before in table 1.2 in table 1.3 in_1 and in_2

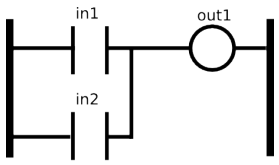

Figure 1.4: Branching Rungs (Alternative)

Action	Result
$@T(in_1 = true \vee in_2 = true)$	$out_1 := true$
$@T(in_1 = false \wedge in_2 = false)$	$out_1 := false$

Table 1.3: Semantics for Fig 1.3 and Fig 1.4

are composed to form our composed action $@T(in_1 = true \vee in_2 = true)$. However it is also possible to represent this action another way.

Action	Result
$@T(in_1 = true)$	$out_1 := true$
$@T(in_2 = true)$	$out_1 := true$
$@T(in_1 = false \wedge in_2 = false)$	$out_1 := false$

Table 1.4: Semantics for Fig 1.3 and Fig 1.4

In table 1.4 we choose to represent in_1 and in_2 as seperate actions. This matches figure 1.4 more closely but also makes any verification harder than table 1.3. For smaller examples table 1.3 make more sense since you can verify relatively simple smaller functions quite fast. If a system becomes resonably large however there might be motivation to use the style shown in table 1.4 since it will allow more complex functions to be decomposed into simpler actions. Since semantically the two are equivalent this paper will focus to the first convention.

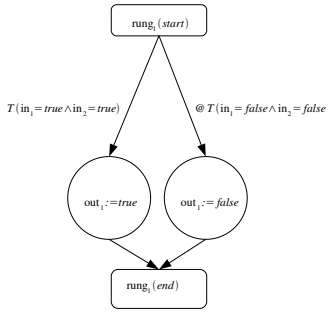

Figure 1.5: State chart conversion for table 1.2

A rung can be defined as a directed acyclic graph with exactly one source and one sink. The state variables form guard conditions along the edges. A branch in this case represents 2 edges leaving one node. For example figure 1.2 can be easily converted into a state chart by observing the results in table 1.2. Each row of table 1.2 is directly converted into an edge with the appropriate guard conditions. Each output assertion is given their own state. Finally in figure 1.5 we observe 1 source for the graph being the start state of the rung, and one sink for the graph being the end state for the rung. We can equivalently take figure 1.3 observe its function table 1.3 and produce an equivalent state chart from its function table.

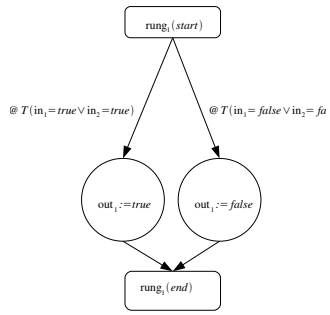

Figure 1.6: State chart conversion for table 1.3

Thus, each rung can be converted into an equivalent state chart by examining its function table, assigning the state variables to guard conditions, and creating a state

for each of the outputs. For completeness we complete this procedure to produce a state chart equivalent representation for figure 1.4 and its corresponding table 1.4.

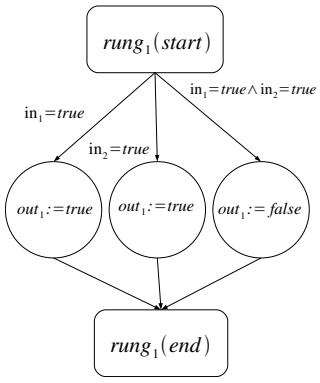

Figure 1.7: State chart conversion for table 1.4

So far we've been looking at logic that has things happening instantaneously however ladder logic does contain methods for expressing more advance behavior. Suppose you have a light that you want to be able to turn on and stay on after you push a button. With any of the ladder diagrams shown above the light would go out as soon as the buttons were all released. In order to keep the light on and constantly on after the button is pressed we introduce the concept of a latch. We modify our diagram shown in figure 1.4 making the output feed back into one of the inputs to our OR circuit. The result is show in figure 1.8.

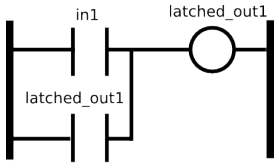

Figure 1.8: Latched ladder logic circuit.

The latched circuit operates the same way as the behavior show in 1.4 the key difference is the feedback of the OR circuit replaces the second row of the function table.