Aquarius’ Object-Oriented, Plug and Play Component-Based Flight Software

Alexander Murray
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
818-354-0111
alex.murray@jpl.nasa.gov

Mohammad Shahabuddin
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109
818-354-4993
shahab@jpl.nasa.gov

Abstract—The Aquarius mission involves a combined radiometer and radar instrument in low-Earth orbit, providing monthly global maps of Sea Surface Salinity. Operating successfully in orbit since June, 2011, the spacecraft bus was furnished by the Argentine space agency, Comision Nacional de Actividades Espaciales (CONAE). The instrument, built jointly by NASA’s Caltech/JPL and Goddard Space Flight Center, has been successfully producing expectation-exceeding data since it was powered on in August of 2011. In addition to the radiometer and scatterometer, the instrument contains an command & data-handling subsystem with a computer and flight software (FSW) that is responsible for managing the instrument, its operation, and its data.

Aquarius’ FSW is conceived and architected as a Component-based system, in which the running software consists of a set of Components, each playing a distinctive role in the subsystem, instantiated and connected together at runtime. Component architectures feature a well-defined set of interfaces between the Components, visible and analyzable at the architectural level (see [1]). As we will describe, this kind of an architecture offers significant advantages over more traditional FSW architectures, which often feature a monolithic runtime structure.

Component-based software is enabled by Object-Oriented (OO) techniques and languages, the use of which again is not typical in space mission FSW. We will argue in this paper that the use of OO design methods and tools (especially the Unified Modeling Language), as well as the judicious usage of C++, are very well suited to FSW applications, and we will present Aquarius FSW, describing our methods, processes, and design, as a successful case in point.

Table of Contents

1 INTRODUCTION ........................................ 1
2 MISSION & FSW REQUIREMENTS OVERVIEW .... 1
3 ARCHITECTURAL APPROACH & FEATURES ..... 2
4 TECHNIQUES & PROCESSES ..................... 6
5 THE USEFULNESS OF PLUG & PLAY COMPONENTS .................. 7
6 VERIFICATION ....................................... 9
7 FSW QUALITY ........................................ 9
8 CONCLUSIONS ....................................... 10
9 FUTURE WORK ..................................... 10
ACKNOWLEDGMENTS ................................. 10
REFERENCES ......................................... 10
BIOGRAPHY .......................................... 10

1. INTRODUCTION

The use of Object-Oriented (OO) techniques and languages in flight software (FSW) is not without controversy: there are justifiable concerns about the efficiency and complexity of OO software in the demanding and resource-constrained arena of embedded software in general, and FSW in particular. We argue in this paper that the advantages of OO architecting and implementation, as well as the lack of realization of some of the typically feared disadvantages, have been amply demonstrated by Aquarius FSW and its success in development, test, and in Earth orbit.

We begin with an overview of the mission and the requirements on the FSW, followed by a presentation of the architecture of the FSW, which will focus on the Component-based architecture employed. In this approach, the software is conceived and designed as a set of Components, with well-defined interfaces, visible at the architectural level, which are dynamically instantiated and connected together at run time.

This has important advantages for testability and operability, by enabling a plug-and-play capability that allows significant modifications to the FSW, including whole new applications, to be unplugged and safely installed in the running software with very little disruption to the operation of the instrument (in particular, with no reboot needed).

We continue with a discussion of the software systems engineering approach employed, and a description of techniques used in architecting, design, and implementation of the FSW, which include the use of modeling with the Unified Modeling Language (UML), and coding in C++.

We go on to discuss verification and testing techniques used to ensure the robustness and reliability of the FSW. These include a high degree of test automation, careful architecting of test tools, and use of random scenario generation for the testing of some of the software’s capabilities.

We conclude with discussions of quality metrics and comparisons with other missions, and a description of current and possible reuse of Aquarius FSW.

2. MISSION & FSW REQUIREMENTS OVERVIEW

The primary goal of the SAC-D/Aquarius mission is to produce global maps of Sea Surface Salinity (SSS) on a monthly basis. SSS is a key variable in understanding water cycles, ocean currents, and glacial processes, to name a few. TBF
integrated by the Argentine space agency, CONAE. While Aquarius is the primary instrument of the mission, SAC-D hosts other instruments as well. The spacecraft has been in a Sun-synchronous, low Earth orbit since June of 2011.

