Launch Control Systems: Moving Towards a Scalable, Universal Platform for Future Space Endeavors

Jonathan Sun

Kennedy Space Center

Major: Computer Science

USRP Summer 2011

Date: 08/05/11

Launch Control Systems: Moving Towards a Scalable, Universal Platform for Future Space Endeavors

Jonathan Sun[1]
University of Southern California, Los Angeles, CA 90007

The redirection of NASA away from the Constellation program calls for heavy reliance on commercial launch vehicles for the near future in order to reduce costs and shift focus to research and long term space exploration. To support them, NASA will renovate Kennedy Space Center's launch facilities and make them available for commercial use. However, NASA's current launch software is deeply connected with the now-retired Space Shuttle and is otherwise not massively compatible. Therefore, a new Launch Control System must be designed that is adaptable to a variety of different launch protocols and vehicles. This paper exposits some of the features and advantages of the new system both from the perspective of the software developers and the launch engineers.

Nomenclature

ANTLR	=	ANother Tool for Language Recognition
API	=	Application Programming Interface
ASF	=	Application Services & Framework team
CMS	=	Common Services team
Compiler	=	A program that turns source code into executable machine code
COTS	=	Commercial Off The Shelf software
CUI	=	Compact Unique Identifier
DDS	=	Data Distribution Services
DSF	=	Display Services & Framework team
GOAL	=	Ground Operations Aerospace Language
GUI	=	Graphical User Interface
Hashmap	=	A data structure mapping elements of one data type to another
IA	=	Information Architecture team
IDE	=	Integrated Development Environment
ILOA	=	Integrated Launch Operations Applications team
JViews	=	A graphical diagramming product made by ILOG, a subsidiary of IBM
KSC	=	Kennedy Space Center
LCC	=	Launch Control Center
LCS	=	Launch Control System
Lexer	=	A program that reads and recognizes text expressions according to rules
LPS	=	Launch Processing System
Matisse	=	NetBeans' built-in Swing display builder
MPLV	=	Multi-Purpose Launch Vehicle
Plugin	=	An external tool that gets integrated into an existing program
SCL	=	Spacecraft Command Language
Script	=	A sequence of commands and behaviors for a program
SDK	=	Software Development Kit
Swing	=	Java's built in GUI framework
TCID	=	Test Configuration Identifier
Widget	=	A reusable dynamic graphical component that displays or receives data
XML	=	Extensible Markup Language

[1] Intern, Launch Control Systems, Kennedy Space Center; University of Southern California

I. Introduction

A. The Demand for a New Launch Control System

The culmination of NASA's Space Shuttle Program and cancellation of the Constellation Program[i] brings major changes both to NASA's organization and engineering direction. Though many are saddened by the Shuttle Program's end, the resulting transition period hopes to bring about a wider array of space exploration opportunities as well as lower cost and overhead for the administration. While the Constellation Program called for NASA to continue to launch manned space missions to the moon, a major component of the new space program will be heavy reliance on private, commercially manufactured and operated launch vehicles[ii] to transport American astronauts to the International Space Station (ISS) while NASA focuses on long term goals like missions to Mars and distant asteroids. Although a successful result would bring about a lower overhead cost for the administration per launch, NASA will still play a major role in supporting the development and operation of commercial launch vehicles. In particular, Kennedy Space Center (KSC) in Florida, where all the Space Shuttle launches took place, will make its existing launch pads open for commercial use.

Because of the specialized nature of the existing launch structures and hardware, NASA will have to renovate the pads to be compatible with future vehicles designated to launch at KSC. Similarly, the Shuttle's launch/ground operations software, known internally as the Launch Processing System (LPS)[iii], while robust, has been developed over decades in narrow accordance with the Shuttle's specifications and will need to be altogether redesigned in order to support future launches. The new infrastructure, simply called the Launch Control System (LCS), has many updated requirements that take advantage of advances in software development to make it ready for the future. Not only does it need to be safe and stable during operation, but needs to provide fail-safe and recovery mechanisms in case any parts of the software malfunction, whether due to internal errors or external obstacles. It needs to be modular so that individual components can be added, modified, or removed with ease to suit the needs of different launch protocols. Finally, it needs to be versatile and powerful enough to give launch engineers full control and awareness over their relevant systems, without compromising usability. In this respect, much of the software development for LCS resembles application development in the commercial world.

B. COTS Products: A Paradigm Shift

Development of the new LCS began in 2005 along with the enactment of President George W. Bush's NASA Authorization Act[iv], and, being a complete redesign of the old LPS, has the advantage of decades of advancement in software architecture and new technologies like the internet. The most significant paradigm shift in the design is the use of Commercial Off The Shelf (COTS) software for as many of the components as possible. In the same way that NASA will purchase future launch vehicle use from private space companies, it will also purchase commercial software packages to compose the new LCS. This methodology is very different from the development of LPS, which was almost exclusively designed and implemented from scratch by NASA engineers. Using COTS software has powerful consequences but also carries risks and tradeoffs. Many COTS packages already provide major portions of the functionality NASA desires, and thus by purchasing these packages and developing on top of them, NASA inherits much of the base functionality without having to reinvent the wheel. Furthermore, NASA no longer has to spend money and developer time and resources maintaining most of the software, as maintenance is handled by the company providing the COTS product.

