Payload Configuration Graphical User Interface (GUI): Simplifying Customer Software Integration

Tim Myers  
University of Colorado at Boulder  
Chris Koehler  
myerstm@colorado.edu  
March 21, 2011

Abstract

In order to solve the problem of every new student satellite project tasking themselves with the creation of an entirely custom software system, the Phoenix architecture was born. It consists of a modular, adaptable, and hardware-independent core architecture that can be configured to fit a variety of needs and purposes. The existing system greatly reduces the amount of code that needs to be written for a potential new project: a generally error prone and time consuming process that leads to late deliveries and buggy software. This system is configurable through the addition of individual modules that serve scoped purposes and have no hard dependency on other modules. Currently, these modules have to be implemented by hand, resulting still in a fair amount of code that allows for the possibility of bugs and delays. The Payload Configuration GUI aims to further reduce this dependence. By allowing the user a graphical drag and drop interface for configuring behavior of the satellite, the code necessary to implement entire modules can be automatically generated. The primary goal of this software is to allow the customer of an ALL-STAR satellite to configure a module which will control their payload hardware, greatly simplifying the integration of customer and core software.

1. Introduction and Justification

In the Spring semester of 2010, the first full team began work on the ALL-STAR project. ALL-STAR is a 3U CubeSat that will serve as a reproducible bus for customer payloads. It aims to provide an impressive set of capabilities that will allow for interesting and demanding missions, yet still be general enough to allow for a wide variety of payloads. Its greatest strength is in its low cost, and fast contract to readiness time. Purchase of an ALL-STAR bus should cost around $150,000 and will be fully integrated within six months of contract.

Of all the challenges that ALL-STAR provides, this six month time period is possibly the most difficult, in particular for the software subsystem. There are extreme challenges in creating a system that is capable of facilitating the specific needs of the satellite bus, but also flexible and powerful enough to be capable of handling the needs of any potential customer payload. To mitigate this, the Phoenix architecture was created. Its purpose was to provide a core software system that is modular, adaptable, and hardware-independent. This architecture, in its minimum state, provides basic functionality that can then be taken advantage of by additional modules. This allows us to build a system that provides both can adapt to any specific requirement, unknown at the time of development.

The core architecture provides utilities such as abstracted data types, and a communication protocol. Modules that then extend the functionality can implement control servers for various hardware subsystems, such as an attitude determination and control system. The simplicity of the code and care that was taken into implementing it, ensure the function and robustness of the core software. In addition the architecture has been and will continue to be extensively tested.

The other piece of the puzzle is the modules themselves. These are comprised mostly of control servers for various hardware subsystems in the satellite, and are what truly define the behavior of the satellite. If the core architecture could be thought of as a software API, these modules are then the clients that take advantage of it, creating an actual entire software system for the satellite.

As a more detailed example, consider again a server meant for the purpose of controlling an attitude determination and control system, and another module that is a control server for a power system. If the attitude control hardware is about to perform a maneuver, there is the problem of whether sufficient power exists. The attitude server, which defines the behavior of how the subsystem operates, would send a message to the power server, utilizing the core architecture. The power server would then check with the power hardware how much power the satellite currently has, and based on some logic of whether the maneuver should be allowed or not, return a message to the attitude server with the response. This
server then in turn either tells the attitude hardware to perform the maneuver, or to wait.

With this organization of the system, the current challenge then lies only in implementing the various modules that will control hardware subsystems, and also the payload’s hardware. These modules may be implemented by hand, either by the software team, or by the respective hardware subsystem’s team. This process however, is somewhat time consuming, prone to bugs (as writing any software is), and requires a good knowledge of the core architecture. In particular, this would provide a great challenge for the payload control server, as it needs to integrate an unknown payload, in the short timeframe of six months, with no prior knowledge of the core architecture.

In order to allow for the rapid integration of the customer’s software with the Phoenix architecture, a Payload Configuration Graphical User Interface (GUI) was designed. The purpose of this GUI was to allow for graphical configuration of behavior between the payload and core architecture without ever having to write the actual code that links the two. Actions and reactions are created through drag and drop composition of visual components that define complex behavior through simple implementations. The actual code that is necessary to facilitate this behavior is then automatically generated and integrated as a module in the Phoenix architecture.

