UNIVERSITY OF CALIFORNIA, SAN DIEGO

AUTOMATED TESTING TOOL FOR PLC BASED INDUSTRIAL APPLICATIONS

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science

by

Feng Zhang

Committee in charge:

Professor Bill Howden, Chair
Professor Rajesh Gupta
Professor Ingolf Krueger

2011

The Thesis of Feng Zhang is approved and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

Chair

University of California, San Diego

2011

DEDICATION

To my wife, Yinghui, who gave me unconditional support throughout the course of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Bill Howden for advising my thesis and for his support as the chair of my committee.

I would also like to acknowledge Professor Rajesh Gupta and Professor Ingolf Krueger for their supports as the members of my committee.

ABSTRACT OF THE THESIS

AUTOMATED TESTING TOOL FOR PLC BASED INDUSTRIAL APPLICATIONS

by

Feng Zhang

Master of Science in Computer Science

University of California, San Diego, 2011

Professor Bill Howden, Chair

PLCs (Programmable Logic Controllers) are the work horse of industrial control systems. Industrial control systems are usually both mission-critical and safety-critical systems. A single software bug in a control system program could cause hardware equipment damage and human life loss. The PLC programs in control systems must provide bug-free and failure-free behaviors in order to avoid accidents. A PLC program must be completely tested for correctness in functionality, reliability, predictability, and safety before it's released for production systems. At a PLC program development stage,

hardware devices in the industrial control system are usually not available for testing the PLC program for safety reasons. The often used solution is to use a simulator to simulate the hardware devices' behaviors. The simulator is usually built as a hardware test stand which consists of toggle switches, lamp indicators, and analog signal generators. The shortcoming of this kind of simulators is they are not automated and require lots of user interactions. As a result, they cannot guarantee the accuracy of behaviors of the hardware devices being simulated.

This research presents an automated testing tool which automates the hardware device simulation process by using "virtual" wires. The hardware device simulation is part of a test case which is defined in the presented testing tool and downloaded to the PLC controller. This testing tool requires no user interaction during a test run so it reduces the testing cost and time and it can precisely simulate the behaviors of hardware devices.

Chapter 1. Introduction

1.1 PLC in Control Systems

PLC stands for Programmable Logic Controller. A PLC is essentially an embedded system which consists of a programmable microcontroller and a variable number of I/O modules. An I/O module has a variable number of I/O channels which are wired to the hardware devices in the field. A PLC is designed as a hard real-time system that monitors the inputs (which are wired to sensors like level sensor, pressure sensor, pH sensor, or temperature sensor) and controls output actuators such as pumps, valves, generators, or boat locks in real-time. Figure 1.1 shows a typical PLC hardware configuration. PLC controllers are usually RISC (reduced instruction set computer) microprocessor. PLC was invented to replace hardware relay logic circuits. When it was first invented in 1960s, a PLC could only be programmed by a specialized program called ladder logic which is similar to a schematic of relay logic. Now the modern PLCs support multiple languages defined by IEC61131 standard (which is an international standard for PLC programming languages). The five languages defined in IEC61131-3 standard are Ladder Diagram (LD), Instruction List (IL), Structured Text (ST), Function Block Diagram (FBD), and Sequential Function Charts (SFC). Allen Bradley's Logix PLCs support all languages but IL. IL is mostly used in European PLCs like Siemens PLC. IL is the European counterpart of LD. A PLC program consists of multiple routines and each routine can be programmed in any language supported by the PLC. This allows the PLC programmer to choose the language that is best suited for each individual task. Only one program can be downloaded to a PLC. A PLC executes its program repeatedly. Each

execution iteration is called a scan cycle. The time consumed for a PLC scan cycle is very fast, in the units of milliseconds. Figure 1.2 shows the major steps of a PLC scan cycle. At the Scan Input step, the PLC reads the real-time values of the sensors that are connected to the PLC input modules and records the values in memory. At the Execute Program step, the PLC executes the program synchronously (from left to right and top to bottom) based on the real-time input values recorded in memory. At the Update Outputs step, the PLC updates the outputs (to the actuators) based on the results of executing the program in the current scan cycle. The I/O values are updated asynchronously.

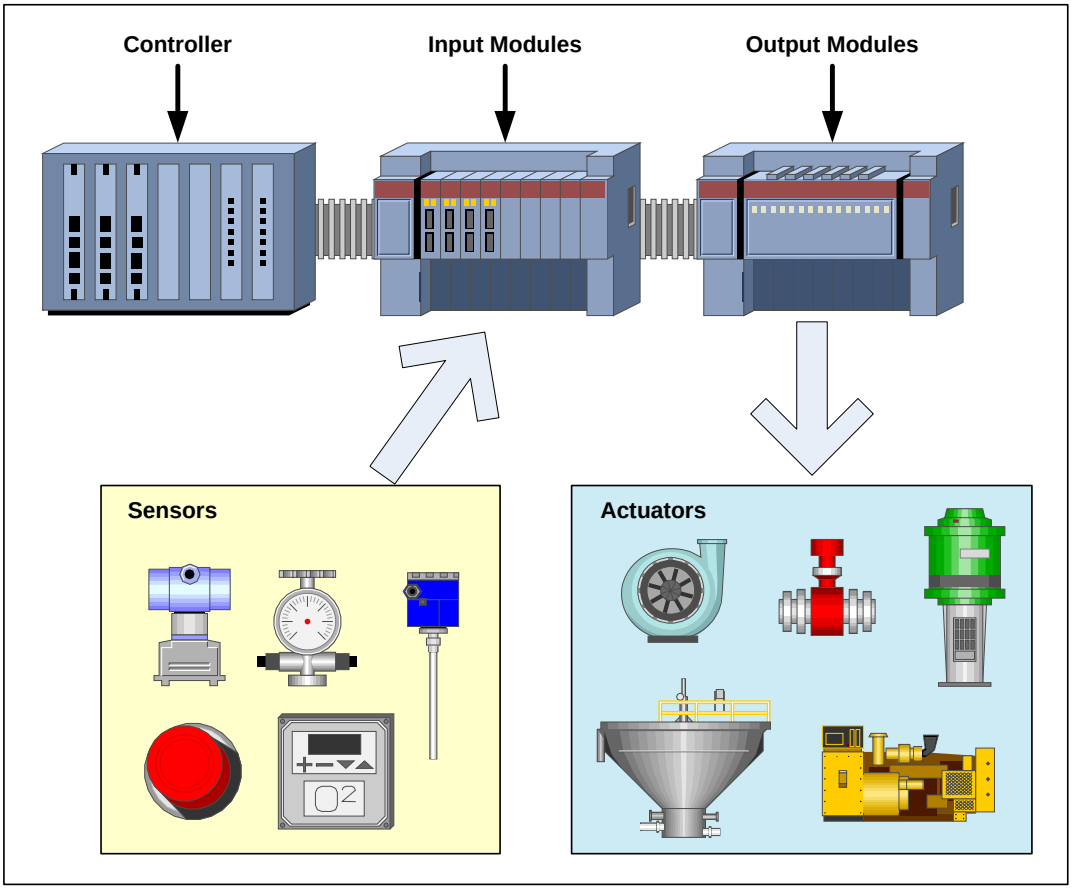


Figure 1.1: Typical PLC configuration.

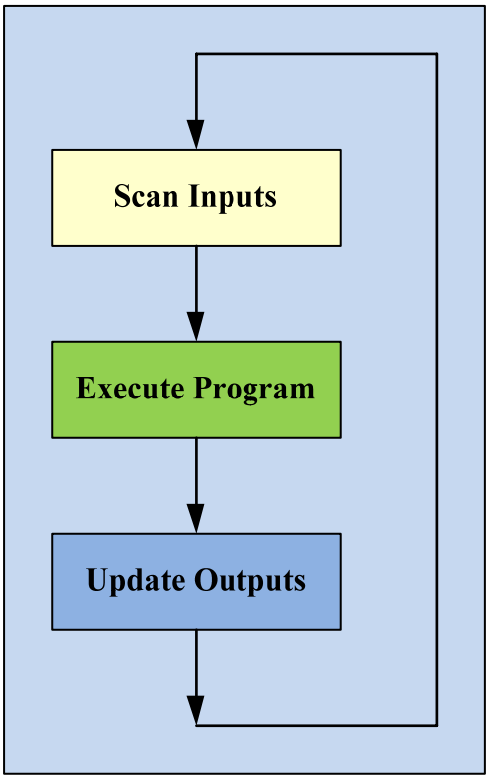


Figure 1.2: PLC scan cycle.

PLCs have made major contributions to industrial automation and they are widely used in the industrial automation control systems. As mentioned earlier, PLCs were originally invented to replace the hardware relays. But as the PLC technology advanced, PLCs provide more and more functionality and play more important roles in industrial automation control systems. Today's PLCs provide faster scan cycle time by using high performance CPU chips, drive high-density I/O systems including wireless I/Os, support more popular industrial communication protocols such as DNP3.0, Modbus, Ethernet/IP, and Profibus, provide redundancy for CPU and I/O, and support various I/O signal types such as discrete, analog (current/voltage), RTD (Resistance Temperature Detector), high speed pulse, and SOE (Sequence of Events) [1]. All these new PLC features allow them to accommodate more complex applications. PLCs are widely used as the reliable plant-

level controllers and deliver a wide range of functionality for relay control, motion control, process control in Distributed Control Systems (DCS) and Supervisory Control and Data Acquisition (SCADA) systems in various of industries including water/waste water, power, automotive, packing, food, and pharmaceutical.

1.2 Software Testing in Control Systems

Industrial control systems are usually mission-critical and safety-critical real-time systems. A real-time system must guarantee both logical correctness and temporal correctness. A PLC program should be designed and developed as a hard real-time application that must guarantee the response time to handle all events even in the worst case scenarios. If a signal process is delayed by a PLC then a system failure could occur. For example, a PLC is used in a power generation system to control a power generator. The PLC must verify that the generator's frequency, voltage and phase angle are synchronized with the power grid before it can output a command to close the main breaker to connect the generator to the power grid. If the PLC closes the main breaker when the frequency, voltage, or power phase angle is not synchronized between the generator and the power grid the hardware equipment such as circuit breakers could be damaged. Nevertheless, if the PLC has a one-second delay for outputting the main breaker close command after it confirms the generator and the grid are synchronized it will also cause the same problem due to a temporal failure.

Software Testing is a very important stage in the software development life cycle in general-purpose systems. It's even more important in industrial control systems. In software development for an industrial control system application, you want to find as

many bugs as possible before you release the application program for the control system. As mentioned before, control systems are usually mission-critical and safety-critical systems. A bug in a production control system could damage the controlled hardware devices and could even cause human life loss. So testing is extremely important and critical in control systems vs. in general-purpose systems.

A program in a control system is essentially a black box which takes some inputs and generates some outputs based on the inputs. Inside the black box it's the control logic. It also can be put in this way, a control program answers "what-if" questions. If the program covers all possible operation scenarios it most likely will answer all the "what-if" questions correctly. However this is a tough task. Often times, when a control system goes into an unknown condition or an unexpected condition the control program may not know how to react. It just sits there and does nothing instead of commanding the system to a safe state. This is very dangerous in safety-critical system. If the control program fails to handle a control scenario an accident could happen and that could cause hardware device damages and even human life loss. A control program in a safety-critical system must provide failure-free behavior by considering all possible operation scenarios. A simple example is a control program ignores the quality bit of an analog I/O point from a sensor. In this case, the control program will continue to use the I/O value of the analog point even after its quality bit became bad (due to a broken sensor). The control program will use the wrong reading for its control logic and eventually will make incorrect control decisions. For example, in a pump control scenario, the pump starts to fill water into a tank when the tank level is low and stops to fill water when the tank level is high. While the pump is running to fill water, the tank level sensor becomes defective. As a result the

tank level reading stops updating. But the actual tank level is still rising. Since the control program doesn't check the quality bit of the tank level signal, it will continue to actuate the pump (instead of stopping the pump and generate a sensor failure alarm). Eventually the tank will overflow. This is a real example that shows why a control program should consider all possible operation scenarios and all possible device failures (broken sensors, jamming actuators) at the software development stage. The program must be tested against expected failures as well as unexpected failures at the testing stage in order to prove the correctness of handling system failures. Ensuring high reliability of a PLC program is critical in industrial control systems.

Chapter 2. PLC Programming Languages

The Allen Bradley CompactLogix PLC and ControlLogix PLC support four languages defined in the IEC61131-3 standard and they are Ladder Diagram (LD) [2], Function Block Diagram (FBD) [3], Sequential Function Charts (SFC) [4], and Structured Text (ST) [5].

2.1 Ladder Logic

Ladder Logic is the most often used language in PLC programming because it was the first and only supported language when PLC was invented; it's made to mimic the existing relay logic wiring schematics; it's easy to debug. Using Ladder Logic reduces the training needs for electricians, technicians and engineers to learn how to program a PLC because they're already familiar with the style of the Ladder Logic diagram. Figure 2.1 shows a hardware switch circuit diagram and a PLC Ladder Logic diagram for the same lamp control circuit.

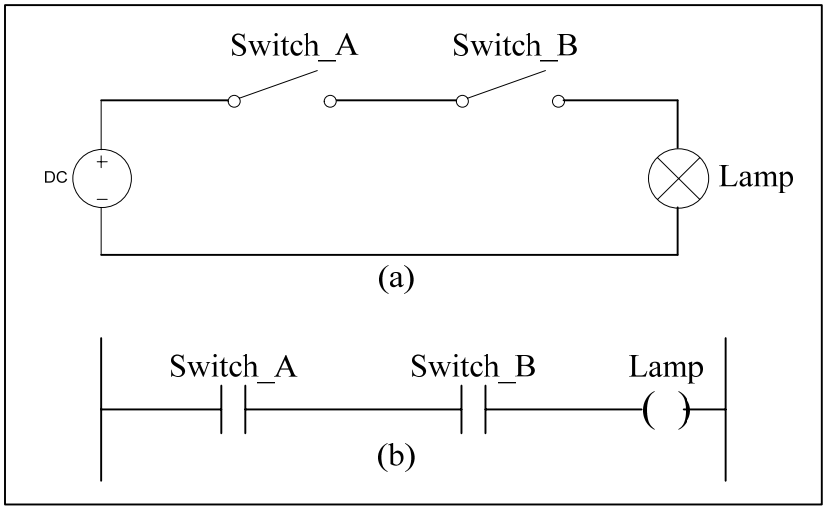


Figure 2.1: (a) Hardware switch circuit diagram. (b) PLC Ladder Logic diagram.

2.2 Function Block Diagram (FBD)

Function Block Diagram (FBD) is a graphical programming language. The basic element of FBD is called a block. A block describes a function between input variables and output variables. A set of function blocks can be connected together (like an electrical circuit) to form a new function. Inputs and outputs of blocks are connected by connection lines. A function block encapsulates its implementation and it makes it possible to develop modular programs and reuse them from one PLC project to another. Figure 2.2 shows a Function Block Diagram program that buffers the value of a digital input module and maps the buffered value to an internal pump structure.

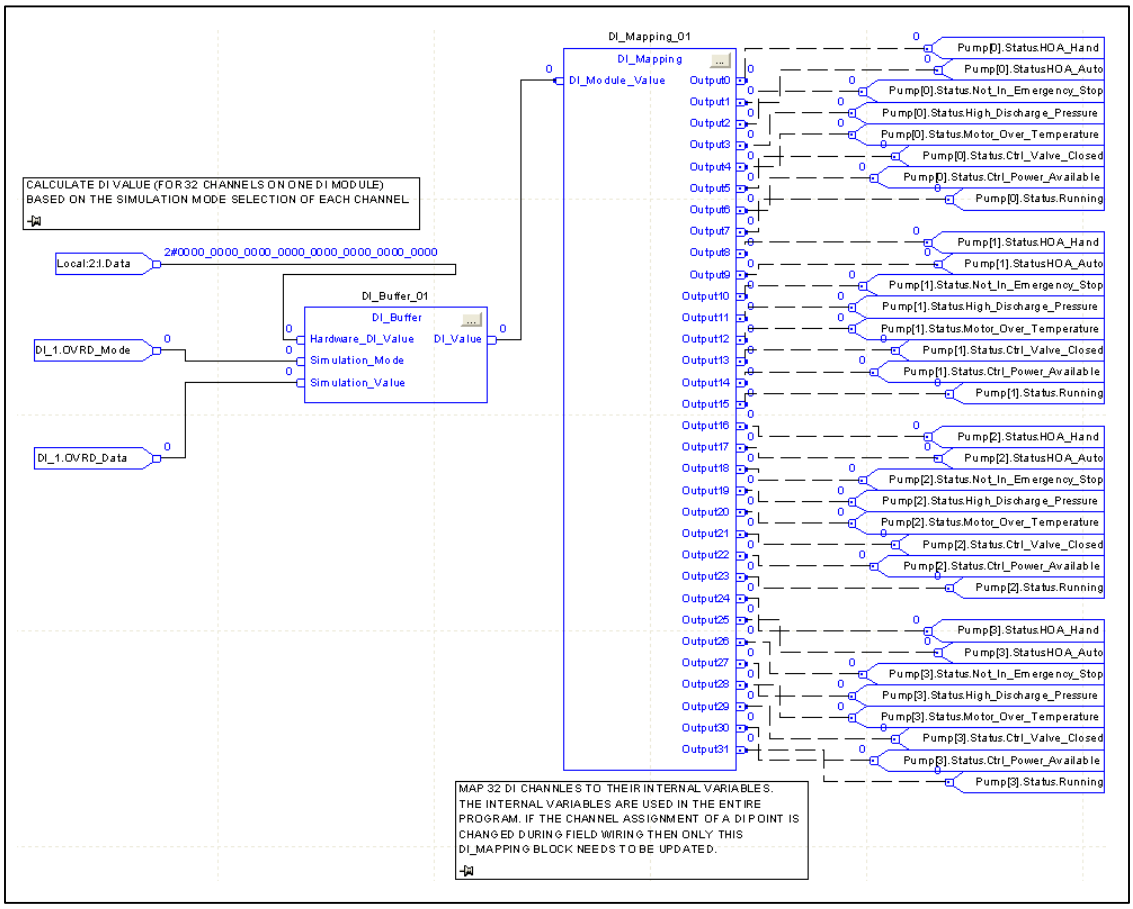


Figure 2.2: A sample Function Block Diagram program.

2.3 Sequential Function Charts (SFC)

Sequential Function Charts (SFC) is another graphical language supported by Allen Bradley Logix PLCs. It's designed for complex sequential process controls. The basic elements in SFC are step, action, and transition. A step defines a major function of the control process. It contains the actions that execute at this step. An action is one of the functions that a step performs. A transition is a condition that is checked before the SFC can go to the next step. Sequential Function Charts are similar to flowcharts and they are the industrial implementation of Petri Net.

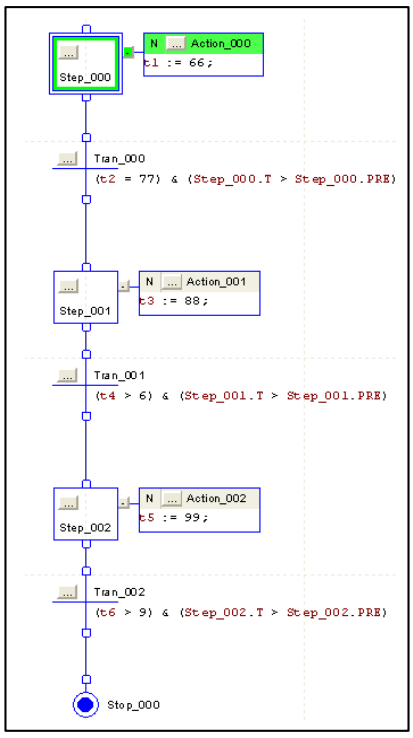


Figure 2.3: A sample Sequential Function Chart program.

Figure 2.3 shows a 3-step SFC program. Each step has a single action. A transition is defined between two consecutive steps to determine when the process control can move to the next step. SFC simplifies the logic for a complex sequential control by a

graphical representation. With SFC It becomes very easy to design and troubleshoot a PLC program. You can easily see what the program is doing and locate the logic where the problem occurs during a process control sequence. SFC is self-documented.

2.4 Structured Text (ST)

Structured Text (ST) is a high-level procedural language that is similar to the Basic language or the Pascal language. Structured Text is very useful when developing complex functions in the PLC such as complicated math calculations or algorithms. People who are familiar with high-level programming languages will feel comfortable to program a PLC in Structured Text. Structured Text programs can be created and edited in any Text Editor. Most of the modern PLCs support Structured Text language. However, debugging a Structured Text program can be very hard because there is no any debugger available for Structured Text in the PLC program development environment. Therefore, Structured Text is not an ideal language for developing process control logics because in Structured Text you cannot track which stage the program is currently running at. Figure 2.4 shows a Structured Text program that calculates pump runtimes.

```
FOR PRT_Index := 0 TO 4 BY 1 DO
  // Start of a new day
  IF PLC_DateTime.Hour = 0 THEN
    Pump_Runtime[PRT_Index].Temp_Bit := 1;
  ELSE
    Pump_Runtime[PRT_Index].Temp_Bit := 0;
  END_IF;
    IF Pump_Runtime[PRT_Index].Temp_Bit AND (NOT Pump_Runtime[PRT_Index].One_Shot_Bit_1) THEN
    Pump_Runtime[PRT_Index].Previous_Day_Runtime := Pump_Runtime[PRT_Index].Current_Day_Runtime;
    Pump_Runtime[PRT_Index].Current_Day_Runtime := 0;

    Pump_Runtime[PRT_Index].Previous_Day_Pump_Starts := Pump_Runtime[PRT_Index].Current_Day_Pump_Starts;
    Pump_Runtime[PRT_Index].Current_Day_Pump_Starts := 0;
  END_IF;
  Pump_Runtime[PRT_Index].One_Shot_Bit_1 := Pump_Runtime[PRT_Index].Temp_Bit;

  // Start of a new week
  IF DOW.DAY_OF_WEEK = 1 THEN
    Pump_Runtime[PRT_Index].Temp_Bit := 1;
  ELSE
    Pump_Runtime[PRT_Index].Temp_Bit := 0;
  END_IF;
  IF Pump_Runtime[PRT_Index].Temp_Bit AND (NOT Pump_Runtime[PRT_Index].One_Shot_Bit_2) THEN
    Pump_Runtime[PRT_Index].Current_Week_Pump_Starts := 0;
  END_IF;
  Pump_Runtime[PRT_Index].One_Shot_Bit_2 := Pump_Runtime[PRT_Index].Temp_Bit;

  // Start of a new month
  IF PLC_DateTime.Day = 1 THEN
    Pump_Runtime[PRT_Index].Temp_Bit := 1;
  ELSE
    Pump_Runtime[PRT_Index].Temp_Bit := 0;
  END_IF;

  IF Pump_Runtime[PRT_Index].Temp_Bit AND (NOT Pump_Runtime[PRT_Index].One_Shot_Bit_3) THEN
    Pump_Runtime[PRT_Index].Current_Month_Runtime := 0;
    Pump_Runtime[PRT_Index].Current_Month_Pump_Starts := 0;
  END_IF;
  Pump_Runtime[PRT_Index].One_Shot_Bit_3 := Pump_Runtime[PRT_Index].Temp_Bit;

  IF Pump_Runtime[PRT_Index].Temp_Bit AND (NOT Pump_Runtime[PRT_Index].One_Shot_Bit_4) THEN
    Pump_Runtime[PRT_Index].Current_Year_Runtime := 0;
  END_IF;
  Pump_Runtime[PRT_Index].One_Shot_Bit_4 := Pump_Runtime[PRT_Index].Temp_Bit;

  IF Sixty_Second_TMR.DN AND (NOT Pump_Runtime[PRT_Index].One_Shot_Bit_5) THEN
    IF Pump_Runtime[PRT_Index].Running THEN
      Pump_Runtime[PRT_Index].Current_Day_Runtime := Pump_Runtime[PRT_Index].Current_Day_Runtime + 1/60;
      Pump_Runtime[PRT_Index].Current_Month_Runtime := Pump_Runtime[PRT_Index].Current_Month_Runtime + 1/60;
      Pump_Runtime[PRT_Index].Current_Year_Runtime := Pump_Runtime[PRT_Index].Current_Year_Runtime + 1/60;
      Pump_Runtime[PRT_Index].Accumulative_Runtime := Pump_Runtime[PRT_Index].Accumulative_Runtime + 1/60;
    END_IF;
  END_IF;
  Pump_Runtime[PRT_Index].One_Shot_Bit_5 := Sixty_Second_TMR.DN;

  IF Pump_Runtime[PRT_Index].Running AND (NOT Pump_Runtime[PRT_Index].One_Shot_Bit_6) THEN
    Pump_Runtime[PRT_Index].Current_Day_Pump_Starts := Pump_Runtime[PRT_Index].Current_Day_Pump_Starts + 1;
    Pump_Runtime[PRT_Index].Current_Week_Pump_Starts := Pump_Runtime[PRT_Index].Current_Week_Pump_Starts + 1;
    Pump_Runtime[PRT_Index].Current_Month_Pump_Starts := Pump_Runtime[PRT_Index].Current_Month_Pump_Starts + 1;
  END_IF;
  Pump_Runtime[PRT_Index].One_Shot_Bit_6 := Pump_Runtime[PRT_Index].Running;
END_FOR;

COP(Pump_Runtime[2], Stormwater_Pump_Runtime[0], 3);
```

Figure 2.4: A sample Structured Text program.

Multiple programming languages can be used in the same PLC program. In order to choose an appropriate language, many factors need to be evaluated such as the programmers' skill, the complexity, modularity, and structure of the programming task,

the type of control logic of the application, who will troubleshoot and maintain the program, and how often the program needs to be modified. A right choice of the programming language will shorten both development time and troubleshooting time and deliver more efficient and reliable programs.

An Allen Bradley Logix PLC controller is a preemptive, multitasking controller. A single PLC project runs in a PLC controller, which supports multiple tasks. Each task supports multiple programs and each program supports multiple routines. Logix PLC supports three types of tasks: Continuous, Periodic, and Event. A continuous task has the lowest priority so it can be interrupted by a periodic task or an event task. Figure 2.5 shows an example of scheduling three tasks. T1 and T2 are periodic tasks and T1 has a higher priority than T2. T1 runs every 10ms and it takes 2ms to run. T2 runs every 7 seconds and it takes 4ms to run. T3 is a continuous task.

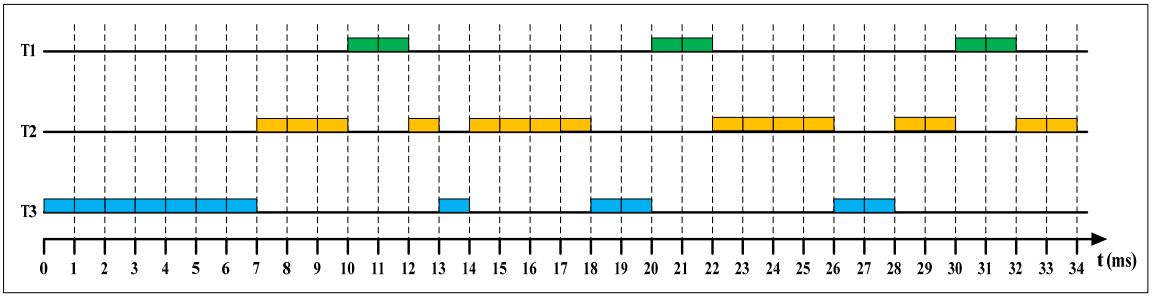


Figure 2.5: PLC project task scheduling example.

