Building Distributed Embedded Systems from Large Software Components

Mikael Åkerholm[1,2], Thomas Nolte[1], Anders Möller[1,2]
[1]MRTC, Mälardalen University, Västerås, Sweden
[2]CC Systems, Västerås, Sweden
email: mikael.akerholm@mdh.se

Abstract

Developers of distributed embedded systems face challenges of high demands on reliability and performance, requirements on lowered product cost, and supporting many configurations, variants and suppliers.

Component-based development is a promising software engineering approach to cost efficiently deal with software variability, reusability and maintainability. However, we claim that existing tools and methods do not address problems related to assembling applications based on large and complex commercial off-the-shelf components developed by different vendors and sub contractors.

In this paper we present our ongoing work towards a method that deals with integration problems present when assembling large software components into distributed embedded systems. Our approach makes it possible to execute the software from several electronic control units in one electronic control unit - which in turn decrease the amount of hardware needed in the system and, also, facilitates large software components to be trade units between different organisations as replacement for electronic control units (hardware components).

1 Introduction

In this paper we discuss our ongoing work on software components for distributed embedded systems. We address large systems that are developed by several organisations, where components can be classified as large and complex Commercial Of-The-Shelf (COTS) components, developed by different vendors and sub contractors. This is different from many other research projects within the Component Based Software Engineering (CBSE) community focusing on software components for embedded systems. Their focus is on smaller software components, where components typically are developed within a single company, and aimed for dealing with configuration and maintenance of a product (often a product-line), e.g., Koala [34], Space4u [30], SaveCCM [15], and Pecos [23].

Distributed embedded systems developed by several organisations are common in, e.g., the vehicle industry and in the automation industry. The systems consist of several computer nodes connected with one or several networks, where each node can be developed by different organisations specialised on different areas. For example, a modern car in the premium segment has 40 or more computers (Electronic Control Units (ECUs)), where the engine control ECU comes from a specialized engine developing organization, and the climate control ECU is developed by an organisation that focuses on the passenger cabin [14]. These different ECUs can be seen as large and complex COTS components, they contain hardware (processor, communication hardware, memories, and I/O units), and software (operating system, device drivers, and control system software).

We investigate the possibility to adopt a software component based approach in development of these systems, as replacement for the ECUs (i.e., hardware components) used today. In other words we investigate the possibilities of changing the trade unit between developers of these systems from hardware- to software-components. This is visualised by Figure 1, the future scenario we will help to realise is that suppliers will deliver software components that are partitioned on hardware nodes by the system integrator, instead of state of practice today, where suppliers deliver hardware components that are connected to a network by the integrator. The relevance of that scenario is motivated by recent efforts by industry, e.g., AUTOSAR [2] within the vehicle industry. These industry efforts indicate that major actors in the domain foresee the benefits in moving from hardware components into software components. The most important expected benefits are:

1. A decreased number of hardware components, means lower cost since each computer node requires expensive cabling and communication hardware, it means lower weight, less space, and lower power consumption.

2. Less expensive logistics related to production and deliveries.

3. Transferability of functions (software components) across different computer nodes enables to optimize the use of hardware resources throughout a system's electronic architecture.

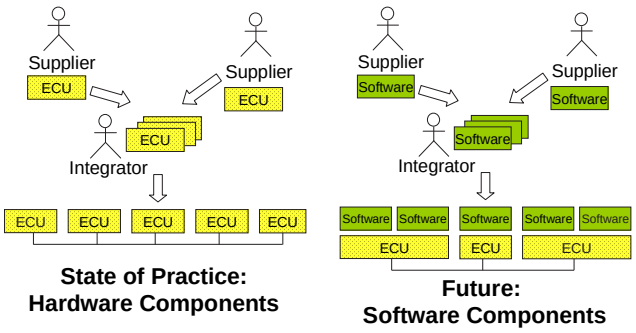

Figure 1. State of practice vs. a possible future scenario. Today suppliers deliver an Electronic Control Unit (ECU), consisting of both hardware and software that is connected to a bus by the integrator. The future scenario is that suppliers deliver software components that are partitioned on a smaller number of ECUs by the integrator.

