The Component Software Challenge for Real-Time Systems

Alessandro Pasetti and Wolfgang Pree
Doppler Lab. for Software Research, University of Constance, D-78457, Constance, Germany
E-mail: pasetti@fmi.uni-konstanz.de, pree@acm.org

Abstract

In some fields – such as desktop applications – software component technology has become established that allows software systems to be built as hardware systems are: as assemblies of pre-defined, off-the-shelf items. The component approach has led to important gains in productivity with consequent reductions in cost and development times.

Component technology has never been applied to real-time systems. Adapting it to satellite real-time on-board software is the "Grand Challenge" discussed in this paper. The discussion is made in the framework of an on-going project with the European Space Agency to redesign the software for satellite Attitude and Orbit Control Subsystems as a component-based system[1]*.*

This paper first describes the benefits of component software and then addresses the technological challenges that must be met to adapt it to the real-time constraints of satellite systems. The final section discusses the relevance of these challenges to real-time systems in general.

1 The Problem

Satellites are mission-critical systems controlled by real-time, embedded software. As in other mission-critical systems, software accounts for a significant – and growing – share of total development costs. This is largely due to the one-off nature of the software that is developed as a completely new application for each mission.

Indeed, the experience of one of the authors on several major European space projects is that reuse of satellite software across projects, when it occurs at all, concerns at most code fragments – and only if the same software team is responsible for more than one project.

[1] The views expressed in this paper are those of its authors only. They do not in any way commit the European Space Agency or reflect official European Space Agency thinking.

The software development process typically begins with the definition of user requirements. Requirements are normally formulated under the tacit assumption that all the software will be developed anew. Consequently, they are narrowly targeted at a specific mission and result in a monolithic piece of code that, while highly optimized for the application at hand, has to be developed entirely from scratch and cannot be reused in other missions.

This approach is in sharp contrast to that adopted on the hardware side. Here, designers begin by surveying the market for available components and then specify their system in terms of these components. Widely accepted standards (on bus interfaces, on electrical interfaces, on mechanical interfaces, etc.) allow components from different suppliers to be plugged together. The resulting system is perhaps not optimal in terms of mass or power consumption (the characteristics of standard components seldom match perfectly the requirements of a specific project) but it is certainly cheaper and faster to assemble than if it were developed from scratch.

This difference between software and hardware development processes is not an ineluctable consequence of a "special" nature of software systems. In fact, outside the real-time field, it is disappearing thanks to the introduction of *software component standards* [1].

The best known examples of such standards are Microsoft's (D)COM, OMG's CORBA, Sun's JavaBeans. Their function is analogous to that of hardware standards: they define interfaces and communication protocols that allow pieces of code developed as self-contained units, and potentially supplied by different vendors, to cooperate.

It is important to emphasize that software components go far beyond traditional subroutine libraries. The latter do not qualify as components because their interfaces are idiosyncratic and because calling and called code must normally be written in the same language and compiled with the same compiler. Component standards by contrast specify industry-wide interfaces that allow independent units of code to interoperate across language, compiler and even platform barriers.

This is a crucial difference as it is leading to the emergence of a market for third-party software components not dissimilar to the markets for hardware items. As a consequence, complex software systems are now coming to be developed very much as hardware systems are developed: as collections of cooperating pre-defined components. It is this new component-based approach that is largely responsible for the recent gains in software productivity in desk-top applications.

The research group at the University of Constance to which the authors belong, has traditionally been concerned with software architectures for complex systems in the business field where it has advocated and successfully applied the component approach. Recently, it started a project with the European Space Agency (ESA) to re-design the real-time software for the Attitude and Orbit Control Subsystem (AOCS) of satellites[2]. The AOCS is one of the most critical subsystems on a satellite. The question that naturally arose was: can component technology offer the same benefits to satellite real-time applications as it does to desk-top and other applications?

2 The Grand Challenge

Demonstrating that this last question has an affirmative answer is the "Grand Challenge" this paper wishes to discuss.

The expected benefits from a transition to component technology are lower costs and reduced development times stemming from the availability of reusable components that can be bought as off-the-shelf items.

Components are not an entirely new concept in the real-time field. Real-Time Operating Systems (RTOS) are examples of components with non-standardized interfaces. After their introduction, engineers no longer had to develop operating systems for their applications. Instead, they could choose a commercial product, configure it for their purposes, and integrate it with their own code.

The challenge is to extend this approach to other parts of real-time applications – sensor and actuator management, control law implementation, failure detection, etc – ultimately arriving at applications that consist mainly of assemblies of separately procured components.