The GUI will be used to configure ALL-STAR’s first payload, THEIA, and therefore receive intensive testing through real use. It is expected that using this method of payload software integration will drastically reduce both time and effort, while also greatly improving the robustness of the integrated system, helping to create a successful mission.

2. Payload Configuration GUI

2.1 Development Environment

GUI development is a notoriously involved process, and for this reason, a development environment was sought that would allow for rapid development of a quality graphical interface. The programming language Python [1] was chosen for its ability to allow for complex systems to be developed quickly due to its nature as a high level language. Also, a GUI library called wxPython [2] was chosen which allowed for the simple creation of attractive and functional GUI components. Both Python and wxPython are well supported and documented. The Integrated Development Environment (IDE) Eclipse [3] was used for editing of the code. This is the same IDE that is used for the development of the core architecture’s C++ code.

2.2 Design Practices

The software team of ALL-STAR practices Agile development. This is a diversion from the traditional software project development structure, often called the waterfall model. In the waterfall model, the entire system is planned upfront, developed, and then tested. This has historically lead to bugs and delayed finish dates, due to the impossibility of planning everything correctly the first time, and the lack of consideration given to testing and debugging during the development process.

Agile development takes the waterfall model, shortens it greatly, and repeats it many times. A small subset of the program was planned, implemented, and tested at a time. This allowed us to be flexible as we realized different needs and ideas for the system. When it was discovered that a redesign was needed of any portion of the program, it was a minor issue instead of a major one.

Designing of portions was done as an interactive exercise between members of the software team, and usually done on a whiteboard. These designs were always preliminary, as our initial ideas are bound to be wrong in some way and it is not worth wasting time on details that will change (see Figure 1 below, and Figures 7 and 8 in section 6).

![Figure 1. Main Window Planning](image)

2.3 General Structure

The goal of the GUI is to allow configuration of a module through defining its behavior. From a system level perspective, controlling the behavior of the satellite is managed through the various modes that the satellite, and each hardware subsystem can be in. Behavior differs depending on these modes. For example, in a low power mode, the attitude control system may not perform maneuvers that would be part of the normal operations.

The GUI should allow for the configuration of behavior for each individual mode. The structure of the program was decided to be similar to an IDE, like the one
used to write the code. Behavior would be configured through multiple files which will be called items. These items will be managed in a panel to the left which is similar to the file explorer panel on the left in Eclipse. A large center panel would then allow for the editing of individual items, just like the window where code is edited in an IDE. To the right of this main panel would be various utilities that provide the tools for creating behavior. These panels differ from a regular IDE because there is normally no need from them, as all information is input as text by the user. On the right, three panels exist, one for messages available to send to other modules, one for logical control blocks (conditional statements, looping), and one for managing variables. Additionally, a console panel exists on the bottom to provide feedback to the user as they interact with the program.

Through the use of all of these panels, a complex module can be developed with the capability of controlling the payload or a subsystem.

2.3.1 Item Panel

To not limit the capabilities of the behavior which can be defined, mode specific actions are necessary. In addition, certain actions may need to be performed upon entry to a mode, or periodically throughout its duration. This requirement led to the decision to implement behavior through numerous items, each residing in a certain mode, and within that mode a certain handler (OnEntry for example). Any given handler and mode combination could have many separate items, each implementing a small piece of the overall behavior. These modes, handlers, and items create a sort of directory structure of files which are managed via the Item Panel (see Figure 2).

![Figure 2: Item Panel](image)

This panel is very flexible, and can differ depending on the module being implemented. When a new project is created, the module it is being created for is chosen, loading an XML file that defines the modes and handlers for the module. Any number of modes may exist as are necessary to fully define the behavior. The handlers are slightly more limited. Currently, seven different handlers exist. OnEntry and OnExit define the behavior when a module enters and exits a mode. Events allow behavior to be scheduled to occur periodically throughout the duration of a mode. Analog Sensors allow the configuration of any analog signals that a hardware
subsystem wants to have managed by the FSW system’s control module. *Digital Indicators* allow for the configuration of simple digital signals that may signify whether a subsystem is in nominal operation, similar to a status indicator on a PC. *Message Handlers* allow the user to define the messages that the module will respond to, and how it will handle them. For example the power control server module may implement a message handler that allows retrieving information about the current power state of the satellite. *Error Handlers* are very similar to message handlers, but only respond to error messages, and are separated for conceptual clarity.

