Components in Real-Time Systems

Damir Isovic and Christer Norström
Mälardalen University, Västerås, Sweden
{damir.isovic, christer.norstrom}@mdh.se

Abstract

Component-based Software Engineering (CBSE) is a promising approach to improve quality, achieve shorter time to market and to manage the increasing complexity of software. Still there are a number of unsolved problems that hinder wide use of it. This is especially true for real-time systems, not only because of more rigorous requirements and demanding constraints, but also because of lack of knowledge how to implement the component-based techniques on real-time development.

In this paper we present a method for development of real-time systems using the component-based approach. The development process is analysed with respect to both temporal and functional constraints of real-time components. Furthermore, we propose what information is needed from the component providers to successfully reuse binary real-time components.

Finally, we discuss a possibility of managing compositions of components and suggest how an existing real-time development environment can be extended to support our design method.

1 Introduction

Embedded real-time systems contain a computer as a part of a larger system and interact directly with external devices. They must usually meet stringent specifications for safety, reliability, limited hardware capacity etc. Examples include highly complex systems such as medical control equipment, mobile phones, and vehicle control systems. Most of such embedded systems can also be characterized as real-time systems, i.e., systems in which the correctness of the system depends on time factors. Real-time systems are usually used to control or interact with a physical system and the timing constraints are imposed by the environment. As a consequence, the correct behavior of these systems depends not only on the logical results of the computation but also at which time the results are produced [1]. If the system delivers the correct answer, but after a certain deadline, it could be regarded as having failed.

The increased complexity of embedded real-time systems leads to increasing demands with respect to requirements engineering, high-level design, early error detection, productivity, integration, verification and maintenance. This calls for methods, models, and tools which permit a controlled and structured working procedure during the complete life cycle of the system [2]. When applying component-based software engineering (CBSE) methodology on the development of real-time systems, an important factor is reusability of real-time components. Designing reusable real-time components is more complex than designing reusable non-real-time components [3]. This complexity arises from several aspects of real-time systems not relevant in non-real-time systems. In real-time applications, components must collaborate in meeting timing constraints. Examples of timing requirements can be deadline, period time, and jitter.

Furthermore, in order to keep production costs down, embedded systems resources must usually be limited, but they must perform within tight deadlines. They must also often run continuously for long periods of time without maintenance.

A desirable feature in all system development, including the development of real-time systems is the possibility of reusing standard components. However, using any particular operating system or database system for a real-time application is not always feasible, since many such systems are designed to maximize the average throughput of the system but do not guarantee temporal predictability. Therefore, to guarantee predictability, we must use either specific COTS developed for real-time systems or an appropriate subset of the functionality provided by the COTS. Some commonly used real-time COTS are real-time operating systems, communication protocols (solutions), and to some extent real-time databases. This type of components provides an infrastructure to the application. Other commonly used infrastructures in non-real-time systems are JavaBeans, CORBA and COM. However, they are seldom used for real-time systems, due to their excessive processing and memory requirements and unpredictable timing characteristics, which is of utmost importance in the class of application we consider. They have, however, one desirable property which is flexibility, but predictability and flexibility have often been considered as contradicting requirements, in particular from the scheduling perspective. Increased flexibility leads to lower predictability. Hence, a model for hard real-time systems cannot support flexibility to the same extent as the above mentioned infrastructures.

Further, we require to reuse application specific components. Example of two application specific component models are IEC-1131 [5] which is a standard for programming industrial control systems and port based objects which is a programming model developed for robotics [4]. Both these models provide support for hierarchical decomposition, parameterization, communication and synchronization between components. These types of models are quite similar to pipes and filters model, the difference is that the pipe only accommodates one data item, which means if the data has not already been processed when the new data arrives, it will be overwritten. However, both models lack the ability to specify timing attributes besides period time and priority which is not sufficient to specify timing sensitive systems.

The development of standard real-time components which can be run on different HW platforms is complicated by the components having different timing characteristics on different platforms. Thus a component must be adapted and re-verified for each HW-platform to which it is ported, especially in safety-critical systems. Hence, we need to perform a timing analysis for each platform to which the system is ported. Given a system composed of a set of well-tested real-time components, we still face the composability problem. Besides guaranteeing the functional behavior of a specific component, the composition must also guarantee that the communication, synchronization and timing properties of the components and the system are retained. The composability problem with respect to timing properties, which we refer to as *timing analysis*, can thus be divided into (1) verifying that the timing properties of each component in the composed system still hold and (2) *schedulability analysis* (i.e. system-wide temporal attributes such as end-to-end deadlines can be fulfilled).