Chapter 3. Review of PLC Testing Methods

3.1 Hardware Test Stand

Since PLC programs control massive hardware devices such as pumps, valves, generators in the field, it's almost impossible to physically move the hardware devices from the field to the test environment. The hardware devices are usually very expensive and they may be still working in the production systems, it's impossible to test the PLC programs in such production systems. Therefore, a PLC I/O simulation system is required to help develop the PLC program during the development stage and test the PLC program in the testing stage. A hardware Test Stand is often used to simulate the I/Os of hardware devices in the field. Figure 3.1 shows a picture of a typical Hardware Test Stand.

Figure 3.1: Typical hardware Test Stand.

The Test Stand is made by many LED indicators, toggle switches, analog signal generators, and meters that are wired to the PLC's input and output modules. The toggle switches and the analog signal generators are used to simulate the sensor inputs and the LED indicators and the meters are used to indicate the PLC outputs. The Test Stand has the following disadvantages and limitations:

1. It’s expensive and limited to the number of I/O channels it’s designed for
2. It requires the physical presence of all the PLC I/O modules because the signals from the Test Stand will need to be wired to the PLC’s I/O modules.
3. It requires user interaction during a test run. In a complex process control, it’s difficult to simulate multiple signals in a certain sequence or to simulate multiple signals simultaneously.
4. It’s a manual simulation and cannot conduct automatic simulation.

3.2 HMI Based Simulator

Another approach that is often used for simulating the PLC I/Os is a HMI (Human Machine Interface) based software simulation. In this approach, a HMI control screen is developed and used for the PLC I/O simulation. The HMI control screen can be made as a SCADA system screen or a standalone Operator Terminal screen. Figure 3.2 shows an Allen Bradley Operator Terminal (called PanelView [6]) screen. Since this is a software simulator it eliminates the requirement of the PLC I/O modules. However, it is still a manual simulation and it’s time-consuming to make the I/O simulation screens themselves. A HMI based simulator can be used to test all functions of a PLC program. But the timing between a command and the response for the command is not automated. It requires user interaction to click a button on the screen to trigger each I/O simulation. So it cannot be used for automatic control sequence simulation and the accuracy of a test depends on the user’s actions too.

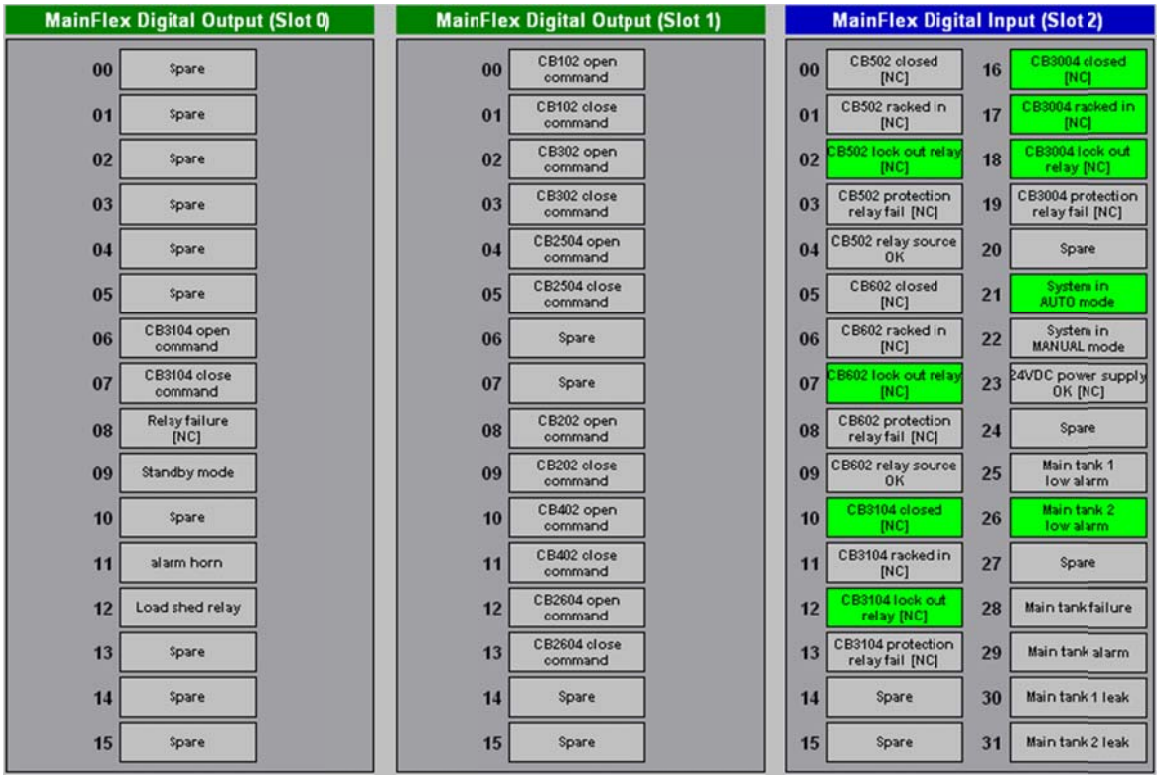


Figure 3.2: Allen Bradley PanelView I/O simulation screen.

3.3 New Approach: "Virtual" Simulator

This research proposes an automated testing tool that can simulate the PLC I/O signals via "virtual" wires and automate the test execution. There are no physical hard wires connected between LogixPlcTester and the PLC being tested. This tool helps PLC software developers and testers test PLC programs during the entire software development stage and the testing stage. It can be used for both unit testing and system testing. The goal of this tool is to assure quality of PLC programs and to deliver reliable PLC programs for industrial control systems.

Chapter 4. LogixPlcTester

To overcome the shortcomings of the hardware Test Stand and the HMI based simulator, this research presents a Windows based application called LogixPlcTester that can precisely simulate behaviors of hardware devices in the field. LogixPlcTester can automatically read the outputs of the PLC and simulate inputs to the PLC by sending values directly to the input memory of the PLC controller based on inputs' trigger events (either time-based event or condition-based event). Unlike the other simulation methods, LogixPlcTester doesn't require the presence of any PLC I/O module in order to simulate signals. LogixPlcTester is connected to the PLC that is being tested through "virtual" wires. LogixPlcTester doesn't require user interaction in order to run the simulations. LogixPlcTester has the following main features:

- Define test cases offline.
- Run test cases against the PLC program.
- Simulate incidents occurring in the production system.
- Simulate "odd ball" cases that cannot be tested in the production system.
- Verify bug fixes before applying the production system.
- Good for both unit testing and system testing.
- Can be also used as a good training system that can demonstrate how the system works in every scenario.
- A good troubleshooting tool for debugging PLC programs when it's set to monitor-only mode.

4.1 Design Diagram

In a typical PLC-based industrial control system, all the field hardware devices' feedback signals are wired to the PLC's input modules, and the PLC control output signals (sent from the PLC's output modules) are wired to the actuators of the field hardware devices. The PLC knows the status of the field hardware devices by reading the input signals and the PLC control program runs based on the input signals and generates output signals to control the field hardware devices. The diagram at the left hand side of Figure 4.1 shows a typical industrial PLC control system. The diagram at the right hand side of Figure 4.1 shows the concept diagram of the automated testing tool presented in this research. Instead of connecting the real devices to the PLC using real wires, the presented testing tool acts as a virtual device or virtual devices and connects to the PLC through virtual wires. LogixPlcTester is the name of the automated testing tool application and is programmed with C# language in Visual Studio 2005. LogixPlcTester simulates the field hardware devices' signals by sending the configured signals in the test case as sequence of events (time triggered simulation). LogixPlcTester can also read the PLC control output signals and based on the PLC output signals it can send the configured signals to the PLC (condition triggered simulation). The diagram at the bottom of Figure 4.1 shows the system diagram at the network level. LogixPlcTester communicates with the PLCs using Ethernet/IP protocol [7]. Ethernet/IP stands for Ethernet Industrial Protocol which was originally developed by Rockwell automation (the vendor of Allen Bradley PLCs). Ethernet/IP is an application layer protocol. Ethernet/IP uses all the transport and control protocols of standard Ethernet including Transport

Control Protocol (TCP), the User Datagram Protocol (UDP), the Internet Protocol (IP) and the media access and signaling technologies. Ethernet/IP protocol is transferred in a TCP/IP packet. Ethernet/IP uses an open application layer protocol called Common Industrial Protocol (CIP).

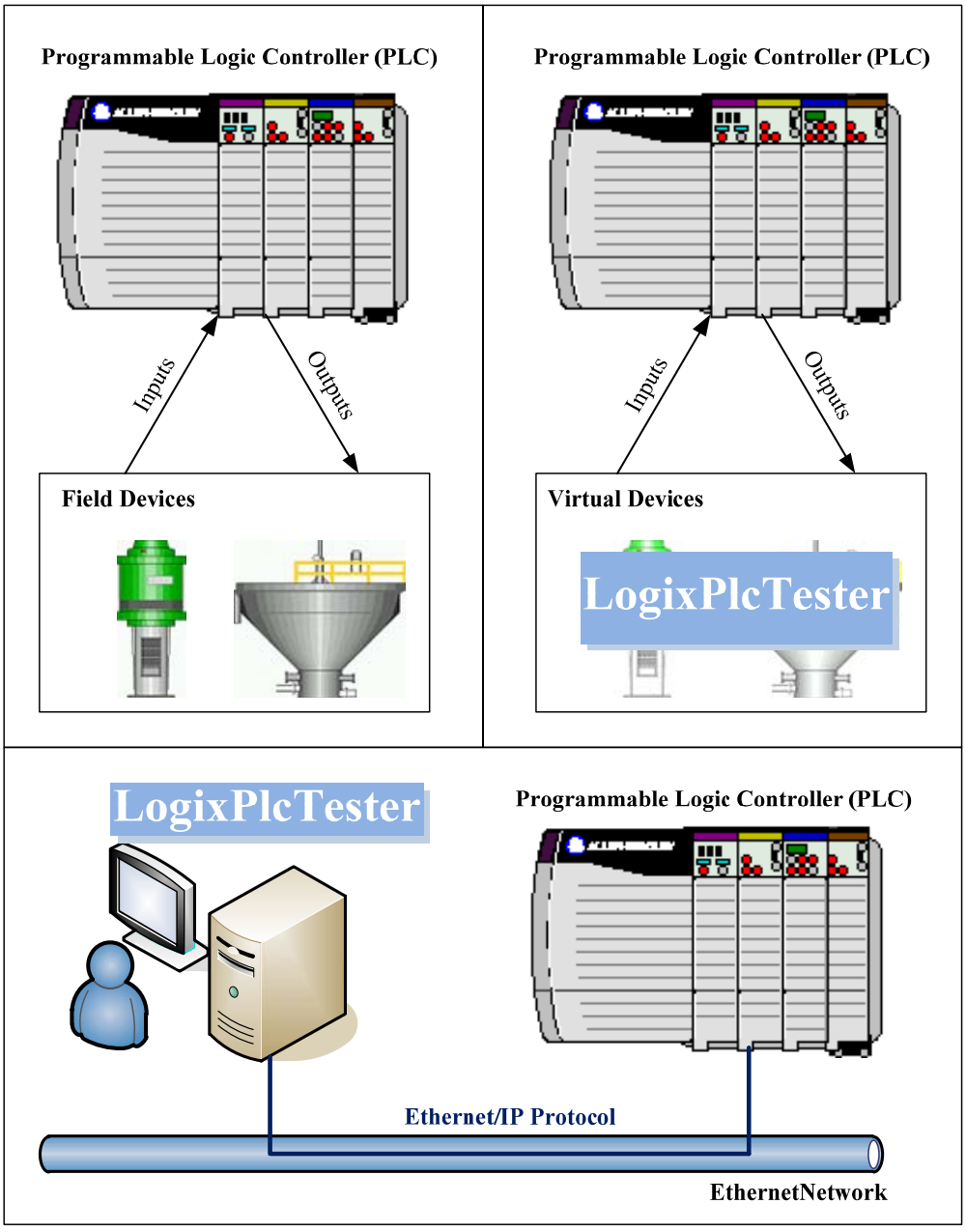


Figure 4.1: LogixPlcTester design concept diagram.

As shown in Figure 4.1, Ethernet/IP protocol has been implemented in the PLC controller's firmware, which allows direct read from and direct write to the PLC's input and output memory. LogixPlcTester uses the corresponding packets of Ethernet/IP to read data from the PLC and write data to the PLC.

4.2 Inhibit PLC I/O Modules

In order for LogixPlcTester to work correctly, no code change is required in the PLC program that is being tested. However, a PLC I/O module configuration change (in the PLC project file) may be required in order to disable the communications between the PLC controller and its I/O modules. During testing, since LogixPlcTester will simulate values for all the PLC input modules by directly updating the PLC's input memory, we need to make sure that the PLC controller won't update its input memory with the hardware values read from the input modules. We can disable the communication between the PLC controller and an I/O module by inhibiting the module. Figure 4.2 shows the I/O modules at slot 2 to 15 are inhibited in a PLC project. The communications between the PLC controller and the output modules can be left enabled as far as LogixPlcTester is concerned. However, we want to disable them as well from the safety standpoint. For example, if the PLC controller updates its output memory during a test it will energize the relay outputs based on the program execution results and that may accidently actuate the field hardware devices if they were wired to the PLC that is being tested and that may cause some unexpected incidents and damages to the hardware devices. The module inhibition step can be ignored if an I/O module is not physically present in the PLC rack because the PLC controller doesn't update the I/O memory for a

module if it cannot communicate with the module. Because the module inhibition is a configuration change (not a code change) it won't reduce the level of truthfulness of the tests in LogixPlcTester. After all the tests are complete, before installing the program in the production system, the Inhibit Module checkboxes must be unchecked in the PLC project in order to allow the PLC controller to update its I/O memory with the real-time hardware values while running in the production system.

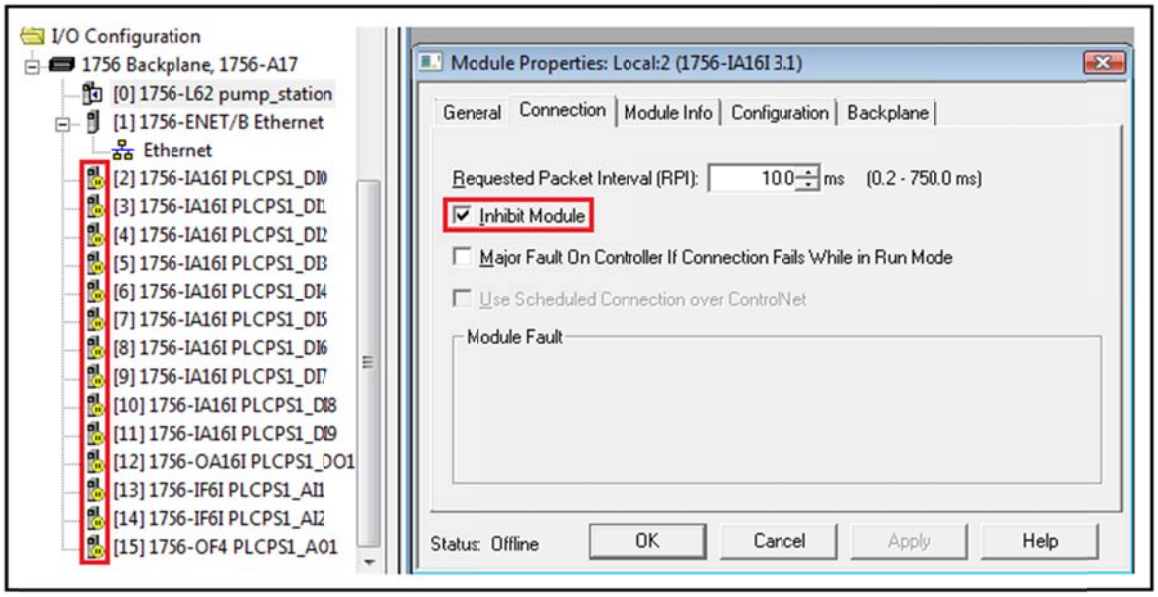


Figure 4.2: Inhibit PLC I/O modules.

4.3 Application Overview

LogixPlcTester is a Windows .NET application developed with C# language in Microsoft Visual Studio 2005. LogixPlcTester creates customized test cases in an Extensible Markup Language (XML) format. Test case XML files can be shared among QA engineers. LogixPlcTester displays a test case in a tree view structure. Users can edit a test case either online or offline. LogixPlcTester displays real-time values of PLC tags defined in a test case and it supports online changes on the fly while a test case is running. A log view feature is supported to log alarms, tag value change notifications, system operation messages online in a spreadsheet format. The logs can be exported from LogixPlcTester to a Comma-Separated Values (CSV) file for further review. Once initiated, a test case runs in LogixPlcTester automatically based on the time triggers and condition triggers in the test case. Figure 4.3 shows the main application window of LogixPlcTester.

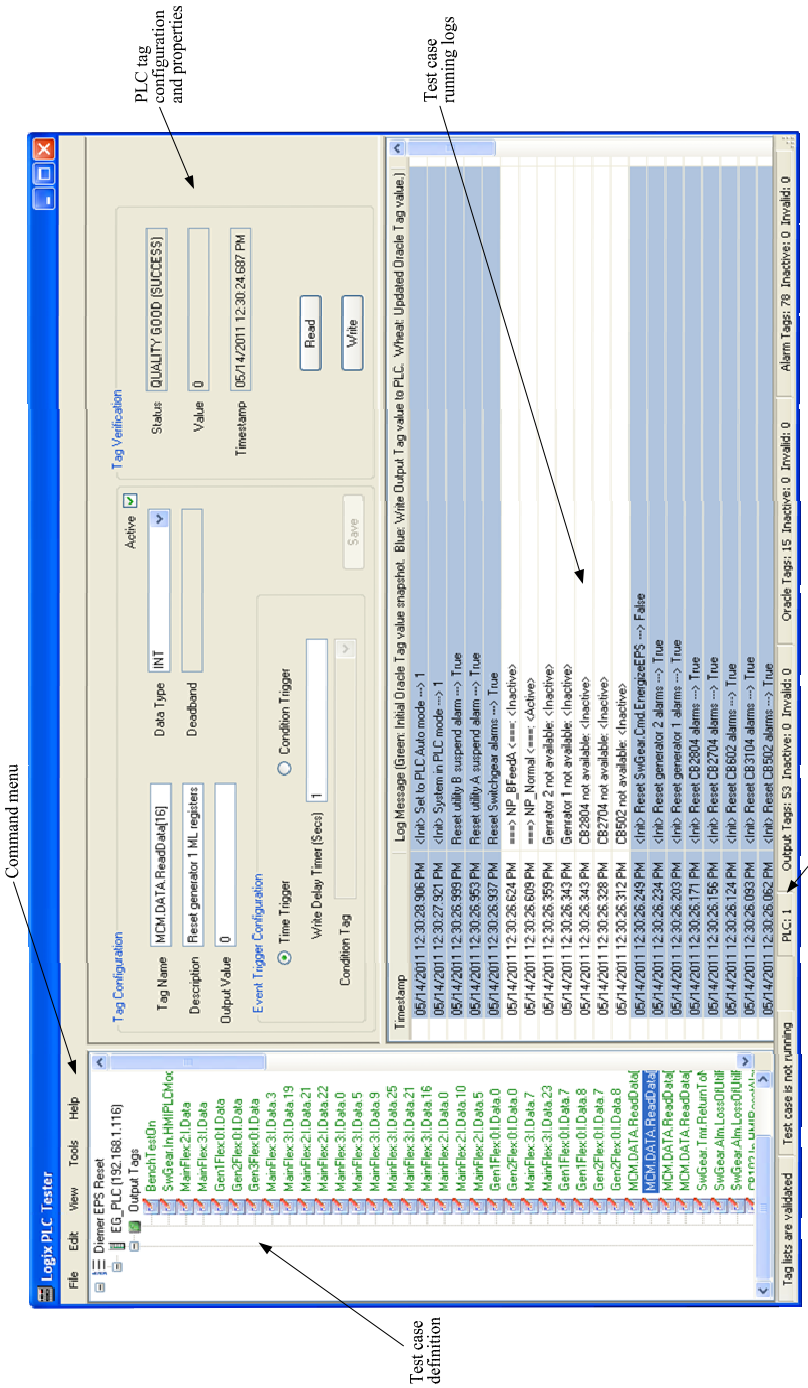


Figure 4.3: LogixPlcTester application overview.

4.4 Test Case Structure

In LogixPlcTester, a test case supports multiple PLCs and each PLC consists of three PLC tag lists that belong to different tag categories. Figure 4.4 shows a typical structure of a test case in LogixPlcTester.

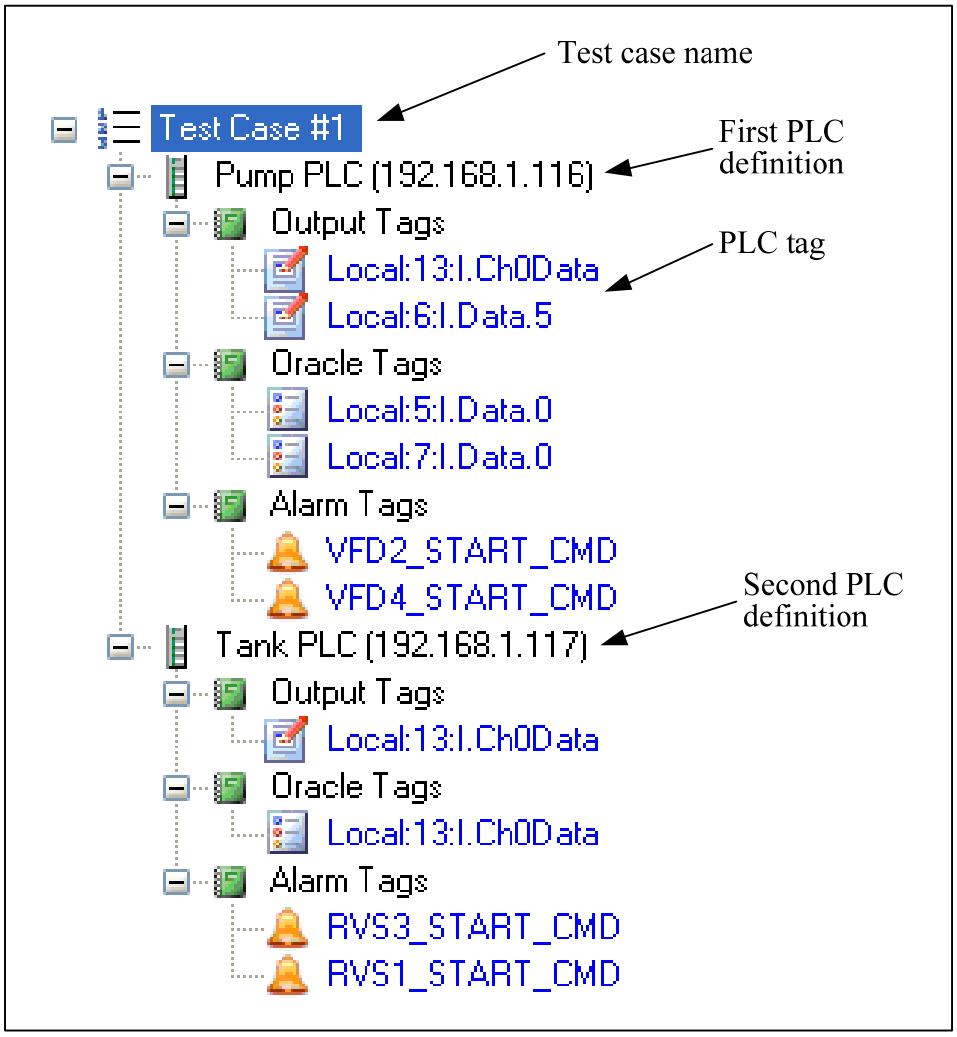


Figure 4.4: A typical test case structure.

In Figure 4.4, the test case that is called “Test Case #1” defined two PLCs (Pump PLC and Tank PLC). Each PLC has Output Tags, Oracle Tags, and Alarm Tags.

An Output Tag is used to send a value to a tag in the PLC based on a time trigger or a condition trigger while the test case is running. Output tags are responsible for simulating field hardware devices by sending configured data to the PLC's I/O channels in real-time. An Oracle Tag is used to monitor the value of a PLC tag. Oracle tags read values in a report-by-exception basis. Only changed values in the PLC are reported to LogixPlcTester. The report-by-exception mechanism dramatically reduces the network communication traffic between LogixPlcTester and the PLCs that are being tested. An Alarm Tag is used to define a system operation alarm or to define an event trigger. For example, you can use an Alarm Tag for a pump overload alarm. You can use an Alarm Tag to monitor a pump run command sent by the PLC and configure an Output Tag that uses the Alarm tag as a condition trigger to simulate a pump running signal once the pump run command is detected by LogixPlcTester.

A tag name in LogixPlcTester must be identical to a tag name in the PLC. The definition of a LogixPlcTester tag name is referred to the same tag name in the PLC.

4.5 Create Test Case

There are two ways to create a test case. One is to manually create an XML file (using the elements and attributes definitions listed in Table 4.1) in any Editor and another is to create it in LogixPlcTester.

As mentioned above, a test case definition is stored in a XML file. Table 4.1 shows the element and attributes that are used to define a test case. Figure 4.5 shows the hierarchical structure of a test case XML file in Microsoft XML Notepad 2007 that is a free XML editor.

Table 4.1: Test case XML elements and attributes.

Type	Name	Description
Element	Test_Case	Test case element
Attribute (of Test_Case)	Name	Test case name
Attribute (of Test_Case)	Duration	Test case execution time (in seconds)
Element	PLC	PLC element
Attribute (of PLC)	Name	PLC name
Attribute (of PLC)	IP_Address	PLC IP address
Element	Output_Tag	Output tag element
Element	Oracle_Tag	Oracle tag element
Element	Alarm_Tag	Alarm tag element
Element	Tag	Tag element
Attribute (of Tag)	Name	PLC tag name (it must resides in PLC)
Attribute (of Tag)	Description	PLC tag description (It should match the tag description in PLC)
Attribute (of Tag)	Active	Enable/disable tag
Attribute (of Tag)	Data_Type	Data type of PLC tag name. See Table 4.2 for details
Attribute (of Tag)	Write_Delay_Timer	Time trigger (in seconds): time to elapse before the tag value is sent to PLC
Attribute (of Tag)	Output_Value	Value to write to PLC tag
Attribute (of Tag)	Event_Trigger	An alarm tag that is used as event trigger for a PLC tag
Attribute (of Tag)	Deadband	Deadband for oracle tag. Tag value change within the deadband won't be logged in LogixPlcTester
Attribute (of Tag)	Enabled_Delay_Timer	Event trigger is set to true after the condition has been active for the defined time (in seconds)
Attribute (of Tag)	Alarm_Expression	Alarm or event trigger expression
Attribute (of Tag)	Log_Option	Log as event (for event trigger) or log as alarm

Table 4.2: Supported PLC data type.

