Challenges in Designing Reusable Flight Software Modules

Robert W. Vick[1]
SAIC, Albuquerque, New Mexico, 87106

The terms "modular software design" and "reusable software" have come to be a joke in much of the aerospace industry and for the government officials managing aerospace programs. Every program touts its "modular software design" but this modularity is only realized with the realm of that individual project. Part of AFRL's Space Plug-and-play Avionics (SPA) concept is the idea of modular, plug-and-play software that can be reused across multiple space programs developed by disparate teams of developers. Just as the idea of a parts depot for plug-and-play components has been advocated where you can take a part off the shelf and plugs it in and it "just works," the idea of a software application store has also been put forward. The need for an overall software architecture involved in the design of reusable software for SPA is discussed.

I.   Introduction

THE Space Plug-and-play Architecture (SPA) [1] was created to make possible the extremely rapid creation of systems that self-organize from intelligent, black box components. Just as a personal computer employs mechanisms like plug-and-play to simplify the integration of disparate components with relatively modest requirements for effort and expertise, the intent of SPA is to provide a similar paradigm for aerospace systems. An important part of SPA is the notion of smart, single point interfaces, analogous to the USB cable of a keyboard or mouse. These interfaces permit the rapid integration of complex devices by hiding their complexity behind the interface, presenting a uniform, simple standard connection. In effect, components are turned into metaphorical "black boxes". These black boxes can be freely plugged into networks that discover them and automatically organize a system around them, ideally with minimal human interaction.

The key to rapid system development is in interfaces. In the prevailing terminology, a "SPA-*x*" *interface* combines a standard physical layer technology *x* (for data transport and control) with other commonly needed "services," such as power and time synchronization. For example, a SPA-U interface is based on the USB standard from the personal computer, but combines the four pins that make up USB with other pins for power and time synchronization. A SPA-*x device* is a spacecraft component bundled with a SPA-*x* interface connector and an embedded electronic datasheet, making components self-describing (these datasheets play a role similar to the software drivers that are associated with devices on a personal computer). Figure 1 describes the current set of standard interfaces utilized in SPA systems. In every USB device on a personal computer is embedded a small bit of silicon that negotiates USB, and translate USB in a form that is easy to meld with the raw circuitry inside the device. Users never see this, of course. Similarly, in SPA, to assist developers in creating SPA devices, the concept of an appliqué sensor interface module (ASIM) was devised. ASIMs implement a particular SPA interface, serving as adapters between SPA and non-plug-and-play devices. They simplify the rendering of "raw" spacecraft components into a uniform plug-and-play model. Software in SPA-based systems automatically detects the existence of components using ASIMs (or otherwise comply with SPA interface definitions), reads the embedded datasheets, and adds the descriptions to an internal "public" database that can be queried. Components use this mechanism to "subscribe" to pieces of data advertised in this database, resulting at the highest level in the system in the sort of "magical" effect that we simply call "plug-and-play".
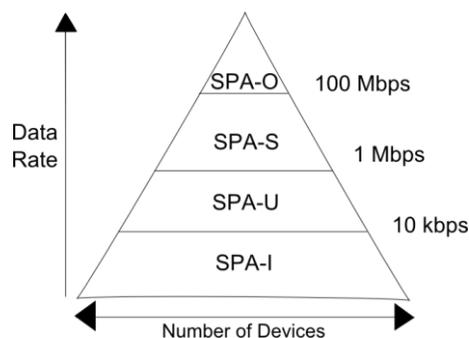

Figure 1. Hierarchy of SPA interface technologies.

---

[1] Software Engineer, Applied CBRNE and Directed-Energy Operation, 2109 Air Park Rd SE, Member.

The metaphor of reducing an aerospace system into a set of smart, plug-and-play building blocks is an enticing one, but engineering the appearance of simplicity is not easy for the developers of this infrastructure. For example, smart interfaces necessarily add overhead, and reducing this overhead is very important. In designing interfaces, this made it necessary to realize that "one size doesn't fit all". A simple thermometer has different characteristics than a hyperspectral imager. One produces bits of data per second, the other gigabits per second. Recognizing the inefficiencies in trying to cover nine (or more) orders of magnitude of component performance with a single interface, the SPA development team identified several tiers of interface, each tuned to a different level of performance.