The Aquarius instrument consists of radiometer, which takes the primary measurement of L-band brightness temperature, and a scatterometer, responsible for acquiring a secondary, corrective measurement of water choppiness.

The high-level block diagram in Figure 1 shows the FSW and the key subsystems with which it has interfaces, some direct, some indirect. There are three boards in the chassis with the computer upon which the FSW runs, which in turn provide communication with other subsystems, including the spacecraft bus, the radiometer, and the scatterometer. Each of the four active thermal control assemblies is responsible for one thermal region of the instrument (which includes, in addition to the radiometer and scatterometer, a deployable 2.5 meter antenna, 3 large feedhorns, and a power distribution unit, none of which are shown in the diagram).

The diagram shows science data flowing from the instruments to the FSW, which in turn packages it into science blocks of combined data along with a great deal of housekeeping telemetry, and sends it back to the spacecraft (the SciData flows), via the communications board. Ground commands are received from the spacecraft bus, again via the communications board, and the FSW sends commands to the instrument components via the scatterometer processing board.

The scatterometer processing board generates a 100-Hz interrupt, which the FSW uses to drive its thread scheduler.

Not shown in the diagram, there are two physical links between the communications board and the spacecraft bus: a redundant MilStd 1553 bus for commands and telemetry, and a high-speed serial bus for science data.

The following list of functional capabilities gives the FSW requirements in a nutshell:

1) Command reception (1553) and processing
2) Science data collection from the radiometer and scatterometer,
3) Science data formatting, storage and output at .5MB/s
4) Analog and digital housekeeping telemetry collection from all subsystems
5) Housekeeping telemetry formatting and transmission (1553 bus)
6) Radiometer command & control
7) Scatterometer command & control
8) Reflector deployment mechanism temperature control

The science data collection, formatting, and sending rate group runs at 100 Hz, while 1553 communication with the spacecraft occurs sporadically based on message traffic.

3. ARCHITECTURAL APPROACH & FEATURES

Armed with an understanding of the FSW’s requirements, we can present our approach to architecting the FSW.

Architectural Principles

We begin with a set of principles and guidelines that govern the design:

1) Achieve a high degree of modularity, a clear partitioning of the software with clear assignment of distinct responsibilities to each part: One of the most fundamental problems of software engineering is complexity, both intrinsic and domain-derived. Dividing software into manageable pieces is a fundamental necessity to reduce complexity.
2) Maintain visibility, clarity and minimalism of interactions among parts of the FSW: Developing, testing, debugging, maintaining, understanding and explaining the software are all made easier the more this principle is achieved, and these in turn are crucial to the reliability and success of the software.
3) Make information visible in the smallest scope possible:
This is related to the previous principle, in that it tends to avoid unanticipated interactions among parts of the software. It also reduces complexity by minimizing the amount of information present in any given context.

4) **Follow patterns**: for tasks or logic that must be done more than once in the software, use a pattern, instead of doing the task in an ad hoc way easy time. Patterns can be implemented once, thoroughly analyzed and tested, and reused.

These principles are not unusual, and software engineers have been striving to work by them for decades. Our contention is that an Object-Oriented approach, as opposed to traditional structured programming, and a Component-based architecture, naturally enable and simplify the realization of these principles to a high degree. Some of the features of OO design that make this so are:

1) **Polymorphism**: the concept that a given type (including behavior declared by that type) may be inherited and refined by more specific types (which inherit the attributes and behaviors of the general type)

2) **Substitutability**: an application of polymorphism as well as information hiding, by which different implementations of the same interface (expressed in a base type) can be swapped without an impact on the user of the interface. This allows easy replacement of parts of software, enabling better, more complete, more flight-like testing at all levels, as well as incremental development

3) **Object Paradigm**: this is a paradigm in which software engineers look at software as sets of things that have behavior, rather than as a call tree of behaviors. The concept involves objects, or data hidden behind behavior, which is a natural extension of the principle of information hiding. It’s worth mentioning here that, while OO software engineering has permeated and become the norm in most domains of software engineering, it still is not as prevalent in flight software.