Data Type	Description	Memory Bits	Range
BOOL	Boolean	1	0 or 1
SINT	Short Integer	8	0
INT	Integer	16	-128 to 127
DINT	Double Integer	32	-2,147,483,648 to +2,147,483,647
REAL	Floating Point	32	+/-3.402823E38 to +/-1.1754944E-38

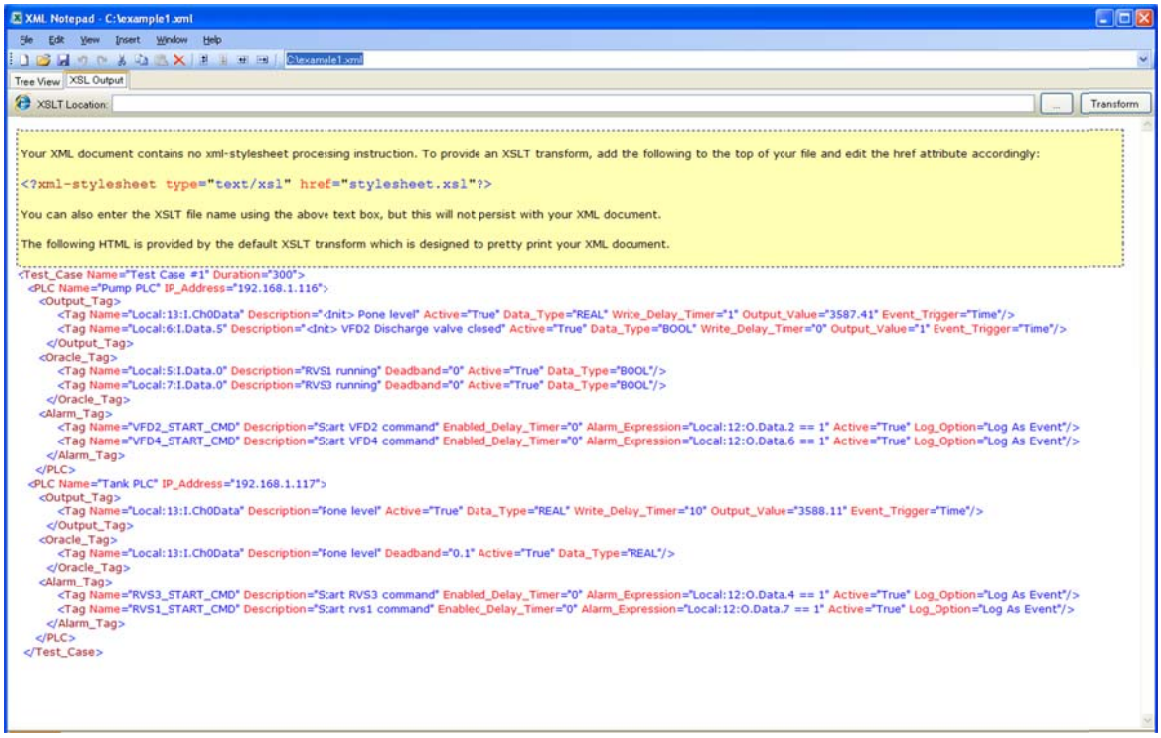


Figure 4.5: A test case XML file in Microsoft XML Notepad 2007.

The following basic steps show how to create a new test case in LogixPlcTester:

1. Click **File > New… > Test Case** as in Figure 4.6. The **New Test Case** dialog appears as shown in Figure 4.7.
2. In the **New Test Case** dialog, type in the test case name and the execution duration (in seconds) of the test case then click **OK**. The execution duration specifies how long the test case will run.
3. Click **File > New… > PLC** as shown in Figure 4.8. The **New PLC** dialog appears as shown in Figure 4.9.
4. In the **New PLC** dialog, type in the PLC name and the PLC IP address then click **OK**.

5. A test case with one PLC is created. Under the PLC, there are three empty tag lists that are created automatically as shown in Figure 4.10.

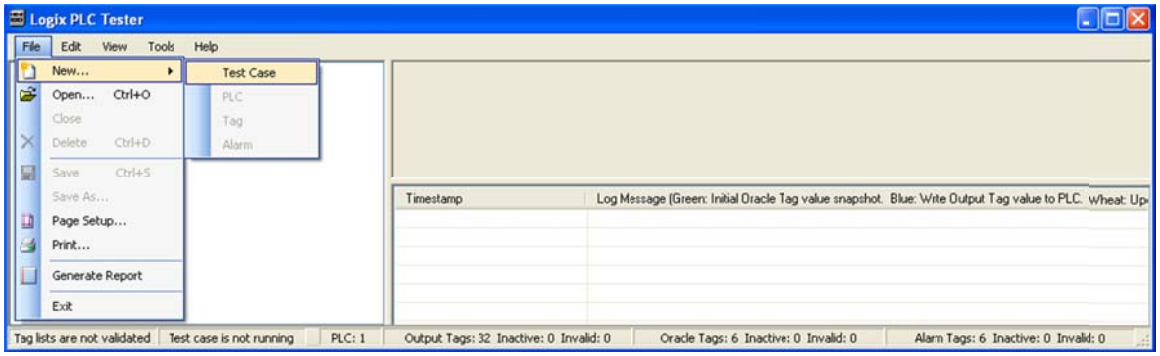


Figure 4.6: Create a new test case in LogixPlcTester.

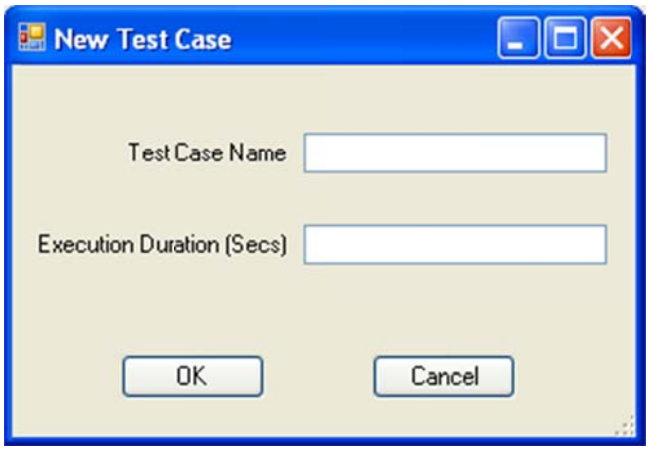


Figure 4.7: New Test case dialog.

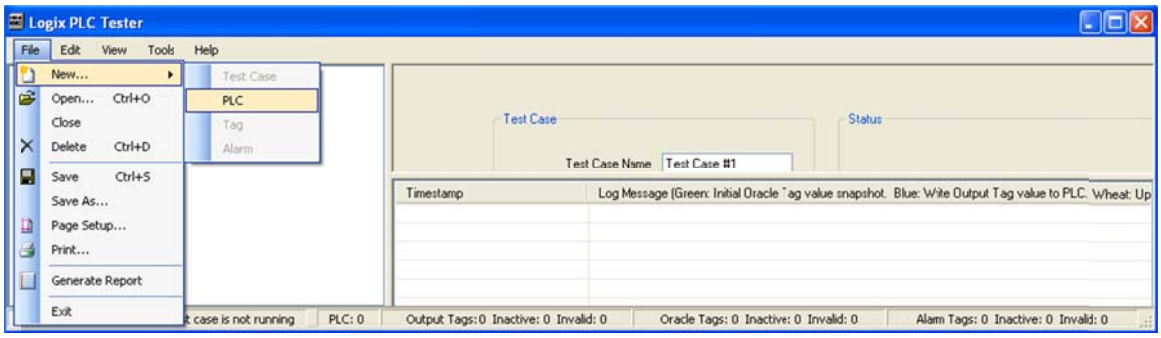


Figure 4.8: Define a new PLC in LogixPlcTester.

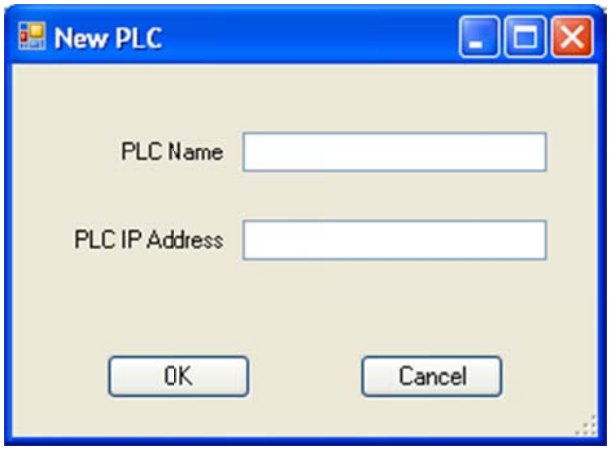


Figure 4.9: New PLC dialog.

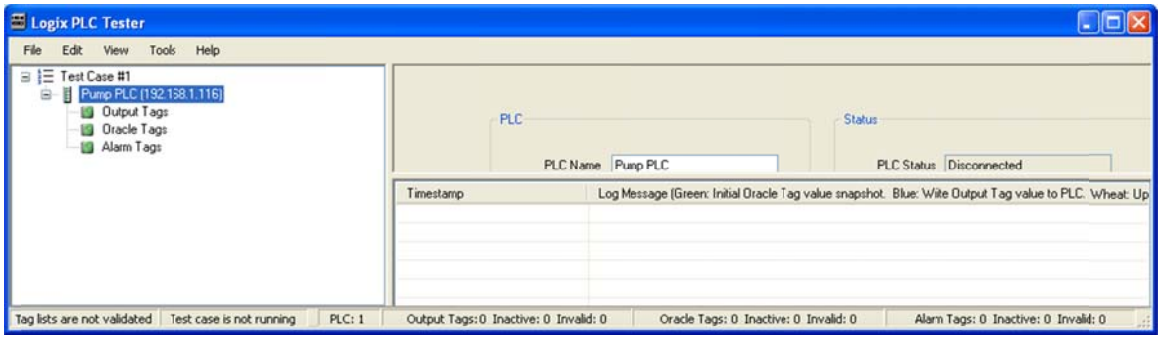


Figure 4.10: A PLC with three empty tag lists in LogixPlcTester.

4.6 Create an Output Tag

Output Tags can be added to the Output Tag list. Right click on **Output Tags** then click **Add** as shown in Figure 4.11. The **New Output Tag** dialog appears as shown in Figure 4.12 (for time triggered Output Tag). When the Condition Trigger is selected, the **New Output Tag** dialog changes its appearance as shown in Figure 4.13. When Time Trigger is selected, the output value will be written to the tag in the PLC as soon as the **Write Delay Time** has elapsed after the test case is initiated. For example, if 30 is specified as the **Write Delay Time,** then the value will be written to the PLC tag after the test case has been running for 30 seconds. When **Condition Trigger** is selected, the

output value will be written to the tag in the PLC after the **Condition Tag** has become active for the amount of time defined in the **Condition On_Delay Time** parameter.

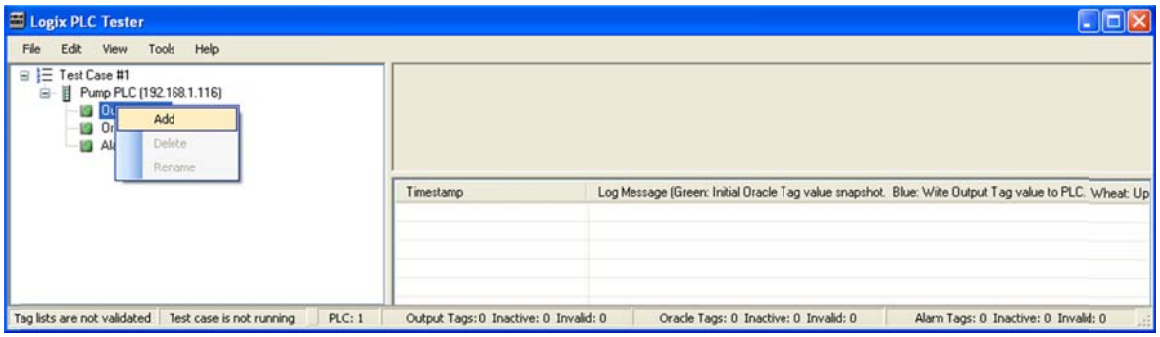


Figure 4.11: Create a new output tag in LogixPlcTester.

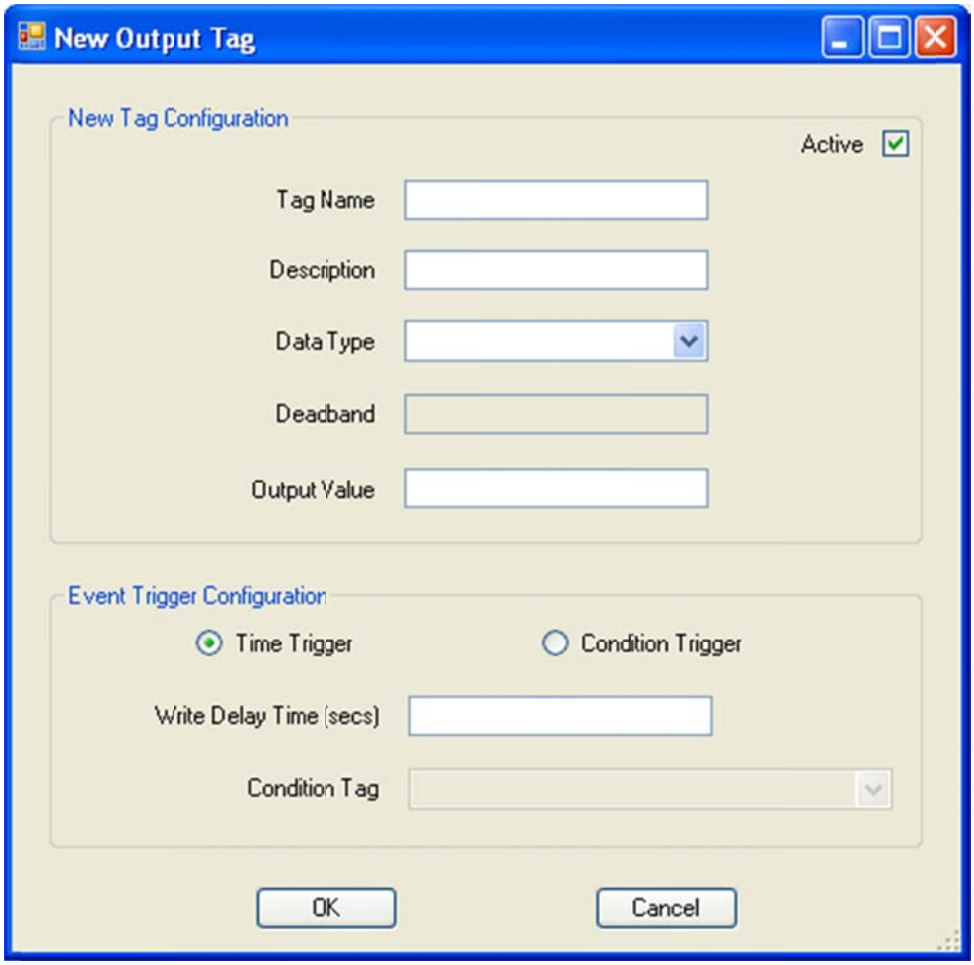


Figure 4.12: New Output Tag dialog (for Time Trigger).

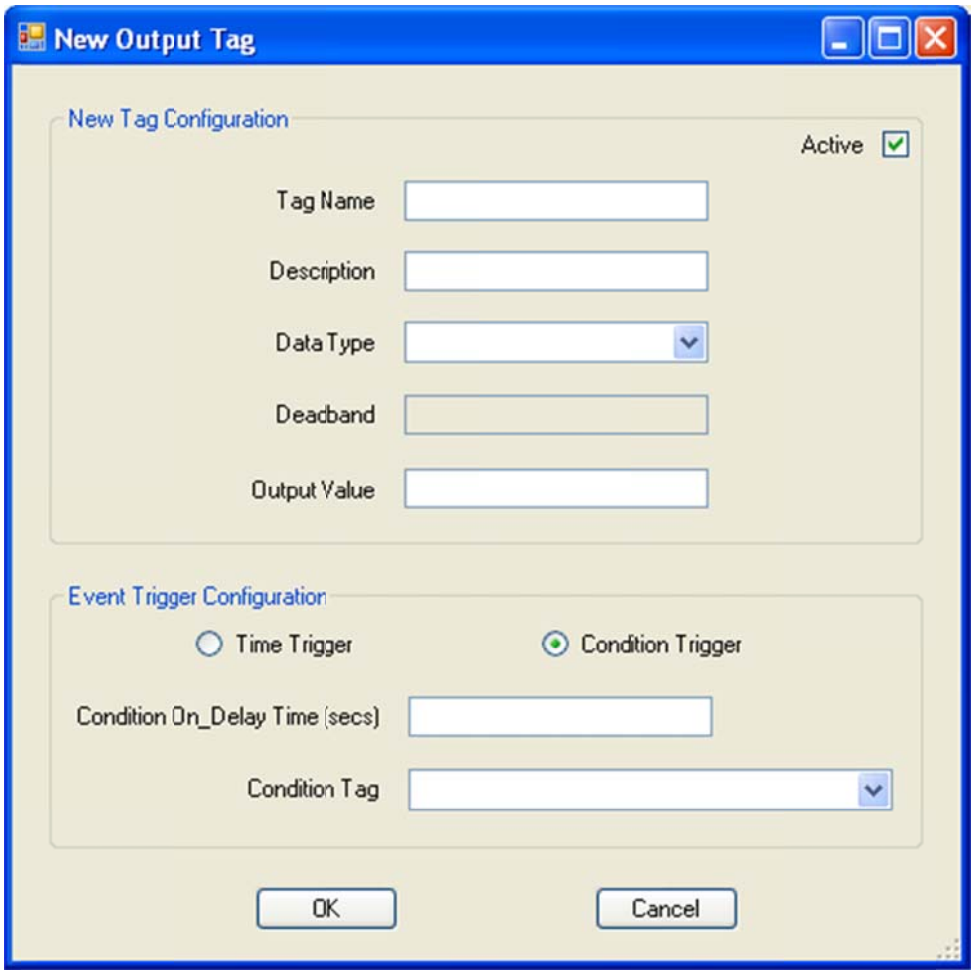


Figure 4.13: New Output Tag dialog (for Condition Trigger).

4.7 Create an Oracle Tag

Oracle tags can be added to the Oracle Tag list. Right click on **Oracle Tags** then click **Add**. The **New Oracle Tag** dialog appears as shown in Figure 4.14. Oracle Tags are used to monitor PLC tags. For example, you can create oracle tags to monitor pump running status, tank level, or device health status**.** Deadband is used to eliminate similar readings of a tag so only the values outside of the deadband will be reported and logged in LogixPlcTester. For example, if the deadband of a pipeline pressure is set to 1 PSI,

then only values that are at least 1 PSI higher than (or less than) the current reading will be seen in LogixPlcTester.

Figure 4.14: New Oracle Tag dialog.

4.8 Create an Alarm Tag

Alarm tags can be added to the Alarm Tag list. Right click on **Alarm Tags** then click **Add**. The **New Alarm Tag** dialog appears as shown in Figure 4.15. Alarm Tags are used to define alarms or Condition Triggers for Output Tags. A lexical analyzer and a math parser have been implemented in the application to parse the alarm expression. The

parser evaluates an alarm expression and returns the value of the alarm expression. Table 4.3 shows the math and logical operators supported in an Alarm Expression.

Example 1: To detect if a pump running.

Pump_Running == 1

Pump_Running is a Boolean tag in the PLC.

Example 2: To detect if there is a pressure is greater than 80 PSI while a pump is running.

Pump_Running == 1 && Pressure > 80

Pressure is a floating point tag in the PLC.

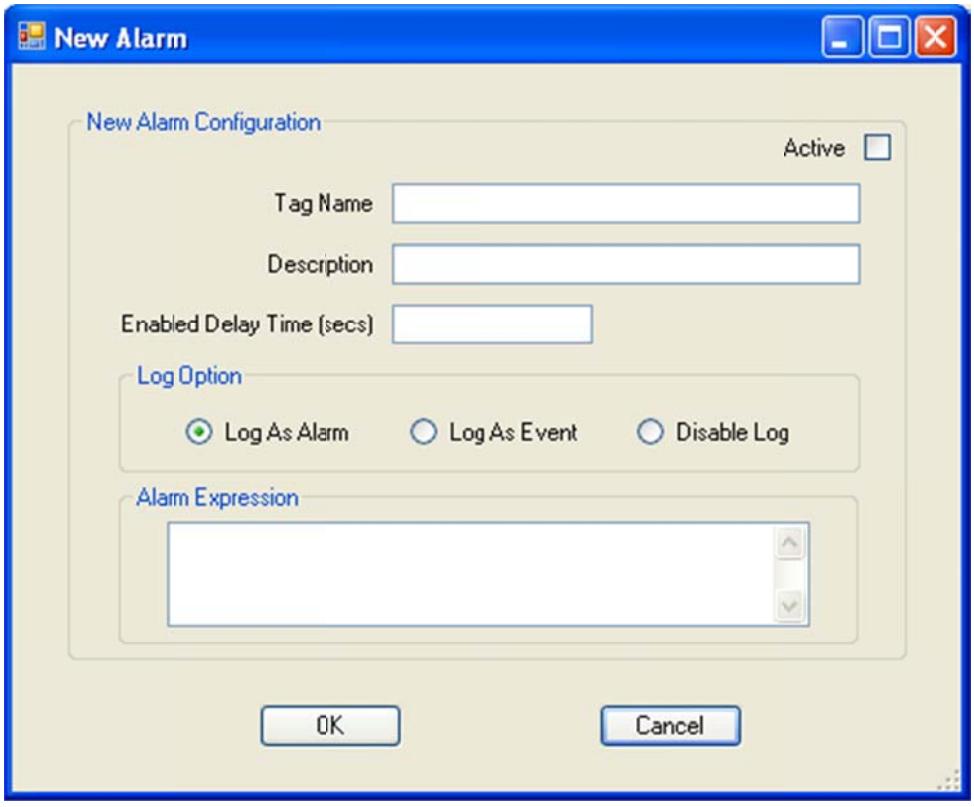


Figure 4.15: New Alarm Tag dialog.

Table 4.3: Supported operators for Alarm Expression.

Math Operators	Logical Operators
+, -, *, /, ^, %	!, ==, !=, ||, &&, >, < , >=, <=

4.9 Online Edit

Once a test case is loaded in LogixPlcTester, it can be edited online. Figure 4.16 shows a Tag Details Window in which you can make change to the tag's configuration. While a test case is running, you still can make tag configuration changes. The test case running process can automatically pick up the changes made on the fly and they will take effect immediately for the rest of the test case run process. You can click the Read button to read the real-time value of the selected tag in the PLC. Clicking the Write button (for Output Tag only) will write the configured Output Value to the selected tag in the PLC. The Read and Write buttons are helpful when doing a step-by-step testing. They can also be used for troubleshooting the PLC program. For instance, you can create an Output Tag for a pump reset command and then you can click the Write button to reset the pump alarms. You can create an Oracle Tag for the tank level tag in the PLC and then you can click the Read button to read the real-time value of the tank level in the PLC when you need it during the troubleshooting process. You can create an Alarm Tag to monitor a pump control output command in the PLC when debugging a pump control problem.

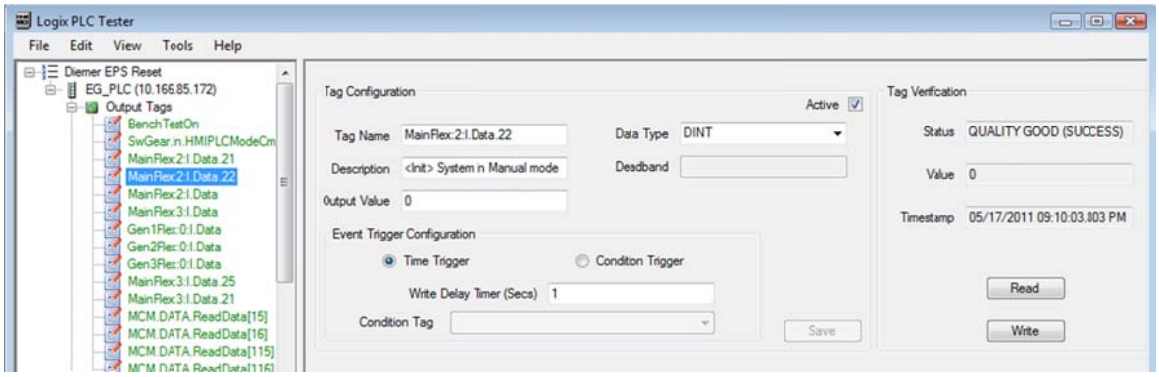


Figure 4.16: Tag Details Window in LogixPlcTester.

4.10 LogixPlcTester as an Operator Console / a Logger

When there are only tags in the Oracle Tag list and/or the Alarm Tag list, LogixPlcTester will turn into an Operator Terminal which is similar to a control system control console (a HMI terminal). In this case, LogixPlcTester will only read Oracle Tags and/or Alarm Tags from the PLC and won't execute any write operations. It can be connected to the production control system to perform a monitoring function. You can create your customized configuration in LogixPlcTester for a specific purpose. For example, if you want to monitor a hydro-electric generator startup sequence and log the entire sequence for further study or analysis, then you can create a test case in LogixPlcTester with only Oracle Tags and Alarm tags associated with the generator startup sequence. The test case shall be started before the hydro-electric generator startup sequence is initiated in the production system. Once the generator startup sequence is complete you can export the LogixPlcTester log messages to a flat file for future reference. Compared to a formal control console or a formal logger, LogixPlcTester is more cheaper, flexible, customized and quicker to setup.

4.10 LogixPlcTester as a Training System

Besides being an efficient tool for testing and troubleshooting purpose, LogixPlcTester can also be used as a training system. A training system basically needs to mimic all the activities that the production system has. In LogicPlcTester, a test case can simulate one or more system behaviors. By running a test case, the trainees will learn how the system works under the scenario that the test case presents. If the test case describes a system failure scenario, the trainees will learn what to do under that circumstance by watching the simulated actions by LogixPlcTester. This is an easy, efficient and safe way to conduct the system operation training without interfering with the production system. The scenario based test cases can be run over and over again without wearing out any hardware devices or causing any damage to hardware devices.

4.11 Data flow Path

Figure 4.17 shows the data flows between LogixPlcTester and the PLC that is being tested. LogixPlcTester simulates field hardware devices' signals by writing values into the PLC using Output Tags. LogixPlcTester reads PLC output commands and PLC internal tags using Oracle Tags. Alarm Tags are similar to Oracle Tags and they're used to generate alarms in LogixPlcTester. Alarm Tags can also be used as Condition Triggers for Output Tags.