3 The Lesser Challenges

This section identifies six milestones on the way to a component-based AOCS software. The technical issues that have to be addressed to pass them represent "Lesser Challenges" ("lesser" only with respect to the "Grand Challenge" of the previous section). As discussed below, some of them have already been met, others are being addressed in on-going projects and still others remain open.

3.1 Hardware Support for Components

The mechanisms through which software components interact with each other typically introduce several layers of indirection. A simple subroutine call in a traditional architecture may get translated to a chain of calls in a component-based system. Component technology, in other words, entails considerable overheads in both memory and CPU usage which is one reason why it has not spread to real-time systems.

Processors currently in use in most space applications are based on the (16-bits, CISC) mil-std 1750 architecture. In the case of the AOCS software, its capacity is already exploited to the limit with memory and CPU margins being close to zero at the time of launch. Hence, the first step towards the adoption of software components in space is the development of a space-qualified processor with memory and speed performance perhaps 2 to 5 times greater than that of 1750 processors.

This step has already been taken with the space-qualification by ESA of the ERC32 processor [2]. This is a 32-bits processor with floating point unit based on the SPARC V7 architecture. Its memory and CPU power are in the same range as those available on ordinary desk-top computers making it adequate to support component technology.

3.2 Software Support for Components

Satellite software is traditionally written in Ada83 with sprinklings of assembler. This choice is justified by the safety features offered by the Ada language but is inadequate to support component technology which relies heavily on object orientation.

A first challenge is therefore to identify an object-oriented language suitable for critical applications. The obvious options are Ada95 and C++. ERC32 development environments exist for both. In the AOCS project, the choice will probably be C++ because of its support for multiple interface inheritance which Ada95 lacks and which is important for constructing a component-based architecture.

C++ is not as "safe" a language as Ada but when making the transition to a component approach, language issues become less important. Language choice affects mainly the "inside" of a component (which is just an ordinary application). However, when a system is built by assembling pre-defined components, one assumes that the components are error-free as they should have been tested by their suppliers (RTOS buyers are seldom concerned with the language in which the RTOS was originally written!). Attention therefore shifts to the component

[2] Estec contract No. 13776/99/NL/MV. ESA technical officers are: J.L.Terraillon and T.v.Overbeek of ESA-Estec.

configuration and assembly process itself that takes place at a level higher than that of an ordinary programming language.

Here, work remains to be done as methodologies for component-based software assembly do not yet exist. This has not prevented the adoption of components in desk-top applications but it is a definite obstacle in the space sector where safety concerns demand adherence to tested and proven formal procedures (eg. ESA's HOOD [3] or HRT-HOOD [4]).

A second challenge therefore lies in the definition of methodologies for the development of component-based applications. This is a challenge that, to the authors' knowledge, is still open.

3.3 Component Architecture

Satellite software is currently developed as a monolithic mission-specific application. A component approach is only useful if an application can be broken up into smaller, loosely coupled, units.

In the AOCS project, a preliminary architecture has been proposed for the AOCS software that reconceptualizes it as a set of independent components encapsulating functionalities that are potentially re-usable across missions [5, 6]. Reusability was achieved by splitting the *management* of a functionality (which is mission-independent) from its *implementation* (which remains mission-specific). The approach is the same as taken by RTOS developers who separate the scheduling of tasks from their implementation and encapsulate the former function in re-usable components.

Thus, the new AOCS architecture provides re-usable components to handle telecommands and telemetry, to manage units, to perform failure detection tests, to manage failure recovery strategies, etc. Each such functionality is packaged as a component exposing operations that allow it to be configured to suit the needs of a specific AOCS. The reconfiguration is done at run-time and the component itself can be re-used as a binary entity (much as an RTOS is reused).

The proposed architecture will be tested next year on an ERC32 breadboard and this challenge is regarded as on the way to being met.

3.4 Real-Time Component Standard

A component standard specifies the kind and format of information that components exchange. At the most basic level, it specifies the way in which code in component A can call code in component B, covering issues like call syntax, parameter passing conventions, bit ordering, etc.

Existing standards (CORBA, (D)COM, JavaBeans) are unsuitable for real-time applications because they do not address issues of timeliness and predictability of service. There is no way for component A to specify that its call to component B should be serviced within a certain time and there is not even a way to put an upper bound on the time component B will take to service the request.

Access control is another component interface aspect not covered by conventional interface specifications but essential in critical systems (and most real-time systems are critical). Such systems typically need a way of restricting access to critical services to a small number of authorized clients. For instance, in a satellite control system, the operation to reconfigure a set of redundant sensors should only be callable from the failure detection manager or from the ground control centre. Component interfaces should ideally include information on which operations can be performed by which clients.