The remaining part of this paper is outlined as follows. Section 2 addresses central technical concepts of CBSE. Section 3 discusses relevant questions when trying to apply CBSE principles on embedded distributed systems developed by different organisations. Finally Section 4 concludes the paper.

2 Component-Based Software Engineering

This section provides an introduction to central technical concepts and research within the CBSE community. For more information about CBSE we refer to Crnkovic and Larsson's book *Building Reliable Component-Based Software Systems* [9].

Fundamental to CBSE is that software applications are built from existing components. The components are to be *composed* or *assembled* into applications, i.e., combined with some connection logics that give a desired behaviour. A *component technology* is a central technical concept. It often contains various development tools for simplifying the engineering process as *modelling tools*, *analysis tools*, and *component repositories*. It provides the necessary run-time support for the components, and imposes certain patterns for assembling components. Figure 2 illustrates the basic elements of a component technology. It is a photograph of a table top in a playground, on which is placed a tray on which different building blocks can be arranged in different combinations. This playground table will be used as a metaphor for a component technology in the following description of its elements.

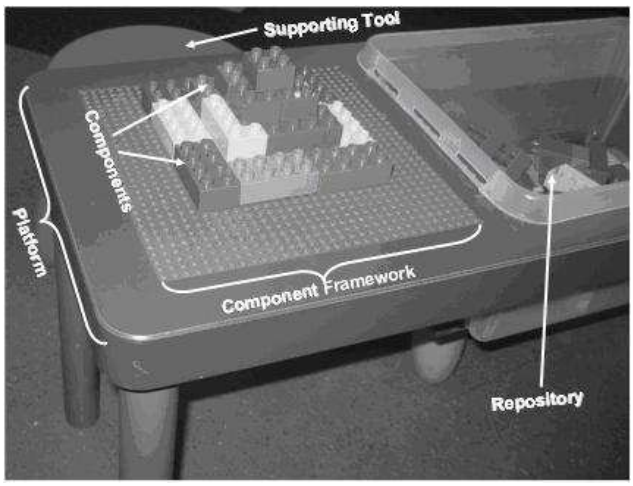

Figure 2. A component technology for building arbitrary shapes.

One of the most important parts of a component technology is the *component framework*, which provides necessary run-time support for the components not provided by the underlying execution platform (i.e., operating system or similar). In the playground table metaphor, the blocks represent the components, the tray on which they stand represents the framework which provides the components with support, and the table on which the tray stands represents the execution platform. In the metaphor the component framework mainly provides strength to the construction that is not offered by the underlying execution platform. While for software components, the component framework typically handles component interactions, and the invocation of services provided by the components, in addition to providing services frequently used within the application domain targeted by the technology.

A component technology is a concrete realisation of a *component model*. A component model defines a set of rules to be followed by users. It defines different component types that are supported by the technology, possible interaction schemes between components, and clarifies how different resources are bound to components. In our playground example, the component model is represented by the abstract rules that the children must follow when as-

sembling blocks because the blocks can only be assembled in a certain pattern. The supplier of the blocks also follows the component model when manufacturing the blocks, to ensure that the blocks are compatible with each other and the tray on which they are supported.

Software components themselves are also of basic importance. A software component can be different in different component technologies. They can be distinguished from other forms of packaged software by compliance with a component model [4]. Furthermore components shall not be mixed with objects, it is fundamental that components are not aware of each others to simplify composition and reuse. In contrast to objects that call each others in their implementation and depend of each others through inheritance, components shall be composed without touching their internal implementation (subject to external composition). However, to date several questions are discussed within the CBSE community [10]. Similar questions can even be found in the simple playground metaphor. For example, do several components assembled together to build an element (such as a wall), make a new component or should they be treated as a set of assembled components?

One of the key terms in current CBSE research is predictable assembly [26], which also represents a main difference between former approaches of integrating pieces of software, and the CBSE approach. Predictable assembly of components goes beyond integration. Integration of components is concerned with fitting components APIs together, e.g., as in current commercial component technologies Corba CCM, COM, and EJB. While predictable assembly also takes properties of the resulting component assembly in consideration, the approach is to predict the system properties based on properties of the components and how they are interconnected. In chapter 9 in [9] Stafford and Wallnau very illustrative explains the difference by the following analogy:

> *"...consider the incompatibility of connecting a very powerful audio amplifier to low-wattage speakers. The speakers will plug in with no problem and at low volumes will probably function acceptably, but if the volume is raised the speakers will most likely be destroyed."*
>
> Stafford and Wallnau

Predictable assembly take both *plug* and *play* in consideration, as in contrast to integration. The amplifier and speakers in their analogy had compatible APIs, it was possible to integrate or *plug* them, but they did not *play* well together.