While the use of COTS software reduces the time and resources required of NASA's software developers, it does not diminish their importance or necessity. Most COTS products still have to be heavily modified or extended before they are appropriate for launch use. Fortunately, most enterprise-level products provide Software Development Kits (SDKs) that allow external users (NASA, in this case) to make modifications to their software as needed. More importantly, since COTS products are not usually designed to communicate directly with each other, NASA developers must write code to bridge the gaps by providing channels for transferring data between the different products. The connections between different software packages, colloquially called "glue code", are the key to unifying all the COTS products under a single Launch Control System. Finally, COTS products may have errors of their own, or may change over time. Large software packages coded by hundreds of developers are rarely perfect, and NASA developers are responsible for ensuring that COTS products continually meet the pedigree of space exploration by designing ways to rigorously test and debug the software and by working closely with the software provider to identify potential fixes for issues. The remainder of this paper discusses several case studies of LCS teams finding advantages through using COTS products and overcoming various obstacles posed by them.

II. LCS Front End Services and Framework Overview

A. LCS Front End Architecture

Consisting of a graphical user interface (GUI) framework as well as tools for creating user-defined commands and automated scripts, the front end architecture of LCS is being developed concurrently by Application Services & Framework (ASF) and Display Services & Framework (DSF) with collaboration from Integrated Launch Operations Applications (ILOA). ILOA is the overarching term that refers to the team of launch engineers who work directly on mission hardware and will be working in the Firing Room (where launches are controlled) on the day of liftoff. LCS provides them with an interface with which they can monitor and manipulate their relevant hardware, ranging from electrical systems, engines, weather monitors, etc. This interface is actually only a small portion of LCS—data does not travel directly from the front end interface to the physical hardware itself. Rather, it is added to a central database that manages and diverts all the data associated with the launch equipment, ensuring that it reaches wherever it is needed using the Data Distribution Service (DDS) protocol[v] and logging its transfer. Although the feedback that the user sees in the display seems instantaneous, in reality, data is moving rapidly between different buffers behind the scenes in order to execute the user's command. The performance and maintenance of this database is handled by another LCS team called Common Services (CMS) and is outside of the scope of this paper.

Rather than waiting for the entire LCS front end to be finished and tested before deployment to ILOA, NASA designed LCS with iterative releases in mind. That is, features are released every *iteration period* (8 weeks in this case), and ILOA can begin development and testing of their end without waiting months or years for every part of LCS to be completed. This release methodology saves a lot of time and allows testing of LCS components to begin much earlier, but requires more care on the developer end in order to ensure that each iteration release is compatible with earlier and future versions.

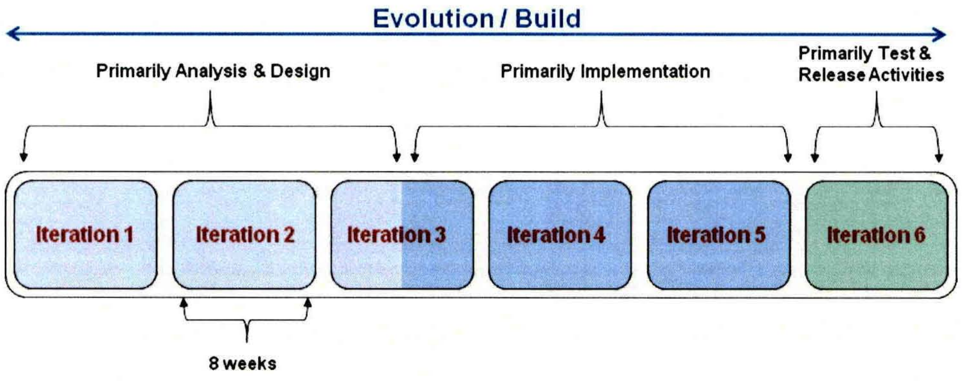


Figure 1. Iterative Development. *The design, implementation, and test phases are still observed in this model, but releases are made at the end of every iteration period rather than at the very end of development.*

B. Overview of DSF and the Display Editor

The advent of modern computer graphics technology allows traditional, text-based launch control displays to be replaced with dynamic graphical displays that use a wide variety of fonts, colors, images, and widgets (graphical components that display dynamic information), are thusly rich in content and can closely model the physical hardware that they represent. This is far more than an aesthetic advantage—by getting rid of vast nondescript arrays of uniform switches or numerical outputs and making displays truly look like the physical systems they manipulate and monitor, launch engineers gain instant familiarity with the controls and can identify relevant data with ease. Launching a rocket requires utmost precision and timing on the part of the engineers, and graphical controls that are familiar, intuitive, and clear result in safer, more efficient operation. Requiring users to identify buttons and numbers by their names and labels only is prone to error, but by placing such buttons and numbers next to images of the physical components they represent, users can now intuitively look for the controls and output relevant to their hardware.

