

Automated Translation of Safety Critical Application Software Specifications into PLC Ladder Logic

Kurt W. Leucht, Glenn S. Semmel
National Aeronautics and Space Administration
Kennedy Space Center, Florida 32899, USA
Kurt.W.Leucht@nasa.gov, Glenn.S.Semmel@nasa.gov

Abstract—The numerous benefits of automatic application code generation are widely accepted within the software engineering community. A few of these benefits include raising the abstraction level of application programming, shorter product development time, lower maintenance costs, and increased code quality and consistency. Surprisingly, code generation concepts have not yet found wide acceptance and use in the field of programmable logic controller (PLC) software development.

Software engineers at the NASA Kennedy Space Center (KSC) recognized the need for PLC code generation while developing their new ground checkout and launch processing system. They developed a process and a prototype software tool that automatically translates a high-level representation or specification of safety critical application software into ladder logic that executes on a PLC. This process and tool are expected to increase the reliability of the PLC code over that which is written manually, and may even lower life-cycle costs and shorten the development schedule of the new control system at KSC.

This paper examines the problem domain and discusses the process and software tool that were prototyped by the KSC software engineers.^{1,2}

TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. APPLICATION	5
3. CONCLUSIONS AND FUTURE WORK	12
ACRONYMS.....	13
REFERENCES	13
BIOGRAPHY.....	14

1. INTRODUCTION

A New Direction

NASA Kennedy Space Center (KSC) engineers are responsible for pre-launch ground checkout of the Space Shuttle and associated ground support equipment (GSE). On January 14, 2004, the President of the United States announced a new vision for space exploration which changed NASA's goals from low earth orbit operations to lunar operations and beyond [1]. In response to the presidential announcement, NASA formulated what is called

the Constellation program to develop a new generation of spacecraft and infrastructure for return to the Moon [2]. A concept image of the new spacecraft is shown in Figure 1. At KSC, where these new space vehicles will be launched, a new ground checkout and launch processing system is currently being developed in support of this effort.

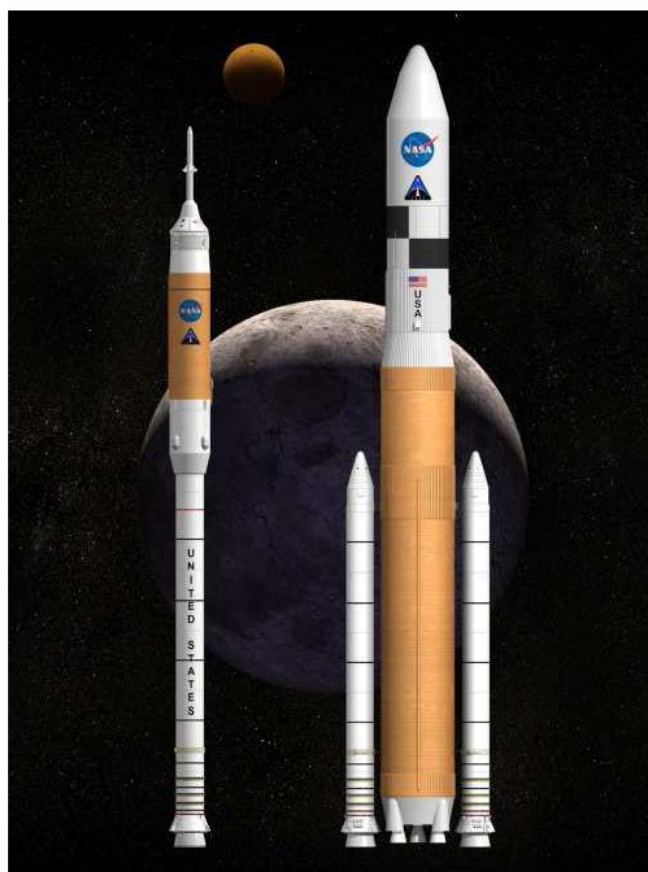


Figure 1 - Artists concept of NASA's next-generation launch vehicle systems.

Image credit: www.nasa.gov

Many large and complex systems that have been implemented by NASA in the past, including previous ground checkout and launch processing systems, contained functionality or performance requirements and specifications that forced custom solutions. As software processes and technologies have matured, commercial off the shelf (COTS) hardware and software components have become more reliable and provide higher performance. Thus, NASA expects to save development schedule and cost by integrating various COTS solutions together in order to

¹U.S. Government work not protected by U.S. copyright.

²IEEEAC paper #1402, Version 5, Updated November 26, 2007

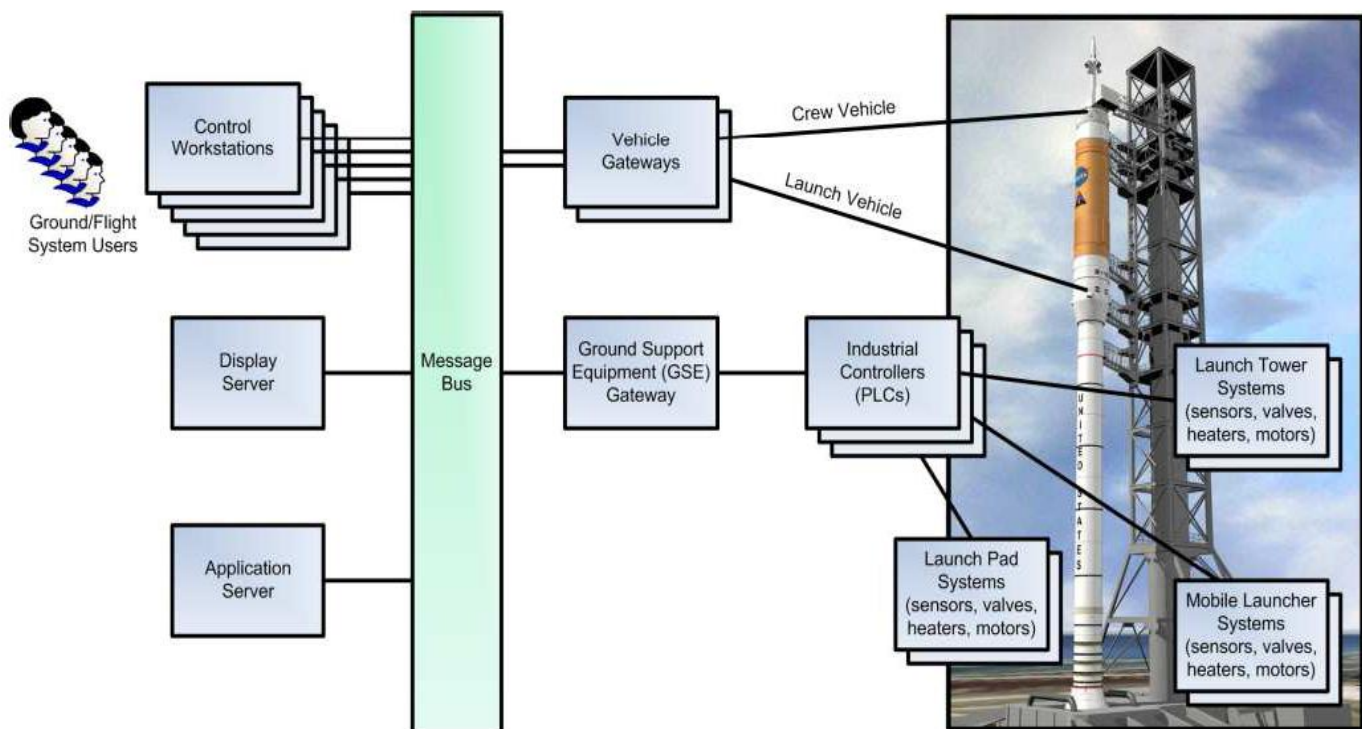


Figure 2 - Simplified Architecture of KSC's Launch Control System (LCS).

