Architectural Implications of Common Operator Interfaces

Richard N. Taylor **Nenad Medvidovic** **Peyman Oreizy**

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
+1 714 824 6429
{taylor, neno, peymano}@ics.uci.edu

Abstract

If a common operator interface (COI) for satellite ground systems is built with conventional user interface (UI) technology, then there is substantial risk that the UI will end up determining a significant part of the ground system's architecture. UI technology typically forces applications to be constructed as a set of call-backs to the UI. To avoid this situation an approach is required which sharply divides the user interface part of the system from the rest of the software. If the ground system is to run on heterogeneous hardware platforms a second sharp separation is also recommended: presentation and dialog decisions from platform specific toolkits. This presentation suggests a multi-level solution to this problem which is established at the architecture level and maintained in the implementation.

Introduction

The economic benefits of common (human) operator interfaces (COIs) are easy to see: operators can move from platform to platform and application to application with reduced or eliminated training. Operator error rates are reduced; "unfamiliarity" no longer is a problem. Software maintenance expenses are reduced as well, since only a single interface style exists.

Common operator interfaces can be obtained by isolating user interface functionality in one part of the application system, and then either implementing that portion of the system directly on each chosen hardware platform or by implementing it atop some single[1] user interface software substrate that has already been implemented on each target hardware platform, such as Galaxy [1].

While "isolating user interface functionality in one part of the application system" is clearly a good idea, the particular way the user interface functionality is isolated can have enormous consequences. In typical systems, whether implemented in C++, Java, C, or Ada, most user interface toolkits (whether platform specific or not) end up dictating the structure of the application code. They do this by

- assuming that only a single thread of control exists and
- requiring that application code be executed as the result of a call-back from the user interface event monitor loop.

This constraint is a very large one, and seems a bit like the tail wagging the dog. We argue that

- the focus of ground system design should be at the architecture-level,
- ground systems architectures should be chosen with the full range of ground systems domain characteristics and evolution issues in mind, and
- implementation technologies should not constrain that architecture.

We believe that with appropriate architectural design and implementation techniques COIs may be achieved without improper constraints on the architecture of the non-UI part of the system.

An Architecture-Based Approach

The key aspects of the approach we recommend spring from recognition of key aspects of the ground systems domain. First, concurrency is prevalent in the domain. Telemetry is received and must be processed very frequently, if not continuously. Human users may at any time request particular processing to be done, or displays to be shown. Other application processing may need to go on in parallel as well. Second, the ground system application may be physically distributed. Third, the user interface inputs and telemetry streams may be viewed as "event-based". With these attributes at the fore, we can recommend the C2 architectural style as appropriate for designing ground systems with a COI but without the architectural constraints that result from typical user interface technologies. Use of the C2 style in designing the applications can permit commercial UI technologies to be used in the implementation, but their effect on the overall architecture is removed as a result of the particular style of event-based communication which characterizes all inter-component interaction in architectures designed "the C2 way".

1.Whether the *identical* user interface appears on each platform is a separate question, and indeed is not necessarily a good thing. For example, some platforms have a three-button mouse, others have two, and others have a single-button mouse. From a human factors perspective it may be superior to allow some platform specific look-and-feel aspects to appear. Either strategy can be effectively pursued from an implementation perspective, and this choice does not affect the remainder of our presentation.

While the C2 style has been described elsewhere in the literature, it seems appropriate to provide a synopsis of it here. The description in the remainder of this section is adapted from the description used in [2].

C2 is an architectural style designed to support the particular needs of applications that have a graphical user interface aspect. The style supports a paradigm in which UI components, such as dialogs, structured graphics models (of various levels of abstraction), and constraint managers, can readily be reused. A variety of other goals are supported as well. These goals include the ability to compose systems in which: components may be written in different programming languages, components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple toolkits may be employed, multiple dialogs may be active, and multiple media types may be involved.

The C2 style can be informally summarized as a network of concurrent components hooked together by connectors, i.e., message routing devices. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. No direct component-to-component links are allowed. There is no bound on the number of components or connectors that may be attached to a single connector. When two connectors are attached to each other, it must be from the bottom of one to the top of the other (see Figure 1).