**2.3.2 Main Panel**

The main panel is where nearly all information entry into the program takes place. It is very nearly akin to the editor window of an IDE where code is entered by the user. The main difference is rather than the user merely typing in code, here they drag and drop visual blocks that represent different components such as messages, variables, and logical control statements. Some text entry does occur to then fill in details of these blocks. For example, an “if” statement block may be dragged in, where the user would then fill in various fields, such as the values being compared, and the comparison’s operator (‘less than or equals’, for example).

A few different items can be seen partially populated in Figures 9 through 11 in section 6. The nature of an item depends on the handler to which it belongs. Those that belong to OnEntry and OnExit are the simplest. These handlers are meant for the user to define behavior that occurs one time, whenever a mode is entered or exited, and a new item consists only of a target for the user to begin dragging in blocks.

For comparison, the blank item for an Analog Sensor item contains a few static blocks that may not be added or removed by the user. These allow the user to provide a variable for the contents of the analog sensor to be stored in, the rate it should be sampled at, and the channel of the sensor. After these blocks, which must be filled in, other blocks may be dragged in from the user, so that some behavior may be implemented every time the sensor is sampled. One example of how this may be used is in the power server. An analog current sensor may be constantly monitoring the current output of the batteries. If this current spikes, there is some sort of issue, and power should be disconnected. An analog sensor item could be configured to monitor this current sensor, store it in a variable, and every time it is sampled, check if it is above a maximum value. If this value is exceeded, the power server may then actually send a message to itself to disconnect the power.

**2.3.3 Messages Panel**

The Messages Panel is one of the aforementioned panels that exist to the right of the Main Panel. It allows for the user to drag in messages that can be sent to other modules into an item. Two kinds of messages exist here. The first is messages that can be sent to any other software module. For example, while configuring the payload control server, control servers for all of the hardware subsystems on ALL-STAR should likely already exist. Each of these modules will have messages that it can handle defined. Information about these messages is loaded into the panel, and they can then be used in items. The second type of messages is those that are sent to the hardware of the module being implemented. For example while implementing an attitude control server, it will be necessary to communicate with the actual hardware. Messages that the hardware may handle (such as to perform a specific maneuver) should already be defined. The hardware’s control server then handles the logic of whether or not it should perform the maneuver via an item, and if it decides appropriate, sends a message to the hardware to actually do so. See Figure 3 for a partially populated Messages Panel.

The goal of the GUI is to allow for the configuration of all subsystem modules, in addition to that for the payload. The timing of various components of the ALL-STAR project however will probably require that some of these be written by hand. For this reason, we have developed a couple of methods for messages to be parsed into this panel. In the case that the code for these messages has already been written by hand, specially formatted comments may be inserted into the code. These comments entirely and succinctly describe the messages. The other option is an XML format. This format is how the messages are saved by the GUI, and so modules that are implemented with it will be saved in this fashion.
XML file may be written by hand though as an alternative to the comments. In either case, either the location of the code that has been properly commented, or an XML file may be chosen through the use of the panel to populate it with the available messages.

### 2.3.2 System Calls Panel

The system calls panel is somewhat of an aggregate for various unrelated blocks that may be inserted into an item. Logical control statements reside here, including *if/else*, *while/do-while*, and a *for loop*. Also a *return* statement is available here for when implementing message handlers. Several other utilities can be found in the tree structure of options here. In addition to the control statements, things like access to the file system are available. A snapshot of the current state of the system calls panel is below in Figure 4.

![System Calls Panel](image)

**Figure 4 (System Calls Panel)**

### 2.3.2 Variable Panel

The last panel on the right hand side is for the management of variables. Variables in the GUI come in two forms, global and local. Global variables are shared between all items. One item may access a variable whose value has been set by another. Local variables are scoped to a single item, and only have meaning in the context of that individual item. The panel will show global variables at all times, regardless of what is open. Local variables are shown only when an item is opened, and only those variables that have been created for that item are shown. Variables can be created in seven basic data types: integer, unsigned integer, boolean, float, double, string, and array. These are the types that are supported by the datatype abstraction provided by the core architecture. The variable panel with a both a global and local variable can be seen below in Figure 5.