Timing analysis is performed at two levels, the task level and the system level. At the task level the worst case execution time (WCET) for each task is either analyzed or estimated. If the execution time is measured, we can never be sure that we have determined the worst case. On the other hand if we use analysis, we must derive a safe value for the execution time. The estimated execution time must be greater than or equal to the real worst case and in the theory provided, the estimate can be excessive. The challenge here is thus to derive a value as close as possible to the real worst case execution time. Puschner gives a good introduction to this problem in the seminal paper [7]. At system level we analyze to determine if the system composed fulfils the timing requirements. Several different mature analysis methods exist, for example, analysis for priority-based systems and pre-run-time scheduling techniques [8][9]. Both kinds of analysis have been proven to be useful in industrial applications [10][11].

When designing a system, we can assign time budgets to the tasks which are not implemented by intelligent guesses based on experience. By doing this we gain two positive effects. Firstly, the system level timing analysis can be performed before implementation, thus providing a tool for estimating the performance of the system. Secondly, the time budgets can be used as an implementation requirement. By applying this approach we make the design process less ad hoc with respect to real-time performance. In traditional system design, timing problems are first recognized when the complete system or subsystem has been implemented. If a timing problem is then detected, ad hoc optimization will be begun, this most surely making the system more difficult to maintain.

The paper is organized as following: In Section 2 we present a method for system development using real-time components which support early analysis of the timing behavior as well as the synchronization and communication between components. The method enables high-level analysis on the architectural design level. This analysis is important to avoid costly re-design late in the development due to the detection in the integration test phase that the system as developed does not fulfill the timing requirements. The presented method is an extension of [10], and it is a standard top-down development process to which timing and other real-time specific constraints have been added and precisely defined at design time. The idea is to implement the same principles, but also taking into consideration features of existing components which might be used in the system. This means that the system is designed not only in accordance with the system requirements, but also with respect to existing components. This concept assumes that a library of well-defined real-time components is available. The development process requires a system specification, obtained by analyzing the customer's requirements.

Furthermore, in Section 3, we propose a method for composing components and how the resulting compositions could be handled when designing real-time systems. In Section 4 we describe how an existing real-time development environment can be extended to support our design method. Finally, in Section 5, we provide guidelines about what one should be aware of when reusing and online updating real-time components.

2 Designing component based real-time systems

In this section we present a method for system development using real-time components. This method is an extension of [10], which is also in use in developing real-time systems within a Swedish automobile manufacturing company. It is a standard top-down development process to which timing and other real-time specific constraints have been added and precisely defined (or more correctly, have been predicted) at design time. The idea is to implement the same principles, but also taking into consideration features of existing components which might be used in the system. This means that the system is designed not only in accordance with the system requirements, but also with respect to existing components. This concept assumes that a library of well-defined real-time components is available. The development process requires a system specification, obtained by analyzing the customer's requirements. We assume that the specification is consistent and correct, in order to simplify the presentation of the method.

The development process with real-time components is divided into several stages, as depicted in Figure 2-1. Development starts with the system specification, which is the input to top-level design. At the top-level design, which includes the decomposition of the system into components, the designer browses through the component-library and designs the system, making selections from the possible component candidates.