**Component Architecture**
In a Component-based architecture, the running software consists of a set of objects called Components, working in tandem to perform the required functionality. A simplified runtime view of Aquarius FSW, showing Components and abstract data flows among them, is given in Figure 2, in order to lend a clearer idea of what we mean by Component. A Component is a key building block of the software, and each component fulfills a unique role in the system. So for example, the object RadiometerCmp shown in the figure is a Component that has the job of containing all of the FSW’s “knowledge” of and interfaces to the radiometer instrument.
The Component named CommBoardCmp embodies all of the FSW’s "knowledge" of and interfaces to the communications digital electronics board, and so on. A Component may be thought of as an application program.

Component architecture, which is enabled by and an extension of OO design concepts, further enhances the attainment of the principles. In this approach, the definition of the interfaces between components (which is to say, the definitions of the components), is a primary concern and architecting activity. Rigorously defining the interfaces between components forces the engineer to concentrate on understanding and defining the interactions among parts of the software, which enhances the clarity and visibility of the interactions within the software.

Another key aspect of Component architecture is that Components are deliverable in binary form, which means they must be able to be compiled and linked in isolation. To be sure, there are common elements needed to compile the Component - e.g. framework libraries, or the public interface declarations of other Components - but once these shared interfaces are defined and stabilized, each Component can be developed largely independently. To achieve the ability to compile Components in relative isolation requires that the dependencies on other source packages be carefully understood and controlled. This has the desirable side effect of making it more difficult to introduce unexpected, invisible interactions among the software parts.

A key part of the Component management framework in Aquarius is a Component Manager (CM), which is responsible for all Component creation, connection, etc. For example, during initialization, the CM instantiates all of the Components that will make up the running FSW.

The framework also consists of a set of interfaces used for managing the initialization and management of the Components and their connections. We call some of these interfaces architectural interfaces, to differentiate them from operational interfaces. The architectural interfaces are used only during initialization, swapping or adding Components, and finalization, while the operational interfaces are used when the FSW is in normal, functional mode. The architectural interfaces allow the creation, connection, disconnection, and finalization of the set of Component instances that make up the running software.

These interfaces are shown in Figure 3. The architectural interfaces are shown in light turquoise and in light green, and a few operational interfaces are shown in violet. Some of these interfaces may be provided by a given Component (i.e. implemented and available for calling on that component), and others are provided by the CM (i.e. available for calling on the CM by Components). The former are shown in light turquoise in the figure, and latter in light green.

The abstract class Component shown in the figure is a base class for all Component implementations, and it provides an implementation of all of the turquoise-colored architectural interfaces. The implementation it provides for each is trivial, i.e. does nothing successfully. So for example, a call to installHandlers on this class does nothing but immediately return successfully. A given Component must override the base class’ trivial implementation of a given architectural interface only if that Component needs to provide a non-trivial implementation of the interface. So for example, if a Component needs to have one or more threads created,
then that Component must override the TaskOwner interface to register those threads with the Scheduler implementation provided by the CM in the scheduleTasks call.

To see how these interfaces are used, consider this fragment of the scenario of initialization: for each component ComponentX, the CM invokes the operation installHandlers (which is possible because ComponentX implements the interface CommandProcessor), passing to ComponentX an implementation of the interface CommandDispatcher. That latter interface is then used by ComponentX to register its CommandHandler implementations (one for each unique command that ComponentX handles).

This is the pattern for establishing connections among Components: the CM invokes the architectural interfaces provided by each Component, in so doing providing an implementation of the relevant interfaces that the Component needs to carry out the connection (i.e. registration) process. In this way, the CM gives each Component the opportunity to connect into the architecture.

The CM does not “know” the true type of any of the Component instances; it knows only that all of the Component instances are derived from the abstract class Component (shown in the figure), and therefore that it may invoke all of the architectural interface operations on each. This is an important usage of the concepts of polymorphism and substitutability, and it is a key feature that enables the flexibility needed for testing in different environments, which we’ll discuss later.

