Unit Testing and Remote Display Development

Nicholas Costa
Kennedy Space Center
Computer Science
USRA Summer Session
18 07 2014

Unit Testing and Remote Display Development

Nicholas Costa
Denison University, Granville, Ohio,

Abstract

The Kennedy Space Center is currently undergoing an extremely interesting transitional phase. The final Space Shuttle mission, STS-135, was completed in July of 2011. NASA is now approaching a new era of space exploration. The development of the Orion Multi-Purpose Crew Vehicle (MPCV) and the Space Launch System (SLS) launch vehicle that will launch the Orion are currently in progress. An important part of this transition involves replacing the Launch Processing System (LPS) which was previously used to process and launch Space Shuttles and their associated hardware. NASA is creating the Spaceport Command and Control System (SCCS) to replace the LPS. The SCCS will be much simpler to maintain and improve during the lifetime of the spaceflight program that it will support. The Launch Control System (LCS) is a portion of the SCCS that will be responsible for launching the rockets and spacecraft. The Integrated Launch Operations Applications (ILOA) group of SCCS is responsible for creating displays and scripts, both remote and local, that will be used to monitor and control hardware and systems needed to launch a spacecraft. It is crucial that the software contained within be thoroughly tested to ensure that it functions as intended. Unit tests must be written in Application Control Language (ACL), the scripting language used by LCS. These unit tests must ensure complete code coverage to safely guarantee there are no bugs or any kind of issue with the software.

Nomenclature

AccuRev	=	A software configuration management tool
ACL	=	Application Control Language
API	=	Application Programming Interface
ASF	=	Application Services and Framework
C++	=	Programming Language
COTS	=	Commercial of the Shelf
CUI	=	Compact Unique Identifier
DDE	=	Desktop Development Environment
DSF	=	Display Support Framework
GUI	=	Graphical User Interface
IDE	=	Integrated Development Environment
ILOA	=	Integrated Launch Operations Applications
LCC	=	Launch Control Center
LCS	=	Launch Control System
LPS	=	Launch Processing System
MPCV	=	Multi-Purpose Crew Vehicle

NASA	=	National Aeronautics and Space Administration
Netbeans	=	An open source integrated development environment
PLC	=	Programmable Logic Controller
SCCS	=	Spaceport Command & Control System
SLS	=	Space Launch System
STS	=	Space Transportation System
VM	=	Virtual Machine

I. Introduction

. The Kennedy Space Center (KSC) is being transformed to serve as a multi-user spaceport that will service both commercial and government users. The SCCS that is being developed at KSC will include software with the purpose of launching the rockets and spacecraft of the future to new and exciting destinations throughout the universe. Source code has been created by the Application Services and Framework (ASF) group that serves as an abstract layer to simplify the coding process for ILOA end users, who are not necessarily software programmers. Thoroughly unit testing the ASF code is an extremely important process. It is imperative that every single line of the source code be executed to ensure it is bug free and therefore avoiding serious errors and malfunctions. In this report I will discuss the process of unit testing the software developed by the ASF group. Another important part of the SCCS system is the ILOA team. Members of ILOA are responsible for the creation of the local and remote displays and control applications. I will also be focusing on the development of remote displays in this report.

II. Unit Testing with ACL Scripts

It is the responsibility of the ASF group to create software that converts ACL scripts written by the ILOA user into C++ code and ensuring this code is valid before sending it on to a gateway server. This allows us to remotely control the hardware through ACL scripts, preforming operations such as open and closing valves, or obtaining measurements. The ASF team is not only responsible for producing the software that translates and validates the ACL scripts, but also ensuring the correct functionality of this software through testing. Essentially, we must create ACL scripts to test all of ACL's Application Programming Interface (API) calls. ACL is simply an abstraction layer sitting on top of the C++ programming language that abstracts away some of the complexity of C++ code. This allows people not intimately familiar with the ins and outs of C++ to still understand the source code for their subsystem. This also makes it much simpler to maintain this technology for a longer period of time. In order to write the ACL scripts, we will be using the Netbeans Integrated Development Environment (IDE). Specifically, we will be using a customized version of the Commercial of the Shelf (COTS) Netbeans IDE with added plugins so that ACL code may be compiled. This scripts are developed and compiled locally on a Linux Virtual Machine (VM). The next step is to promote the script in AccuRev. AccuRev is a COTS product that NASA uses as a hierarchical repository for code. Promoting the script from your local workspace into a stream will allow all the workspaces beneath that stream to see the new script. The final step is to test the script in the Desktop Development Environment (DDE). The DDE is a virtual environment that simulates the software running on an actual machine in the Launch

Control Center (LCC). It is at this point that we can see the output of the ACL test script and determine whether or not the ACL API call we are testing is preforming as expected.