During the Space Shuttle Program, dedicated teams of software developers would program the graphical displays and command scripts/buttons for the launch engineers, a process that was slow and difficult to maintain due to the changing demands of launches. Because launch engineers specializing in hardware components were unlikely to be proficient programmers, every time the engineers desired a modification to the application, LPS had to be updated by the software developers, rigorously tested, and redeployed in time for launch. The design of LCS shifts responsibility of creating these displays from the software developers to the ILOA members without requiring them to have heavy programming experience by including a visual, accessible software package that can create and modify displays on the fly. The Display Services & Framework team (DSF) is thusly responsible for the creation and maintenance of this program, internally called the Display Editor. Because ILOA members know their hardware best, it is natural to allow them to build the displays that they will finally use. In turn, development and maintenance of the Display Editor requires far less time and effort than the maintenance of hundreds of different displays, saving costs and freeing up developer resources to be used elsewhere. Because these displays are self-contained, different commercial launch companies can create entire graphical systems that can be swapped out at launch time, making it easy to adapt to different launch demands and protocols. Despite these advantages, development of the Display Editor carries many extra considerations between the developers and users. DSF must implement enough features to make the Display Editor powerful enough to satisfy all the needs of ILOA users, but must also be careful not to make the program too complex to use intuitively.

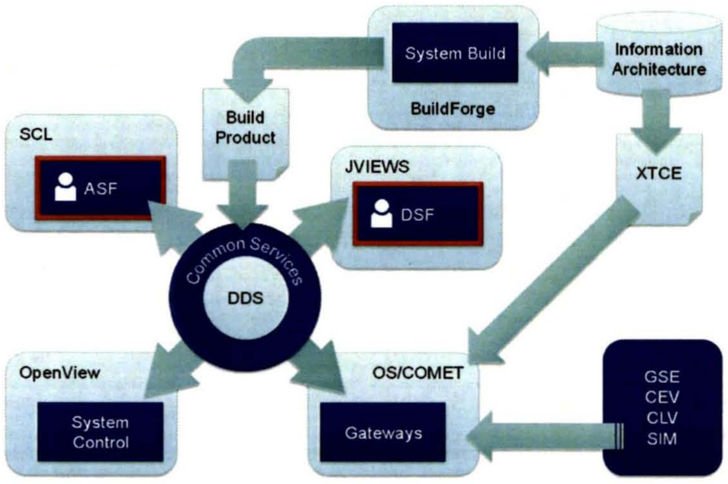


Figure 2. LCS Software Infrastructure. *The Common Services database provides a shared repository of all data from internal and external sources. This diagram shows the relationship between ASF, DSF and the various other LCS services that rely on the database to send and receive data.*

C. Overview of ASF and Spacecraft Command Language

Launch displays provide the primary visual input/output method for ILOA users, who can view numerical measurements on the screen as well as click buttons that send commands to hardware systems, but many launch operations contain steps that must be timed with machine precision, or consist of many complex simultaneous actions that a human operator could not possibly handle in real-time. Such operations must be done programmatically, using code that executes the required logic at launch time. During the Shuttle Program, NASA planned for the launch engineers to write their own scripts (sequences of commands that enact the desired behavior) to define these complex actions, but a lack of willingness from launch engineers coupled with a lack of deep programming expertise forced NASA to move development of such scripts to a team of dedicated software developers. The scripts were developed in Ground Operations Aerospace Language (GOAL), a programming language created internally at KSC just to serve the Shuttle Program. As was the case with displays, the resulting system, though successful for its purposes, was monolithic and difficult and costly to maintain.

```
READ <GMT $GREENWICH MEAN TIME$> AND SAVE AS (REG OUT
       GMT NEW);

LET (REG DECAY LOW LIMIT) = (REG OUT PR NEW) - 1 PSIA;
LET (REG DECAY HIGH LIMIT) = (REG OUT PR NEW) + 1 PSIA;

HIGH LIMIT TO (REG DECAY HIGH LIMIT)
LOW LIMIT TO (REG DECAY LOW LIMIT);
```

Figure 3. Example GOAL code. *GOAL was developed internally for Shuttle-specific applications and has been successful for decades. However, compared to modern programming languages, it is difficult to use and maintain, making it undesirable for future use.*

In order to make development of launch operation scripts more modular, NASA has moved coding of the scripts back to the responsibility of ILOA members. However, GOAL has been replaced by Spacecraft Command Language (SCL)[vi], a commercial language developed by the SRA Corporation, as the language in which the scripts will be written. This change is crucial, as SCL, which resembles English more naturally than GOAL, is designed to be easy to use even for non-programmers and, being a language designed specifically for space operations, already implements much of the functionality desired (GOAL developers had to implement most basic functionality from scratch). Because it is well documented and already widely used elsewhere in the aerospace industry, NASA software developers and ILOA members will have an easier time learning how to use SCL. As with displays, there are obstacles to overcome before SCL can be fully embraced for launch, and these are handled by the Application Services & Framework (ASF) team. Since SCL is a general space operations language, ASF still has to add some specific functionality manually to fully support KSC-specific launch capabilities. On the other hand, some SCL functions are either irrelevant or unfavorable to NASA's launch protocols and must be removed or disabled by ASF before ILOA can safely use SCL. Finally, although SCL scripts will define the launch behaviors programmed by their ILOA engineers, ASF must provide channels for communication between these scripts and the rest of LCS by passing SCL commands on to the previously mentioned CMS database so that they can successfully reach their hardware counterparts.