The challenge, then, is to extend the interface between components to include, at a minimum, access control and timing data. This challenge remains wide open although efforts are being made to address the timing part. An ORG working group has proposed a specification for a real-time CORBA [7, 8]. Sun is considering a real-time extension of Java [9], possibly the first step towards turning JavaBeans into suitable vehicles for real-time components but this remains a long-term goal.

The TAO project [10] offers an interesting alternative approach. Its designers, rather than waiting for a new specification, have provided an implementation of the current CORBA standard that guarantees that calls across component boundaries preserve priority levels and that the overhead in servicing a call request is statically predictable. This makes the ensuing system amenable to the same static schedulability analyses as are used in traditional real-time systems. After the architectural design phase is completed, a TAO solution will be considered for the AOCS software.

3.5 System Robustness

Satellites are critical systems with stringent reliability requirements. When computing overall system reliability, the software is generally assumed to have a reliability of 1. This is because software can only fail if its design is defective and current satellite failure protection strategies do not cover design errors. Satellites must withstand single failures but failure is understood as "failure to meet stated specifications". There is no protection against *design* failures.

Single failure robustness is usually obtained through redundancy: if, for instance, a sun sensor fails, the system switches to a redundant sun sensor. However, prime and redundant equipment are identical and therefore if the failure was due to a design flaw, redundancy offers no protection.

Software failures in space systems are regarded as catastrophic events catered for only through a "safe mode" which merely guarantees the physical survival of the satellite but does not allow it to continue its mission.

This approach works (but only just!) when the complexity of the on-board software is comparatively low. When using very complex software, as component software is bound to be, the notion that "there are no design errors" may become untenable and new failure handling strategies must be introduced to cope with them, perhaps by introducing intermediate levels between "fully operational" and "safe mode".

Failure handling and failure recovery for complex systems are grand challenges in themselves and the issues they raise are far from resolved or even well-understood.

4 Conclusions

The Grand Challenge discussed in this paper is the transition to a component approach for the AOCS software. The attendant technical challenges relate to the adaptation of component technology to the real-time environment of satellite software.

Hardware and software environments are now available that meet both the requirements of real-time space systems and those of component technology, although work remains to be done to define design methodologies for component-based applications. The AOCS architecture is being redesigned as a collection of mission-independent, reconfigurable components. However, crucially, component standards have not yet been adapted to cater to the needs of real-time systems and the implications of using very complex software on failure handling philosophy have not yet been explored. Both these challenges must be addressed if the benefits of components are to become available to satellite real-time systems.

This paper has focused on satellite applications but many of its considerations are applicable to real-time systems in general. They too, and for the same reasons, can benefit from the introduction of software components. The technical challenges are the same: migration to higher performance hardware platforms, adoption of object-oriented languages, definition of real-time component standards.

Real-time software is traditionally treated as "special". It differs from "conventional" software in several key respects summarized in the next table:

Real-Time	Conventional
16- / 8-bits processors	32-bits processor
CISC architecture	RISC/SPARC architectures
Kilobytes memories	Megabytes memories
C/Ada83/Assembler	C++/Java
Monolithic application	Components

The technical challenges outlined in this paper cover all the differences listed in the table. Thus, the Grand Challenge discussed in this paper can be extended to all real-time systems and can be reformulated as an attempt to close the gap between real-time and other software and to apply to the former the same techniques that are bringing so many benefits to the latter.

5 References

[1] C. Szyperski, *Component Software*, Addison Wesley Longman Limited, Harrow (UK), 1998

[2] http://www.estec.esa.nl/wsmwww/erc32/erc32.html

[3] http://www.estec.esa.nl/wmwww/WME/oot/index.html

[4] A. Burns, A. Wellings, *Hard Real-Time Hood*, Elsevier, 1995

[5] A. Pasetti, W. Pree, *A Component Framework for Satellite On-Board Software*, 18-th Digital Avionics Systems Conference, St. Louis (USA), Oct. 99

[6] A. Pasetti, W. Pree, *A Component Framework for Real-Tiem Systems*, Paper submitted to 20-th Real Time Systems Symposium, (Work-In-Progress Session), Phoenix (USA), Nov. 99

[7] N. Murphy, *Introduction to CORBA for Embedded Systems*, Embedded Systems Programming, Miller Freeman, Oct. 98, p. 60-73

[8] http://www.omg.org/homepages/realtime/index.html

[9] http://java.sun.com/aboutJava/communityprocess/jsr/jsr_001_real_time.html

[10] http://www.cs.wustl.edu/~schmidt/TAO.html