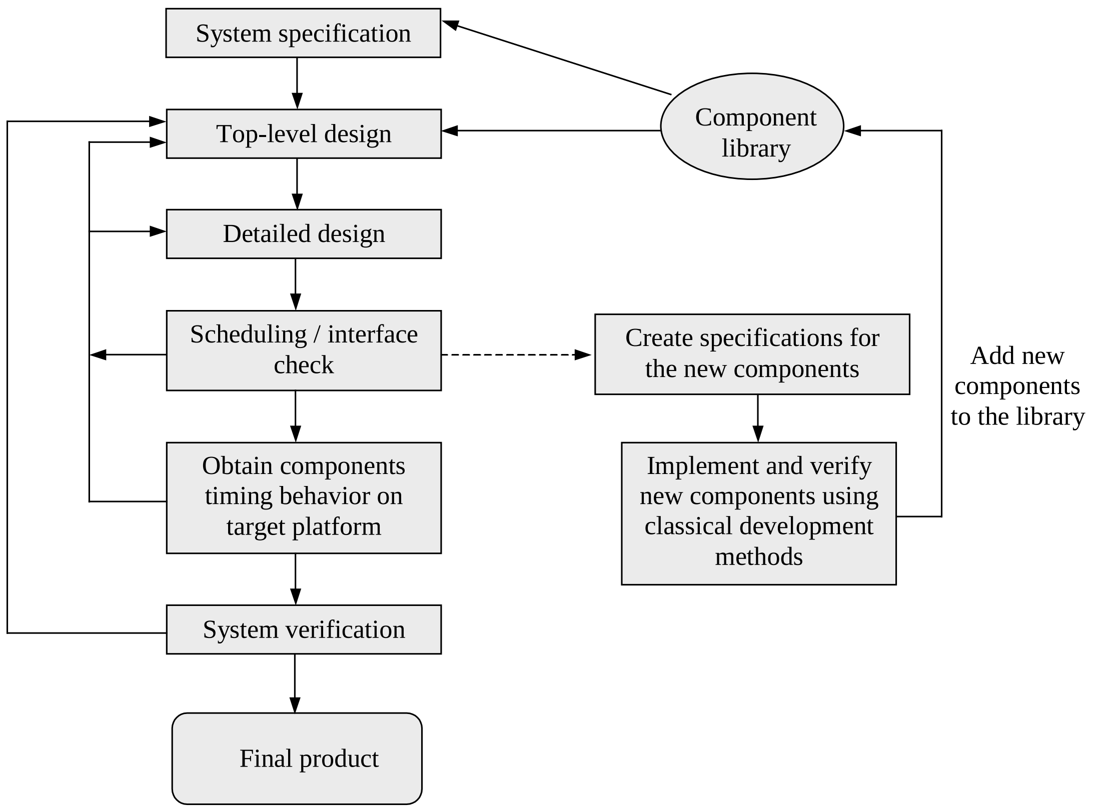
System specification
Top-level design
Detailed design
Scheduling / interface check
Obtain components timing behavior on target platform
System verification
Final product
Component library
Create specifications for the new components
Implement and verify new components using classical development methods
Add new components to the library

Figure 2-1: Design model for real-time components

The detailed design will show which components are suitable for integration. To select components, both real- and non real-time aspects must be considered. The scheduling and interface check will show if the selected components are appropriate for the system, if adaptation of components is required, or if new components must be developed. The process of component selection and scheduling may need to be repeated several times to refine the design and determine the most appropriate components. When a new component must be developed, it should be, (when developed and tested) entered into the component library. When the system finally meets the specified requirements, the timing behavior of the different components must be tested on the target platform to verify that they meet the timing constraints defined in the design phase. A detailed description of these steps is given below.

2.1 Top-level design

The first stage of the development process involves de-composition of the system into manageable components. We need to determine the interfaces between them and to specify the functionality and safety issues associated with each component. Parallel with the decomposition, we browse the component library to identify a set of candidate components, (i.e., components which might be useful in our design).

2.2 Detailed design

At this stage a detailed component design is performed, by selecting components to be used in each component from the candidate set. In a perfect world, we could design our system by only using the library components. In a more realistic scenario we must identify missing components that we need according to our design but which are not available in the component library. Once we have identified all the components to be used, we can start by assigning attributes to them, such as time-budgets, periods, release times, precedence constraints, deadlines and mutual exclusion etc.

A standard way of performing the detailed design is to use the WCET specified for every task which specifies the upper limit of the time needed to execute a task. Instead of relying on WCET values for components at this stage, a time budget is assigned to each component. A component is required to complete its execution within its time budget. This approach has also been adopted in [14], and shown to be useful in practice. Experienced engineers are often needed to make correct assignments of time budgets.

2.3 Scheduling

At this point we need to check if the system's temporal requirements can be fulfilled, assuming time budgets assigned in the detailed design stage. In other words, we need to make a schedulability analysis of the system based on temporal requirements of each component. A scheduler which can handle the relevant timing attributes has been presented in [14], however other approaches such as fixed priority schedulability analysis can easily also be used.

The scheduler in [14] takes a set of components with assigned timing attributes, and creates a static schedule. If scheduling fails, changes are necessary. It may be sufficient to revise the detailed design by reengineering the temporal requirements or by simply replacing components with others from the candidate set. An alternative is to return to top-level design and either select others from the library or specify new components.

During the scheduling we must check that the system is properly integrated; component interfaces are to be checked to ensure that input ports are connected and that their types match. Further, if the specified system passes the test, besides the schedules, the infrastructure for communication between components will be generated.