III. Case Study: Integrating SCL into NetBeans

A. ASF Modification of SCL

As a scripting language designed with aerospace applications specifically in mind, SCL is not as powerful for general applications as languages like C++ or Java. However, general purpose languages are often much more complex than scripting languages and require deep understanding of programming and software architecture. The fact that SCL includes some launch functionality and contains a simple, English-like syntax makes it an ideal platform for ILOA because engineers without programming experience can still design and write SCL scripts to their specifications. Before it is appropriate for use, however, ASF must make heavy modifications to SCL in accordance with LCS requirements. SCL contains numerous *keywords* (reserved words with predefined functionality) that have undesirable behavior in the context of LCS. For example, ASF bans ILOA members from using the keyword **exit** because its invocation immediately shuts down the program that is currently executing, a behavior that is rarely desired and that could prove dangerous if used improperly in a launch script. However, because SCL is a packaged commercial product, its internal functions cannot be directly edited by ASF. Thus, in order to deal with this issue, ASF is developing a program called the Command Application Compiler Tool (CACT), a code analyzer written in the Java language that analyzes SCL scripts and reports errors, such as use of banned keywords. Any SCL scripts that do not pass CACT's evaluation must be edited until they do so. This way, ASF can control the functionality that ILOA members are allowed to use, working around SCL's defaults. In total, there are dozens of illegal keywords that are caught by CACT for a variety of reasons.

```
script AccelerateRocket toSpeed
  global HasIgnited

  -- If rocket is not ignited, do not add fuel
  if HasIgnited = 0 then
    return
  end if

  -- Continue increasing speed until goal is reached
  repeat while Speed < toSpeed

    -- Add fuel until max output is reached
    if FuelFlow < 1  then
      FuelFlow = FuelFlow + 0.1
    end if

    -- Ensure that fuel is not added too quickly
    wait 20 ticks

  end repeat

  -- Goal speed reached, cut off fuel
  set FuelFlow to 0

end AccelerateRocket
```

Figure 4. Example SCL Script. *SCL has features common to most programming language and has a syntax that resembles plain English. This example script increases fuel flow to accelerate the rocket to a specified velocity.*

In addition to banning undesirable and potentially unsafe keywords, ASF is also responsible for filling in the base functionality gaps between SCL's defaults and the demands of LCS. While ILOA is majorly responsible for writing their scripts, there exists some shared, low-level functionality that ASF can provide as a base for higher level programs. This makes script development easier for ILOA members, who can call upon ASF base functions rather than write the functions from scratch. Because ASF can test and certify their own base functions separately, this saves ILOA time both in development and testing. Specifically, adding functionality to SCL can involve creating new built-in keywords as well as writing entire scripts to handle the most commonly demanded actions.

B. The Demand for an SCL Syntax Highlighter

Once all the modifications to SCL are completed and the communication channels between SCL scripts and the CMS database are finished, the language will be ready for deployment to ILOA. However, though SCL scripts are inherently just text files, effective development requires more advanced tools than just a text editor. Most software developers make full use of powerful Integrated Development Environments (IDEs) like NetBeans[vii] and Eclipse to make writing code easier, safer, and much faster. IDEs are vastly extended text editors whose common features include automatic compiling, project management, testing frameworks, version control and backup for source code, auto-complete while typing, syntax highlighting, and extendibility in the form of plugins (external tools that integrate into the program) that extend the base functionality of the IDE. Syntax highlighting is especially important because it increases the readability of the code, allowing developers to quickly identify different data fields, functions, and errors graphically in the text editor. The SCL software package includes a *compiler* that turns SCL scripts into executable files, but does not include any syntax highlighting support in any IDE. Fortunately, because NetBeans is a highly extendable IDE, ASF has developed an internal plugin that adds such functionality for SCL in NetBeans.

C. Building the SCL NetBeans Plugin

While not technically a COTS product within LCS, NetBeans is a commercial IDE, currently developed by Oracle (also the current developer of Java), that is widely used by LCS software developers. It fully supports development in many of the most widely used programming languages. Its power and flexibility has made it one of the most widely used IDEs by professional developers and its flexibility comes from the existence of NetBeans plugins (also called modules) that are coded in Java and extend virtually any element of the IDE imaginable. As such, NetBeans was chosen as the IDE of choice for ILOA members to develop SCL. However, because SCL is not widely used outside of the aerospace industry, official NetBeans support for it is nonexistent, but the NetBeans API (Application Programming Interface) allows support for any language to be added by defining a module and installing it into NetBeans. The resulting IDE is a slightly modified NetBeans that now carries syntax highlighting support for SCL in addition to all the default languages.