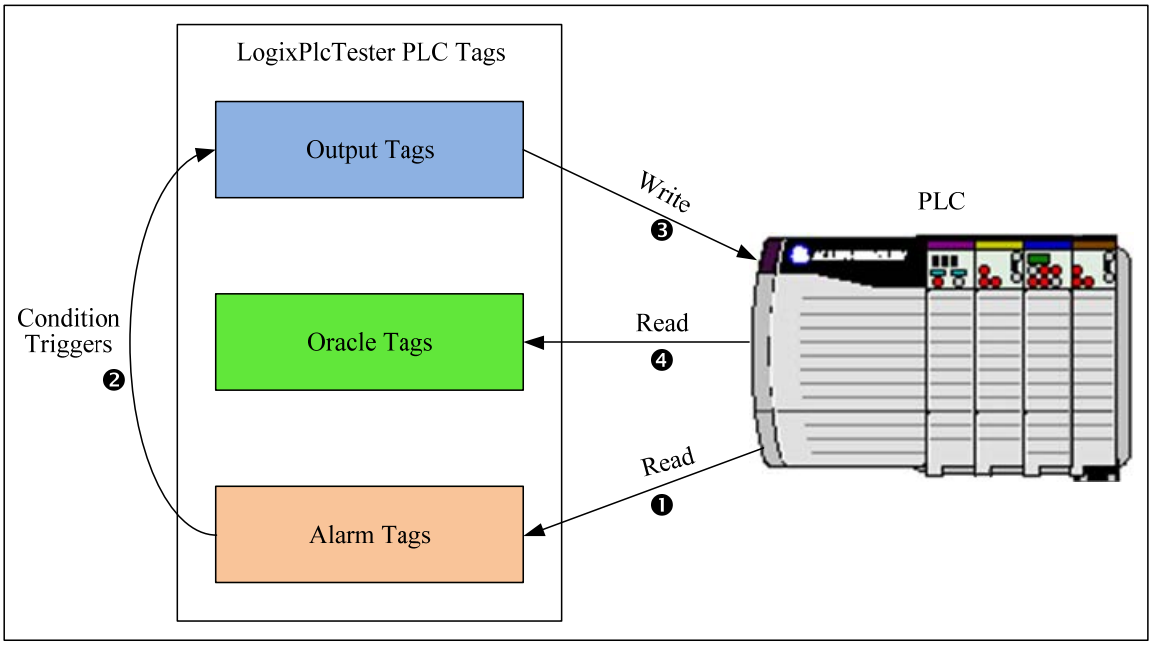


Figure 4.17: LogixPlcTester data flow path.

4.12 Test Case Example

The following example uses LogixPlcTester to test a typical Pump-Tank control logic (Figure 4.18). The control logic is defined as follows:

- Call the pump to start filling water to the tank when the tank level is below 15 feet.
- Call the pump to stop filling water to the tank if the tank level is above 28 feet.
- The PLC generates a “pump failed to start” alarm if it hasn’t received the pump running signal for 5 seconds after the pump start command is issued.
- The PLC generates a “pump failed to stop” alarm if it hasn’t received the pump stopped signal for 5 seconds after the pump stop command is issued.

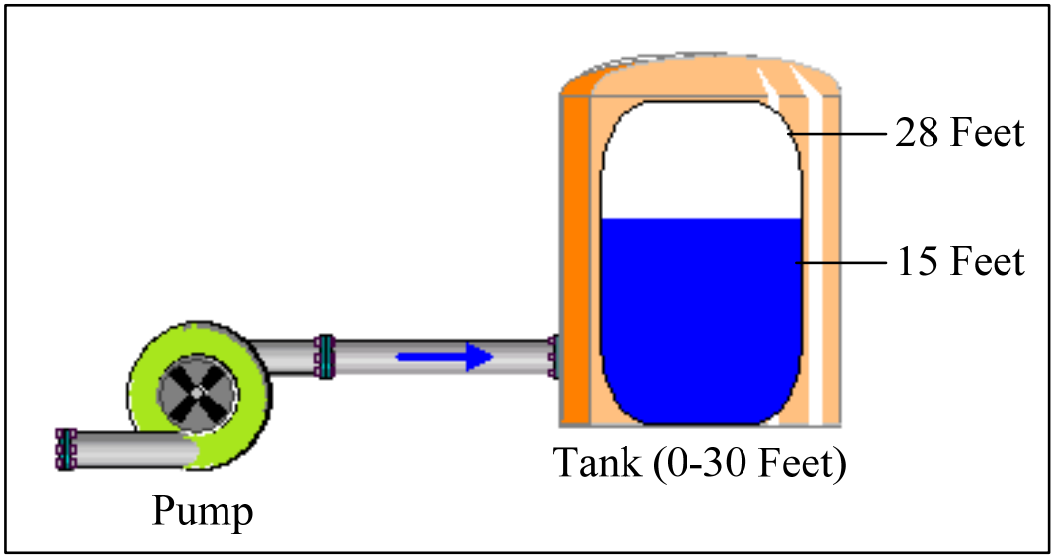


Figure 4.18: Pump-Tank control scenario overview.

Table 4.4: PLC tags for pump-tank control logic.

PLC Tag	Description	Data Type
Pump_Running	1: running 0:stopped	Boolean
Pump_Start_Cmd	PLC call pump to start	Boolean
Pump_Stop_Cmd	PLC call pump to stop	Boolean
Pump_Fail_To_Start	Pump failed to start alarm	Boolean
Pump_Fail_To_Stop	Pump failed to stop alarm	Boolean
Tank_Level	Tank level real-time reading	Floating Point
Pump_Reset_Cmd	Reset pump alarms	Boolean

Table 4.5: Pump-Tank control test case definition.

Test Case Tag	Type	Description	Trigger Type	Condition Tag	Value
Pump_Running Note 1	Output Tag	Write pump running signal to PLC	Condition	Pump_Start_Cmd	1
Pump_Running Note 1	Output Tag	Write pump stopped signal to PLC	Condition	Pump_Stop_Cmd	0
Tank_Level Note 2	Output Tag	Write tank level to PLC to simulate tank is above the Pump_OFF threshold (Write Delay Time is set to 2s)	Time		28.1
Pump_Reset_Cmd	Output Tag	Write pump alarm reset command to PLC (Write Delay Time is set to 1s)	Time		1
Tank_Level Note 2	Output Tag	Write tank level to PLC to simulate tank level is below the Pump_ON threshold (Write Delay Time is set to 3s)	Time		14.9
Pump_Running Note 3	Oracle Tag	Monitor real-time pump running status in PLC			
Tank_Level Note 3	Oracle Tag	Monitor real-time tank level reading in PLC			
Pump_Fail_To_Start	Alarm Tag	Monitor pump failed to start alarm in PLC			
Pump_Fail_To_Stop	Alarm Tag	Monitor pump failed to stop alarm in PLC			
Pump_Start_Cmd	Alarm Tag	PLC output command to start pump			
Pump_Stop_Cmd	Alarm Tag	PLC output command to stop pump			

Notes:

1. In LogixPlcTester, a same tag name can be defined multiple times with different functions. The tag Pump_Running is defined twice. One is to simulate pump running signal and another is to simulate pump stopped signal.
2. The tag tank_Level is defined twice. One is to simulate high tank level and another is to simulate low tank level.

3. In LogixPlcTester, the same PLC tag can be used multiple times in different tag lists. The tag Pump_Running and the tag Tank_level are defined in both the Output Tag list and the Oracle Tag list.
4. The order of the Output Tags in Table 4.5 won't affect the test. The write operation sequences are defined by the Write_Delay_Timer parameter of each tag when using Timer Trigger.

Table 4.4 shows the tags used in the PLC program for the Pump-Tank control logic. Table 4.5 shows the tags used in the test case. Based on the settings in Table 4.5, this test case is used to verify if the PLC control program sends an output command to stop the pump when the tank level rises higher the upper limit (28 feet) and verify if the PLC control program sends an output command to start the pump when the tank level drops below the lower limit (15 feet). After the test case is initiated, at the first second, LogixPlcTester sends a pump reset command to the PLC to reset the pump alarms if there is any. At the third second, LogixPlcTester writes 14.9 (feet) to the Tank_Level tag in the PLC. If the PLC control logic is correctly implemented then it will send an output command to start the pump as soon as it sees the tank level (14.9 feet) written by LogixPlcTester. If the PLC sends an output command to start the pump then LogixPlcTester will read this command through the alarm tag (Pump_Start_Cmd) defined in the test case. Since the Alarm Tag (Pump_Start_Cmd) was configured as a Condition Trigger to initiate the write operation of the Output Tag (Pump_Running), LogixPlcTester will write 1 to the Pump_Running tag in the PLC. Once this is done, the PLC gets the pump running signal feedback so its logic is satisfied in this scenario. Figure 4.19 shows the log messages in LogixPlcTester after the execution of the test case

defined in Table 4.5 is complete. In Figure 4.19, green color indicates snapshot values of Oracle Tags before the test case starts to run; blue color indicates writing Output Tag values to the PLC; wheat color indicates updated Oracle Tag values; white color indicates updated Alarm Tag (or Condition Tag) values.

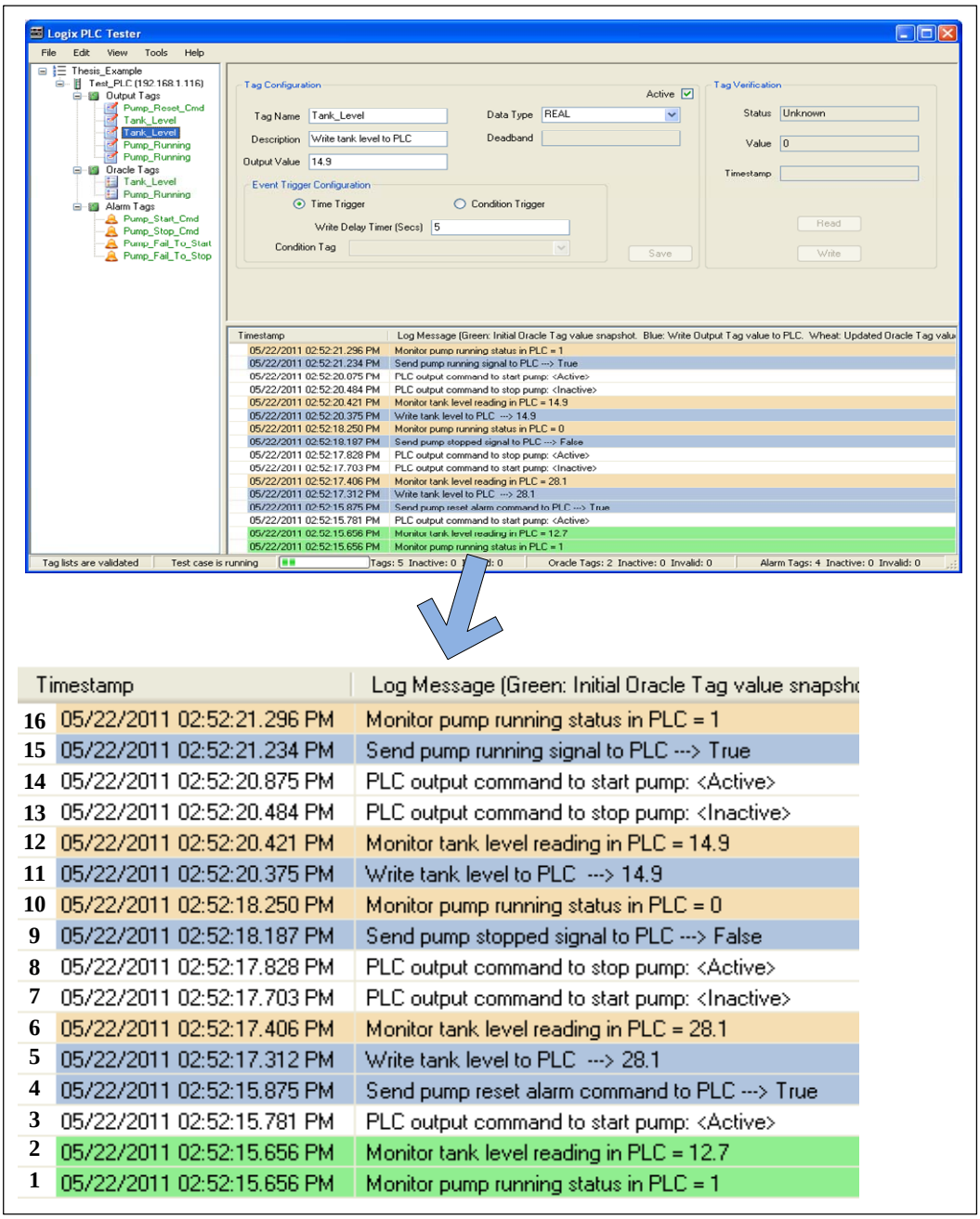


Figure 4.19: Log messages of a test case for the Pump-Tank control logic.

There are 16 log messages generated while the test case defined in Table 4.5 was running.

1. Snapshot of the Oracle Tag “Pump_Running” before the test case was started. The pump was running.
2. Snapshot of Oracle Tag “Tank_Level” before the test case was started. The tank level was 12.7 feet.
3. Status of the Alarm/Event Tag “Pump_Start_Cmd”. The PLC command for running the pump was active.
4. LogixPlcTester sent the value (1) of the Output Tag “Pump_Reset_cmd” to the PLC. This command reset the pump alarms (if there is any).
5. LogixPlcTester sent the value (28.1 feet) of the Output Tag “Tank_Level” to the PLC.
6. LogixPlcTester read the value of Oracle Tag “Tank_Level” tank level from the PLC. The tank level in the PLC was 28.1 feet. So Step 5 was successful.
7. Status of the Alarm/Event Tag “Pump_Start_Cmd”. The PLC command for running the pump was inactive.
8. Status of the Alarm/Event Tag “Pump_Stop_Cmd”. The PLC command for stopping the pump was active.
9. LogixPlcTester sent the value (0) of the Output Tag “Pump_Running” to the PLC to simulate the pump was stopped because the trigger event “Pump_Stop_Cmd” was active.
10. LogixPlcTester read the value of Oracle Tag “Pump_Running” from the PLC. The pump was shown stopped in the PLC. So Step 9 was successful.

11. LogixPlcTester sent the value (14.9 feet) of the Output Tag “Tank_Level” to the PLC.
12. LogixPlcTester read the value of Oracle Tag “Tank_Level” tank level from the PLC. The tank level in the PLC was 14.9 feet. So Step 11 was successful.
13. Status of the Alarm/Event Tag “Pump_Stop_Cmd”. The PLC command for stopping the pump was inactive.
14. Status of the Alarm/Event Tag “Pump_Start_Cmd”. The PLC command for starting the pump was active.
15. LogixPlcTester sent the value (1) of the Output Tag “Pump_Running” to the PLC to simulate the pump was running because the trigger event “Pump_Start_Cmd” was active.
16. LogixPlcTester read the value of Oracle Tag “Pump_Running” from the PLC. The pump was shown running in the PLC. So Step 15 was successful.

Actually, more test cases can be derived from the test case described in Table 4.5. The following are some example test cases that are derived from the base test case in Table 4.5.

1. Add a time trigger (with the Write Delay Time as 6s) based Output Tag to write 28.2 to the Tank_Level tag in the PLC. This will verify if the PLC issues an output command to stop the pump when the tank level rises higher than the upper limit (28 feet). This will complete a pump stop-running-stop control cycle.
2. Change the Condition On_Delay Time of Pump_Running (Output Tag to simulate pump running) to be greater than 5 seconds or completely deactivate the Output

Tag and verify if the PLC drops the pump start output command and generates a "pump failed to start" alarm.

3. In item #1 above, change the Condition On_Delay Time of Pump_Running (Output Tag to simulate pump stop) to be greater than 5 seconds or completely deactivate the Output Tag and verify if the PLC drops the pump stop output command and generates a "pump failed to stop" alarm.
4. Add a time trigger based Output Tag to simulate tank level reading becomes invalid while the pump is running and verify how the PLC reacts to it. (In this case, the PLC should drop the pump start output command and generate an invalid tank level alarm.)
5. Add a time trigger based Output Tag to write a value (between 15 and 28) to the Tank_Level tag in the PLC while the pump is running and verify the PLC continues to run the pump. (The pump should only stop when the tank level is above 28 feet in this case).
6. In item #1 above, add another time trigger based Output Tag to write a value (between 15 and 28) to the Tank_Level tag in the PLC while the pump is stopped and verify the PLC won't issue an output command to start the pump. (The pump should only start when the tank level is below 15 feet in this case).
7. Add a time trigger based Output Tag to simulate pump stopped signal while the pump start command is active and the pump is running and then verify how the PLC reacts to it. (In this case, the PLC should drop the pump start output command and generate a "pump stopped without a PLC command" alarm.)

8. Add a time trigger based Output Tag to write a value (not in the range of between 15 and 28) to the Tank_Level tag in the PLC and verify how the PLC reacts to it. (In this case, the PLC should seize control of the pump and generate a tank level out-of-range alarm.)

If we deactivate the Output Tag "Pump_running" to stop simulating the pump stopped signal (Figure 4.20) when the trigger event (PLC output command to stop the pump) is active, then we have a test case for the example #3 above. Figure 4.21 shows the generated log messages while running the test case in example #3. In Figure 4.21, a "pump failed to stop" alarm was generated after the PLC has energized the pump stop command for 5 seconds. This test case was passed because the PLC did generate a "pump failed to stop" alarm when it hadn't received the pump stopped feedback signal for 5 seconds.

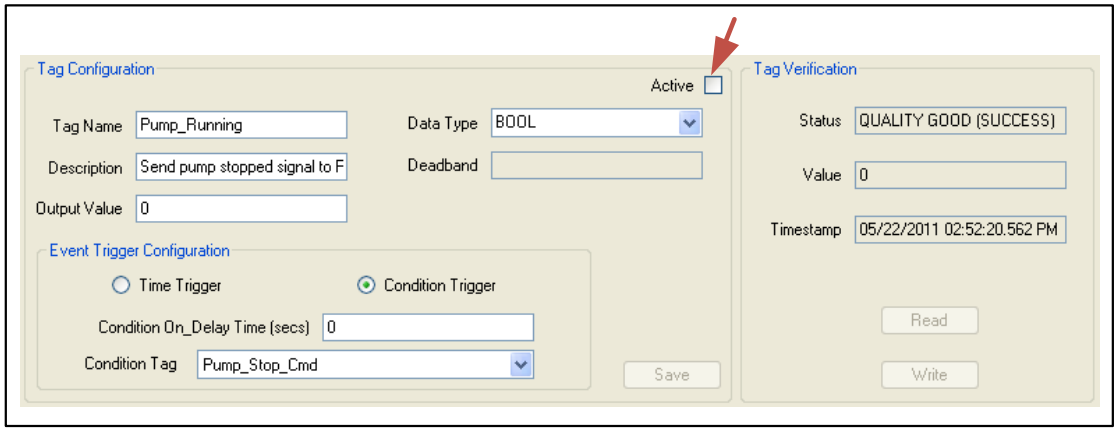
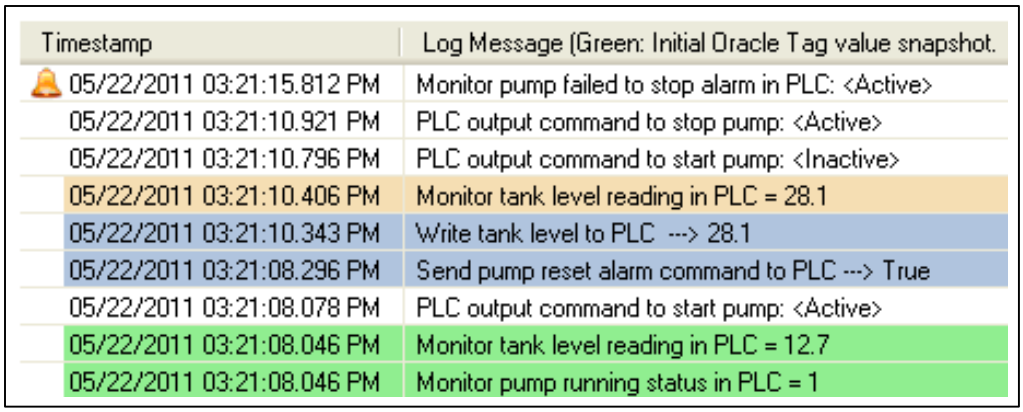


Figure 4.20: Deactivate an Output Tag.



Timestamp	Log Message (Green: Initial Oracle Tag value snapshot.
05/22/2011 03:21:15.812 PM	Monitor pump failed to stop alarm in PLC: <Active>
05/22/2011 03:21:10.921 PM	PLC output command to stop pump: <Active>
05/22/2011 03:21:10.796 PM	PLC output command to start pump: <Inactive>
05/22/2011 03:21:10.406 PM	Monitor tank level reading in PLC = 28.1
05/22/2011 03:21:10.343 PM	Write tank level to PLC ---> 28.1
05/22/2011 03:21:08.296 PM	Send pump reset alarm command to PLC ---> True
05/22/2011 03:21:08.078 PM	PLC output command to start pump: <Active>
05/22/2011 03:21:08.046 PM	Monitor tank level reading in PLC = 12.7
05/22/2011 03:21:08.046 PM	Monitor pump running status in PLC = 1

Figure 4.21: "Pump failed to stop" alarm log message.

The above example illustrates that after you have created the first test case for a PLC program, it's very easy to build more test cases based on the previous ones. LogixPlcTester is a method neutral test tool that can be used with a variety testing methods including Random Testing, Black-Box Testing, Combinational Testing, Bounded Exhaustive Testing, Model Based Testing, and Error-Based Testing.

4.13 Validate and Run Test Case

After a test case has been created, the existence of the tags in the Output Tag list, Oracle Tag list, and Alarm Tag list need to be verified in the PLC. The execution of a test case is prohibited if any of its tags is not verified or is verified with error. All the tags defined in a test case must exist in the PLC and defined as controller scoped tags (accessible to all routines). (Note: An Alarm Tag itself may not be in the PLC. But the tags used in an Alarm Tag's expression must exist in the PLC.) LogixPlcTester cannot access a program scoped tag (accessible to only the routines within a single program). This is specified by the PLC CIP protocol. This won't be an issue in common PLC based control systems. A PLC in a production system is normally on a plant-level Ethernet

network so that it can exchange data with the Human Machine Interface (HMI) such as Supervisory Control and Data Acquisition (SCADA) systems or Distributed Control Systems (DCS). In order for SCADA or DCS to access the tags in the PLC, they must have existed in the PLC as controller scoped tags. To verify the tags of a test case in LogixPlcTester, click **Tools** > **Validate**. The tags that are validated successfully will be changed to green color in the tag lists and the tags that cannot be validated will be changed to magenta color in the tag lists so they can be easily identified. If all the tags are validated successfully then the test case can run. To run a test case in LogixPlcTester, click **Tools** > **Run**. While a test case is running, the log messages will be filled in the Log View window as they occur. A running test case can be terminated at any time by clicking **Tools** > **Stop**. Figure 4.22 shows a test case that is running. The tags in the left panel are all in green color indicating they are validated. The Log View window shows the log messages generated by the test case. The Status Strip bar at the bottom of the application shows various information about the test case including tag validation status, test case running status, test case running progress bar, the number of PLCs defined in the test case, statistics of the Output Tag list, Oracle Tag list and alarm Tag list.

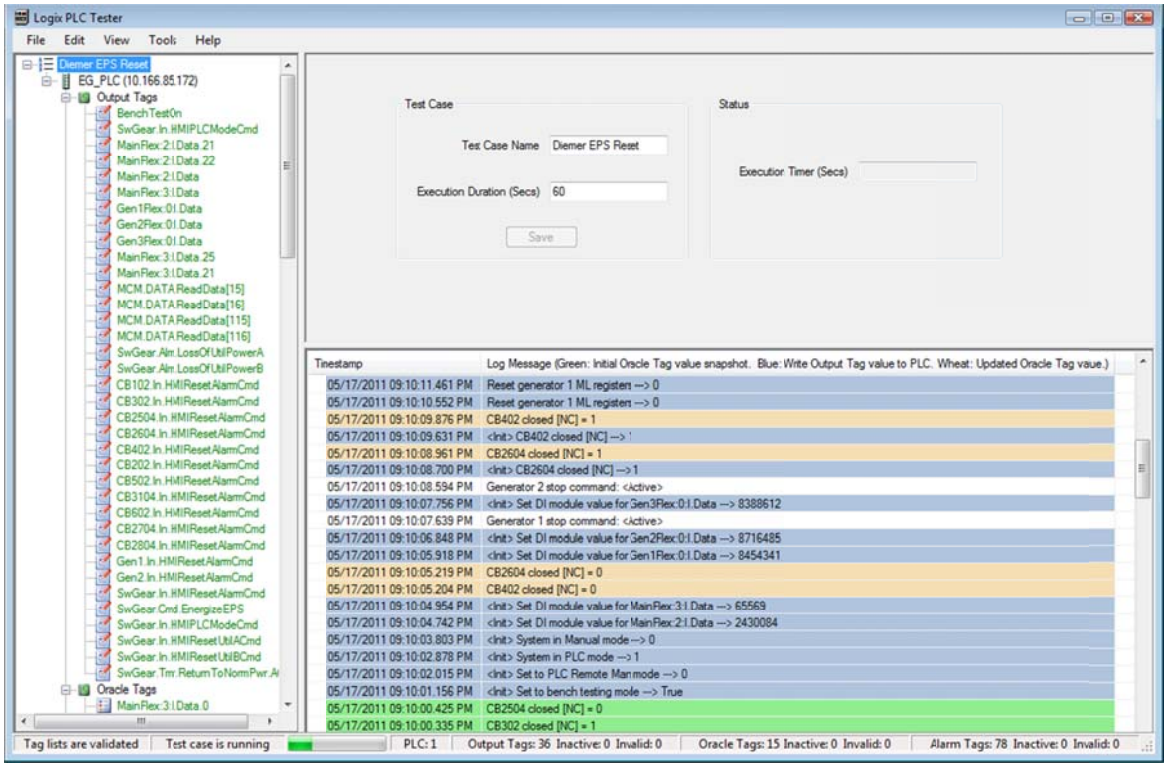


Figure 4.22: A running test case window in LogixPlcTester.

4.14 Control Graph

In order to help testers observe the test process and results graphically, LogixPlcTester has developed its own HMI interface that can edit and run control graphs. The GLG Toolkit from Generic Logic [8] was embedded in LogixPlcTester to support graphical displays. The GLG Graphics Builder (Figure 4.23) is used to create and edit graphical drawings. The GLG Graphics builder has a graphical objects library that includes tanks, pumps, dials, meters and other industrial symbols that can be used directly on a display. The API (Application Program Interface) functions provided by the GLG Toolkit were called in LogixPlcTester to incorporate a GLG drawing into the application and update the drawing with real-time data (from Oracle Tags or Alarm Tags). Figure 4.24 shows a tank level animation in GLG.