Rocket concept image credit: www.nasa.gov

implement large and complex systems. COTS solutions can generally offer significant life-cycle cost savings.³

A simplified high level architecture of the new ground checkout and launch processing system is shown in Figure 2. This system is called the Launch Control System or LCS. All the computer hardware in the LCS is planned to be COTS, including Industrial Controllers or PLCs that are connected to the sensors and end items out in the field. A significant portion of the software in the LCS is also planned to be COTS, with only small adapter software modules that must be developed in order to interface between the various COTS software products.

PLCs are basically environmentally ruggedized computers that are specifically designed to automate or control a process in a factory or plant. PLCs are commonly distributed throughout a factory or plant near the motors, valves, heaters, etc. they are controlling and near the sensors from which they are reading values. Control logic or control software typically executes in a PLC to read data or state values from input channels, and to control output channels accordingly. KSC is not technically a factory or plant, but it is an industrial type of environment that is well suited for industrial (e.g., PLC) control of end items located in the field.

In the launch vehicle and spacecraft processing domain the control logic in the PLCs that monitors and controls high energy equipment or machines, such as pressurized fuel tanks, is categorized as safety critical. A safety critical component or system is defined as one whose failure can cause injury or death.

Application Software

Application software is the high level layer of computer software that the end user actually interacts with. It allows the user to perform specific and productive tasks on the computer. Some common examples of application software are word processors, database programs, and media players. Application software is usually contrasted with system software which is the low level layer that interacts directly with the computer at a very basic level (e.g., the hardware level, the driver level).

In the LCS, application software is the set of software programs conceived, written, and executed by the user that is intended to monitor specific end item measurements and user inputs and use that information to control the user's remote system located in the field. The LCS architecture allows application software functionality to be distributed between an application server in the control room and the PLCs out in the field. The LCS will likely distribute the time-critical closed loop control functionality and the low level end item command and response functionality in the field-located PLCs and house the higher level supervisory control and sequencing functionality in the application servers located in the control room.

²

³ Risks associated with using COTS products, such as vendor dependency, vendor propriety, product quality, product support, and code ownership were considered and are beyond the scope of this paper.

Figure 3 shows a typical example of application software source code⁴ that might be found in the existing legacy ground processing system at KSC. This small portion of code, written in a custom high level test procedure language called Ground Operations Aerospace Language (GOAL) [3], commands a valve to the open state and then looks for the appropriate valve position indications within the appropriate time constraints.

```
$      SEND PRIMARY OPEN COMMAND $
      TURN ON <VLV1_PRI_OPEN_CMD>;

$      DELAYS UP TO 8 SEC FOR INDICATION OF MOTION. STEP 20
      LOOPS TO STEP 10 UNTIL CLOSED INDICATOR TURNS OFF OR
      8 SEC PASSES. $
      READ <GMT> AND SAVE AS (GMT1);
STEP 10 READ <GMT> AND SAVE AS (GMT2);
      LET (VLVTM) = (GMT2) - (GMT1);

$      ADD 24 HOURS IF DAY WRAPS $
      IF (VLVTM) IS LESS THAN 0.0 SEC,
      LET (VLVTM) = (VLVTM) + 86400 SEC;

      IF (VLVTM) IS LESS THAN OR EQUAL TO 8 SEC
      THEN GO TO STEP 20;

$      ERROR MESSAGE $
      RECORD <GMT>,
      TEXT ( VALVE1 INITIAL MOTION GREATER ),
      TEXT (THAN 8 SEC) TO <PAGE-A> YELLOW TO <CNLS-PP>;

$      OPEN VALVE1 USING SECONDARY COMMAND & VIEW SECONDARY
      INDICATORS $
      PERFORM PROGRAM (OpenValve1Secondary);
      GO TO STEP 100; $ DONE $

STEP 20 VERIFY <VLV1_PRI_CLOSED_IND> IS OFF
      ELSE GO TO STEP 10;

$      DELAYS UP TO 26 SEC FOR INDICATION OF MOTION COMPLETE.
      STEP 40 LOOPS TO STEP 30 UNTIL OPEN INDICATOR TURNS
      ON OR 26 SEC PASSES. $

STEP 30 READ <GMT> AND SAVE AS (GMT2);
      LET (VLVTM) = (GMT2) - (GMT1);

$      ADD 24 HOURS IF DAY WRAPS $
      IF (VLVTM) IS LESS THAN 0.0 SEC,
      LET (VLVTM) = (VLVTM) + 86400 SEC;

      IF (VLVTM) IS LESS THAN OR EQUAL TO 26 SEC
      THEN GO TO STEP 40;

$      ERROR MESSAGE $
      RECORD <GMT>,
      TEXT ( VALVE1 OPEN TIME EXCEEDED LIMIT)
      TO <PAGE-A> YELLOW TO <CNLS-PP>;

$      OPEN VALVE1 USING SECONDARY COMMAND AND
      VIEW SECONDARY INDICATORS $
      PERFORM PROGRAM (OpenValve1Secondary);
      GO TO STEP 100; $ DONE $

STEP 40 VERIFY <VLV1_PRI_OPEN_IND> IS ON
      ELSE GO TO STEP 30;

$      SUCCESS MESSAGE $
STEP 50 RECORD <GMT>,
      TEXT ( VALVE1 OPEN TIME IS ), (VLVTM)
      TO <PAGE-B> TO <CNLS-PP>;

STEP 100 TERMINATE;
```

Figure 3 - Typical example of existing legacy application software source code

It is evident from the above code example that software programming skills are necessary to write application software in the existing legacy ground launch processing system at KSC. This necessity led to the formation of a group of programmers at KSC that specifically takes

requirements from ground and flight system users and writes the application software code. Hiring and maintaining a group of dedicated application software programmers was chosen over the option of training all the ground and flight system users to simultaneously be software developers. But this decision has turned out to be quite costly over the total life-cycle of the legacy control system so alternative processes are being investigated for the LCS.

A Language Of Their Own

The developers of the LCS are investigating processes and tools that will allow non programmers (i.e., system engineers, domain experts and end users who are not software savvy) to write safety critical control applications using a high level representation or format. This high level representation is actually considered a “model” of the control system from the perspective of the Model-Driven Software Development (MDSO) discipline [4]. MDSO uses domain-specific abstractions or domain-specific languages to formulate the model of a system that is being developed.

A domain-specific language (DSL) is a programming language that is designed to perform tasks and to solve problems in a particular domain, such as operating a power plant or processing launch vehicles. For example, UNIX® shell scripts are a good example of a DSL for data organization because they are very useful for taking user input and manipulating data in files. Although powerful and robust, UNIX® shell scripts are not very conducive for creating complex data structures, such as lists and trees, and most do not support object oriented design [5].