Imagine for example if we wanted to test the method that allows us to obtain the string representation of an object of type example. The script that we would write to test the TO_STRING(EXAMPLE) method would look something like the following example script.

```
#include<iostream>
using namespace std;

SCRIPT aclTest_Example() {

   WRITE_FILE(Utilities::logFile, "TO_STRING(EXAMPLE) ---### test begin ###---");

   EXAMPLE case1 = 1;
   EXAMPLE case2 = 0;
   EXAMPLE case 3 = -1;

   myCall_tostring(case1, "1", ACLRESULTOK);
   myCall_tostring(case2, "0", ACLRESULTOK);
   myCall_tostring(case3, "-1", ACLRESULTOK);
   myCall_tostring(NULL, "", ACLRESULTFAILURE);



   WRITE_FILE(Utilities::logFile, "TO_STRING(EXAMPLE) ------ test end ------");

}


void FUNCTION( myCall_tostring(EXAMPLE input, STRING expectedResult, AsfStatus expectedAclResult)
{
   STRING valueReturned = TO_STRING(input);

   if ( (expectedAclResult != APIRESULTLAST()))
   {
      OUTPUTSTRING logMsg;
      logMsg << "TO_STRING(EXAMPLE) failed, parameter = " << input << ", APIRESULTLAST() = ["
            << APIRESULTLAST() << "] not equal to expectedAclResultLast [" << expectedAclResult << "]";

       WRITE_FILE(Utilities::logFile, logMsg);
   }

   else if ( (expectedResult != valueReturned ) && (expectedAclResult == ACLRESULTOK) )
   {
      OUTPUTSTRING logMsg;
      logMsg << "TO_STRING(EXAMPLE) failed, parameter = " << input << ", valueReturned = ["
            << valueReturned << "] not equal to expectedResult [" << expectedResult << "]";

      WRITE_FILE(Utilities::logFile, logMsg);
   }

}
```

In this example script we will make sure to test all possible inputs, making a special effort to include all of the corner cases. This test script tests input strings that are positive, negative, and even the NULL value. This allows us to completely and thoroughly test the method of TO_STRING(EXAMPLE), making sure we obtain the expected outcome for any possible input. Notice that in the *myCall_tostring* method, we are attempting to confirm to results. We are first checking to see if the running the API call executed successfully. If the API call executed as expected, we then check to see if the value returned from the API call was the value that we expected it to return. If there is an issue with any of these tests, a message describing the issue will be written to a log file which we can examine after the test is run in order to discover any errors and determine what caused them.

III. Developing Remote Display Interfaces

An important task that the ILOA team must perform is the development of application software for the various subsystem teams at Kennedy Space Center. The ground subsystems, including Hypergolics, Crew Access Arm, and Ground Special Power, and the flight subsystems, including Avionics, Communication, and Cryogenics, need this application software in order to communicate with their hardware in the field. This software has the capability to control the hardware, for example turning a valve on an off. The subsystem teams also require this software to monitor the hardware and return needed measurements and values. This application software falls into four distinct categories. The first type of software is local control applications. This software is comprised of logic that will run on the PLCs out in the field in close proximity to the hardware. The second class of application software is local displays. Local display software is simply Graphical User Interfaces (GUI) that also located out in the field not far from the end items. These two groups of local application software are needed to provide the capability of closely monitoring and controlling specific parts of the hardware out in the field. The third category of application software is remote control applications, also known as firing room applications. This is software logic that runs in the firing room environment instead of out in the field. This software logic is written in the ACL programming language, whose development, use, and testing was discussed in the previous section. The final class of application logic, remote displays, will be the main focus of this section. Remote displays are simply GUIs that are used in the firing room, away from the end items. This remote application software is also necessary as the subsystem teams need the capability to monitor and control their hardware from a safe, centralized, remote location. A remote display developer of the ILOA software development team is required to adhere to the software development process. This procedure mainly involves reviewing the requirement of the user, designing and developing the software, providing documentation for the design and of the software, unit testing, and eliminating any discovered issues. It is especially important for a software developer to document their work, as this makes maintaining and updating the code much simpler, especially if other developers become involved with the project. The remote displays are created in a Linux VM using the Display Support Framework (DSF) Editor. This Development Environment uses simple drag and drop technology, implemented by hidden Java code. Specifically, IMB ILOG JViews is used to create the GUI.

IV. Conclusion

It is imperative that the ASF software function exactly as intended to ensure correct functionality as well as safety. We can guarantee the correction functionality of this software through extensive and thorough unit testing of the source code, making sure to attain complete code coverage. This means that we must make a special effort to create ACL scripts that will cover all possible cases of inputs to the ACL API call being tested. The development of remote display software for use in the firing room is also extremely important. This application software gives subsystem teams the ability to monitor and control their hardware out in the field from a safe, remote location. Both responsibilities of unit testing and developing firing room displays are small yet significant pieces of the puzzle to the development of the SCCS, the launch of the Orion, and ultimately a new generation of space exploration.

Acknowledgments

I am extremely thankful for the amazing opportunity to perform an internship at the Kennedy Space center this summer. The work I have done throughout my internship has allowed me to gain experience with the exciting field of software development. It has been a privilege to work in the NE-C1 department, surrounded by many other bright individuals who love their jobs and are passionate about their daily work. I would like to especially thank my mentor Kurt Leucht, as well as Linda Crawford and Randy Lane. They have made time in their busy schedules to answer any questions I may have, and have given me direction throughout my internship. I am very blessed to have had the opportunity to be a part of the development of the LCS while gaining real-world experience that I will carry with me into my future.

References

Leucht, K. (2013, March 4). Hitchhiker's Guide to Subsystem Applications and Displays. NASA.