3 Application of CBSE principles on distributed embedded systems

In this section we discuss the application of CBSE principles on large distributed embedded systems that are developed by several organisations. In general, such a component technology should support general domain characteristics of embedded systems, which often can be classified as resource constrained safety critical real-time systems. We stress related CBSE research that address some of the requirements of embedded systems described by Wolf [36]:

- **System architecture.** The architecture of an embedded system should be defined to serve the functions of a particular application using resources efficiently. This calls for specialised component technologies which use resources efficiently, to meet the demands of a specific application domain, not for a broad range of (embedded) applications. There are several component technologies that target the requirements of specific domains, e.g., vehicular systems [15, 20], consumer electronics [34, 30], and automation systems [23, 33].
- **Modelling.** To simplify analysis and understanding of the system, developers need models of different aspects of applications at a higher abstraction level than the source code provides, e.g., timing models for real-time analysis, and behaviour models for safety analysis. Most analysis methods in existing CBSE related work are based on some specialised models of the systems, e.g., Markov chains [28], and real-time models [6, 15].
- **Analysis.** A suitable component technology should provide support for reasoning about such quality attributes as the performance and size of systems at an early stage in the design process. This is a goal the designers of embedded systems strive to achieve. Currently much work within the CBSE community focus on predictable assembly, which can be explained as predicting system attributes from component attributes. The Prediction-Enabled Component Technology (PECT) [21, 35] is a development infrastructure, building on the idea that any component technology can be used in the bottom but composition rules enforced by the development tools guarantee critical runtime properties. Reussner et al [28] shows how reliability can be calculated using Markov chains. Furthermore, among other contributions in the field of quality attributes of component-based applications real-time attributes are considered in [6, 15].
- **Verification.** Applications must be verified in accordance with functional and non-functional specifica-

tions. For a component technology, analysis made during design stages, should be verified by practically oriented simulation and test methods. Built-in Self Tests (BIT) is a way to enhance testability of software components; the idea is that components should provide an interface for testing its specification. The European project Component+ was entirely focused on BIT [8].

However, in the above mentioned related CBSE research for embedded systems the component based approach is used mainly for dealing with reuse and product line management within a company. Software components for these purposes are small in comparison to the amount of functionality that is traded between different organisations in the class of systems we are focusing on. In the following paragraphs we will discuss, the problem, our initial research questions, and some possible paths in related research.

3.1 Problem description

On a high level we want to explore the requirements for the described type of distributed embedded systems, built form large components developed by several vendors. We want to assess if current CBSE principles for embedded systems scale to large software components, as large as trade units encapsulating applications traded between different organisations. Where current principles are not applicable we want to explore other solutions. It might be the case that it is easier, or even only applicable, to deal with small components in combination with principles as in the examples above.

We expect that the methods for, e.g., component specification, analysis, modelling, and typical services offered by the component framework will not scale to the use of large complex components. The components in the existing work is more like an encapsulation of a single function (i.e., often a function in C) with no active thread of execution at all e.g., [15, 23, 30]. In these cases the dynamics in form of allocation to execution threads and scheduling are not target for encapsulation, these things are handled in other ways. In other cases, e.g., [20, 33] the components are typically an encapsulation of a single task. To specify components encapsulating whole applications as we consider, will probably not be feasible with these methods. Furthermore analysis, modelling, and services offered by the component framework comes as consequences of how components are specified and implemented. It is naturally easier to specify and analyse something small, than something large and complex and as a consequence it might be the case that completely other types of specifications and analysis techniques are required.

Also the goals for the system integrator may be different when working with small components, in comparison to working with integration of large components. As component complexity increases with larger components, we expect optimisation to be a minor concern in comparison to controlling the increased complexity.

3.2 Research questions

In particular interest from the above discussion we find questions as:

How shall large complex software components for embedded systems be specified?

