Position Paper
Workshop on Evaluating Software Architecture Solutions

The Future of Software Architecture: Special Issues for Complex, Embedded Applications

Prepared by
Carolyn Boettcher and Richard Falcioni
Raytheon Company
El Segundo, Ca.
Cbboettcher@west.raytheon.com

This position paper on future software architecture is written from the perspective of the types of software applications that are embedded in Raytheon products, particularly those that have hard, real-time requirements. The software in these applications is constrained by the interfaces with various specialized hardware devices. In addition, the correctness of the results is dependent on their timeliness. In particular, the predictability of the amount of time that elapses between the initiation of a request and the time that the request is satisfied is an important quality of the architecture. Moreover, the platforms in which the software is embedded have very long life times, often 20 to 40 years. During the system lifetime, the software and hardware is expected to evolve to incorporate new technologies and provide additional functionality. The adaptability of the software architecture to facilitate this evolution is a second critical architecture quality. However, despite these special issues, domain-specific software architectures are seen as the centerpiece of a long term strategy for creating future affordable, adaptable, software-intensive products.

Over the past fifteen years, Raytheon developed and used a common software architecture in several airborne radar systems. However, the software architecture was closely tied to a particular processing architecture, so that it was difficult and not cost effective to port it to new generations of processors. With the rapid advancement in processors and networks, it is highly desirable to be able to insert new processing technology into systems with minimal software impact. For that reason, a software representation is desired that is independent of the hardware architecture, but that facilitates mapping the software onto alternative processing architectures.

A further deficiency in past architecture efforts was the lack of isolation of hardware/software interfaces, so that considerable reprogramming was needed to interface with different hardware devices. Because the hardware of interest includes sensors and other specialized devices, their interfaces are not expected to be included in general purpose, standard APIs being established in the commercial sector. Therefore, representations are needed that enable the establishment of domain-specific hardware/ software interface specifications.

One of the most important architectural concepts to emerge in recent years is that of plug and play. Inherent in this concept is a presumption of certain system and module design standards, the most fundamental being that the plug and play unit is designed to be completely configurable by means of software. To facilitate software controlled configuration, the plug and play unit must provide the system with information about the services it offers, the resources it requires, and, if necessary, the driver software that supports it.

There are several approaches being taken to extend plug and play, e.g. Jini from Sun Microsystems, and Universal Plug and Play from Microsoft. Although there are differences, both include common principles that may permit the plug and play paradigm to be applied beyond desktop systems to complex, embedded systems. Perhaps the most significant extension is the peer model for system components, where any node within an extended plug and play configuration can be viewed as a potential user of services, a potential provider of services, or both.

A second important extension is the incorporation of peer capabilities/requirements discovery . Since plug and play peers are designed so that they do not make assumptions about the peers they will serve as clients or employ as resources, the required information must be gathered at run-time. At system start up, or whenever a peer comes online, there is a period of discovery during which the new peer communicates to the system what capabilities it has and what resources it needs. In this way, every peer will have access to the capabilities and requirements of every other peer in the system.

An especially intractable issue in applying extended plug and play architectural principles to embedded, real-time systems is that of scheduling the use of shared resources. In desktop systems, the scheduler usually does not have to make any guarantees about when a resource can be obtained by a potential user of that resource. In contrast, in real-time systems, there is a need to guarantee that the most critical functions will get the resources they need in a highly predictable and timely fashion. To accomplish this, there must be an understanding of the overall objectives and priorities of the system. The question, then, is the following: Can a mission critical, embedded system be architected in such a way as to readily resolve system resource scheduling in a manner consistent with mission objectives, while minimizing configuration effort in a manner that retains the many advantages of the plug and play paradigm?

To help illuminate the issue, the assumption is made that there is a way to quantify the importance and time criticality of all system objectives such that they can be compared to each other in a meaningful and consistent way — all tied to the same frame of reference, so to speak. Then, the issue is whether or not there is a straightforward method of deriving the importance and time criticality of specific resource requests from those system level objectives, so that units and subsystems that are developed separately can autonomously adjust their resource requests in a manner consistent with the overall mission objectives of the system in which they are being configured.

This position paper has mentioned just a few of the special issues that need to be considered in representing and evaluating software architectures for real-time embedded systems. They include the need for predicting system performance, for supporting system evolution over an extended life time, and for a method of resolving resource conflicts to provide predictable performance within a plug and play paradigm. The resolution of these issues is needed if the architectural advances expected in more conventional systems are to be applied to the specialized realm of real-time, embedded systems.