A DSL has been created by the LCS Application Services Product Group for developing test sequences of ground checkout and launch operations of Constellation vehicle elements [6]. Figure 4 shows the software functionality that was previously shown in Figure 3 for opening a valve, but it is now written in the new DSL that was created for the LCS. This example is a simplified snippet from a much larger and more complex DSL-based program that was recently developed as an LCS prototype user application.

```
# SEND THE PRIMARY OPEN COMMAND
send_command(discrete("VLV1_PRI_OPEN_CMD", "ON"))

# VERIFY THE PRIMARY INDICATORS CHANGE APPROPRIATELY WITHIN
# THE REQUIRED TIME CONSTRAINTS
if verify_within_voting (2,
    [lambda : read.VLV1_PRI_CLOSED_IND == OFF,
    lambda : read.VLV1_PRI_OPEN_IND == ON],
    ["VLV1_PRI_CLOSED_IND", "VLV1_PRI_OPEN_IND"],
    [8, 26], DIALOG,
    "VLV1_PRI_CLOSED_IND and VLV1_PRI_OPEN_IND") > 0 :

    # A FAILURE OCCURRED. GENERATE ERROR MESSAGE.
    send_message ("VALVE1 PRIMARY COMMAND FAILED",
        SYS1, WARNING)

    # USE SECONDARY COMMAND & SECONDARY INDICATORS
    perform("OpenValve1Secondary", [], BLOCKING)

# SUCCESS. GENERATE MESSAGE TO USER.
send_message ("VALVE1 OPENED SUCCESSFULLY", SYS1, INFORMATION)
```

Figure 4 - Typical example of new application software source code using a DSL

³ _____
⁴ Due to International Traffic in Arms Regulations (ITAR) restrictions, all source code examples have been simplified from their original form and all Space Shuttle program specific information has been removed.

The DSL code that was created for the LCS uses keywords and functions that are familiar to the ground and flight system user, such as “send command”, “send message”, and “voted verify within”. The “voted verify within” scenario will now be explained briefly as it will be used later in numerous examples.

In the spacecraft ground processing domain, a simple “verify within” operation takes a single measurement (e.g., a valve position indicator) and verifies that it changes to a specified or expected value (e.g., OPEN, CLOSED, ON, OFF, etc.) within a specified time period. A “voted verify within” operation is similar to the “verify within” operation, except that it takes multiple measurement parameters instead of just one and it also takes an argument for the total number of voted measurement parameters that must be true in order for the whole voted verify within operation to be true.

This voted verification functionality is very useful in safety critical control systems that utilize a lot of redundancy or duplication of commands and/or measurements. A ground or flight system might contain three separate measurements that all give the state of a single end item, such as the position of a valve. However, the ground or flight system user might consider the valve to be successfully opened and operating within specifications if one of the three indicator measurements was failed and only two of the three actually indicated that the valve was open. This is a “2 of 3 voting” indicator scenario and is quite common in this domain.

The DSL code encapsulates the programming details of the voted verify within operation and attempts to elevate the programming language to the abstraction and level of understanding within the ground and flight system users domain. Unfortunately, some software programming skills

are still necessary to write application software using this new DSL. So the developers of the LCS investigated other ways to represent the application software and is currently developing a tabular specification format that uses the DSL keywords and functions that are familiar to the ground and flight system users. The tabular specification format, or tabular spec, allows most ground and flight system users to document how the application software is intended to function and requires little or no software programming knowledge or experience.

Figures 5, 6 and 7 show small portions of application software tabular spec from the LCS prototype effort. The sample in Figure 5 demonstrates the same application software functionality that was previously shown in Figures 3 and 4. It commands a valve to the open state and then looks for the appropriate valve position indicators within the appropriate time constraints. The sample in Figure 6 configures the LCS to start monitoring the pressure in a tank and to react appropriately when the pressure goes below or above some specific thresholds. The sample in Figure 7 sends a text message to the user console in order to notify the user of an important event.

It is evident from these three simple tabular spec examples that software programming skills are no longer required to write application software in the LCS. Representing application software in this tabular spec format has many advantages. It is a high level semantic that is conducive for end users who are not software savvy, to create their applications themselves. This new representation of the application software is smaller in size and is also less complex than the prior representations which equates to a productivity increase by comparison.

LINE	ROUTINE	DSL API	OBJECT(S)	DESCRIPTION/MESSAGE	LO/VAL	HI	VOTING	DURATION	REACTION
1	# Routine sends primary OPEN command and waits for appropriate indicators								
2	OpenValve1Primary	# Send primary OPEN command							
3		send_command	VLV1_PRI_OPEN_CMD	Valve1 Primary Open Command	ON				
4									
5		# Verify both primary indicators change appropriately within appropriate time durations, on failure call another routine to perform Secondary Open Command							
6		verify_within	VLV1_PRI_CLOSED_IND	Valve1 Primary Closed Indicator	OFF		2 of 2	8 sec	
7		verify_within	VLV1_PRI_OPEN_IND	Valve1 Primary Open Indicator	ON		2 of 2	26 sec	OpenValve1Secondary
8	end								
9									

Figure 5 – Typical example of new LCS application software tabular spec – Opening a valve

LINE	ROUTINE	DSL API	OBJECT(S)	DESCRIPTION/MESSAGE	LO/VAL	HI	VOTING	DURATION	REACTION
41	# Routine starts assert monitoring for high or low tank pressures and opens and closes vaporizer valve appropriately								
42	StartAutoTankPress	# Monitor for high pressure							
43		assert_constraint	TANK_PRESSURE_1	Tank Pressure Indicator #1	15		1 of 3		
44		assert_constraint	TANK_PRESSURE_2	Tank Pressure Indicator #2	15		1 of 3		
45		assert_constraint	TANK_PRESSURE_3	Tank Pressure Indicator #3	15		1 of 3		OpenVaporizerValve
46									
47		# Monitor for low pressure							
48		assert_constraint	TANK_PRESSURE_1	Tank Pressure Indicator #1	25		1 of 3		
49		assert_constraint	TANK_PRESSURE_2	Tank Pressure Indicator #2	25		1 of 3		
50		assert_constraint	TANK_PRESSURE_3	Tank Pressure Indicator #3	25		1 of 3		CloseVaporizerValve
51	end								
52									

Figure 6 - Typical example of new LCS application software tabular spec – Monitoring pressure in a tank

LINE	ROUTINE	DSL API	OBJECT(S)	DESCRIPTION/MESSAGE	LO/VAL	HI	VOTING	DURATION	REACTION
21	# Routine notifies user of failure								
22	OpenValve1Error	# Send a failure message to the user							
23		send_message		Unable to open Valve1					
24	end								
25									

Figure 7 - Typical example of new LCS application software tabular spec – Sending a user a text message

The tabular spec also contains inherent tracing from user requirements to implementation and minimizes the learning curve for capturing application requirements and brings consistency to the process. It also improves the quality of the application requirements with a consistent, structured format. The use of a DSL along with this tabular spec allows the domain expert or end user to focus on solving the domain problem rather than focusing on the software.

The LCS developers realize that this tabular spec format is limited in features and capabilities as compared to the prose programming approach shown in Figure 4, but many existing application software examples from the ground processing domain have already successfully been represented in this tabular spec format. Even if only half of the total user application software can be represented in tabular spec format, that equates to a significant improvement in cost, schedule and manpower necessary to develop, implement and maintain the LCS. Anecdotal evidence suggests that up to 80% of the existing Space Shuttle program user application software might be successfully represented in the new tabular spec format. We are currently investigating this estimate and hope to produce empirical data to further refine it. This level of applicability could translate into significant cost savings for the Constellation program which reuses some legacy Space Shuttle program elements, such as Solid Rocket Boosters.

Problem Description

The LCS developers needed a mechanism or tool to translate application software from tabular spec format into PLC code to execute on the PLC platforms out in the field. The LCS developers were tasked to investigate the possibility of manually and automatically creating the application software portion of the PLC code from the tabular spec representation. A portion of some legacy application software that performs the task of loading fluids into a tank was represented in tabular spec format by a team of NASA KSC ground system engineers. This representative sample tabular spec was used as the starting point for both the

manual and the automatic PLC code creation process.

This work was considered a prototype effort and was performed for the sole purpose of determining if it was even possible to automatically create PLC code from a high level representation of the application software. The tabular spec format (Figures 5, 6, 7) was chosen for this prototype effort over the DSL prose format (Figure 4) for multiple reasons, but mostly because representing user application software in the tabular spec format has many cost and schedule benefits over the DSL prose format. Also, the tabular spec format is simpler and more structured and should lend itself better to parsing and translation.