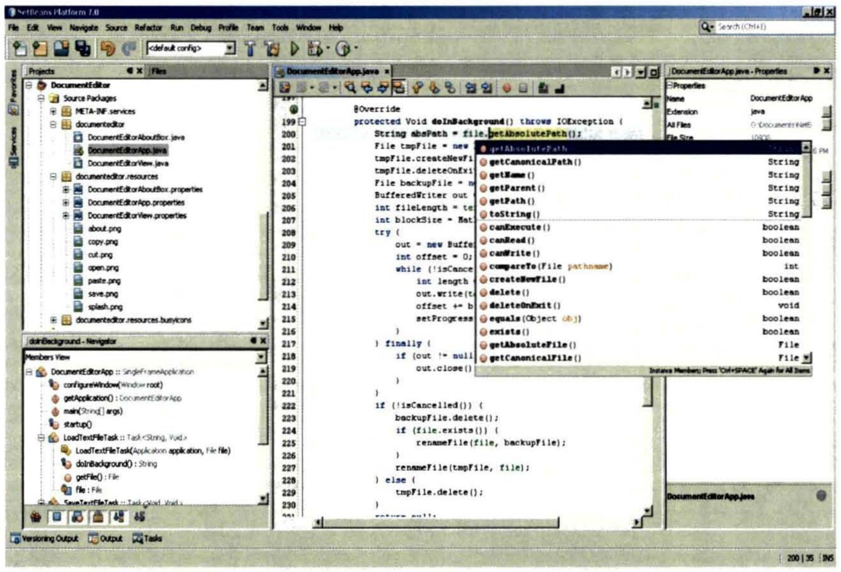
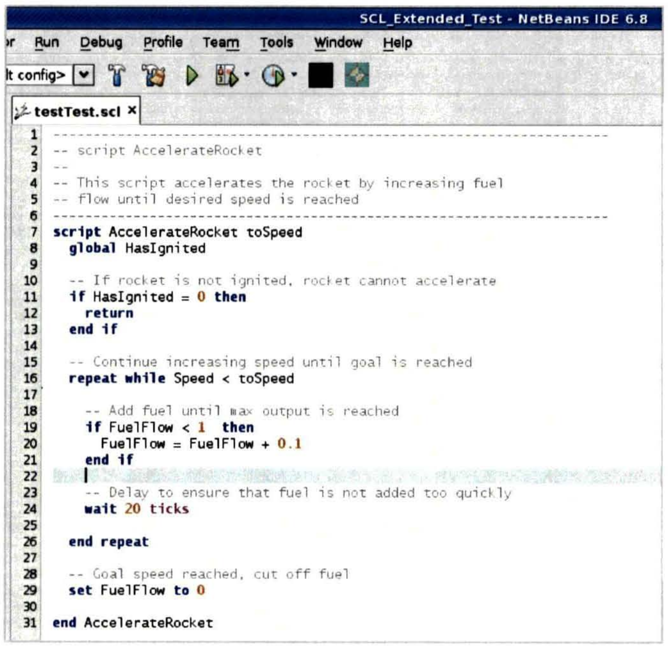

```
@Override
protected Void doInBackground() throws IOException {
    String absPath = file.getAbsolutePath();
    File tmpFile = new
    tmpFile.createNewFi
    tmpFile.deleteOnExi
    File backupFile = n
    BufferedWriter out
    int fileLength = te
    int blockSize = Mat
    try {
        out = new Buffe
        int offset = 0;
        while (!isCance
            int length
            out.write(t
            offset += b
            setProgress
        }
    } finally {
        if (out != null
            out.close()
        }
    }
    if (!isCancelled()) {
        backupFile.delete();
        if (file.exists()) {
            renameFile(file, backupFile);
        }
        renameFile(tmpFile, file);
    } else {
        tmpFile.delete();
    }
```
```
getAbsolutePath
getCanonicalPath()                String
getName()                         String
getParent()                       String
getPath()                         String
toString()                        String
canExecute()                      boolean
canRead()                         boolean
canWrite()                        boolean
compareTo(File pathname)          int
createNewFile()                   boolean
delete()                          boolean
deleteOnExit()                    void
equals(Object obj)                boolean
exists()                          boolean
getAbsoluteFile()                 File
getCanonicalFile()                File
Instance Members; Press 'Ctrl+SPACE' Again for All Items
```

Figure 5. **NetBeans IDE Interface.** *NetBeans makes software development easier and more effective. Project management tools are on the left, syntax highlighting can be seen in the middle, and the popup is a code auto-completion dialog.*

In order to create a syntax highlighting plugin for NetBeans, one must provide a *lexer*, which is a program that reads and recognizes text expressions based on predefined rules, and a *mapping* of text expressions to colors. The SCL Lexer, written by ASF in ANTLR[viii] (ANother Tool for Language Recognition), scans through SCL scripts and recognizes and categorizes text structures according to predefined grammatical rules. Discussion of how the lexer works requires an understanding of natural language processing algorithms and is outside the scope of this paper. However, once the lexer has finished categorizing the text in the SCL script, NetBeans will then color all the text according to the rules predefined by the user's mapping. For example, illegal keywords may by default highlight red, while SCL and ASF keywords highlight blue. Furthermore, once the plugin is installed in NetBeans, users can change the color mappings to their own choices via the NetBeans options dialog. Thus, ILOA users who prefer a different color scheme than the default can define their own without any ASF intervention. The advantages of the syntax highlighting plugin are much more than aesthetic—readable code is far easier to review, debug, and maintain, and very few developers write code today without syntax highlighting.