II. Hardware versus Software

As SPA has continued to mature and test systems are developed to exercise the various technologies that enable the creation of plug-and-play systems, new approaches and ideas have been identified. One of the more recent of these is the idea of standard interface templates for component xTEDS (extensible Transducer Electronic Data Sheet). The xTEDS is an electronic data sheet stored on the component it describes and is transmitted to the rest of the SPA system at registration-time. The purpose of an xTEDS is to fully describe the information that is provided by the component to the system. This description includes everything from the electronic format of the data (i.e. 8-bit integer, 32-bit floating point, etc) to information on what the data is describing and in what units it is being transmitted. The goal is for any component of any type to be able to transmit any type of data in any possible format and for any component on the network that needs data to be able to find the data needed as well as any information about it that is needed to allow the component to interpret it correctly. Thus while a single source for the "look-up service" is necessary, although it may be distributed, most other operations can be done in a peer-to-peer fashion between individual components. These new interface templates would define a minimal set of required variables and messages for certain common component types. [2] For example, every GPS unit will produce some certain information regardless of the manufacturer, but some may optionally produce addition information. By defining a minimum set, another component developer can rely on certain types of information always being available but can still take advantage of non-standard information to enhance the operation of the component. This is similar to how a mouse on your PC can use either a default driver that will provide basic functionality or the specific driver for that device that will allow you to use the full capabilities of it.

But how do you extend this idea to software? Unlike hardware devices that have a natural set of capabilities, what a piece of software can do is driven primarily by the programmer (or programming team) and not by any physical limitations. While this does not directly impact whether or not the software can be plug-and-play, it does impact what systems the software can be used in. It is not desirable to have overlapping responsibility for the control for systems such as propulsion and power and software that might be usable in one system's software architecture may be unusable in another.

III. Software Architectures

Consider two potential architectures. The first is comprised of a set of functional modules that are tied together in a series of scripts designed to perform the necessary mission. The second is a set of modules in which each module is responsible for managing a specific subsystem within the satellite. A mission is performed through the interactions between these modules. Both of these architectures are modular designs with responsibilities divided up within the satellite by functionality.

A. Stovepiped Architecture

Examine the first example architecture, shown in Figure 2. In this architecture, each of the "pages" shown at the top of the figure, labeled "Script 1" through "Script #*n*", are simple interpreted documents used to describe the the mission operations. Each script connects a sequence of solitary executable processes that are each used to perform a single function using some simple logic. Each of the rectangles at the bottom of the figure represents an executable process. The processes are designed with the intent of performing a single, specific function that is usually tied to a single device or subsystem. For example the "Fire Thruster" process allows you to fire any thruster in the system once. Depending on the system's capabilities, it may have been written to command multiple thrusters or allow the user to command the period of the thruster firing. Each process provides an interface that allows it to be used by the system. In this case that interface will parallel that of the component or system attached to it.
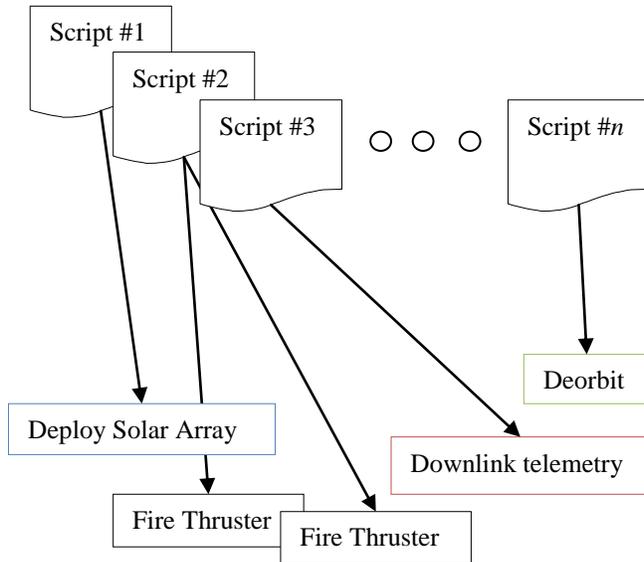
Figure 2. Example of a "stovepiped" flight software architecture.