The operational interfaces shown represent activities that occur only after the architecture is configured. For example, the handling of commands, a normal functional activity, is done through various Components’ implementations of the CommandHandler interface.

The doCycle operation of the Periodic interface is the mechanism by which a periodic thread is given the CPU each period - also an operational activity. In Aquarius FSW, the science manager Component provides an implementation of Periodic upon which doCycle is called at 100Hz to perform science data management.

Figure 4 helps in understanding how polymorphism and substitutability are again used to achieve significant changes in FSW behavior by swapping out an old version of a Component for a new version. The class ComponentX in this figure represents a typical Component. Note that it is derived from the abstract Component base class, the same one that appeared in Figure 3. All of the functionality of the initial version of ComponentX is implemented in the class ComponentX, represented in the figure by the virtual functions operation1(), operation2() and operation3(). However, the class ComponentX is still abstract because of the inherited pure virtual function getVersion(), and so the actual implementation class of the initial version of ComponentX is ComponentXV1. Now suppose at some time during the mission, a change is required in the behavior of existing functionality of ComponentX, as well as the addition of some new behavior. These changes are accomplished by defining a new class, ComponentXV2, in which only operation2() is overridden (thus, the code of the other two operations are reused, unchanged).

ComponentXV2 also has a new operation, newOperation(), which represents the desired new functionality. The instantiated new version of ComponentX, an instance of class ComponentXV2, thus contains code both old and new. This means that it is only necessary to upload the new code (the implementations of the functions of ComponentXV2), and not the old code that is already part of the FSW.

**Design for Reuse**

There is a tendency to think of reuse as being important only in a multi-mission sense. As important as reuse between missions is, reuse internally, within the same software subsystem, has important benefits. But reuse on any level doesn’t just happen; it has to be designed for by carefully separating the general from the specific, layering the software to separate the concerns of each. Designing for reuse forces the definition of minimal and clear interfaces. Internal reuse avoids doing the same thing in multiple ways, decreasing the possibility of errors, and increasing the value of time spent testing. To achieve higher reuse, Aquarius FSW is divided at the root source-code level into two parts: a framework, which is not Aquarius-specific: and the Components’ source code, which tend to be highly Aquarius-specific.

The framework part of the code provides the skeleton of a solution for common but important tasks. To name the most important ones:

1) Handling the validation and routing of commands among the Components of the FSW
2) Handling the collection and bundling of housekeeping telemetry

---

**Figure 4. The inheritance pattern of Components.**
Figure 5. The definition of Component classes and specific UML interfaces implementing the data flows shown in Figure 2.

3) Managing Components: instantiating, connecting, disconnecting, and finalizing them

4) Wrappers around a small number of key operating system capabilities, including queues, threads, and semaphores. These wrappers lend the using code OS and platform independence, for the supported platform and OS configurations.

The code for the Components tends to be highly mission-specific. For example, one Component embodies all knowledge of the radiometer that is contained in the FSW, and is responsible for the management of the radiometer, including all command & control, monitoring, and science data collection. Another Component is responsible for all interfaces between the FSW and the communications board. These functions are generally highly specific to Aquarius.

4. Techniques & Processes

The Use of UML

The Unified Modeling Language (UML) is an enabling technology for Object-Oriented design in general, and for Component architecture as well. UML was made for expressing OO-based architecture and design, and the concept of Component is contained in UML and is fairly well developed. We found the use of UML highly effective for the following activities:

1) Initial architecting of the FSW, including describing the context in which the FSW operates, and a high-level description of the requirements

2) Requirements organization and presentation

3) High-level design of the FSW, definition of Components and interfaces between them

4) Detailed design of the FSW, using Class Diagrams and behavioral diagrams such as State Machines, Activities, and Communication Diagrams.

5) Partial FSW code generation; the tool we used generates the declarations of classes and operations in C++. It also is able to read C++ header files and build or refine a model from them.

6) Partial test tool code generation. Our UML tool was quite good at generating and round-tripping in Java, which we used for test tools.