SCL_Extended_Test - NetBeans IDE 6.8
Run Debug Profile Team Tools Window Help
testTest.scl

```
-----------------------------------------------------------------
-- script AccelerateRocket
--
-- This script accelerates the rocket by increasing fuel
-- flow until desired speed is reached
-----------------------------------------------------------------
script AccelerateRocket toSpeed
  global HasIgnited

  -- If rocket is not ignited, rocket cannot accelerate
  if HasIgnited = 0 then
    return
  end if

  -- Continue increasing speed until goal is reached
  repeat while Speed < toSpeed

    -- Add fuel until max output is reached
    if FuelFlow < 1 then
      FuelFlow = FuelFlow + 0.1
    end if

    -- Delay to ensure that fuel is not added too quickly
    wait 20 ticks

  end repeat

  -- Goal speed reached, cut off fuel
  set FuelFlow to 0

end AccelerateRocket
```

Figure 6. SCL Syntax Highlighting. *This screenshot shows the example script from Figure 4 opened in NetBeans with SCL syntax highlighting installed. In this case, numbers are colored orange, making them easier to recognize and search for. By clicking Tools→ Options, users can manually set the color preferences for different text expressions.*

IV. Case Study: The Display Editor

A. Selection of a COTS Product

While the Display Editor's requirements have remained the same since its inception—to give ILOA engineers a fully graphical tool to edit launch displays—how the program is implemented has changed greatly since the beginning of development. The proof-of-concept for the Display Editor was originally built with Java's Swing framework, a graphical package built in to the Java language that contains base elements like windows, buttons, and text fields. In order to give ILOA users features like manual drawing tools and predefined widgets, DSF performed multiple trade studies and decided upon using a COTS package called JViews to build the final Display Editor on. Although this process required extra effort to research and critique all the available products, the decision ultimately saved DSF from having to code many basic graphical capabilities from scratch, again demonstrating the advantage of using COTS software.

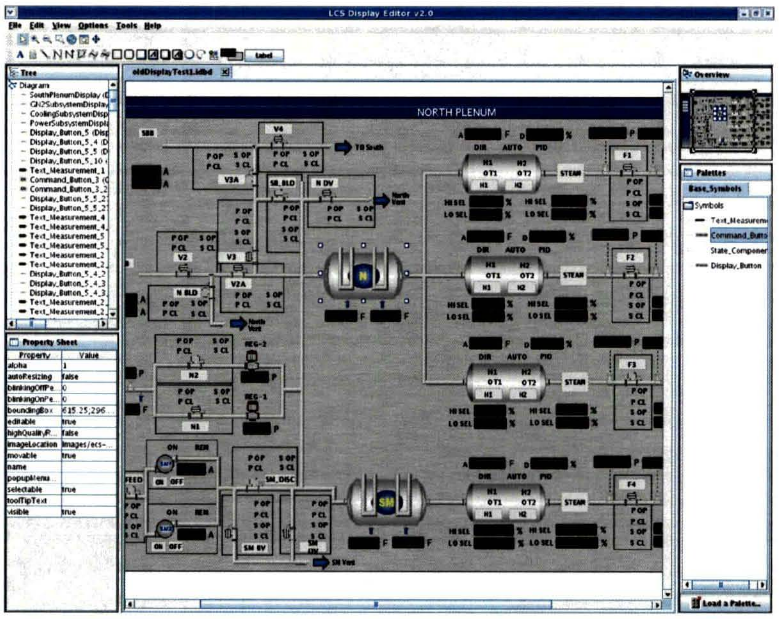


Figure 7. The LCS Display Editor. *The Display Editor makes creation of complex displays like the one shown relatively straightforward. The left panel allows users to select and modify properties for any object in the display. The right panel navigates the display and provides the user with DSF-defined widgets.*