To be able to answer this question, issues like which attributes are required to know about a component and which attribute are possible determine about a component has to be addressed. The answer probably depends of the context of the application, e.g., if the application is time critical or safety critical. If the component has requirements on its environment, e.g., I/O units, processor demands, or depends on input from other parts of the system.

How to compose and model large complex software components, which architectural principles shall be used?

In most existing component models for embedded systems, the components are composed by specifying data connections between components. Connected components form a point-to-point control flow (or pipes-and-filters) architecture, which might also be suitable for bigger components. However developers of large distributed embedded systems are used to connect their hardware components to busses, which also might be a suitable abstraction when using software components.

Is it possible to apply existing analysis methods for software assembled from small components with increased size of components?

An analysis method is applicable if the input parameters it requires are available, and if it capable of handling the size of the problem. It is not enough that the method is possible to apply on problems of realistic size, it is also necessary that the desired attributes are possible to determine from the large complex components! It might be the case that one has to rely on testing or simulation for certain properties. Analysis can and should where it is possible be used early in the process on models specialised on certain critical properties, while test and simulation are more suitable for verifying the actual implementation of the whole system, as well as verification of the analysis and models.

How about possible side effects when software from different vendors share execution platform? Who is responsible if component A causes component B to fail in some way, the system integrator or the supplier of A or B?

The answer to this question may be that components

from different vendors must be separated from each others very clearly. Possibilities to interference between components may have to be totally eliminated by the system integrator.

3.3 Possible answers in related work

In order to support integration of components developed by different organisations, the runtime framework might have to provide methods for spatial and temporal isolation. In general, spatial isolation can be achieved with memory protection and protocols that guarantee a bounded time in access to shared resources as Priority Inheritance Protocols (PIP) [29]. Temporal isolation can be achieved with scheduling algorithms based on General Processor Sharing (GPS) [27] as the EGPS algorithm [17], or some type of hierarchical scheduling algorithm e.g., the proposal by Deng and Liu [12] to use the Total Bandwidth Server (TBS) [31, 32] to serve applications with own local schedulers. More recent work based on server based scheduling are some of the extensions of the Constant Bandwidth Server (CBS) [1] to also handle shared resources. Two of these extensions are the BandWidth Inheritance protocol (BWI) [19] and the Constant Bandwidth Server with Resource constraints (CBS-R) [7]. BWI allows for shared resources using a PIP mechanism. Although supporting hard real-time guarantees, BWI is more suitable for soft real-time systems. CBS-R is using the Stack Resource Policy (SRP) [5] in order to cope with shared resources. CBS-R is scheduling the whole system using servers, where each task has its own server. Looking at all these methods for system and subsystem integration, they all introduce performance compromises compared to more tightly coupled components.

A more recent research project that is related to the work presented in this paper is, e.g., the DECOS project [11]. DECOS is targeting a broad application domain, including automotive and aerospace applications. By providing a Platform Interface Layer (PIL) and a middleware with basic services, components can be developed independently allowing for easy integration of both safety-critical and non safety-critical Distributed Application Subsystems (DAS). A DAS provides a nearly independent distributed subsystem interconnected using virtual networks. The core of DECOS is the time-triggered communication system backbone. On top of this, virtual networks are supported allowing for most types of existing networking technologies to be emulated. DECOS provides both spatial and temporal partitioning, preventing overwriting memory elements of other jobs (data and code), interference among jobs sharing access to devices, as well as the disturbance of timing among jobs holding shared resources. Overall, the DECOS project is highly related to the goals of our project and will be investigated in detail.

Looking at the industrial domain, the latest automotive software standard is AUTOSAR developed by the AUTOSAR consortia [2]. AUTOSAR is scheduled to be complete in 2006, and its goal is to create a global standard for basic software functions such as communications and diagnostics. From an integration point of view, AUTOSAR provides a Run-Time Environment (RTE) routing communications between software components regardless of their locations, both within a node and over networks. Tools allows for easy mapping of software onto the existing architecture of nodes (ECUs). This mapping is depicted in Figure 3. AUTOSAR is working towards integration of standardized tools relying on, e.g., operating system standards such as, e.g., OSEK/VDX OS [25], and various communication standards as, e.g., OSEK/VDX COM [24], FlexRay [13], CAN [16], LIN [18] and MOST [22]. In an automotive domain, the AUTOSAR project is highly related to the goal of our project and will be investigated in detail.