Design Process: Defining the Components

One very key activity in architecting a Component-based software system is defining the components and the interfaces among them. We found an effective procedure for accomplishing this. The most directly applicable of our previously-listed architectural guiding principles here is part of 2), to maintain minimality of interactions among parts of...
the FSW. This implies finding a decomposition of logical responsibilities into Components that results in the least and simplest possible communication among them.

Using a white board, and the kind of diagrams shown in Figure 2, it is easy to try out various candidate decompositions of the FSW into Components, sketching out the data flows them. Reducing the number and complexity of data flows in each candidate decomposition informs the decision among the alternatives.

Another key guiding principle we identified in defining the Component decomposition is that the software should follow the physical modularity of the hardware. In Aquarius, there were three custom-made circuit boards with which the FSW had a direct interface. We wanted to be able to test in configurations in which any combination of those three boards being present or not present could occur.

If a particular board is not present, it needs to be simulated in software, and this is easily accomplished by creating a simulation version of the Component or Components that implement the interface to that board. Clearly, if a single Component has a direct interface to more than one board, then it’s quite difficult to accommodate having one or the other of the boards, but not both, in the software. Also, the ease of substituting just one Component to simulate a non-present board is better than substituting multiple ones; thus, it’s best if there is one software Component for each discrete, removable hardware piece.

Again referring to Figure 4, the simulation version of ComponentX is implemented as the class ComponentXSim. Note that, because that class is derived from the ComponentX base class, the simulation version can exercise much of the flight code, which resides in the ComponentX class.

Having identified the Components and the data flows between them, the next step is to convert the data flows into actual UML Interfaces, with function signature definitions. This step includes making decisions about whether a given interface operation is synchronous or asynchronous, and whether data is pushed or pulled.

Refer again to the simple runtime diagram of Figure 2. The Component instances and data flows shown in that diagram have been converted into specific classes and interfaces in the diagram of Figure 5. For example, the data flow labeled “msg” going from the CommBoardCmp object at the top of Figure 2 to the CommandCmp is shown to be implemented, in Figure 5, as the interface MessageSink, and its being provided by the CommandCmp, and used by the CommBoardCmp.

In the model, the MessageSink interface has a single operation in it: acceptMessage(). (This is not shown in any of the diagrams in this paper). From this model, the header file for the code of interface MessageSink can be generated by the UML tool.

The Use of C++

There has been much debate in the FSW community about the relative benefits and suitability of C versus C++ for FSW. C has been the dominant FSW language in our institution. Among the concerns about using C++ for FSW are:

1) Code bloat, excessive executable size
2) Hidden dynamic memory allocation
3) General complexity
4) Performance.

Before discussion each of these, we need to describe our restrictions on the use of C++ for Aquarius FSW. We use the full C++ language with the following exclusions:

1) No use of Runtime Type Information (RTTI). This includes not only the typeid operator but also dynamic_cast. This feature makes linking more complex, executables larger, and significantly complicates pointer logic.
2) No use of exception handling (try, throw, catch). This causes dynamic memory allocation beneath the covers.
3) No use of multiple or virtual inheritance. This feature can cause virtual functions to run slightly slower, and it implies the use of RTTI.
4) No use of standard string, IO streams, and other standard library features. These can tremendously increase the size of the image, and are not really required in FSW anyway.
5) No extravagant use of templates. We do allow templates, but sparingly. This avoids code bloat.

These restrictions on the language make it possible to avoid realizing the concerns outlined above. We avoid code bloat by carefully using templates, and by avoiding unneeded library features and RTTI. We avoid hidden memory allocation by avoiding exception handling, and also by an educated and careful use of Standard Template Library (STL) container templates, i.e. by allocating memory for these containers up front, and knowing how to use them so that they do not allocate memory thereafter.

The complexity argument, we find, is a combination of unfamiliarity with OO concepts and ways of thinking, as well as a reaction to indeed overly-complex use of inheritance and template usage, often perpetrated by new users of C++. Aquarius’ FSW design and code has been presented and reviewed extensively, and our experience tells us that it is not more complicated than other FSW designs of similar size.