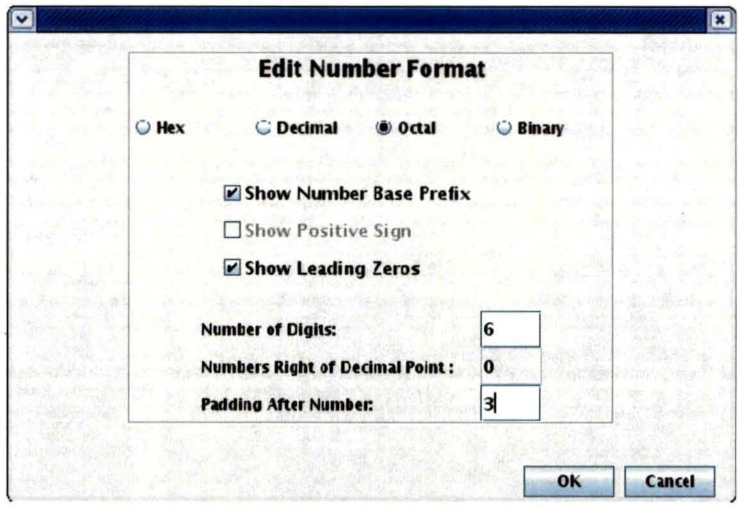
JViews[ix] is a Java Swing interface editor that allows users to create interactive graphical displays without any programming. Developed by ILOG, a subsidiary of IBM, JViews not only provides the ability to drag and drop graphical components onto a display but also has strong support for primitive lines and shapes as well as tools for editing images, gradients, and text. Furthermore, a powerful internal database allows users to define complex graphical interactions without any programming by using the included JViews interfaces. Widgets and text fields can update their displayed output based on the internal database, so by linking the JViews internal database with the CMS database, DSF provides the Display Editor with an easy way to update visual data based on data that flows in from launch hardware. For example, the State Component, a prominent DSF widget, is a dynamic object that can be user-defined to change image based on incoming data. One application of the State Component is the graphical construction of valves and pipes that visually open and close based on the state of the external hardware. Furthermore, DSF is currently working on providing a way to allow ILOA users to build their own custom widgets using the JViews interface.

B. Display Editor/Test Driver Applications

Development of the Display Editor is actually split into two parts, *design-time* and *runtime*. Design-time code refers to parts of the software used only to build the displays. The primary design-time component is the Display Editor itself, which creates and saves displays in .idbd format (ILOG's modified version of Extensible Markup Language (XML)) but does not actually execute them. Runtime code refers to parts that are responsible for actually executing the displays as standalone programs, maintaining communication between the JViews and CMS databases, updating displays as new data flows in, and passing along button clicks in the displays as commands to the CMS database. The runtime is handled by the Test Driver application, which is similar to the Display Editor but executes displays rather than edits them. In addition to its graphical capabilities, JViews also contains a strong, well-documented API that encourages user extension of functionality. As such, DSF has added many additional features to Display Editor and Test Driver that are specific to developing LCS displays. One example is a versatile number formatter, a set of options that allows user to customize how raw quantitative data from launch hardware is displayed on the screen.

C. The Display Editor Number Formatter

Number formatting is one aspect of launch displays that is very important to ILOA users, as properly formatted numbers make quantitative data much easier to track and digest. Traditionally, text-based display systems printed numbers on the screen the way they were received, making it very hard for launch engineers to interpret quantities, especially when hundreds might be displayed at once. In fact, the issue was so significant that, in some cases, professional typists were hired to manually transcribe incoming numbers into a readable format for the engineers! The freedom of a graphical display coupled with the power of the JViews database makes number formatting both highly customizable and automatic, reducing the potential for human error and increasing the amount of output that can be displayed on screen without being confusing. The number formatting system in the Display Editor is an example of functionality that was wholly developed by NASA on top of the COTS product. The DSF-defined JViews widget that displays numerical output is called a *text measurement* and resembles a simple digital counter. Text measurements, like all display components, receive their data from the JViews database, which in turn is updated by the CMS database. Thus, raw, unformatted number data is sent from CMS through DSF, but before this data propagates the JViews database and gets displayed, several options are available to ILOA users so that they can choose exactly how their numbers appear at runtime and launch. These options include simple preferences like showing plus signs for positive numbers and number of significant figures as well as powerful operations like the ability to convert output to hexadecimal, octal, or binary number bases. Furthermore, in the Display Editor, users can preview what their text measurement outputs will look like long before any real data is available. These options, combined with JView's ability to freely drag text measurements around, result in the capability for users to create more meaningful layouts with numerical output.



Edit Number Format

Hex Decimal [✓ Octal] Binary

[✓ Show Number Base Prefix]

Show Positive Sign

[✓ Show Leading Zeros]

Number of Digits: 6

Numbers Right of Decimal Point: 0

Padding After Number: 3

Figure 8. Number Format Options Dialog. *Users can select from a variety of formatting options for numerical output. Certain options are enabled or disabled by default depending on what the physical hardware supports.*

Programmatically, the number formatting system is wholly independent of JViews, relying only on numbers and character sequences (also known as Strings) to produce the formatted output. In some ways, it functions as a standalone package that is used by both the Display Editor and Test Driver (Display Editor uses it to define user formatting options, while Test Driver extracts them and formats the output). ILOA users select their options through dialogs in the GUI, and for each text measurement, a *hashmap* is produced mapping all of the options to their desired properties. In addition to this, a String is produced that encodes all the formatting options—this is required so that when the display is saved into a .idbd file, the formatting options are preserved in the XML. The formatting package is also highly extendable—a factory class generates appropriate formatters as needed for text measurements, and runtime formatting is done through a black-box interface that can be subclassed by new formatters as needed. Occasionally text measurements can be configured to display Strings rather than numbers, and this versatility allows the formatting package to also handle String formatting with minimal extra effort from the DSF developer (currently, the String formatter only defines a maximum length, but additional options can be added modularly as needed).