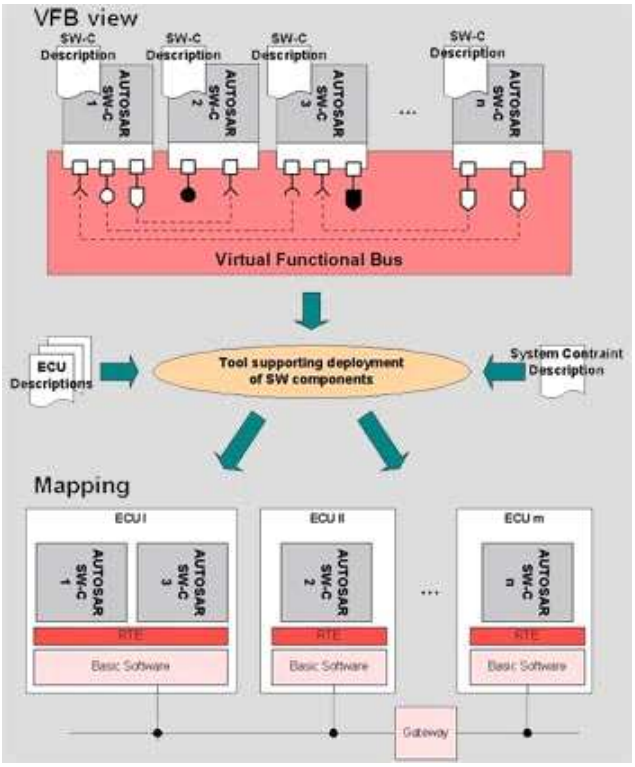

Figure 3. AUTOSAR Virtual Functional Bus and ECU mapping [3].

4 Conclusions

We have discussed our ongoing work on using software components in distributed embedded systems. We address

large systems that are developed by several organisations, and focus on the case when components are as large as suitable trade units between organisations. This focus is different from other ongoing research projects within the CBSE community that address component based development of embedded systems. These projects typically focus on software components for dealing with configuration and maintenance of embedded systems within a company. The expected gains with being able to use software components as trade unit between organisations, instead of today's hardware components are e.g.:

1. A decreased number of hardware components, means lower cost since each computer node requires expensive cabling and communication hardware, it means lower weight, less space, and lower power consumption.

2. Less expensive logistics related to production and deliveries.

3. Transferability of functions (software components) across different computer nodes enables to optimize the use of hardware resources throughout a system's electronic architecture.

We have presented an initial set of questions that mainly focus on finding differences between applicability of CBSE principles in the cases when using smaller or larger components for embedded systems. We expect that parts of the work done with focus on small components might have problems to scale up to this type of big components. Furthermore the goals of a system integrator might be different. When using small components, optimisation might be much more prioritised than in the case with big components developed by different organisations. Other issues might be in focus as complexity control, and assuring that components on the same hardware platform do not interfere with each others. Both temporal and spatial isolation must be provided to allow easy integration of independently developed subsystems into a system. Further study of the ongoing academic and industrial projects will be done.

References

[1] L. Abeni. Server Mechanisms for Multimedia Applications, Scuola Superiore S. Anna, Pisa, Italy, Tech. Rep. RETIS TR98-01, 1998.

[2] AUTOSAR. Homepage of Automotive Open System Architecture (AUTOSAR). http://www.autosar.org/.

[3] AUTOSAR Web Content, V22.4. Available 2005-06-06 from: http://www.autosar.org/.

[4] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering, Volume II, Technical Report, Software Engineering Institute, Carnegie-Mellon University, May, 2000.

[5] T. Baker. Stack-Based Scheduling of Real-Time Processes. *The Journal of Real-Time Systems*, 3(1):97–100, 1991.

[6] E. Bondarev, J. Muskens, P. de With, M. Chaudron, and J. Lukkien. Predicting real-time properties of component assemblies: a scenariosimulation approach. In *Proceedings of the 30th Euromicro Conference*, September 2004.

[7] M. Caccamo and L. Sha. Aperiodic Servers with Resource Constraints. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 161 – 170, December 2001.