And finally performance: The ISO Technical Report on C++ Performance (see [3]) thoroughly examines the performance of C++ in every aspect of execution, and concludes that C++ is in some respects (e.g. virtual functions in a multiple-inheritance hierarchy) very slightly slower than C, and in many respects no different at all. Our experience agrees. Aquarius FSW has quite demanding performance requirements, among them handling interrupts at rates in excess of 100Hz while sending downlink data at 1/2 MB/second, which it meets easily.

5. THE USEFULNESS OF PLUG & PLAY COMPONENTS

To date, the run-time modification capability has been extremely useful in flight as well as testing situations. It has allowed lower-risk and quicker problems fixes in flight and in integration and thermal testing. It has enabled tests on a system level that would normally be possible only in white-box context, allowed rapidly prototyping design modifications and problem fixes, and served other utility purposes.
Following is a brief discussion of some of the applications of this capability.

1) **Fixing problems after the flight instrument had been assembled**: In integration testing with the flight Radiometer, a minor FSW bug was found (not copying a status word from the instrument into the science header). Later still, a deficiency in the design for reporting of radio frequency interference was detected, which had to be fixed by using some spare words in the science block for additional RFI status bits. However, overwriting the EEPROMs at that late date was judged too risky, and the decision was made to fix these problems by uplinking modified components. This was done in flight after the instrument was originally turned on. These fixes together required uplinking modified versions of three Components.

2) **Writing the EEPROMs**: Late in the development, we realized that writing the EEPROMs using a patch instead of the traditional serial port-based EEPROM writing application would save us from having to modify the flight hardware configuration to enable the serial port if we wanted to write the EEPROMs after the flight system had been assembled. (This would also open the possibility of writing the EEPROMs in flight, though this is not planned.) Thus the EEPROM Writer component was developed to write a flight software image to either (or both) of the two on-board EEPROMs. This component is not part of the original flight software component configuration. It is itself uploaded as a new flight software patch. There are two on-board EEPROMs; one on the CPU board and the other on communications board. The EEPROM Writer component is capable of writing to and verifying of either EEPROM, selected via ground command. The EEPROM Writer component, of class **EepromCmp**, consists of an **EepWriter** and **CmdHandler** classes, as shown in Figure 6. During the connection process, the installCommandHandlers method registers all the new ground commands needed for uplinking a complete FSW image, with unique op-codes and their associated command parsers. The scheduleTasks method adds the EepWriter object to the schedule as a new periodic task with pre-selected priority. The EEPROM writes must be timed carefully, and this Periodic runs at 100 Hz to accomplish this. Once installed, the EepWriters CmHandler object wakes up on a series of ground commands to upload a new flight image. The image is written to one of the EEPROMs in chunks during periodic calls from the Scheduler to the EepWriter.

3) **CPU stress test**: we uplink a new component whose sole purpose is to waste CPU time at a high priority, and to progressively use more of it as time goes on. We want to make sure that the performance of the FSW degrades in a predictable way. This patch is called the CpuHog.

4) **Prototyping a different design of 1553 driver**: We wanted to optimize the driver, but we wanted to try it out before changing the baseline and overwriting the EEPROM, so we made a patch of the component that contains the driver. After debugging the driver, we decided to incorporate the new design into the baseline.

5) **Analog telemetry reading changes**: We had to make a change in the FSW timing of reads from an analog accumulation register because the board required more time than its specification said it did. We made the modifications necessary in a patch of the component that contains the interface to that register, and, after some experimentation and modifications (over several patch uploads), we settled on the best fix and made it part of the baseline.

6) **Quick fix in thermal testing**: In thermal testing, we found that a serial bus driver chip became very noisy above a certain temperature, and had to change the FSW to handle this, which included disabling an interrupt and other changes. Under
extreme time pressure, we quickly put the fix into a patch, tested it, and then incorporated it into the baseline. Doing the patch let us test the fix in the thermal chamber before having to write the EEPROMs with a new edition of the FSW.

7) Fault injection in system-level patching tests: we have several components that intentionally fail at different steps in the patching sequence so that we can test the FSW’s response to these failures.