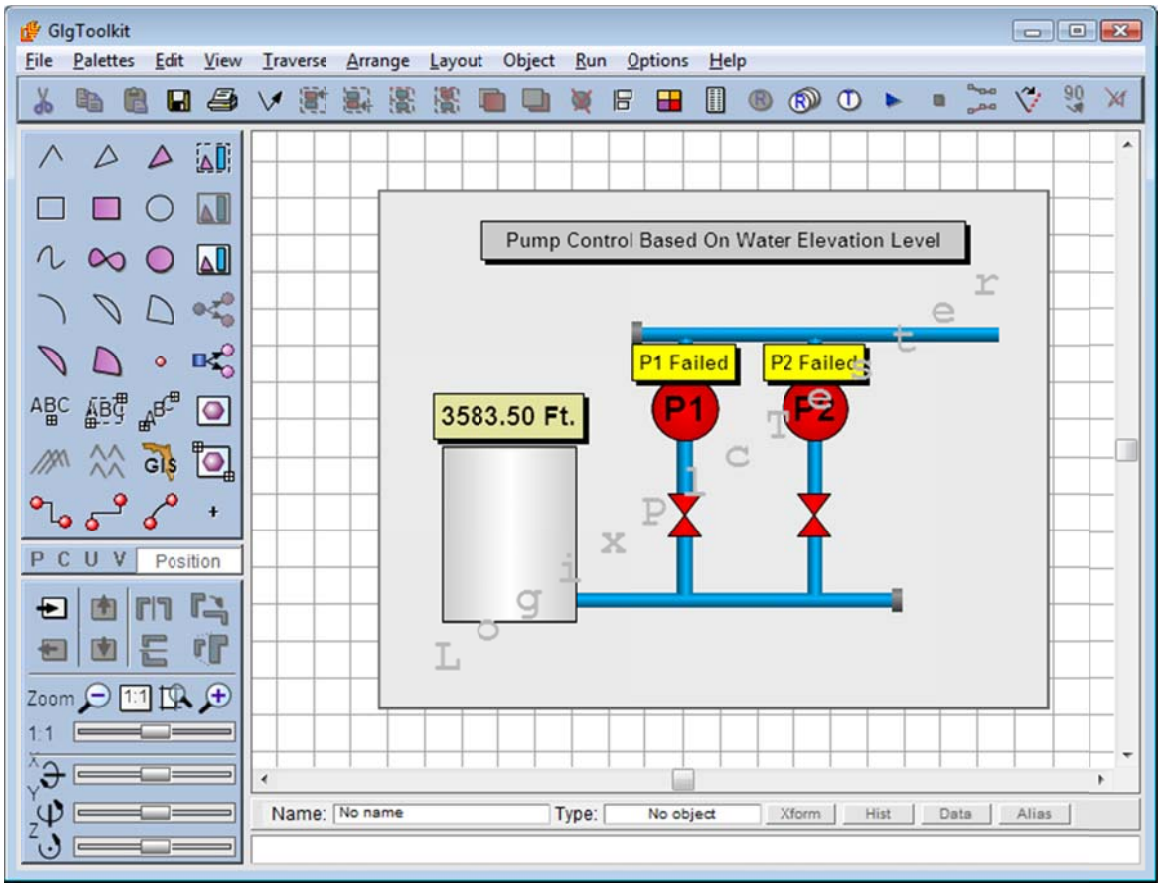


Figure 4.23: GLG Graphics Editor.

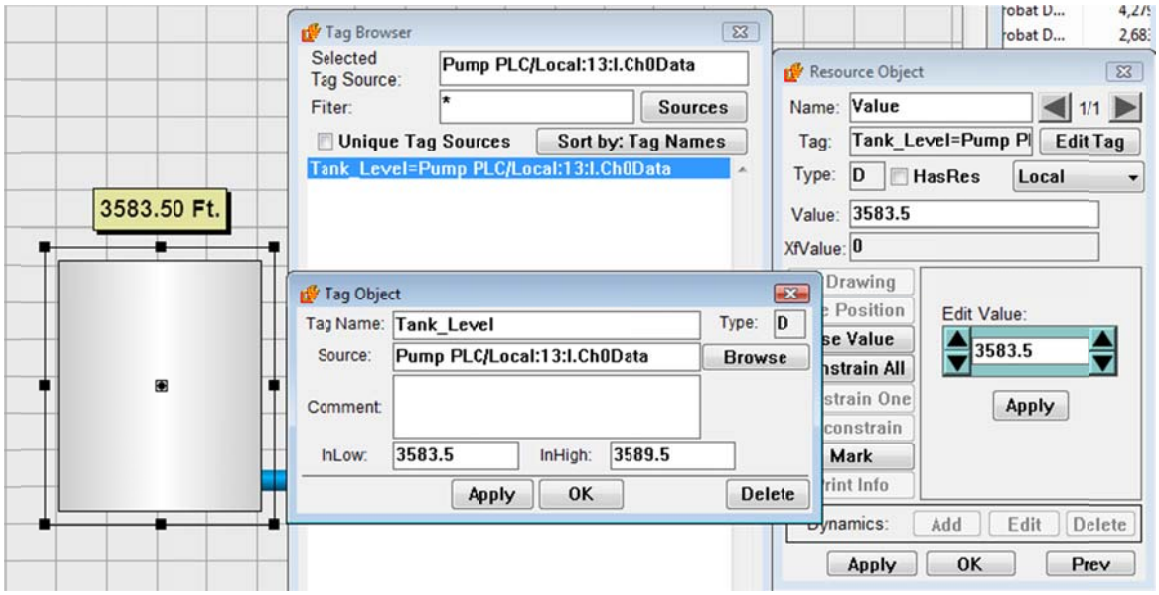


Figure 4.24: Animate a tank object in GLG.

To animate an object in a GLG drawing, a tag source is needed. A tag source is a string that defines a link to a tag in the PLC. The custom syntax of a tag source used in LogixPlcTester is PlcName/Tagname. The PlcName is the same PLC name specified in a LogixPlcTester test case. The Tagname is a valid Oracle Tag or alarm Tag in a LogixPlcTester test case. Figure 4.25 shows the test case to which the GLG drawing in Figure 4.23 is linked.

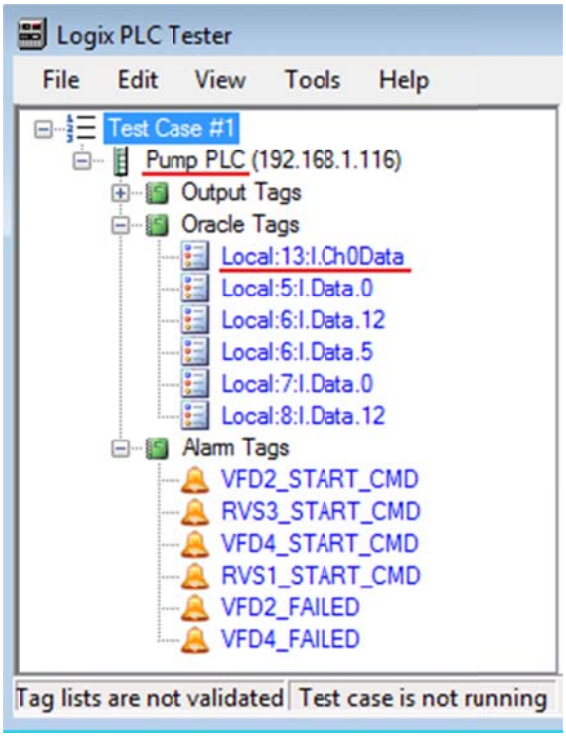


Figure 4.25: A pump control test case.

To run a graphical display in LogixPlcTester, click **Tools > Open Contorl Graph**. (Figure 4.26). Figure 4.27 shows the runtime graphical display for a pump control system.

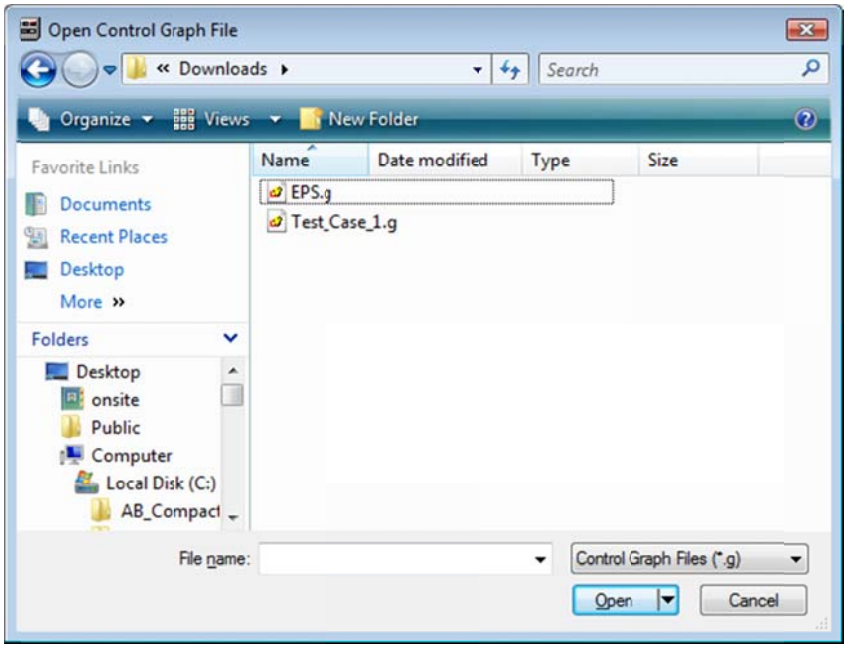


Figure 4.26: Open a GLG graphical display in LogixPlcTester.

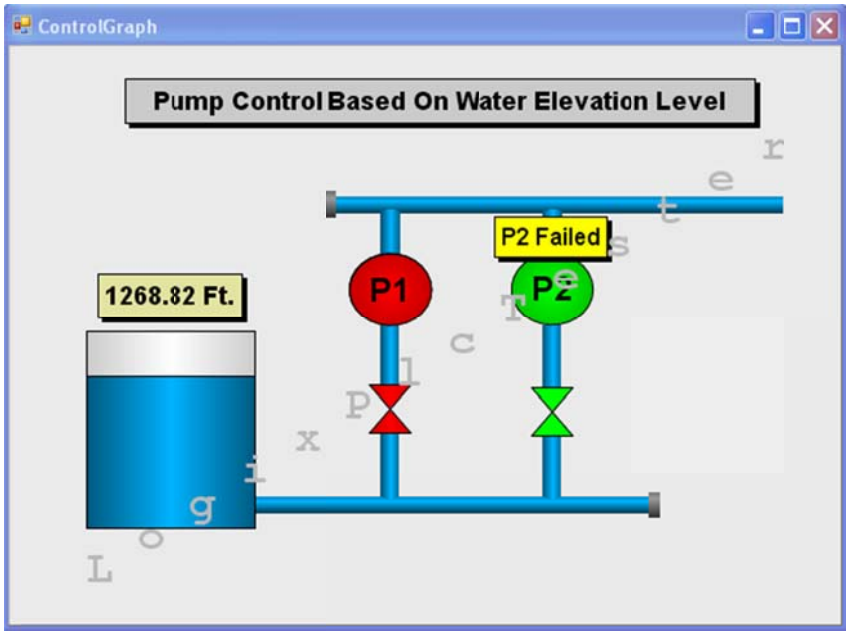


Figure 4.27 Graphical display for a pump control system.

4.15 Import and Export

A test case defined in LogixPlcTester can be saved as a XML file. A test case created externally by using a Text Editor can be imported into LogixPlcTester. Figure 4.28 shows how to import/export a test case.

LogixPlcTester can generate a report for a test case. A report lists a summary of the test case definitions. Compared to a test case XML file, a report is easy to read and understand because the information presented in a report is in spreadsheet formats. A report itself is in a PDF format. Figure 4.29 shows a sample report generated by LogixPlcTester.

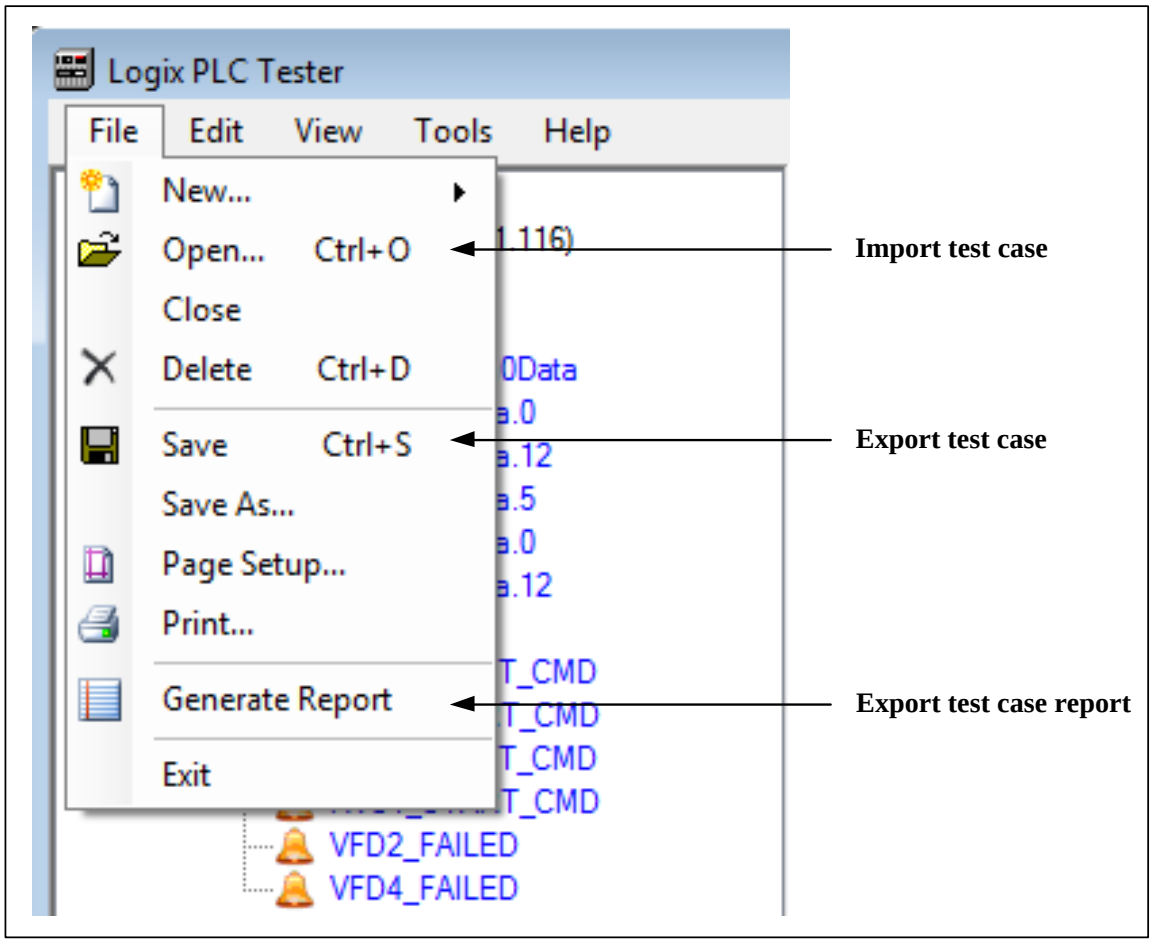
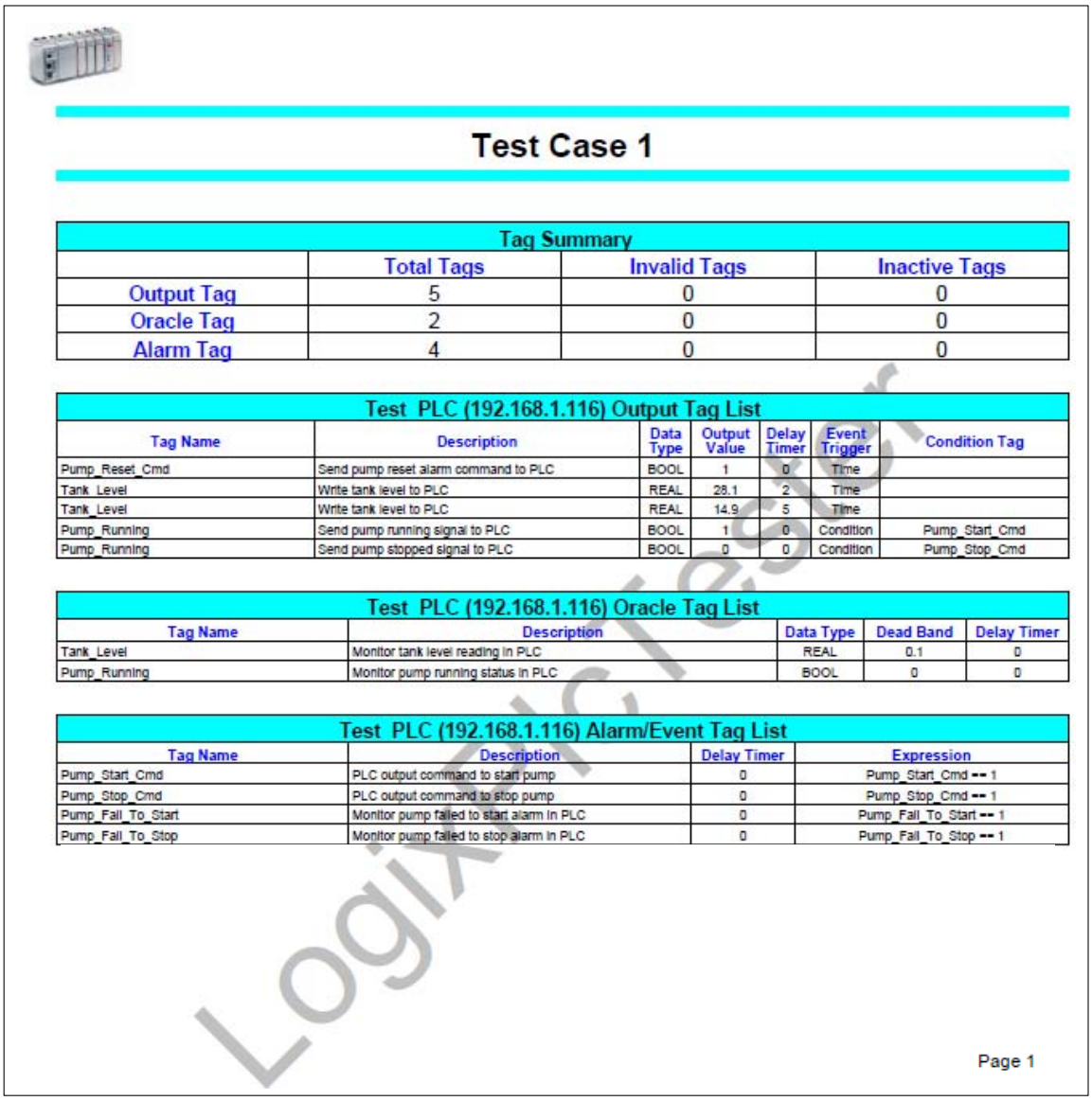


Figure 4.28: Import/Export a test case.



Test Case 1

Tag Summary			
	Total Tags	Invalid Tags	Inactive Tags
Output Tag	5	0	0
Oracle Tag	2	0	0
Alarm Tag	4	0	0

Test PLC (192.168.1.116) Output Tag List						
Tag Name	Description	Data Type	Output Value	Delay Timer	Event Trigger	Condition Tag
Pump_Reset_Cmd	Send pump reset alarm command to PLC	BOOL	1	0	Time	
Tank_Level	Write tank level to PLC	REAL	28.1	2	Time	
Tank_Level	Write tank level to PLC	REAL	14.9	5	Time	
Pump_Running	Send pump running signal to PLC	BOOL	1	0	Condition	Pump_Start_Cmd
Pump_Running	Send pump stopped signal to PLC	BOOL	0	0	Condition	Pump_Stop_Cmd

Test PLC (192.168.1.116) Oracle Tag List				
Tag Name	Description	Data Type	Dead Band	Delay Timer
Tank_Level	Monitor tank level reading in PLC	REAL	0.1	0
Pump_Running	Monitor pump running status in PLC	BOOL	0	0

Test PLC (192.168.1.116) Alarm/Event Tag List			
Tag Name	Description	Delay Timer	Expression
Pump_Start_Cmd	PLC output command to start pump	0	Pump_Start_Cmd == 1
Pump_Stop_Cmd	PLC output command to stop pump	0	Pump_Stop_Cmd == 1
Pump_Fail_To_Start	Monitor pump failed to start alarm in PLC	0	Pump_Fail_To_Start == 1
Pump_Fail_To_Stop	Monitor pump failed to stop alarm in PLC	0	Pump_Fail_To_Stop == 1

Page 1

Figure 4.29: Test case sample report.

Chapter 5. Experiment

LogixPlcTester was used to test the functions of a PLC application program for a real industrial control system. The PLC program runs in a development system that consists of a computer that runs LogixPlcTester, a computer that runs the PLC programming software, a PLC with no I/O modules, and a network switch that connects the computers and the PLC to the same network. LogixPlcTester was used to test the PLC program as it was developed.

5.1 System Overview

The application, the Emergency Power System was built for a water treatment plant in Southern California. The plant utility power is supplied via two main switchgear buses by two Southern California Edison feeders. The two main buses can be tied together so one feeder can power both of the buses in the event of a power outage or scheduled maintenance. The Emergency Power System is comprised of a separate switchgear bus and two generators. The emergency power switchgear bus connects the two generators to both main buses via circuit breakers so that the two main buses can be powered by the two generators during either a loss of both utility feeders, a malfunction of specific circuit breakers, or scheduled maintenance.

An Allen Bradley ControlLogix PLC is utilized in this system to monitor the utility power feeder status, switchgear status, circuit breaker status and control the circuit breakers and the generators to feed power to the plant when an abnormal condition occurs in the system such as a utility power failure or a circuit breaker failure. For example, if

the utility power is lost to one of the two main switchgears the PLC shall operate circuit breakers to feed power to this switchgear from the other switchgear that has normal power. If both main switchgears lose utility power at same time the PLC shall operate circuit breakers and start both generators and transfer power from the generators to both main switchgears. The PLC program for the Emergency Power System is a mission-critical program. If the PLC program fails to deliver power to the water treatment plant in a certain amount of time, then the water treatment process conducted at the plant will stop. As a result, untreated water could flow into the distribution water system and some residential area could be delivered with untreated water as drinkable water. In order to prevent such incident from happening, the PLC program must be tested thoroughly against all possible scenarios to verify it can deliver power to the plant within an allowed time period in every scenario. That is, both logical correctness and temporal correctness of the PLC program must be validated in order to meet the system operation requirements. Figure 5.1 shows the Emergency Power System diagram.

5.2 Software Design

The system specification lists over 200 operation scenarios in which the PLC program must respond correctly. Every scenario is triggered by an external event. The PLC program is designed as an event driven program. When an external event occurs (such as utility power loss, utility power return, or circuit breaker failure) the PLC will run a sequence (a series of hardware operations such as open breaker 1, close breaker 2) to respond to the event. Every sequence is defined by a subroutine, which is called in the main routine. In a sequence subroutine, there are some conditions (trigger events) that

must be met before the sequence can be actually started. Once a sequence is running, other sequences are not allowed to run. Only one sequence is allowed to run at any time. Table 5.1 shows an example of a sequence (sequence 9 in Figure 5.2). As a result, the switchgear configuration will change accordingly. An example scenario, suppose that the utility power A is failed (the utility power B is still normal) while the switchgear is in Normal. In this scenario, the PLC needs to open CB102 and then close CB2604 to power the bus A from the bus B. After the sequence is complete, the switchgear's state will change to BFeedA under normal power. So the system state changes from Normal to Normal BFeedA in this scenario. Another example scenario, both the utility power A and B are failed while the switchgear is in Normal. In this scenario, the PLC needs to open CB102 and CB202 first, then starts both generators, after CB2704 and CB2804 are closed the PLC should close both CB302 and CB403 to feed the plant with the generator power. After the sequence is complete, the system state will change to Split_Feed under generator power.

LogixPlcTester Logview captures all the log messages while a sequence is running. The log messages show the logical order of a series of actions that the sequence executes. The log messages are the main evidence used to verify the correctness of a sequence. To visually monitor the entire switchgear status and the progress of a running sequence, a control graph (Figure 5.1) is developed in LogixPlcTester. Figure 5.1 shows the control graph for system overview. The status of circuit breakers and generators in Figure 5.1 reflects the switchgear state while it's in normal mode in which the plant is fed with both utility power feeders. Appendix A shows the PLC program structure diagram.

Table 5.1: A sequence example.

Sequence 9: Retransfer to Plant Normal State	
Step 0	Open 302, 402, 2604 (Feedback time: 2 seconds)
Step 1	Close 102 (Feedback time: 2 seconds)
Step 2	Close 202 (Feedback time: 2 seconds)
Step 3	Shutdown both generators (Feedback time: 2 seconds)

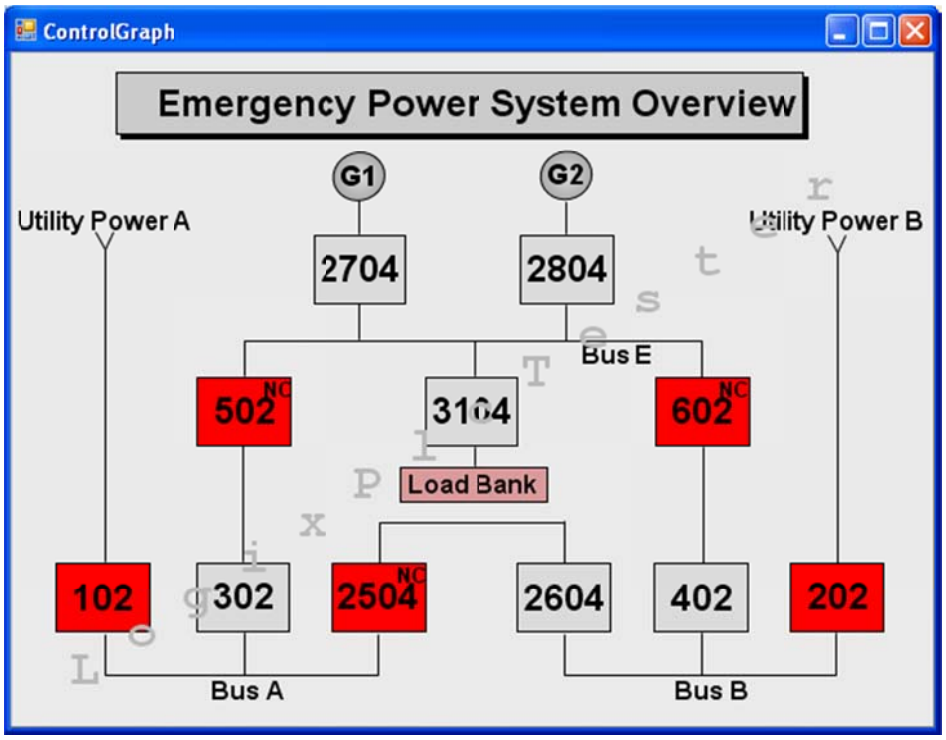


Figure 5.1: Emergency Power System overview in a LogixPlcTester graph.

5.3 Testing with LogixPlcTester

Since the hardware equipment is very expensive and is impossible to be relocated to the location where the software development/testing team works, all software tests must be conducted in a simulation system except the final acceptance test. The final acceptance test has to be conducted on the real hardware equipment after all the simulation tests are passed. Testing the PLC program in a simulation system will protect the hardware from being damaged by software bugs and it will also prevent the hardware

from wearing out by massive number of repeated operations during software development. So running tests in a simulation system will extend the lifetime of the hardware equipment. LogixPlcTester is utilized as both a simulation tool that simulates the hardware equipment's behaviors and a testing tool that tests the PLC program against every scenario of the Emergency Power System. To reduce the software development cycle by finding and fixing bugs at an early stage, the software tests were conducted as the PLC program was being developed. When a scenario becomes available in the PLC program a test case was created for the scenario and executed in LogixPlcTester to test the scenario.