A script may or may not always be running on the satellite. During normal operations, it is likely that all of the mission would be accomplished with little to no outside interaction but it may require a user to manually start the first script remotely. Once the sequence of scripts are started, each is executed by a script interpretation program. This program could be a piece of commercial software, such as the ubiquitous Bourne Again Shell (bash) or perl, or it could be a custom application reading scripts written in a custom satellite scripting language. For the purposes of this comparison, the choice here is not significant. It is important to note that neither the executable processes nor the scripts are running at all times, therefore there will not be one arbiter of the spacecraft's current state that will always be present over time.

B. Subsystem Architecture

A second potential architecture is based on the typical notion the subsystems included in a typical satellite architecture. Unlike in the "stovepiped" architecture, the modules in the subsystem architecture are designed with the functionality of each satellite subsystem in mind such as propulsion, power management, and communcations. Throughout the mission lifetime each of these modules remains active and acts as the final arbiter for all commands directed to devices within their subsystem of responsibility. The interfaces to these modules describe the interfaces to the combined functionalities of all of the subsystem devices. These interfaces may be reused between designs if they have sufficient functionality in common.

Figure 3 shows an example of a subsystem architecture that includes some of the modules that would be present in an actual system. Some of the rectangles at the top of the figure represent the modules responsible for managing the satellite subsystems. Note that not all potential subsystems are shown in the figure. At the bottom of the figure is a dashed rectangle that represents the capability allowing the different modules to communicate with each other. In this case, that capability is provided by the operating system's inter-process communication (IPC) implementation. Mission commands may be either uploaded from an external source via a communications link or injected directly onto the IPC bus and routed to their respective destinations.

Figure 3. Example of a "subsystem" flight software architecture.

C. Reuse

In both of the architectures described above there is the potential for software to be reused between projects intended to perform different missions. Notwithstanding the need for interfaces to the hardware devices, there is also the difficulty interfaces between software. One of the goals of the SPA program is to provide the capability for software reuse and a key element of responsive space is the concept of a depot of flight software modules that are available for all missions to use. However without a common software architecture used in the design of all of the modules, reuse is possible but not implied by the use of SPA. Consider the difficulty of attempting to use a software module from Figure 2 in a system implementing the architecture from Figure 3. The first challenge that becomes apparent is that the modules in the first architecture are not designed to be constantly running but are only intended to be run when specific events trigger it. A possible solution is to build a wrapper around the first module that will stay running continuously monitoring the bus traffic for commands destined for its subsystem. That wrapper would then call the original module when the commands indicate that the original conditions are met for activation.

However, consider the case for reusing the "Fire Thruster" module from the first approach in a design implemeing the subsystem architecture. This module would fall into the responsibility of a Propulsion subsystem module but the

3
American Institute of Aeronautics and Astronautics

propulsion system has the responsibility to do more than just fire thrusters. Depending on the system design it may need to monitor and control the temperatures of the propellant or perform more sophisticated power management. But a Propulsion module designed for the second architecture would not work in conjunction with "Fire Thruster" module from the first architecture. A subsystem module would include all the functionality necessary to manage the devices within that subsystem; including, in this case, monitoring propellant phase, temperature, and power characteristics for the devices.

A similar situation is true when taken in reverse particularly when the modules overlap in their area of responsibility. When the modules do not overlap they may potentially be able to coexist, however they would require compatible module interfaces.

IV. Conclusion

There have been many challenges to realizing the dream to reusable software. Part of the SPA approach includes the capabilities necessary to enabling widespread software reuse. However, this dream cannot effectively been attained until the software architecture is addressed. There are many options for designing flight software architecture; almost as many options as there are programmers. At minimum, each facility attempting to leverage a depot of reusable flight software modules must adopt a consistent architecture for each of the modules to operate within. Any future work to develop reusable software must consider these issues if it hopes to achieve more than yet another modular software design.

Acknowledgments

References

[1]Lyke, J., "Space Plug-and-play Avionics (SPA): A Three-Year Progress Report," Proceedings of the *AIAA Infotech@Aerospace* Conference, AIAA, Rohnert Park, California, 2007.

[2]Lanza, D., Lyke, J., Vick, R., "The Space Plug-and-Play Avionics Common Data Dictionary – Constructing the Language of SPA," Proceedings of the *AIAA Infotech@Aerospace* Conference, AIAA, Atlanta, Georgia, 2010.