8) Cleaning up after and replacing a suspended task: on one of our system test scenarios, we intentionally peek an invalid address, which causes the thread that handles the peek command to get suspended. We don’t want to reboot, so we uplink a replacement patch for the component containing that thread. The finalization process of the replaced component deletes the original thread, and installing the replacement creates a new one. In this case, we install a new instance of the very same version of the component. This could avoid a reboot in flight.

We believe that the effort of designing and implementing the Component management framework and its application to Aquarius FSW, as well as the additional work of thoroughly verifying the Component replacement logic, has been amply rewarded.

6. VERIFICATION

Verification of the FSW, both in terms of implementation of the requirements and also of a high degree of reliability, takes place by both analysis and by testing. Testing is preferable, we resort to analysis when testing is not possible or practical. Testing takes place on two levels: “black box”, system tests, in which only the flight data paths in and out of the FSW are available; and “white box”, subsystem tests, which are implemented as C++ applications which have access to the internals of the parts of the software under test.

We spent a great deal of time writing test tools and test code; in fact we wrote about half again as much test code as we did flight code, and it was well worth it.

An important aid for testing: by wrapping OS functions in a C++ interface, our FSW could run on both the target PowerPC under VxWorks, and on the Solaris/SUN target, using native Solaris Posix threads, queues, etc. Thus, we could run almost any of our tests on either the workstation or the target.

Automated System Testing

We defined a set of scenarios at FSW-system level, using only the flight interfaces: 1553 bus messages and high-speed serial port for science downlink. This set of scenarios was sufficient to verify almost all of the FSW’s requirements. (There were some requirements that had to be verified on a white-box level, and a few that could only be verified by inspection or analysis.)

We automated this system testing to a high degree. We wrote several tools to achieve this:

- **Command Tool**: reads command dictionary, processes scripts with commands in simple English/ASCII format, with delay specifications per command, converts commands to binary, 1553 format commands, sends them on a socket to SIM version of communication board interface Component.

Can run indefinitely, being given new scripts to process by copying files into the input directory.

- **Telemetry Tool**: accepts “1553” messages on a socket from the SIM version of the comm board interface Component, converts binary messages into simple ASCII format.

- **Sci Data Tool**: accepts binary science block messages on a socket from the SIM version of the comm board interface Component, converts binary messages into simple ASCII format.

- **Test Executive**: Reads a suite of tests, executes each one (option to particular tests simultaneously or in series) using the Command/Telemetry and SciData Tools, evaluates results by comparing ASCII housekeeping and science data output files with expected results files in the form of Perl regular expressions.

With these tools, we were able to easily run long suites of test scenarios and quickly evaluate the results. This enabled us to do a comprehensive retest as often as we liked (subject to the limitation that it took many hours to run the entire set of tests).

**Testing Component Replacement or Addition**

Changing out a Component at runtime, or adding in a new one, seems risky, even though it is entirely common-place in ground software. But because it is not so often done in flight software, and because of the higher reliability needed, we perceived a greater risk in doing it, and therefore paid special attention to the verification of this capability in the FSW.

One of the difficult things to verify was that there were no stray interactions between successive versions of a component. For example: if Component A did not completely detach from Component B, and Component B were replaced with a new version, might Component A still have an invalid reference to the old instance? The order of Component replacement then, is meaningful. With about 10 Components that can be patched, Aquarius FSW might be vulnerable to bugs such as this, only in specific Component replacement order.

Verifying the lack of such bugs is a combinatorial problem, resulting from the ordering of Component replacement operations. To test this, we developed a program that generated thousands of pseudo-random sequences of Component replacement operations. We ran this program frequently throughout the development process. This gave us significantly more confidence that we had uncovered any bugs in the connection/disconnection logic.

7. FSW QUALITY

At JPL, software defect rates are tracked on an ongoing basis in order to understand how well (or not well) we are doing in producing quality software.

Metrics are collected on all types of software: ground system as well as flight, and categorized by type. Defect rates are also categorized by phase: development, system test, and operations. These values for instrument flight software are shown in the Table 1.