To capture all the features and behaviors of the system, three Finite State Machine (FSM) models were established. The test cases are created according to the FSM models. Figure 5.2 shows a FSM model for power transfer and retransfer operations under normal conditions (no device failures) in the system. (There are two more models created in this system: the Circuit Breaker Failure Model and the Lockout Relay Failure Model. These two models define more than 150 scenarios.) As in Figure 5.2, each scenario has a trigger event and a sequence. The sequence will be initiated when the trigger event occurs. When the sequence is complete the system state (switchgear configuration) will change. A test case is created for each scenario. In some cases, a power transfer scenario and a power retransfer scenario are combined in the same test case. For example, scenario 1 (transfer upon utility power A failure) and scenario 6 (retransfer upon utility power A return) are combined in a same test case. The logical structure of a test case is illustrated in Figure 5.3.

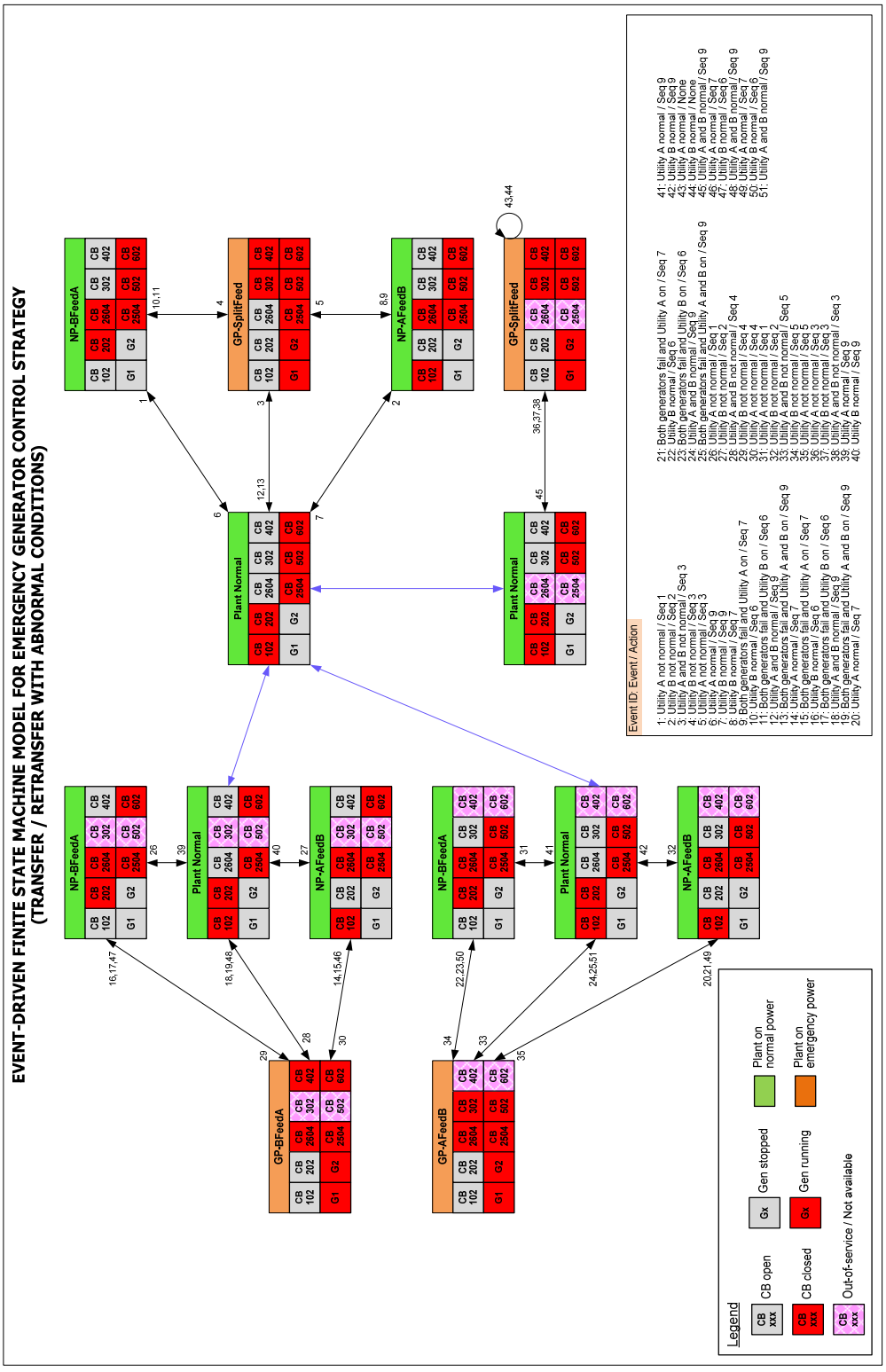


Figure 5.2: a FSM model for the Emergency Power system.

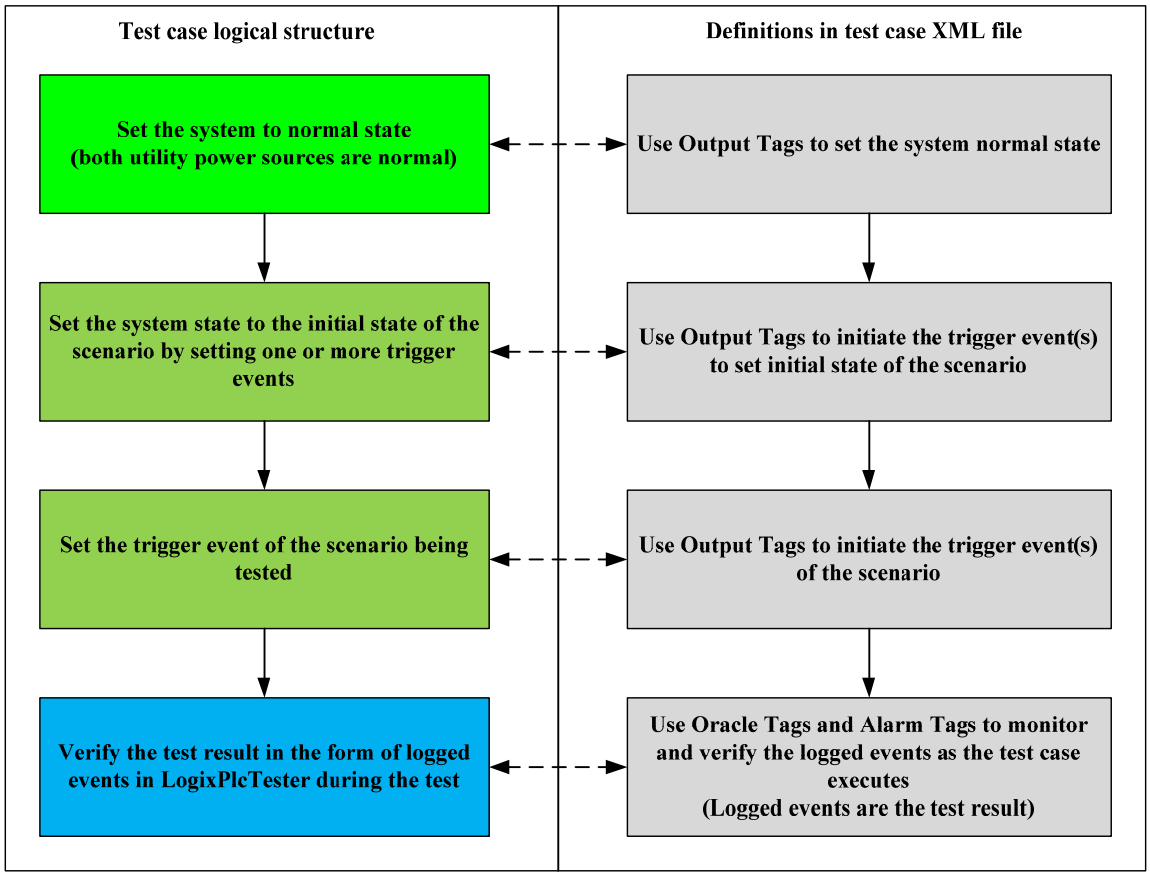


Figure 5.3: Logical structure of a LogixPlcTester test case.

For example, scenario 5 (in Figure 5.4) defines a transition from NP_AFeedB (bus A feed bus B under normal power) to GP_SplitFeed (generator power feed both bus A and bus B) when utility power source A is failed. The test case for scenario 5 consists of the following four major steps (corresponding to the four steps in Figure 5.3).

1. Use Output Tags to set the system to plant normal mode. Set CB102, CB202, CB2504, CB502, CB602 to the closed state by simulating the circuit breakers' feedback signals to the PLC. Use the same way to set the other circuit breakers to the open state and to set both generators (G1 and G2) to the off state.

2. Set the system to the initial state of scenario 5. NP-AFeedB (bus A feed bus B under normal power) is the initial state of scenario 5. The reason that the system is in NP-AFeedB mode is because utility power source B was failed. So at this step, we need to run scenario 2 first (by simulating a utility power source B failure) to get to the initial state of scenario 5.
3. Use an Output Tag to simulate the trigger event of scenario 5 which is utility power source A failure. This will initiate the sequence associated with scenario 5.
4. Use Oracle Tags and Alarm Tags to log necessary events in order to monitor and validate the sequence of scenario 5.

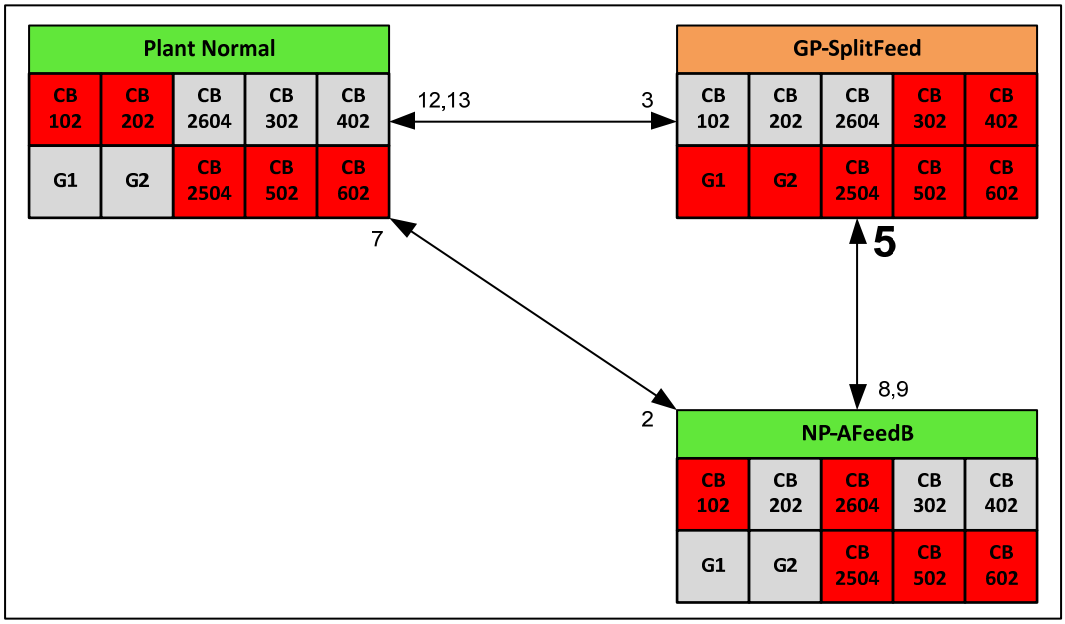


Figure 5.4: Scenario 5 for the Emergency Power System.

Unlike most general purpose systems, the test result of a scenario in a PLC program is usually a sequence of events (including the sequential relationship between the events) that determines the logical correctness of the test result. The timestamps of the events determine the temporal correctness of the test result. For example, in scenario 1, after utility power A is lost, the PLC shall open CB102 first and then close CB 2604. The sequence between actions is very important. Verifying the state of CB102 and CB2604 at the end of the test is not sufficient to verify the logical correctness of the result. You must also verify that CB102 open operation occurred before CB2604 close operation. To verify if a sequence was completed within the allowed time period, the timestamp of each action executed in the sequence must be inspected. Figure 5.5 shows the test results of the test case for scenario 1 and 6.

Timestamp	Log Message (Green: Initial Oracle Tag value snapshot. Blue: Write Output Tag value to PLC. Wh
06/05/2011 02:32:01.609 PM	****** Sequence-9 Re-Transfer to Normal Power Setup Was called to Run: <Inactive>
06/05/2011 02:32:01.500 PM	****** Sequence-9 Re-Transfer to Normal Power Setup Completed: <Inactive>
06/05/2011 02:31:57.125 PM	===> NP_BFeedA <===: <Inactive>
06/05/2011 02:31:57.078 PM	===> NP_Normal <===: <Active>
06/05/2011 02:31:56.984 PM	****** Sequence-9 Re-Transfer to Normal Power Setup Completed: <Active>
06/05/2011 02:31:53.406 PM	CB102 close command: <Inactive>
06/05/2011 02:31:52.734 PM	CB102 close = 1
06/05/2011 02:31:52.687 PM	CB102 closed ---> 1
06/05/2011 02:31:52.656 PM	CB2604 open command: <Inactive>
06/05/2011 02:31:52.609 PM	CB102 close command: <Active>
06/05/2011 02:31:51.921 PM	CB2604 closed [NC] = 1
06/05/2011 02:31:51.890 PM	CB2604 closed [NC] ---> 1
06/05/2011 02:31:51.859 PM	CB2604 open command: <Active>
06/05/2011 02:31:51.609 PM	****** Sequence-9 Re-Transfer to Normal Power Setup Was called to Run: <Active>
06/05/2011 02:31:51.531 PM	****** Sequence-1 NP_BFeedA Setup Was Called to Run: <Inactive>
06/05/2011 02:31:51.281 PM	****** Sequence-1 NP_BFeedA Setup Completed: <Inactive>
06/05/2011 02:31:51.218 PM	####### EMERGENCY to NORMAL countdown timer is up ---> 3599000
06/05/2011 02:31:48.203 PM	####### Transition Trigger: Utility power A resumed (Scenario 6 Active) ---> 0
06/05/2011 02:31:43.453 PM	CB2604 close command: <Inactive>
06/05/2011 02:31:43.296 PM	===> NP_BFeedA <===: <Active>
06/05/2011 02:31:43.281 PM	===> NP_Normal <===: <Inactive>
06/05/2011 02:31:42.937 PM	****** Sequence-1 NP_BFeedA Setup Completed: <Active>
06/05/2011 02:31:42.875 PM	CB2604 closed [NC] = 0
06/05/2011 02:31:42.828 PM	CB2604 closed [NC] ---> 0
06/05/2011 02:31:42.687 PM	CB2604 close command: <Active>
06/05/2011 02:31:42.640 PM	CB102 open command: <Inactive>
06/05/2011 02:31:41.937 PM	CB102 close = 0
06/05/2011 02:31:41.906 PM	CB102 closed ---> 0
06/05/2011 02:31:41.859 PM	CB102 open command: <Active>
06/05/2011 02:31:41.640 PM	****** Sequence-1 NP_BFeedA Setup Was Called to Run: <Active>
06/05/2011 02:31:38.218 PM	####### Transition Trigger: Utility power A failed (Scenario 1 Active) ---> 1
06/05/2011 02:31:28.656 PM	===> NP_Normal <===: <Active>
06/05/2011 02:31:28.343 PM	Generator 2 stop command: <Active>
06/05/2011 02:31:28.328 PM	Generator 1 stop command: <Active>
06/05/2011 02:31:28.171 PM	CB2504 closed [NC] = 0
06/05/2011 02:31:28.171 PM	CB302 closed [NC] = 1
06/05/2011 02:31:28.156 PM	CB2604 closed [NC] = 1
06/05/2011 02:31:28.156 PM	CB402 closed [NC] = 1
06/05/2011 02:31:28.156 PM	CB202 closed = 1
06/05/2011 02:31:28.156 PM	CB102 close = 1
06/05/2011 02:31:28.156 PM	CB602 closed [NC] = 0
06/05/2011 02:31:28.140 PM	CB3104 closed [NC] = 1
06/05/2011 02:31:28.140 PM	CB502 closed [NC] = 0
06/05/2011 02:31:28.140 PM	Generator 2 running = 0
06/05/2011 02:31:28.140 PM	Generator 2 off = 1
06/05/2011 02:31:28.140 PM	CB2804 closed [NC] = 1
06/05/2011 02:31:28.140 PM	Generator 1 running = 0
06/05/2011 02:31:28.140 PM	Generator 1 off = 1
06/05/2011 02:31:28.140 PM	CB2704 closed [NC] = 1

Figure 5.5: Log messages of running scenario 1 and 6 in LogixPlcTester.

Every time after a major change was made in the PLC program, a regression testing process was conducted by LogixPlcTester. Basically all the tests that were verified before must be retested again because a major code change could potentially introduce new bugs in the PLC program. The functions of the PLC program that worked before might not work again in the new code. There are around 200 test cases created for the Emergency Power System. It will take at least a week to run all the tests if using a hardware test stand or testing against the real equipment. But with LogixPlcTester, it only takes a day to run all the tests. Because all the test cases were saved in XML files there is no preparation for a test before running it. Once a test case XML file is loaded, LogixPlcTester will automatically read the test instructions from the XML file and execute them in the pre-defined orders.

The PLC program was written by a programmer and the author did a code review for most parts of the code resulting in familiarizing with the internal data structure and program structure. The author also had a complete knowledge of how the system should work. Most of the test cases were created based on the FSM model. There were some test cases that were created based on knowledge of the internal structure of the program. For example, by code reviewing it was found that there was a operation mode selector switch that sends two hardware signals to a PLC digital input module. The two signals are Automatic mode and Manual mode. In normal operation, only one of the two signals is active. Two test cases were created to test the mode switch logic under abnormal conditions. One is used to simulate both hardware signals are active and another is used to simulate both hardware signals are inactive. These two test cases help the operators understand how the PLC will react to the mode selector switch hardware failure. Another

example is I knew the PLC feedback timer (the time that the PLC waits for a feedback signal after it issues a control command) which is set to 3 seconds for the circuit breaker open and close commands. Two test cases were created to test the feedback timer logic. One is to simulate the feedback signal 4 seconds after the PLC issued a circuit breaker open command and another is to send no feedback. These two test cases test how the PLC handles the circuit breaker control failures.

5.4 Results

The PLC program has 52 subroutines and around 2500 rungs of ladder logic code in total. There are around 220 test cases created in LogixPlcTester to cover all operation scenarios specified in the system specification. The entire testing process took one month to finish due to the availability of the PLC software functions. The software tests were conducted as the PLC software was developed. Sometimes the testing process had to stop to wait for the new functions or bug fixes to be available. The time spent on actual software testing and troubleshooting was about 20 days and over 1000 tests were conducted in LogixPlcTester. There were 23 bugs detected in the PLC program by LogixPlcTester during the entire testing process and the following 10 bugs represent them. Table 5.2 shows the PLC instructions used in the bug examples. Table 5.3 shows the system states used in the bug examples. Table 5.4 shows the abbreviations used in the bug examples.

Because of the bugs were detected and fixed during the simulation tests by LogixPlcTester, there was not a single bug reported during the final tests in the

production system besides few scenarios that were missed in the original design document.

Table 5.2: PLC ladder instructions used in the bug examples.

Ladder Instruction	Description
-] [- (Examine if closed)	The instruction tests the data bit to see if it is set.
-]\[- (Examine if open)	The instruction tests the data bit to see if it is cleared.
-( )- (Output energize)	When the instruction is enabled, the controller sets the data bit. When the instruction is disabled, the controller clears the data bit.
-(L)- (Output latch)	When enabled, the instruction sets the data bit. The data bit remains set until it is cleared, typically by an -(U)- instruction. When disabled, the instruction does not change the status of the data bit.
-(U)- (Output unlatch)	When enabled, the instruction clears the data bit. When disabled, the instruction does not change the status of the data bit.
-]ONS[- (One-shot bit)	When enabled and the storage bit is cleared, the instruction enables the remainder of the rung. When disabled or when the storage bit is set, the instruction disables the remainder of the rung.
JSR (Jump to subroutine)	The instruction jumps execution to a different routine
TON (Timer on delay)	The instruction is a non-retentive timer that accumulates time when the instruction is enabled. A timer's enable bit (.EN) indicates the timer is enabled. A timer's done bit (.DN) indicates the timer times out.
RES (Reset)	The instruction resets a timer.

Table 5.3: System states used in the bug examples.

System State	CB102 Status	CB202 Status	CB2604 Status	CB302 Status	CB402 Status	Description
NP-BFeedA	Open	Closed	Closed	Open	Open	Utility power B feeds both A-side and B-side of the plant. Utility power A is not available.
NP-AFeedB	Closed	Open	Closed	Open	Open	Utility power A feeds both A-side and B-side of the plant. Utility power B is not available.
GP-SplitFeed	Open	Open	Open	Closed	Closed	Generators feed the plant. Both utility power A and B are not available.
Plant Normal	Closed	Closed	Open	Open	Open	Utility power A feeds A-side of the plant and Utility power B feeds B-side of the plant.
NP-AOnly	Closed	Open	Open	Open	Open	Utility power A feeds A-side of the plant. B-side of the plant has no power.
NP-BOnly	Open	Closed	Open	Open	Open	Utility power B feeds B-side of the plant. A-side of the plant has no power.
Plant Dark	N/A	N/A	N/A	N/A	N/A	The plant has no power.

Table 5.4: PLC variable name abbreviations used in the bug examples.

Abbreviation	Description
Alm	Alarm
CB	Circuit Breaker
Cmd	Output Command
DI	Digital Input
DN	Timer is done
DO	Digital Output
GP	Generator Power
In	Input
NP	Normal Power
Seq	Sequence
Seqx	Sequence x
Stat	Status
Step.x	Step x of a sequence (x starts from 0)
SwGear	Switchgear

The following examples illuminate the common variable names that are used in the code examples in this chapter.

Example 1: SwGear.Cmd.SetupNP_B_Feed_A means the switchgear sequence NP-BFeedA is running.

Example 2: CB102.In.OpenDI means the circuit breaker 102 is open.

Example 3: CB102.Cmd.OpenDO means the circuit breaker 102 open command is active.

Example 4: Sq1_NP_BFeedA.Step.0 means the sequence 1 (NP-BFeedA) is running at the first step.

Example 5: CB2604.Alm.FailToClose means the circuit breaker 2604 has a “failed to close” alarm.

Example 6: SwGear.Stat.NP_BOnly means the switchgear’s current system state is NP-BOnly.

1. Incorrect permissive conditions

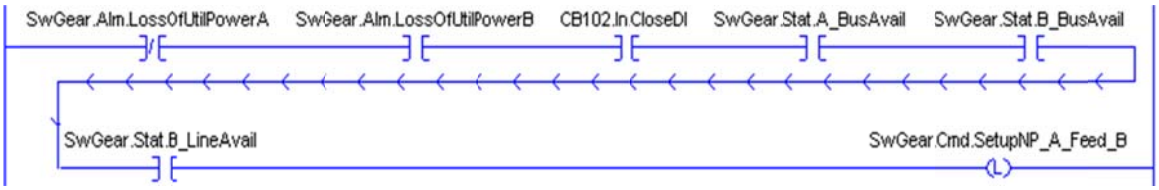
Overview

Permissive conditions are a PLC programming idiom. A permissive is a process condition that must be met before hardware equipment is allowed to operate. Permissive conditions can protect the equipment from unsafe or illegal operations. Usually there are multiple permissive conditions that need to be met in order to operator on equipment. In process control PLC programs, almost all the output commands to equipment have associated with permissive conditions. The permissive conditions must be set accurately in order for PLC to send the right command to the right equipment at the right time. If a permissive condition is missing for an output command then the output command might be sent to the equipment when it shouldn’t be sent. If unnecessary or unrelated

permissive conditions are set for an output command then the output command won't be sent to the equipment or the command will be sent to the equipment unexpectedly.

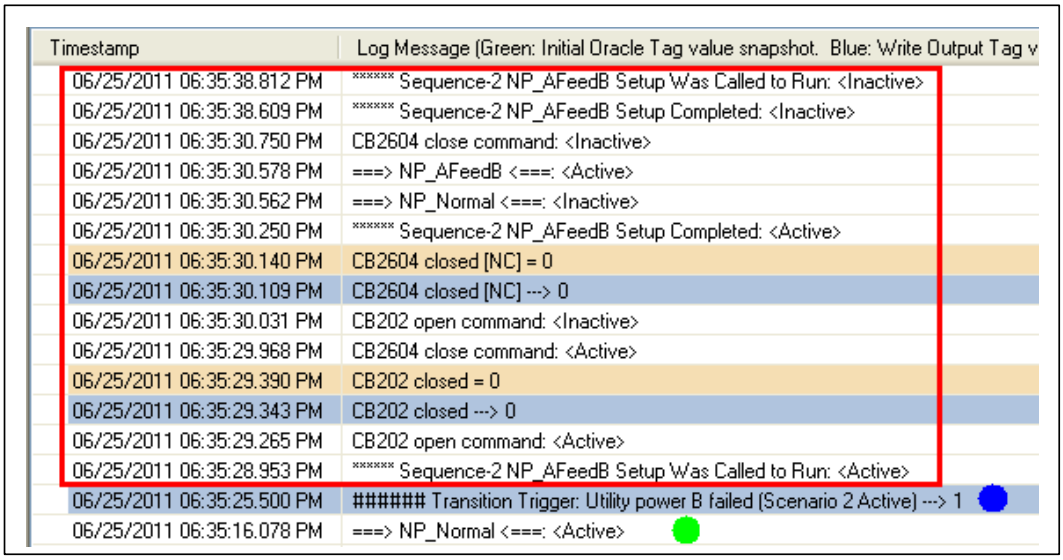
Detection

The test case that detected the bug is the test case designed for the system state transition scenario #2 of the FSM model described in Figure 5.2. The initial system state of this scenario is Plant Normal, the trigger event is the utility power B failure, and the expected system state is NP-AFeedB. During the test, after the trigger event was initiated there was no indication that showed the NP-AFeedB sequence was called in the LogixPlcTester logs. This implied that the permissive conditions for the sequence may not be setup correctly. After debugging the PLC program, a bug was found in this following ladder code. The code lists 6 permissive conditions for calling the NP-AFeedB sequence.



It was found that the permissive condition "SwGear.Stat.B_LineAvail" (a PLC variable) was not related to the sequence and its value was false during the test. This resulted in an unsatisfied permissive condition for the sequence. In this bug, the permissive "SwGear.Alm.LossOfUtilPowerB" and the permissive "SwGear.Stat.B_LineAvail" are mutually exclusive. ("SwGear.Stat.B_LineAvail" is true only when "SwGear.Alm.LossOfUtilPowerB" is false and CB202 is closed.) When the NP-AFeedB sequence is ready to run (the first 5 permissives are met), the permissive "SwGear.Stat.B_LineAvail" is always false. So the sequence will never run.

After removing “SwGear.Stat.B_LineAvail” from the above code, the same test case was rerun and the expected result was observed by reviewing the log messages in LogixPlcTester. Figure 5.6 shows the log messages while running the sequence after the bug was fixed. The green circle shows the initial state of the sequence and the blue circle shows the trigger event of the sequence. The log messages in the red rectangle show the “Normal power A feed B” sequence was called after the trigger event had been initiated. When the bug was present in the ladder code, the log messages in the red rectangle were not seen and that was how the bug was detected by LogixPlcTester.