In the LCS, the application software will be distributed between servers that are located locally in the control room and PLCs that are located remotely out in the field. This paper focuses only on the portion of the application software functionality that is expected to execute on the PLCs out in the field, and not the portion that is expected to execute in the control room.

2. APPLICATION

Manual Translation

A few small executable sections from the previously mentioned tank loading tabular spec were selected for manual translation into PLC code. The first selection was made up of three subroutines, shown in Figure 8, that attempt to open a valve allowing fluid to flow in a pipe. The first subroutine attempts to open the valve using the primary command and response path. The second subroutine is called from the first if the valve fails to open. This second subroutine attempts to open the same valve using the secondary or backup command and response path. The third subroutine is called from the second if the valve fails to open again on the second attempt. This third subroutine generates an error message to the user.

Program Name: ValveOpeningSequence		Prog Description: This program demonstrates sending a command and verifying discrete indicators. Uses multiple routines with error paths.							
LINE	ROUTINE	DSL API	OBJECT(S)	DESCRIPTION/MESSAGE	LD-VAL	HI	VOTING	DURATION	REACTION
1	# Routine sends primary OPEN command and waits for appropriate indicators								
2	OpenValve1Primary	# Send primary OPEN command							
3		send_command	VLV1_PRI_OPEN_CMD	Valve1 Primary Open Command	ON				
4									
5	# Verify both primary indicators change appropriately within appropriate time durations, on failure call another routine to perform Secondary Open Command								
6		verify_within	VLV1_PRI_CLOSED_IND	Valve1 Primary Closed Indicator	OFF		2 of 2	8 sec	
7		verify_within	VLV1_PRI_OPEN_IND	Valve1 Primary Open Indicator	ON		2 of 2	26 sec	OpenValve1Secondary
8	end								
9									
10	# Routine sends secondary OPEN command and waits for appropriate indicators								
11	OpenValve1Secondary	# Send secondary OPEN command							
12		send_command	VLV1_PRI_OPEN_CMD	Valve1 Primary Open Command	OFF				
13		send_command	VLV1_SEC_SELECT_CMD	Valve1 Secondary Select Command	ON				
14		send_command	VLV1_SEC_OPEN_CMD	Valve1 Secondary Open Command	ON				
15									
16	# Verify both secondary indicators change appropriately within appropriate time durations, on failure call another routine to generate a user message								
17		verify_within	VLV1_SEC_CLOSED_IND	Valve1 Secondary Closed Indicator	OFF		2 of 2	8 sec	
18		verify_within	VLV1_SEC_OPEN_IND	Valve1 Secondary Open Indicator	ON		2 of 2	26 sec	OpenValve1Error
19	end								
20									
21	# Routine notifies user of failure								
22	OpenValve1Error	# Send a failure message to the user							
23		send_message		Unable to open Valve1					
24	end								
25									

Figure 8 – First selected executable tabular spec section – Opening a valve with error paths

The functionality of all three of these subroutines was manually coded into PLC ladder logic⁵ and tested using a field item simulator in order to verify the proper operation of the manually coded ladder logic. An Allen Bradley ControlLogix PLC was utilized along with Rockwell

Automation's RSTestStandTM simulator software [7].

Figure 9 shows an example of some of the resulting manually translated PLC ladder logic. For brevity, only a very small portion of the PLC code is shown here.

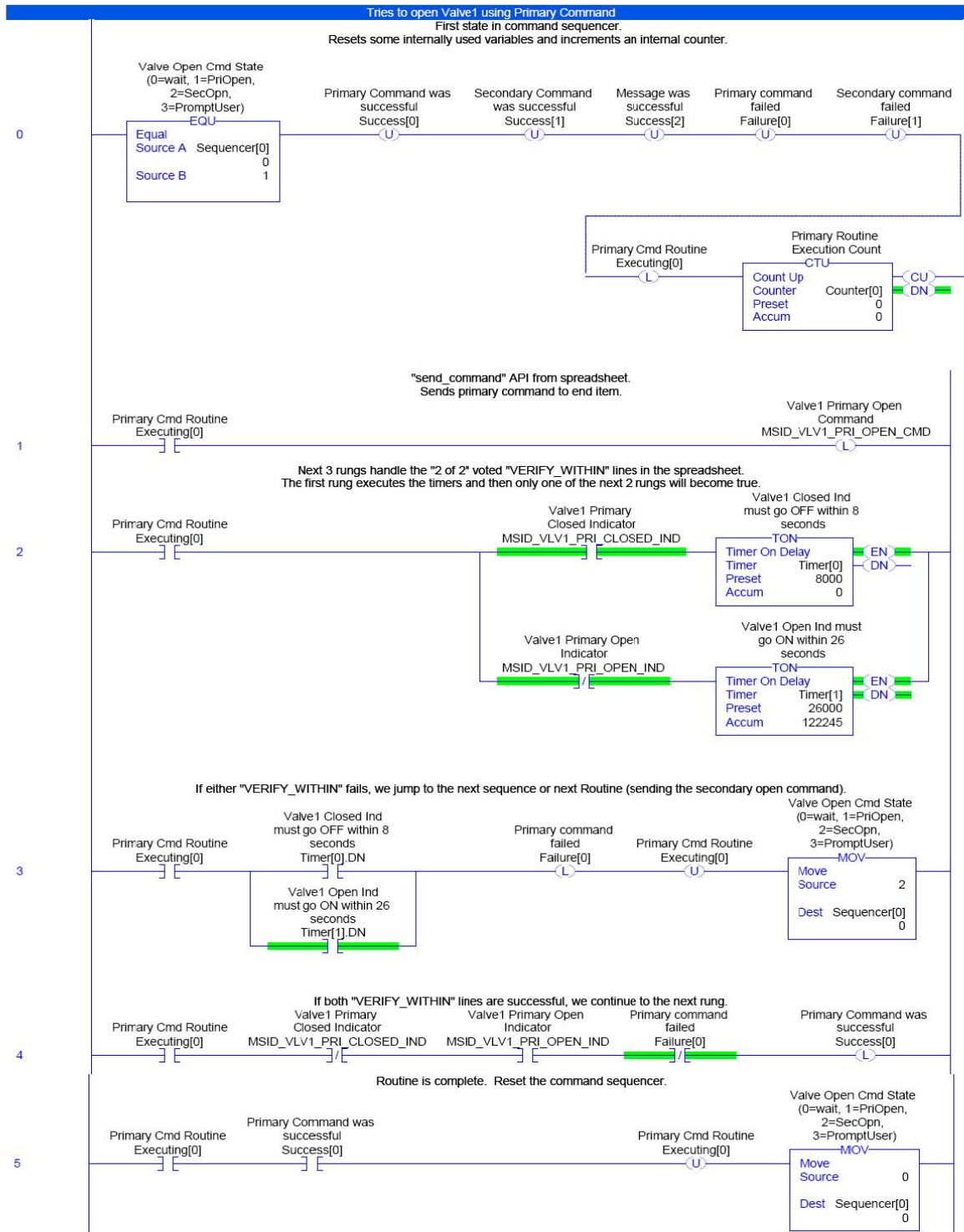


Figure 9 – Typical example of manually translated PLC code

⁵The process described in this paper should also work for other PLC programming languages besides ladder logic, but ladder logic was chosen for this prototype effort.