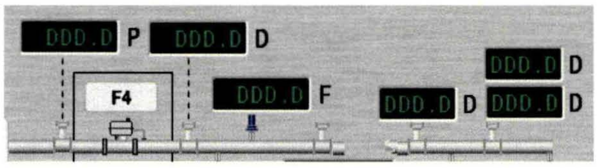


Figure 9. Number Format Preview. *This close-up shows several text measurement previews placed next to their valves and thermometers.*

V. The Future of LCS

A. Commercial Launch Opportunities

While a few commercial space companies have had recent success in launching rockets into orbit, it may be a few years before a final partnership between NASA and a commercial launch vehicle provider is finalized. The lack of a designated launch vehicle, however, does not impede the development of the LCS infrastructure, which has enjoyed continued support from NASA management. Commercial companies that do select KSC for their launch site in the near future will undergo the same development process as ILOA in creating scripts and displays to interface with their hardware. However, by then LCS will be much more comprehensive and robust in functionality, having been in continuous development until then. In addition to the ASF and DSF tools, the LCS front end will contain a program called the Component Build, which packages a display and all its dependencies together into a single archived file. Before launch time, all required displays will go through Component Build and finally be grouped into a Test Configuration Identifier (TCID). The TCID thus represents the configuration of an entire launch system, ready to be swapped in and out of the Firing Room of the Launch Control Center (LCC). Before launch day, however, the TCID will be deployed to simulated firing rooms for testing purposes.

B. Future NASA Initiatives

Until commercial launch vehicles begin arriving at KSC, LCS software will support the early testing of NASA vehicles like the Space Launch System[x], which is the current replacement to the Shuttle Program. Expected to launch beginning in 2017, the Space Launch System is composed of a combination of Ares rocket technology and Shuttle rocket technology and will carry the Multi-Purpose Crew Vehicle (MPCV) (previously known as Orion under the Constellation program). At the time of this paper's authorship, early testing of LCS with the MPCV has already begun, as well as support for various Expendable Launch Vehicles[xi]. The ILOA team is building displays and scripts to operate ground support vehicles and equipment that are native to KSC and will support any future launches that take place. Thus, even in NASA's transition period between vehicles, LCS is busy preparing the infrastructure for the future.

VI. Conclusion

At the time of this paper's authorship, space exploration is at a major crossroads. Dominated for decades by government agencies like NASA and the Russian Federal Space Agency (formerly the Soviet Space Program), space travel will soon become widely available to commercial companies and potentially even the consumer market. While the Space Shuttle Program's end marks the culmination of one of the most successful space programs in history, the opportunities of the future are even more promising. In order to support the launch vehicles of the future, however, software is required that takes full advantage of new technology. The LCS infrastructure, designed from scratch to be powerful, modular, and safe, will help guide space travel to new levels of availability, safety, and ambition by providing a universal framework for future launches.

Acknowledgments

This paper's author thanks Linda Crawford for her mentoring and Caylyne Lapolla for her help and support. Additional thanks go out to all ASF and DSF team members, who were always a pleasure to work with and offered help whenever available. Finally, the author thanks Kennedy Space Center and the University Student Research Program for their internship opportunity.

References

[i] NASA's Constellation Program, URL:
http://www.nasa.gov/mission_pages/constellation/main/index2.html

[ii] NASA Commercial Space Transportation, URL:
http://www.nasa.gov/exploration/commercial/index.html

[iii] NASA's Launch Processing System (LPS), URL:
http://science.ksc.nasa.gov/shuttle/technology/sts-newsref/stsover-prep.html#stsover-lps

[iv] NASA Authorization Act of 2010, URL:
http://commerce.senate.gov/public/index.cfm?p=Legislation&ContentRecord_id=8d7c1465-f852-4835-ba84-25faf56bbb36

[v] Data Distribution Service (DDS) Specifications, URL:
http://www.omg.org/technology/documents/dds_spec_catalog.htm

[vi] Spacecraft Command Language for Mission Critical Command and Control, URL:
http://www.sra.com/scl/

[vii] NetBeans Integrated Development Environment, URL:
http://netbeans.org/

[viii] ANTLR, ANother Tool for Language Recognition, URL:
http://www.antlr.org/

[ix] IBM ILOG JViews Enterprise, URL:
http://www-01.ibm.com/software/integration/visualization/jviews/enterprise/

[x] NASA's Space Launch System & Multi-Purpose Crew Vehicle, URL:
http://www.nasa.gov/exploration/new_space_enterprise/sls_mpcv/index.html

[xi] Kennedy Space Center Expendable Launch Vehicle Status Reports, URL:
http://www.nasa.gov/centers/kennedy/launchingrockets/status/2011/index.html