Timestamp	Log Message (Green: Initial Oracle Tag value snapshot. Blue: Write Output Tag v
06/25/2011 06:35:38.812 PM	****** Sequence-2 NP_AFeedB Setup Was Called to Run: <Inactive>
06/25/2011 06:35:38.609 PM	****** Sequence-2 NP_AFeedB Setup Completed: <Inactive>
06/25/2011 06:35:30.750 PM	CB2604 close command: <Inactive>
06/25/2011 06:35:30.578 PM	===> NP_AFeedB <===: <Active>
06/25/2011 06:35:30.562 PM	===> NP_Normal <===: <Inactive>
06/25/2011 06:35:30.250 PM	****** Sequence-2 NP_AFeedB Setup Completed: <Active>
06/25/2011 06:35:30.140 PM	CB2604 closed [NC] = 0
06/25/2011 06:35:30.109 PM	CB2604 closed [NC] ---> 0
06/25/2011 06:35:30.031 PM	CB202 open command: <Inactive>
06/25/2011 06:35:29.968 PM	CB2604 close command: <Active>
06/25/2011 06:35:29.390 PM	CB202 closed = 0
06/25/2011 06:35:29.343 PM	CB202 closed ---> 0
06/25/2011 06:35:29.265 PM	CB202 open command: <Active>
06/25/2011 06:35:28.953 PM	****** Sequence-2 NP_AFeedB Setup Was Called to Run: <Active>
06/25/2011 06:35:25.500 PM	###### Transition Trigger: Utility power B failed (Scenario 2 Active) ---> 1
06/25/2011 06:35:16.078 PM	===> NP_Normal <===: <Active>

Figure 5.6: Log messages of the NP-AFeedB sequence.

Error Analysis

Since every transition in the FSM models is covered in a test case, every sequence is covered by at least one test case. (Some transitions share one sequence. See Figure 5.2.) For this particular bug, it can be uncovered during the system tests because of the mutual exclusive property of this bug. General speaking, the incorrect permissive conditions related bugs could be detected by a test because if a bug exists in a transition

then the test case that tests that transition will fail due to a control sequence failure (such as an output command is not issued or an output command is issued unexpectedly). But it's not always the case. For example, in the above bug, if the permissive "Line_B_OK" is independent and it has no logical relationship with the rest of the permissive conditions in the ladder rung and its value is true by coincidence during the test, then this bug may be missed. In this example, code review may be more efficient to reveal this kind of bug.

Permissive conditions for the equipment control commands may not be available or may be only partially available in the system specification because the people who wrote the system specification may not have the information. As a result, the programmer may make some assumptions during the software development and at the end of the development those assumptions were not verified with the operators who know all the permissive conditions for operating the equipment. Considering all the possible combinations of the permissive condition a model such as a truth table could be used to help to create test cases to cover all the possibilities. At the software development stage, the programmer may not have a full view of the permissive conditions. It's common and reasonable to make some assumptions in order to continue the software development. However, at a later time these assumptions muse be revisited and verified with the domain expert.

2. Typo

Overview

In the PLC program, there are many tags that are similar to each other. For example, when you create two instances of a generator type, the two new instances will have similar names such as G1 and G2. The similar names in the PLC program are

logical. If an incorrect tag is used in the program due to a typo then the program most likely will behave unexpectedly and the unexpected behavior could be revealed during a system state transition test in which the ladder rung (where the typo resides) is checked.

Detection

The typo bug found by a test was that the system state variable SwGear.Stat.NP_AFeedB was mistakenly written as SwGear.Stat.NP_BFeedA in the following ladder code.

The test case that detected the bug is one of the test cases that test the system failures. In the test, the initial system state is NP-AFeedB, the first trigger event is the utility power B is normal, the second trigger event is CB202 “failed to close” alarm, and the expected system state is the same as the initial state - NP-AFeedB. The sequence that was called to run the transition was the sequence 9 (in scenario 7 of Figure 5.2). At the step 2 (Sq9_NormalPower.Step.2 is true) of the sequence 9, CB202 was commanded to close but it was failed to close (simulated by LogixPlcTester). The ladder code was supposed call the sequence 2 (the NP-AFeedB sequence). But due to the typo, this rung was never executed because the system state SwGear.Stat.NP_AFeedB was one of the permissives to run the NP-AFeedB sequence. During the test, SwGear.Stat.NP_BFeedA was false because it wasn’t the current system state. (The current system state was NP-AFeedB and the system state variable SwGear.Stat.NP_AFeedB was true.) As a result, after observing the CB202 “failed to close” log message in the LogixPlcTester Log Viewer, there was no further PLC action observed and the system state was left at NP-AOnly because the bus tie breaker CB2604 was opened at the step 1 of the sequence 9.

Error Analysis

The bug actually led to a permissive condition problem and it was caught by a test case that was testing a system state transition. The kind of typo bug can be discovered by the tests because similar tags in the PLC program usually are mutually exclusive of each other. One possible earlier error-oriented detection mechanism for this kind of bugs is to add detailed comments in the code that specify the function of each tag and function. When doing code review, this kind of bug can be easily identified. Another possibility is to use assertion based formal verification in the code to inspect the input conditions of the logic. This kind of bug is easy to fix but is difficult to prevent in the code unless using names, comments, or assertions to make a distinction between similar names.

3. One-shot related problem

Overview

In PLC programming, one-shot bit is a mechanism which is used to only execute the ladder rung once when the condition becomes true. The following structured text code demonstrates how one-shot works. A, B, C are Boolean type variables. B is a one-shot bit. The initial value of B is false. When A becomes true, “C := 1” will be executed. In the next scan cycle “C := 1” won’t be executed because B is true. “C := 1” will only be executed again when A goes through a transition of 1 to 0 to 1.

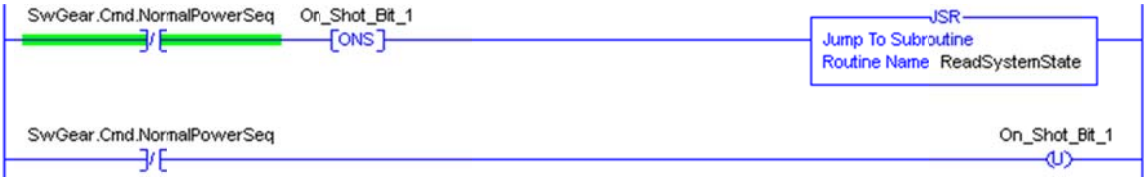
```
IF A  AND (NOT B) THEN C := 1;
B = A;
```

In PLC ladder logic, there is an instruction called -[ONS]- for the one-shot bit function. The following ladder code has the same function as the structured text code above.

In ladder code, a one-shot bit is controlled by the controller automatically. If the one-shot bit (B) is manually set or reset by the code then the one-shot function won't work correctly and that will cause the program to behave unexpectedly.

Detection

The one-shot bug was found in the following ladder rungs.



The code is supposed to run a subroutine called ReadSystemState only once when the "Return to Normal Power" sequence (sequence 9 in Figure 5.2) is completed. However the subroutine ReadSystemState was continuously called in every PLC scan cycle resulting in the system state capturing all the intermediate states while other sequences were running. The bug was detected in multiple test cases. For instance, in the "Early update of system state" example, the intermediate system state – "Plant Dark" was shown in the LogixPlcTester Log Viewer because it was monitored by an Oracle Tag and it caused the sequence to terminate early.

Error Analysis

The ladder rung that contains the one-shot bit problem will be executed during the tests because the test cases that cover all transitions will cover all the branches in the code. In a PLC ladder program, every rung of the program is essentially a branch. Since a one-shot problem definitely changes the system's behaviors and all the system behaviors/scenarios are covered by the test cases, it would be captured during the system

tests. If the intermediate system states showed in the Control system operator terminal then it will confuse the operator. The system state should be updated at the end of a sequence but not while a sequence is running. To detect one-shot bit problems in the PLC program at an early stage, a one-shot bit rule can be added to the programmer's checklist, which should be followed during code review.

4. Early update of system state

Overview

When an external event occurs (such as one utility power source becomes unavailable), a sequence will be called in the PLC code to run a series of commands (open/close some breakers) to transit to a new system state. Once the sequence is completed the system will be in a new state that is usually different from its original state before the sequence was called. When a sequence finishes, it must update the system state (described by a PLC internal variable) at the last rung of the sequence subroutine. If this is done earlier (meaning the system state is updated while a sequence is still running), the PLC program will be confused by the intermediate system states, which will lead to an unexpected termination of the running sequence. Figure 5.6 shows a system state update diagram.

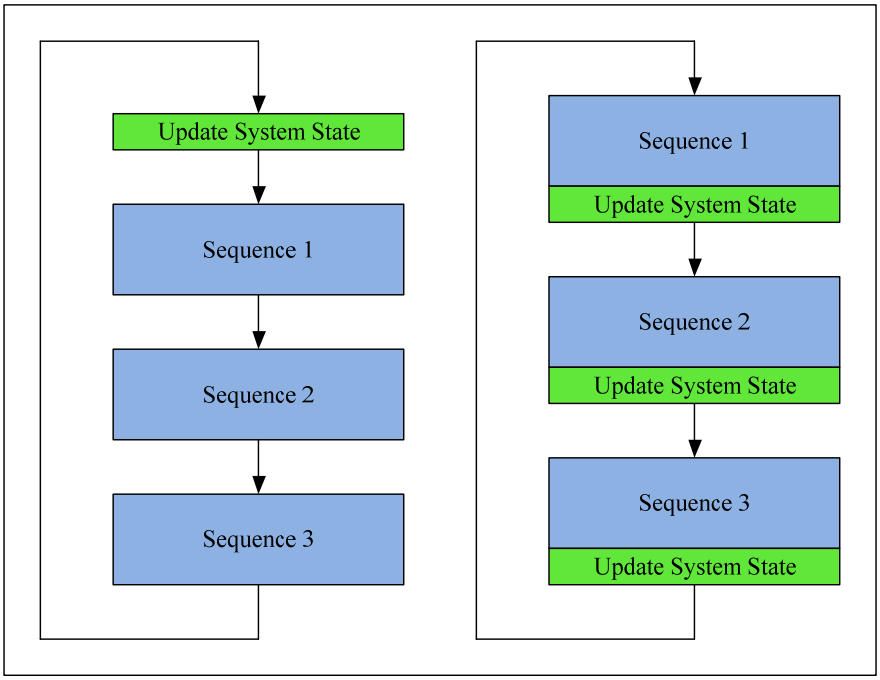


Figure 5.7: System state update diagram.

In Figure 5.7, there are three sequence subroutines and one System State Update subroutine and the arrows indicate the sequence that the subroutines are scanned in the PLC program. In the left chart of Figure 5.7, the System State Update subroutine is called in every PLC scan cycle regardless of the running status of the sequences and this will generate all the intermediate system states as a sequence is running. The intermediate system states will cause unexpected early termination of the running sequence. If the code is structured as in the right chart of Figure 5.7 this kind of bug won't appear because the system state is only updated at the end of every sequence.

Detection

The following ladder code shows the bug discovered during a system test. The initial system state of the test is NP-AFeedB, the first trigger event is the utility power A failure, the second trigger event is the CB2604 "failed to open" alarm, and the expected

system state is GP-AFeedB. When the first trigger event was enabled by LogixPlcTester, the PLC called the sequence 3 (in scenario 5 of Figure 5.2) as it supposed to. In this sequence, the first step was to open CB102 and CB2604. LogixPlcTester simulated the CB102 open signal but not the CB2604 open signal. As a result, CB2604 was failed to open (this is the second trigger event in this test case). As soon as CB102 was opened, the system state was updated and the new (intermediate) system state was “Plant Dark”. This caused the sequence to be stopped at this step because the following ladder rung in the sequence 3’s subroutine expected the system state to be NP-AFeedB (the system state variable is SwGear.Stat.NP_AFeedB) before it could continue. (The following ladder code is supposed to close CB302 to transit to GP-AFeedB after CB2604 fails to open.) So the system was left as “Plant Dark” when the test was done. This means the plant had no power, which is a serious operation incident in the plant.

Error Analysis

As shown in this example, this kind of defect will be revealed by one of the system model transition tests because the problem caused by the defect is obvious and LogixPlcTester can always capture it by monitoring the System State by an Oracle Tag. This kind of bug could be classified as an iconic error of the type of Early State Update. It can occur in different ways and cause different problems. To prevent it, any system state must be only updated when a sequence is completed so the code will always use the stable and final real-time system state.

5. Sequence conflict

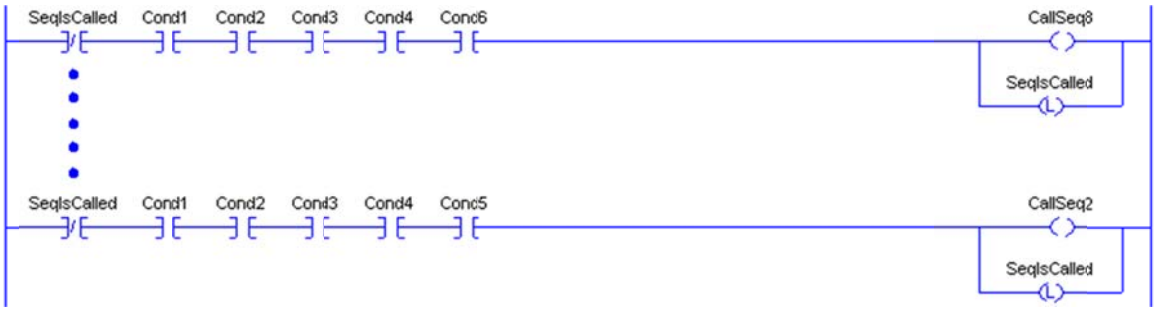
Overview

In the PLC code, a sequence is a series of commands (in a sequential order) to execute a system state transition. In the original design, there were 7 sequences. Later 6 more sequences were added as the development progressed to cover some new scenarios that were missed in the system specification. As the new sequences were added, some of them conflicted with the original sequences. For example, when a certain condition occurs, a new sequence was called unexpectedly instead of calling some original sequence. A sequence is called based on conditions (trigger events). When a new sequence was added the programmer didn't define the trigger event (for the new sequence) to be strict enough to distinguish from the existing sequences' trigger events. As a result, a system state may satisfy multiple sequences' trigger events. In this case, which sequence will be called when such trigger events occur only depends on the locations where the sequence subroutines are called in main routine. The PLC scans the code from top to bottom so the first sequence subroutine that has satisfied trigger condition(s) will be called and the other sequence subroutines that also have their trigger conditions met won't be called because at any time there is only one sequence that can run. Due to the fact that all the condition variables are very similar in the code, it's easy for the programmer to get confused and use the wrong conditions.

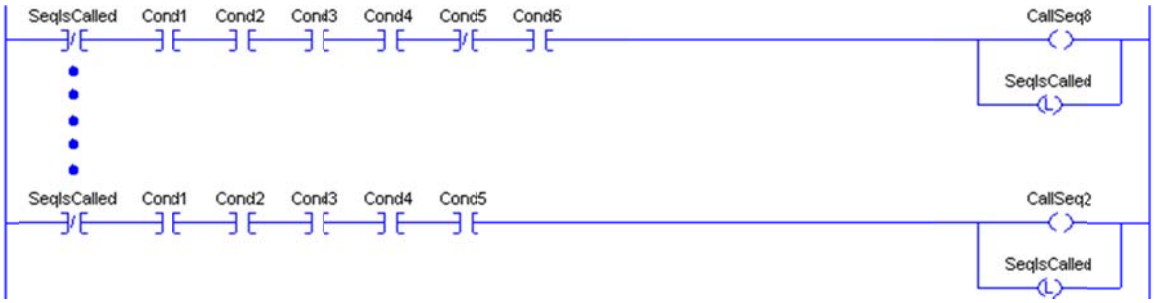
Detection

The original bug was very complicated because it involves multiple subroutines and needs more industrial power operation background to understand. To simplify it, a simple model for the bug is described by the following ladder code to explain the bug.

The variable “SeqIsCalled” is set when there is a running sequence. The variable “CallSeq2” and the variable “CallSeq8” indicate the respective sequence is called to run. The original sequence (the sequence 2) is called when Condition 1 through Condition 5 are met. The new sequence (the sequence 8) is called when Condition 1 through Condition 4 are met and Condition 6 is met. During the test (designed to verify the system transition done by the sequence 2), Condition 1 through Condition 6 are all met. In the test case that tests the sequence 2, the rung of the sequence 8 was examined by the PLC first because its occurrence was earlier than the sequence 2’s occurrence in the ladder code. So the sequence 8 was called and the sequence 2 was not. But the purpose of the test is to exercise the sequence 2. As a result, the wrong sequence was observed in the LogixPlcTester Log Viewer.



The bug in the above ladder code is introduced by a weak trigger condition for a sequence. A possible solution is to tighten the trigger condition for Sequence 8 as follows:



Error Analysis

Sequences get called under certain conditions. At any time there is only one sequence that gets executed. Since every sequence is tested by at least one test case, the presence of a sequence with a wrong trigger condition will be revealed during the system state transition tests. The defect is made worse since it may occur during site testing, with changes being made in real time. It may be a regression error in which there is a solution and a change is made. The old code needs to be completely reanalyzed and tested because of the change, not just the new code. This is an example of how subtle a change is that is made by new code. It looks more like a code addition rather than a change, but the effects can be subtle.

6. Wrong system state

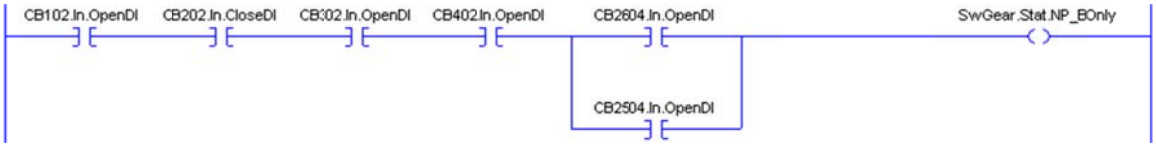
Overview

In the PLC program, the switchgear's system state is the most important variable in the system. All the sequences rely on it to execute successfully. Therefore, the ladder rungs in the "system state update" subroutine are critical. A wrong system state generated form this subroutine can cause the entire system to fail. Since this subroutine handles the calculation of the system state, all the system assets (such as circuit breakers, generators,

and alarms) must be examined to conclude the correct system state. A wrong system state can be produced when an asset is overlooked during the system state calculation.

Detection

The following ladder code is in the subroutine for updating the system state. The subroutine runs once when a sequence is completed. The code has a bug in it. It should have taken a wider and bigger snapshot of the system in order to determine the new state correctly. In this bug, the PLC was only checking the (open/close) states of the breakers to determine the next system state. There are other properties that also affect the system state such as breakers' availabilities, utility power status, generator status, and switchgear alarms.



In order to determine if the switchgear is in the NP_BOnly state, checking the circuit beakers' states is necessary but not sufficient. You also need to check CB202's failure condition in this case. If CB202 "failed to open" alarm is active then the switchgear should be in the Plant Dark state instead of in the NP_BOnly state. The correct version of the ladder code is as follows:

The test that found the bug is designed to test one of the system failure scenarios. The initial system state of this scenario is NP-BFeedA, the first trigger event is the utility power B failure, the second trigger event in this test is the CB202 "failed to open" alarm,

and the expected system state is Plant Dark. During the test, after the first trigger event was initiated, the sequence 3 (in scenario 4 of Figure 5.2) was called. In the sequence, the first step was to open CB202 and CB2604, and the second step was to start both generators. At the first step, LogixPlcTester didn't simulate the CB202 open signal to the PLC resulting in a CB202 "failed to open" alarm (this is the second trigger event). When this alarm was active, the sequence should terminate and the system state should be "Plant Dark". But due to the bug, the wrong system state NB-BOnly was observed in the LogixPlcTester Log Viewer at the end of the test. Since the system state is monitored by an Oracle Tag, any wrong state can be detected in LogixPlcTester.

Error Analysis

A system state variable is maintained to reflect the current system state. The state variable is updated based on various conditions. In some cases the programmer does not know all the information for a correct state because it is not in the system specifications. It could simply be a case of additional details that need to be added as the system is developed. The operators who are familiar with the system know the details that affect the state. During the system development, the programmer should work with the operators to determine for missing information from the system specification.

This kind of bug can be caught early if the programmer asks the operators "what if" questions concerning the additional details. It may not be constructive to do this during the system design or programming. But after the basic system can be constructed, the programmer can revisit and deal with details and unspecified cases. To facilitate this, the programmer needs to document the additional information that is required for programming so that it will not be forgotten later.

Sometimes the programmers need to make some assumptions to push through an initial solution. However, after the initial solution has been acquired, all the assumptions made during the development must be revisited and reexamined. Any wrong assumptions must be fixed.

7. PLC scan direction related error

Overview

There are two different scan styles that PLCs use to scan their programs. One is called Rung Scan, which scans the code form left to right, top to bottom. Another is called Column Scan, which scans the code from top to bottom, left to right. Different scan direction can result in different output from a same code. Figure 5.8 shows a sample code on which the two PLC scan methods lead to different results.

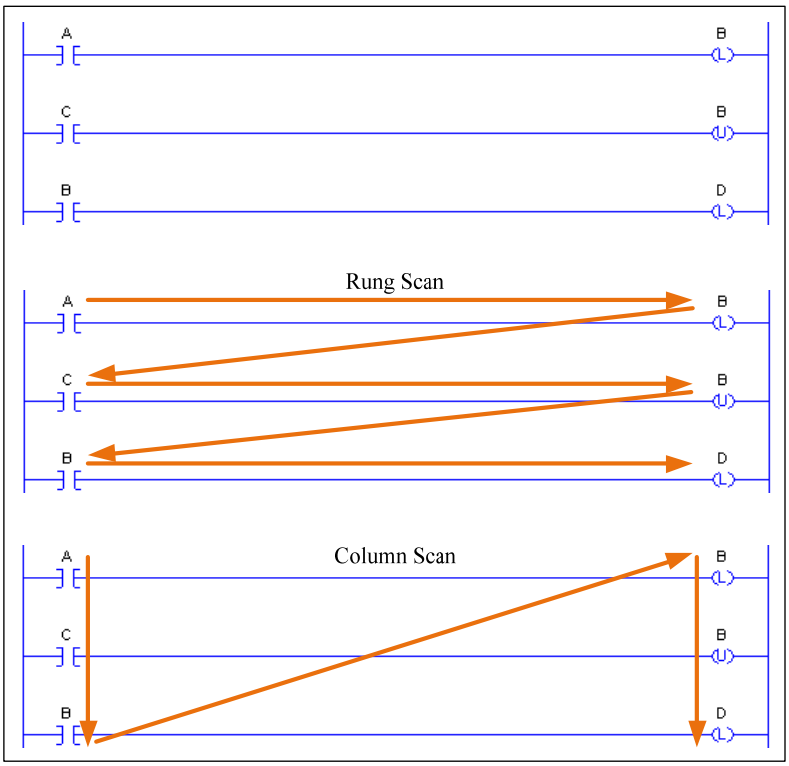


Figure 5.8: A code example for demonstrating scan methods.

The equivalent function in Structured Text language is as follows:

```
if tag_A = true then tag_B := true;
if tag_C = true then tag_B := false;
if tag_B = true then tag_D := true;
```

Table 5.5 lists the results of the above logic when using Rung Scan and Column Scan. If we scan the code with the Rung Scan method the result of the variable D is 0. But if we scan the code with the Column Scan method the result of the variable D is 1. However, if the initial values for (A,B,C,D) changes from (1,1,1,0) to (1,0,1,0) the results of Rung Scan and Column Scan will be the same. So this bug will only reveal when specific inputs are used for testing. A possible test method is to construct a truth table that lists every logical condition as an entry and test all the entries, as in Table 5.5.

Table 5.5: Truth table for Figure 5.5.

	A	B	C	D
Initial Value	1	1	1	0
Rung Scan Result	1	0	1	0
Column Scan Result	1	0	1	1

The bug was in the follow ladder code which is from the sequence 1 (NP-BFeedA) subroutine. The code was written so when the CB2604 “failed to close” alarm is generated at the step 2 of the sequence it will terminate and call another sequence (GP-SplitFeed). When the test case that is designed to test this scenario was running, it was observed in LogixPlcTester that the sequence GP-SplitFeed was never called by the PLC. After checking the code for the permissive conditions for calling the sequence GP-SplitFeed, the bug was identified. Because the PLC uses the Rung Scan method for scanning the ladder code, the first rung is always examined before the second rung. That

means when CB2604 failed to close, the variable “SwGear.Cmd.SetupNP_B_Feed_A” was set to false by the first rung and when the second rung was examined it wouldn’t execute due to the false permissive condition.

Error Analysis

If the PLC scans with the Column Scan method, then this wouldn’t be an issue because in any PLC scan cycle both rungs will read and use the same value for the variable “SwGear.Cmd.SetupNP_B_Feed_A”. This type of bug can be detected during system tests because it usually causes a serious problem in terms of system behaviors and all system behaviors are monitored as Oracle Tags in LogixPlcTester during the tests.

8. Wrong logical relation between timers

Overview

In a sequence, there are usually multiple steps and each step will output several commands to different hardware devices. Each step has a timer to ensure that the step won’t take longer than an allowed time for that step. There is also a timer per hardware device that is used for waiting the feedback from the field hardware device before the PLC sets the “device failed to respond” alarm. The step timer should be always set to be longer than the feedback timer.

Detection

The defect was found during a regression test after a change was made in the code. The programmer made assumptions on the preset values of the feedback timers

during the code development. Before starting to test on real hardware equipment, the programmer was told to increase the feedback timer setting for a generator based on the circuit breaker's response time to control commands. After the change was made, the feedback timer was longer than the step timer. (The programmer forgot to change the step timer's value to be greater than the new feedback timer's value. That caused any sequence in which that particular feedback timer was used to fail every time. In LogixPlcTester, the sequence failure message was logged. The ladder code in Figure 5.9 shows the bug. The first rung defines a 1-second step timer. The second rung sets the sequence failure alarm if the step timer times out. The third rung resets the step timer if the CB102 open feedback signal is received by the PLC while neither the step timer nor the feedback timer has timed out. The last rung defines a 2-second feedback timer for waiting for the CB102 open signal. The feedback timer was set to 0.5 second originally. After it was adjusted to 2 seconds, the programmer forgot to adjust the step timer to be greater than 2 seconds. This bug always led to a failure at this step of the NP-BFeedA sequence.