Tabular Spec Scenario	Brief PLC Code Description of Translation Point
discrete “send_command”	Latch to send ON command and Unlatch to send OFF command.
analog “send_command”	Move an integer or float (real) value to a stimulus tag.
discrete “set” command	Latch to set ON and Unlatch to set OFF.
analog “set” command	Move an integer or float (real) value to a tag.
all voted “verify_within”	A prerequisite rung with timers, then a failure rung if any (Boolean OR) timer finishes, then a success rung with Discrete (XIC/XIO) or Analog (GRT/LES) comparators in series (Boolean AND).
any voted “verify_within”	Very similar to above. A prerequisite rung with timers, then a failure rung if all (Boolean AND) timers finish, then a success rung with Discrete (XIC/XIO) or Analog (GRT/LES) comparators in parallel (Boolean OR).
“verify_within” (non-voted)	Simplification of voted case. Only one timer and only one comparator in the success rung.
“verify” command	Same as “verify_within” cases, only without the timers.
any voted “assert_constraint”	Routine that executes continuously upon activation. Violation rung with Discrete (XIC/XIO) or Analog (GRT/LES) comparators in parallel (Boolean OR).
all voted “assert_constraint”	Very similar to above. Executes continuously with a violation rung with Discrete (XIC/XIO) or Analog (GRT/LES) comparators in series (Boolean AND).
“assert_constraint” (non-voted)	Simplification of voted case. Only one comparator in the violation rung.
“remove_constraint” command	Just halts the “assert_constraint” routine that executes continuously.
“send_message” command	Pass a text string to a PLC System Message routine that was written manually as a System Software mini-layer on the PLCs.
“send_message_id” command	Same as “send_message” case, only using message databank for parameters.
“delay” command	Rung with a timer. Halt execution of routine until timer finishes.
“perform” command	Start a PLC routine by setting the value of a routine sequencer.

Table 1 – Translation points from tabular spec to PLC code

Translation Points

This manual process of conversion or translation from tabular spec representation to PLC ladder logic demonstrated that translation points or patterns existed between portions of the tabular spec and portions of the PLC ladder logic. Table 1 shows the translation points that were identified by this manual translation process.

Using these translation points, a few representative samples of the manually coded PLC ladder logic were exported from the PLC coding integrated development environment (IDE) as plain text. This exported text was then converted by hand into plain text PLC code “libraries” with the intent that a future automatic tabular spec to ladder logic translation utility would use these PLC code libraries during its translation process. In this context, the PLC code libraries could also be considered as code templates.

Figure 10 shows a small sample from the PLC code library that was created by this manual translation process.

```
# Sends a discrete (ON/OFF) command
<LIBRARY object="DiscreteSendCommand">
N:
XIC(<INSERT object="ExecutingTag"/>)
<INSERT object="Ot1OrOtu"/>(<INSERT object="ObjectTag"/>)
/
</LIBRARY>
```

Figure 10 - Typical example of PLC code library

This PLC code library object is intended to be used whenever a discrete (or Boolean) command is sent in the tabular spec. The `<INSERT object="ExecutingTag"/>` portion is to be replaced by the automatic translation utility with a Boolean PLC flag that tells the PLC that the routine that contains this ladder rung is supposed to execute. The `<INSERT object="Ot1OrOtu"/>` portion is to be replaced with an output latch object or an output unlatch object, depending on whether an “ON” command or an “OFF” command is being sent. The `<INSERT object="ObjectTag"/>` portion is to be replaced with the name of the discrete commandable object that is intended to be commanded. After the automatic translation utility employs this PLC code library, a tabular spec line that looks like that shown in Figure 11 should be translated into some PLC code that looks like that shown in Figure 12.

DSL API	OBJECT(S)	DESCRIPTION/MESSAGE	I/O/VAL
send_command	VLV1_PRI_OPEN_CMD	Valve1 Primary Open Command	ON

Figure 11 – Tabular spec example of discrete ON command

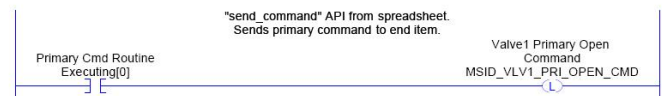


Figure 12 – Manually translated PLC code for discrete ON command

Figure 13 shows another small sample from the PLC code library that was created by this manual translation process.

```
# First line of a voted verify within section.
# This code sets up a timer for each of the voted objects.
<LIBRARY object="VotedVerifyWithinTimers">
N:
XIC(<INSERT object="ExecutingTag"/>)
<LOOP object="TotalVoted">
  <INSERT object="XicXio1"/>(<INSERT object="ObjectTag"/>
  <INSERT object="XicXio2"/>)
  TON(<INSERT object="TimerTag"/>,
  <INSERT object="TimerValue"/>,?)
</LOOP>
</LIBRARY>
```

Figure 13 - Typical example of PLC code library

This PLC code library object is intended to be used to set up a ladder rung with timers for an “all voted verify within” scenario in the tabular spec. The section of code library surrounded by the <LOOP object="TotalVoted"> and the </LOOP> is to be repeated by the automatic translation utility for every voted measurement contained in the voted verify within section of the tabular spec. Inside that repeated loop, the <INSERT object="TimerTag"/> portion is to be replaced with the name of an instance of a timer object that is to be automatically created for every voted measurement. The <INSERT object="TimerValue"/> portion is to be replaced with the actual time value that the timer object will count to in milliseconds. After the automatic translation utility employs this PLC code library, a tabular spec line that looks like that shown in Figure 14 should be translated into some PLC code that starts like that shown in Figure 15.

Automatic Translation

After manually performing some representative samples of translation from tabular spec to PLC ladder logic, and after manually creating a PLC code library for each of the translation points that were contained in the samples, a prototype automatic translation utility was then developed. A limitation of this prototype translation utility is that it only translates tabular spec snippets that match the translation points that were identified during the earlier manual

translation process. The translation capabilities of the utility will be expanded as more and more translation points are identified, translated manually, and then turned into additional PLC code libraries.

Figure 16 shows the simplified process flow for translating a single “send_command” line from tabular spec to PLC code. The spreadsheet that contains the application software in tabular spec format is exported to plain text and is used as input to the translation utility along with the PLC code library. The translation utility processes these input files using program transformation steps. This creates an output file that is capable of being imported by the PLC coding IDE.

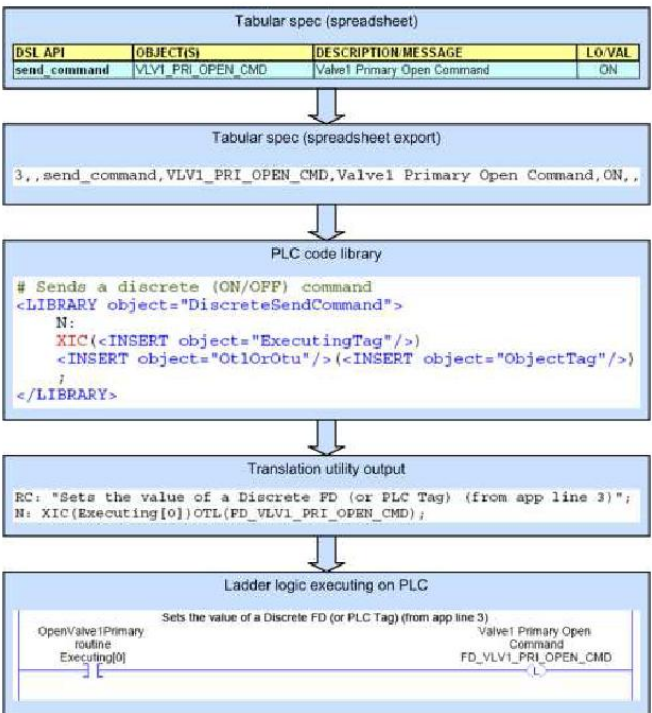


Figure 16 – Flow of tabular spec to PLC code translation

DSL API	OBJECT(S)	DESCRIPTION/MESSAGE	LO/VAL	HI	VOTING	DURATION
verify_within	VLV1_PRI_CLOSED_IND	Valve1 Primary Closed Indicator	OFF		2 of 2	8 sec
verify_within	VLV1_PRI_OPEN_IND	Valve1 Primary Open Indicator	ON		2 of 2	26 sec