[8] Component+ project. http://www.component-plus.org/, (Last Accessed 2005-09-28).

[9] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House publisher, ISBN 1-58053-327-2, 2002.

[10] I. Crnkovic, J. Stafford, H. Schmidt, and K.Wallnau. *Componentbased Software engineering - CBSE 2004 Sympoisum*. Springer Verlag, LNCS 3054 2004-05-17 ISBN: 3-540-21998-6, 2004.

[11] DECOS. Dependable Embedded Components and Systems, homepage, https://www.decos.at/.

[12] Z. Deng, J. W. S. Liu, and J. Sun. A scheme for scheduling hard real-time applications in open system environment. In *Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, 1997.

[13] FlexRay Consortium. http://www.flexray.com/.

[14] J. Fröberg. Engineering of Vehicle Electronic Systems: Requirements Reflected in Architecture. Technical report, Technology Licentiate Thesis No.26, ISSN 1651-9256, ISBN 91-88834-41-7, Mälardalen Real-Time Reseach Centre, Mälardalen University, March 2004.

[15] H. Hansson, M. A. kerholm, I. Crnkovic, and M. Törngren. SaveCCM - a Component Model for Safety-Critical Real-Time Systems. In *Proceedings of 30th Euromicro Conference, Special Session Component Models for Dependable Systems*, September 2004.

[16] ISO 11898. Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. *International Standards Organisation (ISO)*, ISO Standard-11898, Nov 1993.

[17] T.-W. Kuo, W.-R. Yang, and K.-J. Lin. EGPS: a class of real-time scheduling algorithms based on processor sharing. In *Proceedings ot the 10th Euromicro Workshop on Real-Time Systems*, pages 27 – 34, June 1998.

[18] LIN Consortium. LIN - Local Interconnect Network. http://-www.lin-subbus.org/.

[19] G. Lipari, G. Lamastra, and L. Abeni. Task Synchronization in Reservation-Based Real-Time Systems. *IEEE Transactions on Computers*, 53(12), December 2004.

[20] K. Lundbäck, J. Lundbäck, and M. Lindberg. Component-Based Development of Dependable Real-Time Applications. Arcticus Systems: http://www.arcticus.se (Last Accessed: 2005-01-18).

[21] G. Moreno, S. Hissam, and K. Wallnau. Statistical models for empirical component properties and assembly-level

property predictions: Towards standard labeling. In *Proceedings of 5th Workshop on component based software engineering*, 2002.

[22] MOST Cooperation. MOST - Media Oriented Systems Transport. http://www.mostcooperation.com/.

[23] O. Nierstrass, G. Arevalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Proceedings of the First International IFIP/ACMWorking Conference on Component Deployment*, June 2002.

[24] OSEK/VDX-Communication. Version 3.0.3, July 2004. http://www.osek-vdx.org/mirror/OSEKCOM303.pdf.

[25] OSEK/VDX-Operating System. Version 2.2.2, July 2004. http://www.osek-vdx.org/mirror/os222.pdf.

[26] Pacc project, predictable assembly from certified components. http://www.sei.cmu.edu/pacc (Last Accessed: 2005-01-18.

[27] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137 – 150, April 1994.

[28] R. Reussner, H. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66:241–252, 2003.

[29] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transaction on computers*, 39(9), September 1990.

[30] Space4u. software platform and component environment 4 you. http://www.extra.research.philips.com/euprojects/space4u/ (last accessed: 2005-01-18).

[31] M. Spuri and G. C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS'94)*, pages 2–11. IEEE Computer Society, December 1994.

[32] M. Spuri, G. C. Buttazzo, and F. Sensini. Robust Aperiodic Scheduling under Dynamic Priority Systems. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)*, pages 210–219. IEEE Computer Society, December 1995.

[33] D. Stewart, R. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, pages 759 – 776, December 1997.

[34] R. van Ommering, F. van der Linden, and J. Kramer. The koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, March 2000.

[35] K. C. Wallnau. Volume III: A Component Technology for Predictable Assembly from Certifiable Components. Technical report, Software Engineering Institute, Carnegie Mellon University, April 2003.

[36] W. Wolf. What is embedded computing? *IEEE Computer*, 35(1):136–137, January 2002.