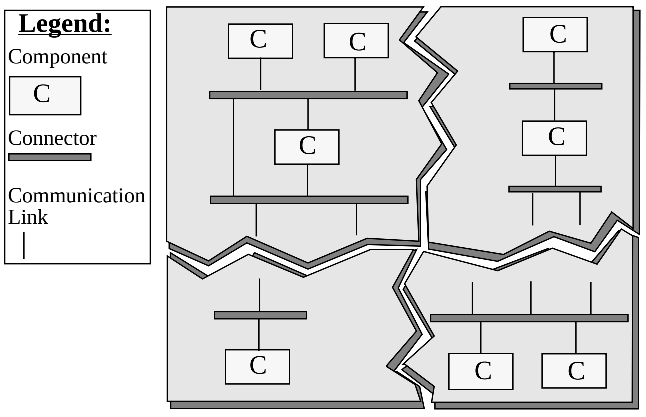

Figure 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

Each component has a top and bottom domain. The top domain specifies the set of notifications to which a component responds, and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds. All communication between components is achieved by exchanging messages. This requirement is suggested by the asynchronous nature of component-based architectures, and, in particular, of applications that have a GUI aspect, where both users and the application perform actions concurrently and at arbitrary times and where various components in the architecture must be notified of those actions. Message-based communication is extensively used in distributed environments for which this architectural style is suited.

Central to the architectural style is a principle of limited visibility or *substrate independence*: a component within the hierarchy can only be aware of components “above” it and is completely unaware of components which reside “beneath” it. Notions of above and below are used in this paper to support an intuitive understanding of the architectural style. As is typical with virtual machine diagrams found in operating systems textbooks, in this discussion the application code is (arbitrarily) regarded as being at the top while user interface toolkits, windowing systems, and physical devices are at the bottom. The human user is thus at the very bottom, interacting with the physical devices of keyboard, mouse, microphone, and so forth.

Substrate independence has a clear potential for fostering substitutability and reusability of components across architectures. One issue that must be addressed, however, is the apparent dependence of a given component on its “superstrate,” i.e., the components above it. If each component is built so that its top domain closely corresponds to the bottom domains of those components with which it is specifically intended to interact in the given architecture, its reusability value is greatly diminished and it can only be substituted by components with similarly constrained top domains. For that reason, the C2 style introduces the notion of event translation. Domain translation is a transformation of the requests issued by a component into the specific form understood by the recipient of the request, as well as the transformation of notifications received by a component into a form it understands. The C2 design and development tools, are intended to provide support for accomplishing this task.

Each component may have its own thread(s) of control, a property also suggested by the asynchronous nature of tasks in the GUI domain. It simplifies modeling and programming of multi-component, multi-user, and concurrent applications and enables exploitation of distributed platforms. A proposed conceptual architecture is distinct from an implementation architecture, so that it is indeed possible for components to share threads of control.

Finally, there is no assumption of a shared address space among components. Any premise of a shared address space would be unreasonable in an architectural style that allows composition of heterogeneous, highly distributed components, developed in different languages, with their own threads of control, internal structures, and domains of discourse.

Conclusion

Creating a common operator interface to ground systems software is a worthy goal and technically feasible. Achieving it requires design of that interface and need not involve creating a complete canonical architecture for ground systems. Unless care is taken in the design and implementation of a COI, however, an inappropriate architecture can be forced upon the rest of the system, namely structuring it as a set of routines under the control of the user interface software. Choosing an appropriate architectural style and supporting implementation techniques can avoid this undesirable situation, however. The C2 style, described above, is one suitable architectural style. It has been demonstrated in a variety of applications and exhibits many properties supportive of incremental system evolution. While other architectural styles may also be effective in the ground systems domain, we believe that any style used must reflect the key characteristics of that domain, namely, concurrency, distribution, heterogeneity of components and platforms, and the need to support continued evolution.

Acknowledgement

Effort sponsored by the Defense Advanced Research Projects Agency, and Airforce Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Reserach Laboratory, or the U.S. Government.

References

1. Visix Corporation. http://www.visix.com
2. Nenad Medvidovic, Richard N. Taylor. Reusing off-the-shelf components to develop a family of applications in the C2 architectural style. Accepted for publication in *IEE Proceedings Software Engineering.* To appear.