Figure 14 – Tabular spec example of an all voted verify within

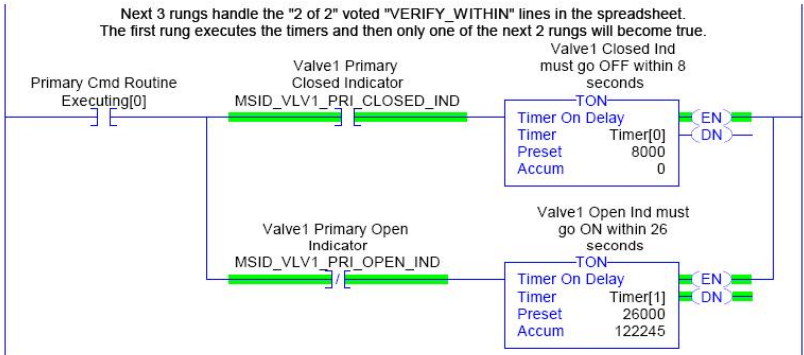


Figure 15 – Portion of manually translated PLC code for an all voted verify within

Perl, a widely used scripting language, was used for the prototype translation utility due to Perl's regular expression and text processing capabilities, and also due to its rapid application development capabilities [8]. Figure 17 shows a short code snippet from the prototype translation utility.

```
if ($foundLibrary eq "true") {
    if ($verbosity >= $someDebug) { print("Library snippet found: \"" . $LibToParse . "\"\n"); }

    $myTotRoutines = getTotRoutinesFromRoutineName($Routine, @RoutineMap);
    if ($verbosity >= $someDebug) { print("Total Routines: \"" . $myTotRoutines . "\"\n"); }

    $myResStart = getResourceStartFromRoutineName($Routine, @RoutineMap);
    if ($verbosity >= $someDebug) { print("Starting Resource Num: \"" . $myResStart . "\"\n"); }

    $LibToParse =~ s/<LIBRARY object=\"RoutineAndSequenceStarter\">([.]*)/$1/;
    $LibToParse =~ s/<INSERT object=\"SequencerTag\"/>/Sequencer[$CurrentSeqId]/;

    $LibToParse =~ s/<LOOP object=\"SuccessTags\">([.]*)/$1/;
    for (my $ctr=$myResStart; $ctr < ($myResStart + $myTotRoutines); $ctr++) {
        if ($ctr < ($myResStart + $myTotRoutines - 1)) {
            $LibToParse =~ s/([^\n]*)(<INSERT object=\"SuccessTag\"/>)([^\n]*)/$1Success[$ctr]$3\n$1$2$3/;
        }
        else {
            $LibToParse =~ s/([^\n]*)(<INSERT object=\"SuccessTag\"/>)([^\n]*)/$1Success[$ctr]$3/;
        }
    }
    $LibToParse =~ s/([.]*)</LOOP>/$1/;

    $LibToParse =~ s/<INSERT object=\"ExecutingTag\"/>/Executing[$CurRoutineId]/;
    $LibToParse =~ s/<INSERT object=\"CounterTag\"/>/Counter[$CurRoutineId]/;
    $LibToParse =~ s/([.]*)</LIBRARY>/$1/;
    $LibToParse =~ s/\n//g;
    $LibToParse =~ s/\t//g;
    $LibToParse =~ s/, /1/g;
}
```

Figure 17 – Sample of translation utility perl code

The syntax for using the translation utility is as follows:
 > TabularToPlcUtility-proto.pl [-i in] [-o out]

Since the PLC code library was not expected to be changed or switched out by the user, it was not deemed necessary or useful to include it as a command line argument during the prototype phase of the project. Figure 18 shows the translation utility being used during the prototype phase of the project.

```
C:\TabularToPlcPrototype>TabularToPlcUtility-proto.pl
-i Appl.csv -o Appl.L5K
Started Tue Sep 25 21:07:03 2007
Application input file: Appl.csv
PLC code output file: Appl.L5K

Total routines found: 3
  Start only: 1 (<*)
  Middle only: 1
  End only: 1
  Start & End: 0 (<*)
  Not Processed: 0

(<*) Total PLC sequencers needed: 1
Total PLC flag resources needed: 3

Total application lines found: 27
  Commands: 9
  Overhead: 10
  Blanks: 8
  Not Processed: 0

Finished Tue Sep 25 21:07:03 2007
C:\TabularToPlcPrototype>
```

Figure 18 - Example of translation utility in use

Rockwell Software's RSLogix IDE was used to demonstrate that the output of the prototype translation utility could be imported into a PLC and executed without modification. RSLogix imports and exports a unique plain text project definition format, although the vendor is in the process of transitioning to an Extensible Markup Language (XML)

import and export spec format [9]. XML is a specification for a widely accepted general purpose markup language that is commonly used to share structured data between different information systems [10]. PLCOpen, a PLC open standards organization, has created an XML-based specification for the exchange of PLC programs, libraries, and projects between PLC development environments [11]. However, most of the major PLC vendors have not adopted this open standard in favor of their own import/export approaches, which many feel are more powerful and robust.

After the prototype translation utility was developed, the same set of executable tabular spec sections from the tank loading application that were used for manual translation were selected for automatic translation into PLC code, in addition to a few other executable examples of tabular spec. Figure 19 shows a small tabular spec subroutine that attempts to open a valve using the primary command and response path. This same subroutine was used earlier in the manual translation section of the paper. Figure 20 shows the PLC code that was automatically translated from this particular tabular spec subroutine.

This automatically translated PLC code looks almost exactly like the manually translated PLC code shown earlier in Figure 9, with the addition of tabular spec line numbers in

LINE	ROUTINE	DSL API	OBJECT(S)	DESCRIPTION/MESSAGE	LO/VAL	HI	VOTING	DURATION	REACTION
1	#Routine sends primary OPEN command and waits for appropriate indicators								
2	OpenValve1Primary	# Send primary OPEN command							
3		send_command	VLV1_PRI_OPEN_CMD	Valve1 Primary Open Command	ON				
4									
5		# Verify both primary indicators change appropriately within appropriate time durations, on failure call another routine to perform Secondary Open Command							
6		verify_within	VLV1_PRI_CLOSED_IND	Valve1 Primary Closed Indicator	OFF		2 of 2	8 sec	
7		verify_within	VLV1_PRI_OPEN_IND	Valve1 Primary Open Indicator	ON		2 of 2	26 sec	OpenValve1Secondary
8	end								
9									