Overall, Aquarius FSW had an order of magnitude fewer
Table 1. Average instrument flight software defects per thousand physical lines of code, by phase

<table>
<thead>
<tr>
<th>Phase</th>
<th>Average</th>
<th>AQ</th>
<th>AQ/Av</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>1.6</td>
<td>.053</td>
<td>3.3%</td>
</tr>
<tr>
<td>System Test</td>
<td>1.71</td>
<td>0.16</td>
<td>9.2%</td>
</tr>
<tr>
<td>Operations</td>
<td>0.14</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

defects than the average for JPL instrument FSW projects. We should note that this dataset was collected over only five instrument flight software projects, so its authority is questionable. But it does provide a rough metric of software quality.

8. CONCLUSIONS

One frequently-heard reaction to the plug-and-play capability provided by our architecture is: That is not necessary in flight. Another is: That increases risk by reducing determinism because of the dynamic connection process.

The first objection is not surprising: After all, once the system is launched, it can hardly change so much that it would need major behavior modifications. This may be true, but it seems to us that it rarely is. Many missions, including Aquarius, have had to make significant FSW changes after integration. Moreover, this objection misses the point that important use cases for the FSW happen well before launch: namely during development and integration, and in the continuous verification that (should be) is done during all phases of development and integration. Our plug-and-play architecture, as we have seen, greatly enhances the ability to do better, faster, more flight-like testing at all phases of development.

And that addresses the second objection: by improving testing, this capability decreases the overall risk to the mission, by making the FSW better tested and thus more reliable. There is only so much test time, and if the FSW is difficult to test because of built-in rigidity, then that degrades the quality and quantity of testing, and so results in a less reliable product.

9. FUTURE WORK

The Aquarius framework (including Component management as well as the OS wrapper facility) is currently being used in a new development for the Gravity Recovery and Climate Experiment (Grace) Follow On mission (see [6] and [7]), for the FSW of an instrument called the Laser Ranging Interferometer. The framework has already proven its value in getting the LRI’s FSW up and running quickly. Also, this will give us the opportunity to make minor improvements and enhancements to the framework.

The next important step to be made with this framework is to modify it to be able to work in a partitioned memory operating system, such as Unix or VxWorks and its Real-Time Processes. This would make the framework suitable for use on larger missions with stringent reliability requirements. We hope to have an opportunity to do that in the near future.

ACKNOWLEDGMENTS

We would like to thank our colleagues on the Aquarius project, in particular the Instrument Engineer, Dalia McWatters, and the command and data subsystem engineers, Mimi Paller (lead), Andy Berkun, Mark Fischman, and Craig Cheetham for their excellent work in producing clear, stable, and exceptionally well-documented hardware and interfaces: they made the FSW team’s work much easier.

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES


BIography

Alex Murray

Alex Murray is a senior software engineer with the Jet Propulsion Laboratory, California Institute of Technology. He is currently leading the development of Grace Follow On’s LRI instrument FSW. He previously developed FSW for the Mars Science Laboratory, led the development of the SMAP spacecraft bus FSW, and previously led the development of the flight software for the Aquarius instrument. He has led and done software development for flight, ground, and simulation software for missions and for technology development projects at JPL. Previously he led and developed software for a variety of projects at TRW (now Northrop-Grumman), and he served as a system engineer for the European weather satellite agency, Eumetsat, as well as software engineer for the Dresdner Bank in Frankfurt, Germany. His experience includes embedded and flight software development, prototype and research development, OS development, AI, analysis and simulation tools, science and image processing applications, business and GUI applications, and databases. He holds BS and MS degrees in mathematics from The Ohio State University.
Mohammad Shahabuddin is a Senior member of the Technical Staff at the Caltech Jet Propulsion Laboratory. His contributions to NASA include design and development of FSW for Mars Science Laboratory, and previously for Aquarius instrument flight software, Mission Data System (MDS) flight framework, multi-mission simulation framework and software simulators for Galileo, Voyager, and Cassini compute data hardware. Object oriented principles were applied in the design and development of Aquarius software, using UML design tools and C++ language. He and his team earned NASA award for developing bit-level High Speed Simulator for Cassini Spacecraft used as a testbed for command and sequence validation. He received his M.S. degree in Electrical Engineering from California State University at Long Beach. He did his undergrad in Electrical Engineering from Engineering University Peshawar, Pakistan.