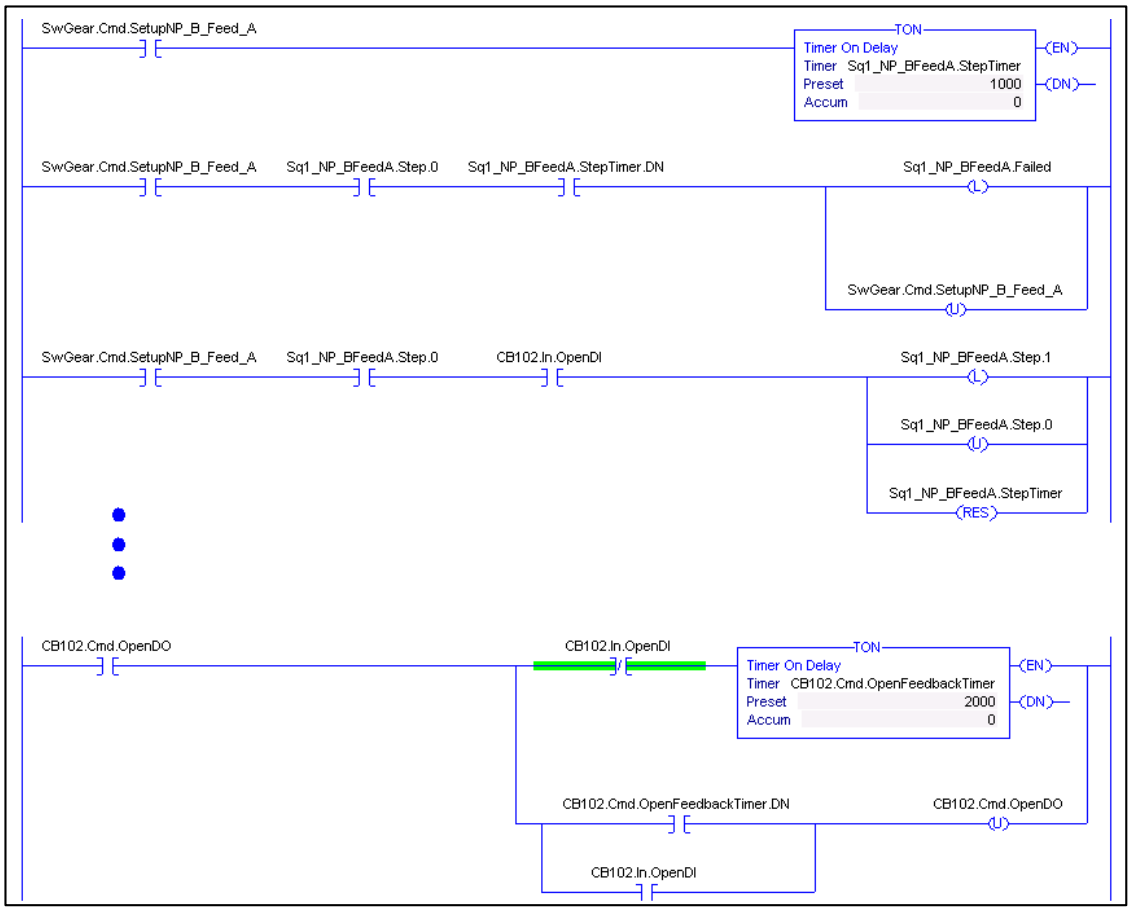


Figure 5.9: A timer related bug in the sequence NP-BFeedA.

Error Analysis

This is a regression error because the same sequence was tested with the same test case and it was working. This kind of bug can be detected by LogixPlcTester because it causes multiple scenarios (that use the defect sequence) to fail and there is at least one test case designed to test each scenario for correctness. This could be labeled as a disturbed invariant error. When any logical relationship involves quantities that are changed, the logical relationship must be analyzed and tested to see if the invariant of the logical relationship has been altered by the change. A PLC programming rule can be added to the checklist to verify the timers' preset values in the PLC program and the

logical relationship between timers. The timer related bugs are very common in PLC programming and some of them are not easy to detect. For example, a hardware device's response time is 10 seconds but it was set to 5 seconds in the PLC program. During the simulation tests, this may not be found because the response time simulated by LogixPlcTester could be always within the threshold (5 seconds). But when conducting the real tests against the hardware device, this bug will reveal because the actual response time from this device is between 5 seconds and 10 seconds. By preparing a checklist with all known PLC issues and using it during the system tests, the known common bugs can be easily sweep out of the PLC code.

9. Confusion between logical AND and logical OR

Overview

In PLC programing, there are two bit instructions called logical AND and logical OR. They perform bitwise AND or OR operations. For example, AND instructions should be used between permission conditions, and OR instructions should be used between alarm conditions. Figure 5.10 shows an example of AND instructions and an example of OR instructions.

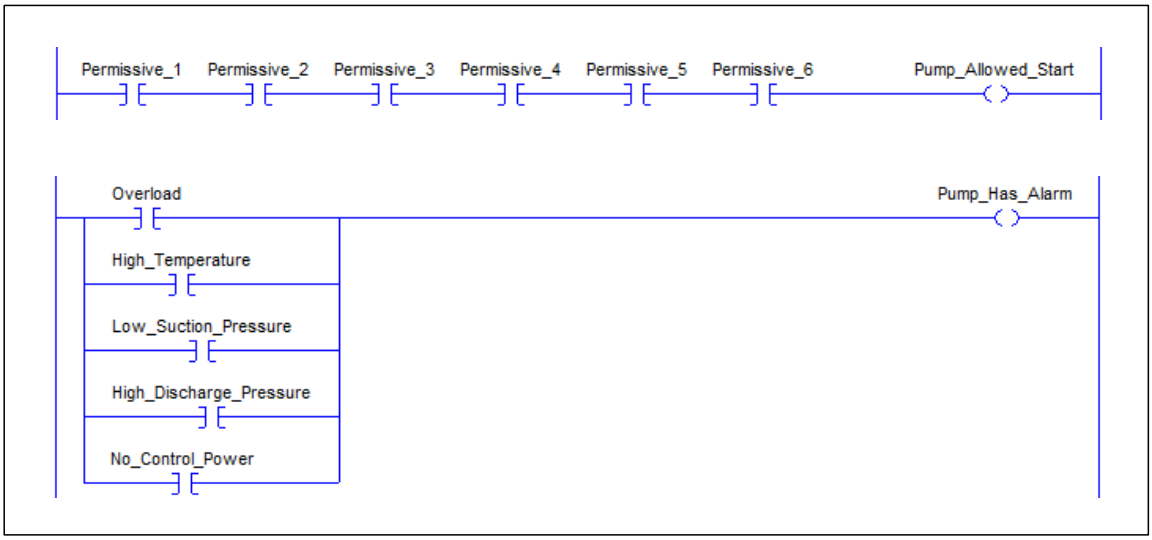


Figure 5.10: Examples of AND and OR instructions.

According to Demorgan's Laws, the logic in Figure 5.10 can also be represented as in Figure 5.11.

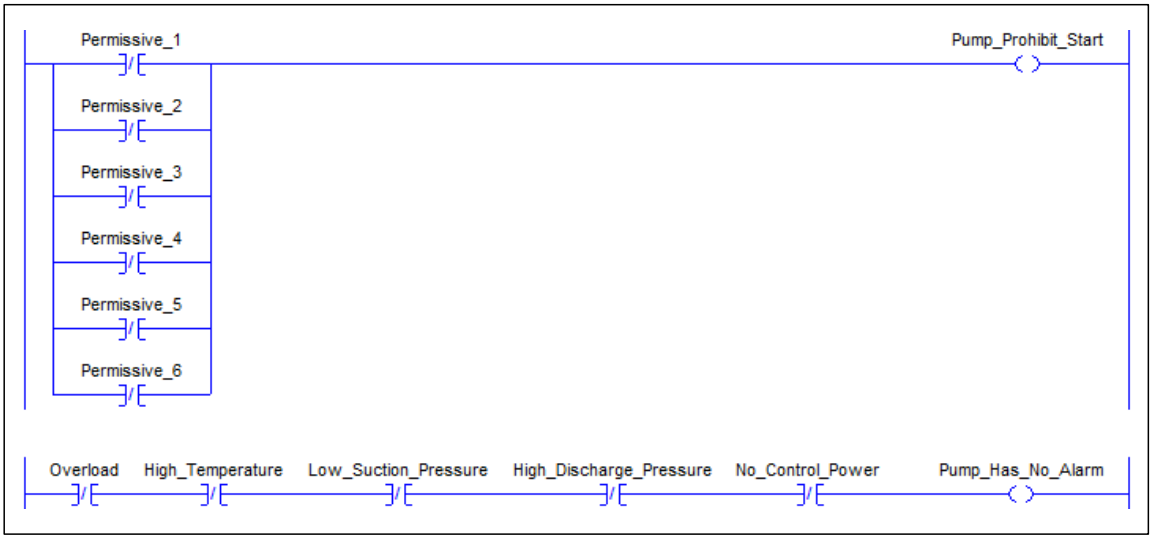


Figure 5.11: Equivalent logic of Figure 5.6 according to Demorgan's Laws.

Detection

Figure 5.12 shows the bug that was discovered during system tests. This logic is meant to set the Boolean variable “SwGear.Stat.SeqRunning” when any of the sequences

is running. (When this variable is set, no new sequence is allowed to run until the current running sequence is done.) However the code in Figure 5.12 obviously violated Demorgan's Laws. Figure 5.13 shows two possible solutions to the problem.

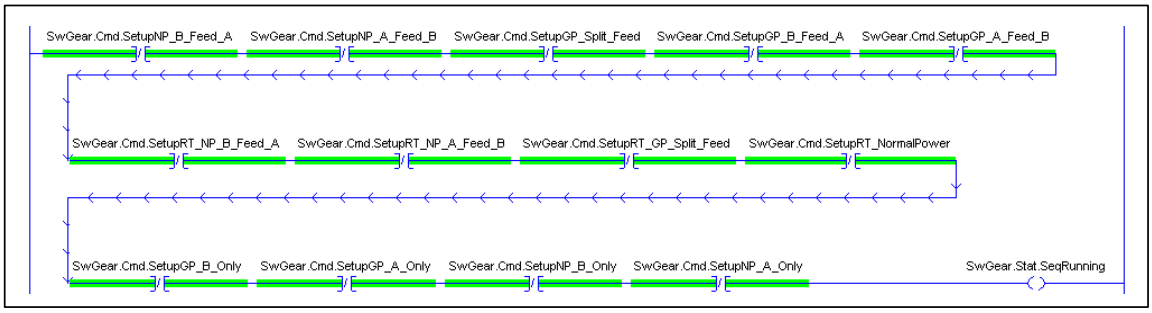


Figure 5.12: A logic that violated Demorgan's Laws.

Figure 5.13: Fixes for the ladder code in Figure 5.12.

This bug was detected by both test cases and code review. Because the function represented by the logic is fundamental and it's used by all the sequence subroutines, it was detected by multiple test cases that test the system state transitions. Due to this bug, when a sequence's trigger event was enabled by LogixPlcTester, the sequence couldn't be started because the variable "SwGear.Stat.SeqRunning" was always true, which was indicating there was a running sequence. As a result, no log messages were observed in LogixPlcTester to indicate the expected sequence was called to run by the PLC.

Error Analysis

Because breaking Demorgan's Laws in PLC programs results in wrong logics in system functions, they are easy to be discovered by some test cases that rely on the system function. Code review method was used in this project to perform an initial screening for bugs in the code and it did find this bug. Code review is usually performed during the software development prior the black-box testing and it's usually performed by another programmer who's familiar with the system specification, the overall structure of the program and its objectives. During code review, the programmer will carefully inspect the logic implemented in the source code rung by rung in order to find errors. A mental simulation of a code execution is helpful to verify the correctness of the functions. Code review will show the location of a bug found so there is no debug process required. Because of the native characteristics of the PLC ladder logic program, it's proved that code review is quite efficient for detecting bugs in PLC programs. However, code review doesn't tend to find more subtle problems in the code because it can only deal with certain level of complexity of the code. Most likely, it won't be able to find a bug that is buried in a complex code such as an algorithm. Code review can be improved with an

error oriented approach. For example, when you're walking through the code you can miss things over and over again such as wrong tag names, wrong comments, counter overflow, incorrect timer presets, etc. But if you do it in iterations, focusing on one type of error at a time, then you may see the errors.

10. Hardware interlock

Overview

There are usually some interlocks built in the hardware to prevent illegal operations (to protect the hardware). The hardware interlocks could be overlooked in the original simulation tests. This is due to the unawareness of the hardware interlocks at the beginning of the tests. They are usually discovered when running the same tests against the real hardware equipment. If we know the interlocks before we start to design the test cases then we can integrate them in the test cases. Missing hardware interlocks could correspond to missing permissive conditions. For instance, the PLC shouldn't try to close a circuit breaker when certain conditions exist. This is an interlock built in the hardware for protective purpose. In the program, this acts as a permissive condition for the breaker's close command. The following ladder code shows a hardware interlock for CB102 close operation. The interlock prevents CB102 from closing when either CB302 lock out relay alarm is active or CB2504 lock out relay alarm is active. Without the interlock, it could potentially cause some hardware damages because it may cause a power loop in the system.

Detection

The above interlock was missing when a test was conducted. The test that found the missing interlock was actually required by a field operator. The test is to verify how the PLC reacts in the scenario where the system state is NP-BFeedA (initial state) and the utility power A is normal (trigger event). It's similar to the scenario 6 of Figure 5.2. But in this scenario, CB302 was simulated to have a lockout relay alarm. It was observed in LogixPlcTester Log Viewer that the PLC called the sequence 9 to close CB102, which was correct if there was no lockout relay alarm on CB302 or CB2504. In this test, the actual expected result was no action from the PLC.

Error Analysis

At the software development and testing stage, the programmer should ask the domain expert – the operator for the hardware interlocks and implement them in the program. For the missed interlocks, most likely they will show themselves during the final acceptance test in the production system. The interlocks should be added in the software program as permissive conditions as they reveal during the final test. Software interlocks and hardware interlocks complement each other. If one fails the hardware is still protected by the other one.

Another lesson learnt from this project is "Difference between designer's mind and equipment's mind". The system designer writes the system specification based on what he knows about the system and what he thinks the system should work. However, the physical equipment may work differently than what says in the specification. As a result, there will be some code changes at the site to remedy the misunderstanding. When you are making changes at the site (with people looking over your shoulder), sometimes

you do not think it through and just patch the code for the current issue. You may introduce some more bugs into the code while patching the code. The sequence conflict bug is a typical example in here. The programmer added a new sequence at the site without considering the impact on the existing sequences resulting in a sequence conflict defect.

In this project, there were three parties involved (engineer/designer, operator, and programmer/tester). The engineer designs the system and writes the system specification. The programmer/tester writes and tests the program based on the system specification. Once the development and testing are done, the operator will test the program against the real equipment to verify if the program works in the production system and meets operation standards. At this step, some discrepancies between the system design and the way the real equipment actually works will be exposed. In this case, onsite code changes are inevitable. As a result, a regression test has to be conducted. Sometimes the engineer and the operator may make contradictory demands to the programmer for changes to the code and in that case the same code can be changed back and forth multiple times.

Chapter 6. Related Work

Allen Bradley has a software product called RSTestStand [9], which enables control system developers to create virtual control system scenarios that can be used to test design configurations and programs. RSTestStand allows you to develop and test your control program in an offline system by simulating the field inputs and outputs signals. RSTestStand uses industry standard OPC (OLE for Process Control) protocol to communicate with Allen-Bradley PLCs using RSLinx. RSLinx is another software product from Allen Bradley that is a communication server providing plant floor device connectivity to support Allen Bradley software applications such as RSLogix 5000 (the Allen Bradley Logix PLC programming software). In addition, RSLinx is an OPC compliant data server supporting the OPC Data Access 2.05 specifications. In order for RSTestStand to work, RSLinx must be installed on the same computer where RSTestStand resides. In the communication link between RSTestStand and the PLC, RSTestStand acts as an OPC client and RSLinx acts as an OPC server. RSTestStand reads and writes PLC tag values by inquiring the OPC server (RSLinx)

Similar to a hardware test stand, RSTestStand provides you with a range of devices like push buttons and pilot lights to interact with your logic program. Figure 6.1 shows the RSTestStand operator console, which is the runtime interface equipped with buttons, selector switches, and pilot lights used by an operator to interact with the logic program test. The operator console is similar to the Control Graph feature in LogixTester.

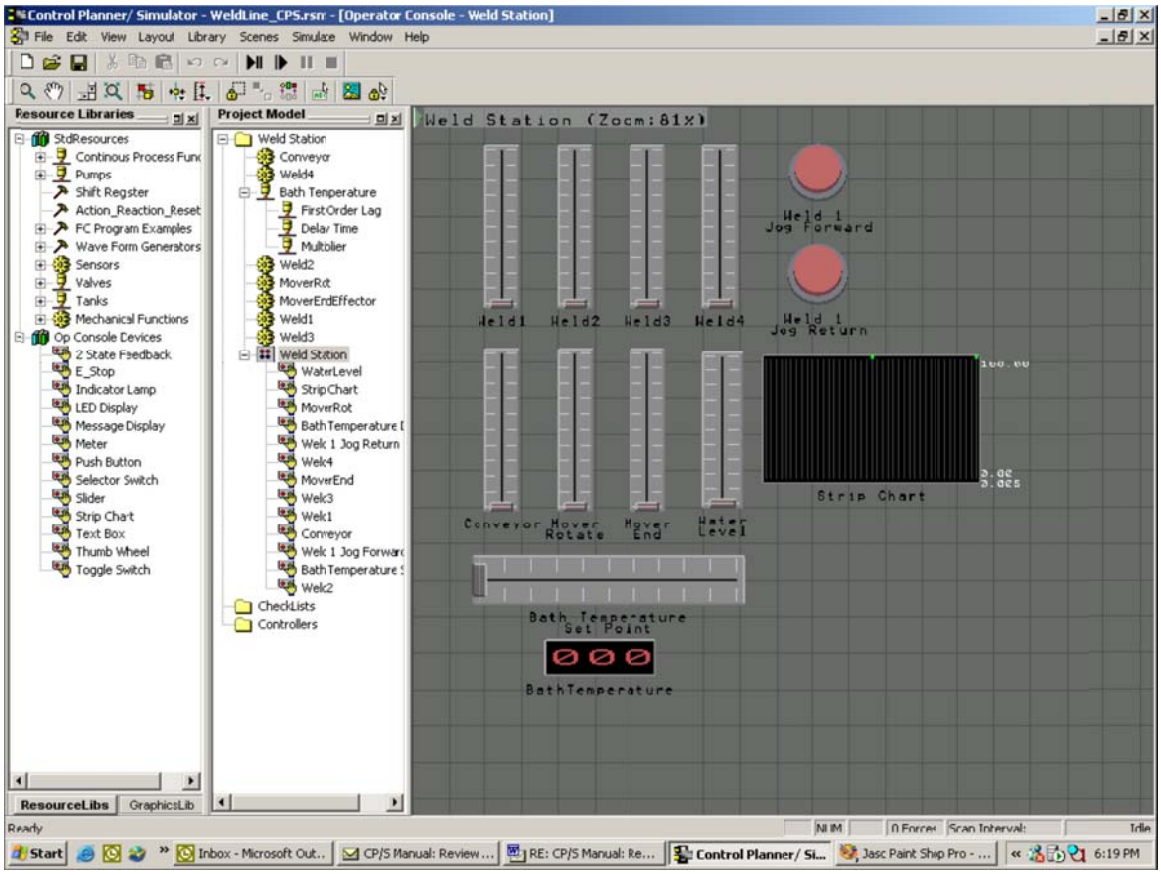


Figure 6.1: RSTestStand operator console.

Like in LogixPlcTester, RSTestStand also uses tags to exchange information with the PLC. RSTestStand uses flowcharts to control the behaviors of the tags. A flowchart is a graphical interface for programming RSTestStand. Figure 6.2 shows an example of RSTestStand flowchart. A test case can be created using a flowchart in RSTestStand. But it won't give users the flexibility of defining time based triggers or condition based triggers as in LogixPlcTester. RSTestStand doesn't provide an explicit log viewer that shows the sequence of events while a test case is running. Since RSTestStand uses OPC as its communication protocol, its communication resolution is limited by the OPC standard. The typical OPC resolution is 1 second. LogixPlcTester uses the PLC's native

communication protocol (Ethernet/IP) to communicate with the PLC so that its resolution is optimal.

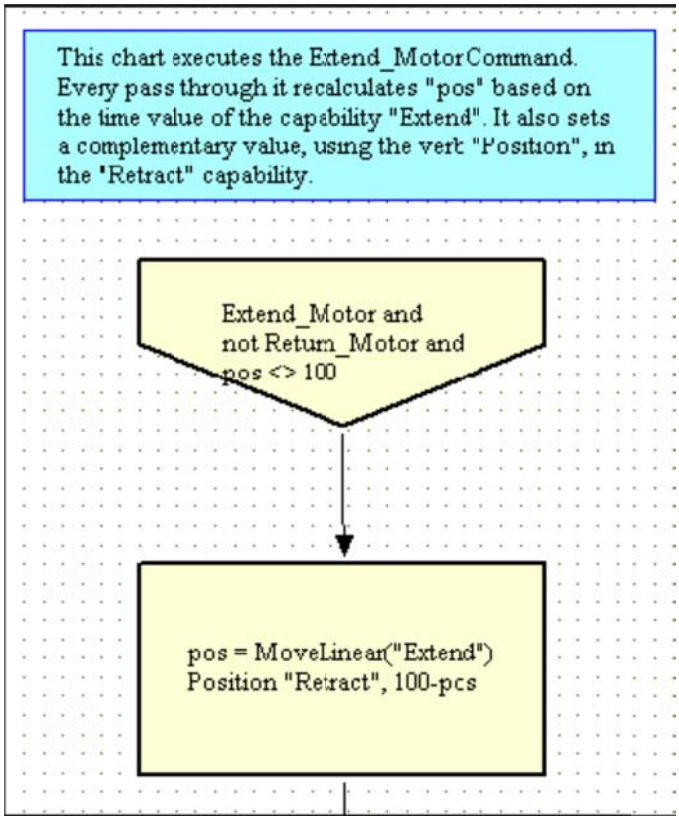


Figure 6.2: RSTestStand flowchart.

Similar to LogixPlcTester, RSTestStand can train operators and technicians in a virtual environment. This approach doesn't require expensive equipment in the plant and it provides a cost-effective and system-safe way to understand the system operations prior to installation.

Siemens' S7-PLCSIM [10] and Allen Bradley's RSLogix Emulate 5000 [11] are another type of PLC simulator tool. They both focus on PLC controller hardware simulation rather than I/O simulation. They can run PLC programs in their software simulators without the actual PLC hardware. They can simulate most of the functions that

a real PLC has. You can debug your program in the software simulator. They are very useful when you want to develop and debug programs without the PLC hardware. But they don’t provide any automated mechanism for I/O simulation. The only way you can simulate I/O points in them is to manually change values in the System Data Reference Table. However, when the PLC controller simulators [10][11] are used with LogixPlcTester together, a pure software testing environment is formed, which requires no PLCs during the entire software testing process. It will be extremely beneficial in terms of costs when multiple testers are working in the system. With this solution, every tester can have a dedicated test environment (a virtual PLC and a virtual test stand for I/O simulation) other than sharing one test environment with others. This will eliminate the interference between testers and their ongoing tests.

Chapter 7. Conclusion and Future Work

7.1 CONCLUSION

The automated testing tool, LogixPlcTester, was designed to test software programs running in Allen Bradley Logix PLCs based on test cases. It acts as a virtual device that can respond to the PLC's commands by writing feedbacks to the PLC. With LogixPlcTester, the testing process can start early in the software development cycle. As a result, software bugs can be discovered at an earlier stage than usual and this makes bug fixing easier and faster. The project development costs can be greatly reduced and stable and high quality programs can be produced. A control program that has been precisely tested by LogixPlcTester in a development environment will make the final onsite testing easier and faster in the production system. Unnecessary hardware equipment damage due to software bugs can be prevented. And the system startup process will become much smoother. Since all the tests can be conducted in a simulation system using LogixPlcTester, only the final acceptance test need to be run on the real hardware system. As a result, the hardware equipment's operation life time can be extended.
LogixPlcTester is a powerful and easy to use automated PLC testing tool that helps building and assuring quality into industrial control programs. LogixPlcTester allows testers to build test cases quickly with a built-in configurator that requires no programming for building test cases. The goal of LogixPlcTester is to provide an effective and time-saving test environment for testing PLC programs to identify potential problems with the PLC programs and eliminate the identified bugs with regression tests

so that high quality programs can be attained before they are downloaded into production systems and start controlling hardware equipment. The goal was achieved in the Emergency Generator Control project.

7.2 Future Work

Currently the time resolution that LogixPlcTester supports is 250 milliseconds meaning any signal changes in the PLC that are faster than 250 milliseconds won't be captured by LogixPlcTester. A time resolution of 250 milliseconds is actually good enough for most of the PLC control programs because the field devices normally have a response time longer than 250 milliseconds. However in some time-critical systems where higher time resolution is required, LogixPlcTester may not produce feedbacks to the PLC fast enough in order to meet the response time requirement. A temporary work-around for this is to enlarge the feedback timers in the PLC so that the LogixPlcTester test cases can be passed as far as the response time is concerned. The work of improving LogixPlcTester's time solution has been started. The code for reading tag values from PLC is being rewritten to improve efficiency. The optimal goal of time resolution for LogixPlcTester is 100 milliseconds.

Hardware interlocks are usually overlooked during software simulation testing. In industrial control systems' testing, hardware interlocks should be put as a rule in the checklist. Testers should always check with domain experts about hardware interlock while creating test cases. All hardware interlocks should be implemented in software for redundant protection.

In the current version of LogixPlcTester, log messages are the only way to verify a test result. Timestamped log messages show the sequence of actions issued by the PLC and the sequence of responses sent from LogixPlcTester. Testers need to review the log messages in order to determine if the test is passed. Although the log messages clearly record the entire test process, sometimes testers misread the log messages resulting in incorrect test results. Even with a graphical control screen supported by LogixPlcTester, testers' interactions are still required. A potential solution is to have LogixPlcTester be able to read a temporal logic specification and use it to automatically verify the result of a test while it's running. This temporal logic specification will act as a "oracle" for the test case. With this feature, LogixPlcTester will generate an explicit result of a test case (either passed or failed). Currently this new feature is under feasibility evaluation.

A batch test mode will be added to process multiple test cases in a predefined sequential order automatically. Currently a test case is loaded manually in LogixPlcTester. When a test case is done, a new test case has to be loaded in LogixPlcTester manually. With the batch mode support, testers just need to specify the location of the test cases that need to run and then initiate the start of tests. After that, LogixPlcTester will automatically execute the tests in the following order: load a test, run the test, unload the test, load a new test, run the new test… until all the tests are done. Testers only need to review the log messages after all test cases are complete. With the temporal logic specification feature, this would become even easier. Basically what testers would get is the final test reports. They would no longer review the log messages from the tests because LogixPlcTester already verified the test results by using the temporal logic specifications.

References

[1] Allen Bradley ControlLogix System User Manual. Rockwell Automation Publication 1756-UM001L-EN-P. 2011.

[2] Allen Bradley Logix5000 Controllers General Instructions. Rockwell Automation Publication 1756-RM003M-EN-P. 2010

[3] Allen Bradley Logix5000 Controllers Function Block Diagram Programming Manual. Rockwell Automation Publication 1756-PM009C-EN-P. 2009

[4] Allen Bradley Logix5000 Controllers Sequential Function Charts Programming Manual. Rockwell Automation Publication 1756-PM006C-EN-P. 2009

[5] Allen Bradley Logix5000 Controllers Structured Text Programming Manual. Rockwell Automation Publication 1756-PM007C-EN-P. 2009

[6] Allen Bradley PanelView Plus Terminals User Manual. Rockwell Automation Publication 2711P-UM001J-EN-P. 2009

[7] EtherNet/IP Technology Overview. 2011. http://www.odva.org.

[8] GLG User's Guide and Builder Reference Manual.2011. http://www.genlogic.com.

[9] RSTestStand Getting Results Guide. Rockwell Automation Publication TSTENT-GR001A-EN-P. 2004

[10] Siemens' Simatic S7-PLCSIM User's Manual. 2007

[11] Allen Bradley RSLogix Emulate 5000 Getting Results Guide. Rockwell Automation Publication LGEM5K-GR015A-EN-P. 2005

Appendix A. PLC Program Structure Diagram