Figure 19 – Portion of tabular spec to be automatically translated into PLC code

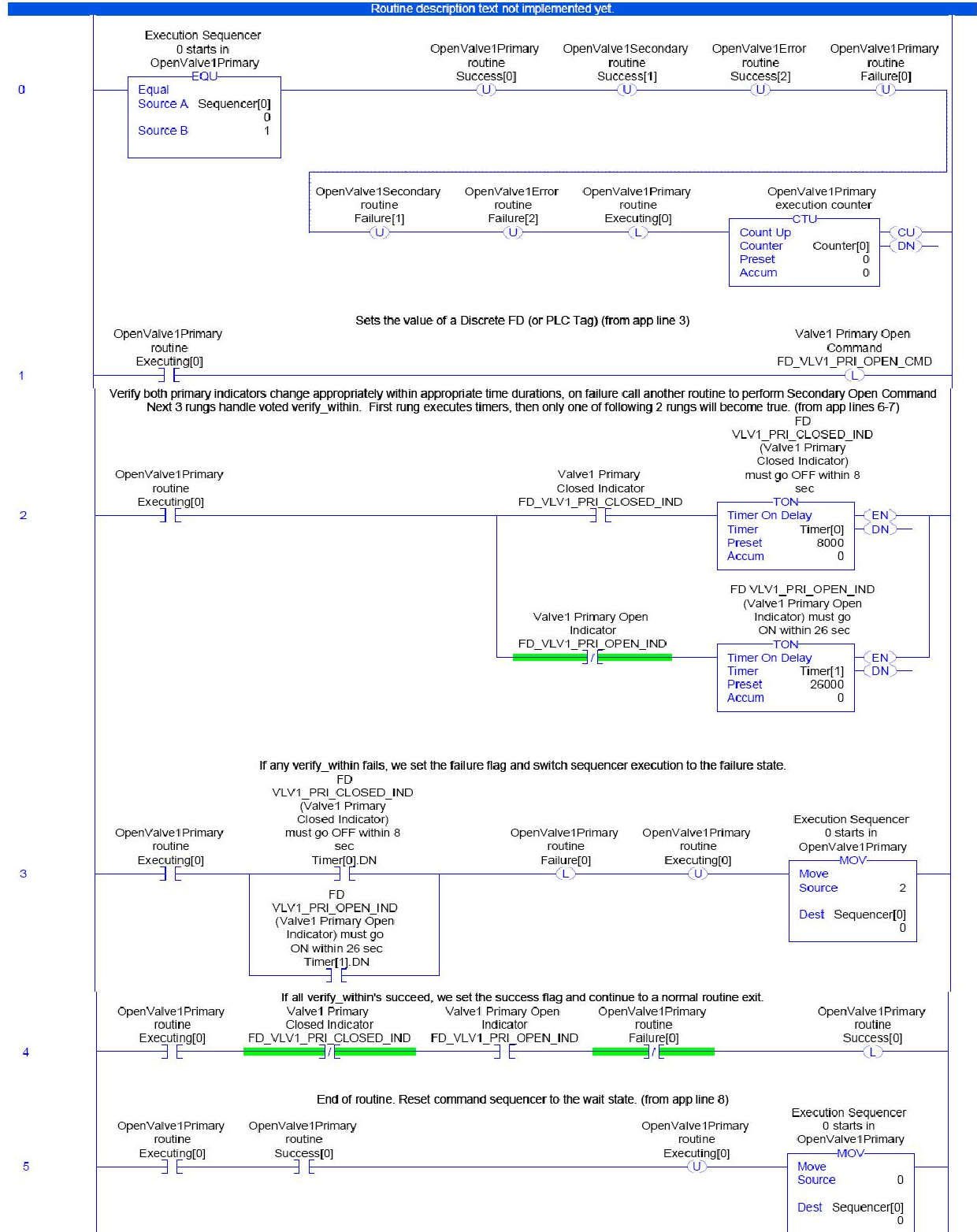


Figure 20 – Automatically translated PLC code from tabular spec portion shown in Figure 19

the rung comments of the automatically translated PLC code. The execution behavior of the manually translated code and of the automatically translated code was exactly the same. The similarity between the two was no accident. This code similarity is inherent to the process of manually creating PLC code libraries or code templates for use by the automatic translation utility. The authors recognize there are tradeoffs here. The manually and automatically translated PLC code may not be the most compact nor efficient. However, we feel that it is very important that the automatically translated PLC code be readable and understandable by PLC programmers. This strategy favors life-cycle costs as long as performance requirements continue to be met.

Some readers may ask why we would need to keep any PLC programmers on staff if we are automatically “generating” PLC code. There are many reasons:

- (1) The resulting PLC code will have to be tested functionally and also for performance.
- (2) The PLC programmers will be needed to troubleshoot and fix any problems that are found.
- (3) Assuming that problems are found and corrected, those corrections will have to be back-implemented into the automatic translation utility and into the PLC code libraries.

Also, there may be PLC code needed by the control system that cannot easily be automatically translated. For example, layers of PLC code that are deemed as “system software” that are used to interface between the application software layer and the rest of the LCS subsystems and components do not have to be written nor maintained in the domain of the ground and flight system user. Thus, these layers can be written directly in low level PLC code by experienced PLC programmers. These system software layers along with some other PLC code will probably need to be merged with the automatically translated PLC code. The automatic translation utility could be modified to perform this merge operation, or a separate utility could be created specifically for the merge task.

Return On Investment

Automatic application code generation from a high-level tabular spec has many economic advantages. In large systems, this technique helps to prevent quality and maintainability problems; it automates recurring software development steps; and it allows shorter product development time. The ability to represent the software design using a DSL that is oriented more towards the problem space than the solution space also contributes to these economic advantages. Also, miscommunications and misunderstandings between the requirements of the domain expert and the code implementation of the software developer are reduced in this process. In addition, code

consistency is increased and maintenance costs are decreased. [4], [12], [13]

Assuming that the automatic translation utility is tested thoroughly and formally validated and certified after development, the use of such a tool during the LCS development and future maintenance phases will increase the reliability and quality of the code that resides in the PLCs. In addition, a certified tool such as that described here will also increase the verification and validation (V&V) pedigree of the final PLC code because much of it will be generated by a certified tool as opposed to being generated by error prone human programmers⁶. This is very important for any software system, but even more so for safety critical software systems.

Using an automatic translation utility, the LCS will contain identical translations of identical portions of application software across different ground systems. This level of code consistency is difficult or impossible to obtain when multiple human programmers, or even one single human programmer is in the loop. Even when following approved and published coding standards, each coder has his or her own style and preferences as to how to solve each problem that is set before them. Even a single coder will sometimes solve the same problem differently on different days. Inconsistency in code can become a burden during troubleshooting efforts, during maintenance activities, and especially during the future upgrade process.

In our domain, the potential exists for very large tabular spec representations of application software and the potential also exists for significant repetition of similar tasks and functions within those potentially very large tabular specs. The automated production of large amounts of repetitious and/or tedious PLC software is expected to be a significant cost and schedule savings in the overall LCS project development life-cycle.

There are also payoffs down the road after the LCS is operational. Operations and Maintenance (O&M) costs of the application software residing on the PLCs could be reduced by this process for system changes and maintenance activities. Not only do these tools and processes have great potential to save the LCS project both cost and schedule, but the use of these tools and processes has the potential of making a better product than could be made manually.

One problem worth noting during this prototyping process was resistance and skepticism from some members of the PLC programming community. Since many PLC programmers generally have an electrical or electronics design background instead of a software engineering

⁶Please note that the authors are both error prone human programmers ourselves and we value the numerous benefits of humans in the loop. However, we also value the numerous benefits of letting computer programs handle tedious and repetitious tasks, at which human programmers are more prone to make errors.

background, the notion of automatic code generation often appeared foreign and the benefits of said technology were not completely obvious to them. It is hoped that our proposed solution will help the PLC programmers become more aware of some software engineering and V&V principles and their benefits.

Related Work

Automatic code generation and automatic code translation technologies have been in use in the Software Engineering industry for a long time. Literature reviews by the authors turned up only a few examples where these techniques and technologies were being used to automatically generate PLC code.

The U.S. Navy (Naval Surface Warfare Center, Philadelphia) has developed PLC based Machinery Control Systems (MCS) for use on various ships. Much of the PLC code, along with the SCADA display code in MCS is automatically generated from information in a hardware and measurement configuration database, however the actual machine control logic (i.e., application software) is written entirely by hand. The generated PLC code is automatically integrated with the machine control logic [14].