![Variables Panel](image)

**Figure 5 (Variables Panel)**

### 2.3.2 Console Panel

The console panel exists simply for the purpose of providing messages and feedback to the user as they interact with the program. It will eventually provide results to the user about the success of validating the existing items and generating code from them.

![Console Panel](image)

**Figure 6 (Console Panel)**
3. Information Management

One of the toughest challenges in creating the GUI was creating a method for managing the vast amount of information that is created. The items that are created, the content inserted into them, variables, and messages all need to be managed in a way that allows for simple saving and opening. The XML format [4] was chosen to use nearly universally for its incredible simplicity and flexibility. XML essentially allows you to create your own file format by following a few simple conventions. Moreover, python provides a library for dealing with XML that makes creating and parsing the files very easy.

XML is a file format that consists of nested items that may have specified attributes. This lends itself very well to most of the information in the program, particularly the item pages, as they consist of nested blocks that each contains information which may be thought of as attributes.

XML files are used to save all aspects of the GUI. Individual files are used to represent items. These files contain all of the blocks that have been used to populate the item, as well as any local variables that have been created. A single project wide file keeps track of any settings, global variables, and various other pieces that may have been loaded into the program such as messages.

The benefit of using XML truly becomes apparent in code generation. The saving of blocks in XML provides a format and order that makes code generation almost trivial. If two nested for loops need to be generated, they are represented by two nested for loop items in the XML, and can be directly converted into C++ from the XML with relatively little work. The XML files can be traversed linearly and the code generated as this is done.

4. Code Generation

The final product of the GUI is the code that it will generate. From the items that have been created in a relatively short amount of time, code will be generated that may have taken much longer to write by hand. The generated code for a module will work directly with the core architecture, and alongside with other modules which may have also been written with the GUI, or else by hand. Code generation is done by parsing through the XML files representing each of the items that have been created. Code will be generated as if it were written by a human—so that it will be readable by a human. This will simplify the debugging process if there happen to be any issues with the generated code, and also increases its probability of working correctly as there is no confusion introduced by making it unnecessarily concise and cryptic.

5. Conclusion and Results

The Payload Configuration GUI is an ongoing project by the flight software team of the ALL-STAR project. Much of the program is completed, and nearly all components of the GUI exist. The major work left to be done is largely to create code generation. Work has been started on this aspect, and has been found to not be difficult, largely thanks to the convenient XML structure of the saved items.

Through testing of the existing program, it has been found to be incredibly fast and easy to build up items with complex behavior. The drag and drop interface, and abstraction of the code through visual blocks allows for programming from a system level, without having to worry about the details of a programming language.

Our goal and current schedule is to finish up all pieces of the GUI, save for code generation, by around the end of May 2011. This will allow ALL-STAR’S first payload, THEIA, to begin creating items to define the behavior of their payload control server. Since code generation is completely independent from this portion of the GUI, configuration and continued development will be able to go on concurrently. Code generation will be finished shortly after so that the GUI can be fully tested. Code for THEIA’s payload control server will be generated and integrated into the flight software system where it can be tested as a whole.

We are incredibly confident in the ability to finish the project in a timely manner due to the amount of work that has been done already in a fairly short amount of time. We have also constantly reconsidered what the needs are of this program, and as a result have revised and changed the GUI so that it will satisfy them. The GUI should be fully capable of configuring a module with any desired behavior. Should actual testing realize that we are missing a component, it can easily be added.

We are very excited about the program that we are creating, and believe that it is truly revolutionary. The GUI along with the Phoenix architecture not only speeds up the development process and decreases the likelihood of bugs, but also makes the development of a software system more accessible to those with little background in software development. We expect to see good results with the ALL-STAR project and THEIA payload that will continue with any future ALL-STAR payloads. Also, the core architecture and GUI have both been developed in a way that would allow them to be completely usable on any future student satellite project, and we hope that this is the case. Our goal was to support the ALL-STAR project, but also to solve the problem of every new student satellite project struggling through the difficult problem of creating a new software system.
6. Additional Figures

Figures 7 & 8 (Additional whiteboard plans for various components of the GUI)

Figure 9 (Main Panel, implementing a startup handler item)
Figure 10 (Main Panel, implementing an analog sensor item)

Figure 11 (Main Panel, implementing an events item with periodic behavior)
7. References