Semantic Designs, Inc. has developed an enterprise level productivity suite called the Design Maintenance System (DMS) ® Software Reengineering Toolkit. This program transformation tool is capable of translating very large high level mechanical process specifications or a DSL into highly optimized PLC ladder logic. The tool is designed to use several small and relatively simple layers of translation stacked on top of one another. This multi-layered transformation tool design is less brittle than other program transformation tools that have to do more work and perform more complex transformations all at one time [15], [16].

This DMS ® toolkit appears capable of producing efficient PLC code from our tabular spec format but was not chosen for the LCS prototype effort due to the added cost and schedule impact of needing to engineer new transformation layers and domain specific knowledge between our new tabular spec format and transformation layers that already exist in the tool. Also, to a lesser extent, highly optimized PLC ladder logic was not as much of a priority on this project as was human readable and understandable PLC ladder logic.

3. CONCLUSIONS AND FUTURE WORK

This work has successfully demonstrated that a process and a software tool are capable of generating executable PLC code from a high level specification representation of a safety critical control system. This process includes some manual work to find translation points and to create PLC code libraries. However, that up front and one-time manual effort is overshadowed in the end by the automatic generation of repetitious and tedious functionality that would be difficult and error prone to perform manually.

Such a process and tool increases the quality, reliability, maintainability, and verification/validation pedigree of the PLC code over that which is coded manually. It also provides a high level of PLC code consistency and could even reduce operations and maintenance costs for the control system after it is deployed.

Follow-on phases of development of the automatic translation utility should include most, if not all, of the following tasks:

- (1) Prototype and demonstrate manual and automatic translation into mixed PLC programming languages as appropriate (e.g., function block diagram, sequential function chart, structured text, instruction list).
- (2) Represent as much of the LCS application software in the tabular spec format as possible without overcomplicating the tabular spec format.
- (3) Manually implement the remaining translation points and any newly discovered translation points along with the matching PLC code libraries.
- (4) Add code to the translation utility to recognize and handle the new translation points along with the new PLC code libraries.
- (5) Test and certify the translation utility for automatic generation of safety critical PLC control logic in the LCS at KSC.
- (6) Extend the translation utility as necessary to generate PLC code that can be imported by various PLC vendor products.

ACRONYMS

COTS: commercial off the shelf
DMS: Design Maintenance System
DSL: domain-specific language
GOAL: Ground Operations Aerospace Language
GSE: ground support equipment
IDE: integrated development environment
KSC: Kennedy Space Center
LCS: Launch Control System
MCS: Machinery Control Systems
MDSD: Model-Driven Software Development
NASA: National Aeronautics and Space Administration
O&M: Operations & Maintenance
PLC: programmable logic controller
V&V: Verification and Validation
XML: Extensible Markup Language

REFERENCES

- [1] President George W. Bush, "President Bush Announces New Vision for Space Exploration Program", Press Release, January 14, 2004. Retrieved September 7, 2007 from the Internet: <http://www.whitehouse.gov/news/releases/2004/01/20040114-3.html>
- [2] NASA, Constellation Program Office, John F. Connolly, "Constellation Program Overview", Media slideshow, October, 2006. Retrieved September 7, 2007 from the Internet: http://www.nasa.gov/pdf/163092main_constellation_program_overview.pdf
- [3] Terry R. Mitchell, "A standard test language - GOAL (Ground Operations Aerospace Language)", ACM IEEE Design Automation Conference, Proceedings of the 10th workshop on Design automation (DAC'73), June 1973.
- [4] Tom Stahl and Markus Völter, "Model-Driven Software Development: Technology, Engineering, Management", Chichester, West Sussex, England: Wiley, 2006.
- [5] Wikipedia, "Domain-specific programming language", Community encyclopedia, August 24, 2007. Retrieved September 7, 2007 from the Internet: http://en.wikipedia.org/wiki/Domain-specific_programming_language
- [6] Michel Ingham, Matthew Bennett, Richard Borgen, Klaus Havelund, and David Wagner, "Development of a Prototype Domain-Specific Language for Monitor and Control Systems", Proceedings of the 2008 IEEE Aerospace Conference, Big Sky, MT, March 2008.
- [7] Rockwell Automation, "Rockwell Automation RSTestStand™ Speeds Control System Development and Deployment", Press Release February 19, 2002. Retrieved September 25, 2007 from the Internet: <http://www.rockwellsoftware.com/corporate/pressrelease/2002rsteststand.cfm>
- [8] Larry Wall, Tom Christiansen, and Jon Orwant, "Programming Perl, Third Edition", Sebastopol, CA: O'Reilly & Associates, Inc., July 2000.
- [9] Rockwell Automation, "Logix5000 Controllers Import/Export Reference Manual", Publication 1756-RM084L-EN-P, January 2007. Retrieved September 25, 2007 from the Internet: http://literature.rockwellautomation.com/idc/groups/literature/documents/rm/1756-rm084_-en-p.pdf

- [10] World Wide Web Consortium, "Extensible Markup Language (XML) 1.1 (Second Edition)", W3C Recommendation, September 29, 2006. Retrieved September 25, 2007 from the Internet: <http://www.w3.org/TR/xml11/>
- [11] PLCopen association, Phil Melore, "PLCopen adds independent schemes to IEC 61131-3", PLCopen spec website, September 4, 2007. Retrieved September 7, 2007 from the Internet: http://www.plcopen.org/pages/tc6_xml/xml_intro/
- [12] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami, "Model-Driven Development", IEEE Software, Volume 20, Issue 5, Sept.-Oct. 2003 Pages:14 – 18. Retrieved September 25, 2007 from the Internet: <http://ieeexplore.ieee.org/iel5/52/27576/01231145.pdf>
- [13] CodeGeneration.net, Jack D. Herrington, "Code Generation: The One Page Guide", Code generation website, 2003. Retrieved September 25, 2007 from the Internet: http://www.codegeneration.net/files/JavaOne_OnePageGuide_v1.pdf
- [14] Naval Surface Warfare Center, Jeffrey Cohen and Adam Sass, "Machinery Control Systems", NAVY slideshow for NASA KSC, January 31, 2007.
- [15] Ira D. Baxter, Christopher Pidgeon and Michael Mehlich, "DMS®: Program Transformations for Practical Scalable Software Evolution", 26th International Conference on Software Engineering (ICSE'04), 2004 pp. 625-634. Retrieved September 7, 2007 from the Internet: <http://www.semdesigns.com/Company/Publications/DMS-for-ICSE2004-reprint.pdf>
- [16] Michael Mehlich and Ira D. Baxter, "Mechanical Tool Support for High Integrity Software Development", Proceedings of Conference on High Integrity Systems '97, 1997, IEEE Press. Retrieved September 7, 2007 from the Internet: <http://www.semdesigns.com/Company/Publications/HIS97.pdf>

BIOGRAPHY

Kurt W. Leucht is a software and test engineer in the Engineering Development directorate at NASA Kennedy Space Center, FL. He has been writing and testing command and control software and advisory tool software for various KSC customers for the past 10 years. He previously worked hardware failure analysis for NASA KSC, but always dreamed of writing software instead. He has a BSEE from the University of Missouri-Rolla (formerly Missouri School of Mines) and an MS in Space Systems from Florida Institute of Technology. He can be contacted at Kurt.W.Leucht@nasa.gov.



Glenn S. Semmel is the Chief of the Application, Simulation, and Support Software Branch within the Engineering Development directorate at NASA KSC. He has previously lead efforts to infuse AI-based technologies for ground support of the Space Shuttle and future space vehicles. He has a BS Electrical Engineering, MS Engineering Management, and MS Computer Engineering from the University of Central Florida. He is currently pursuing a PhD in computer engineering in the field of model based diagnostic reasoning. Contact him at Glenn.S.Semmel@nasa.gov.