2.4 WCET verification

Even if the component supplier provides a specification of the WCET, it must be verified on the target platform. This is absolutely necessary when the system environment is not as in the component specification. We can verify the WCET by running test cases developed by the component designer and measuring the execution time. The longest time is accepted as the component WCET. Obtaining the WCET for a component is a quite complicated process, especially if the source code is not available for the performance of the analysis. For this reason, correct information about the WCET from the component supplier is essential.

2.5 Implementation of new components

New components; those not already in the library must be implemented. A standard development process for the development of software components is used. It may happen that some of the new components fail to meet their assigned time budgets. The designer can either add these to the library for possible reuse in other projects or redesign them. In order to proceed, the target platform must be available at this stage. Once a component is implemented and verified we must determine its WCET on our target platform and verify the WCET of library components, if this has not been done before.

2.6 System build and test

Finally, we build the system using old and new components. We must now verify the functional and temporal properties of the system obtained. If the verification test fails, we must return to the appropriate stage of the development process and correct the error.

2.7 Component library

The component library is the most central part of any CBSE system, since it contains binaries of components and their descriptions. When selecting components we examine the attributes available in the library. A component library containing real-time components should provide the following in addition to component identification, functional description, interface, component binary and test cases:

- Memory requirements - Important information when designing memory restricted systems, and when performing trade-off analysis.
- WCET test cases - Test cases which indicate the WCET of the components WCET for a particular processor family. Information about the WCET for previously used targets should be stored to give a sense of the components processor requirements.
- Dependencies – Describing dependencies on other components.
- Environment assumptions - Assumptions about the environment in which the component operates, for example the processor family.

2.8 WCET test cases

Since the timing behavior of components depends on both the processor and the memory organization, it is necessary to re-test the WCET for each target different from that specified. The process of finding the WCET can be a difficult and tedious process, especially if complete information or the source code is not available. Giving the WCET as a number does not provide sufficient information. What is more interesting in the test cases is the execution time behavior shown as a function of input parameters as shown in Figure 2-2. The execution time shows different values for the different input sub-domains.

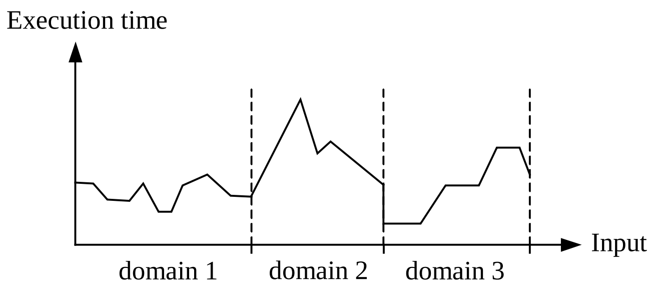

Figure 2-2: An execution time graph

Producing such a graph can also be a difficult and time-consuming process. In many cases, however, the component developer can derive WCET test cases by combining source code analysis with the test execution. For example, the developer can find that the execution time is independent of input parameters within an input range (this is possible for many “simple" processors used in embedded systems but not for others).

The exact values of the execution time are not as important as the maximum value within input intervals, as depicted in Figure 2-3. When a component is instantiated, the WCET test cases are chosen from the appropriate input sub-domain. The timing behavior depends on how the component is instantiated.

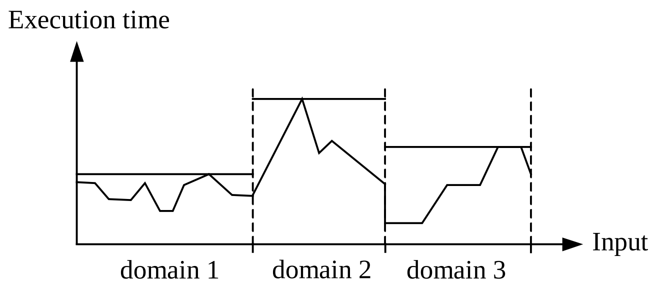

Figure 2-3: Maximum execution time per sub-domain

3 Composition of components

As mentioned earlier a component consists of one or more tasks. Several components can be composed into a more complex one. This is achieved by defining an interface for the new component and connecting the input and output ports of its building blocks, as shown in Figure 3-1.

This new kind of component is also stored in the component library, in much the same way as the other components. However, two aspects are different: the timing information and the component binary. The WCET of a composed component cannot be computed since its parts may be executing with different periods. Instead we propose that end-to-end deadlines should be specified for the input to and output from the component. End-to-end deadlines are set such that the system requirements are fulfilled in the same way as the time budgets are set. These deadlines should be the input to a tool which can derive constraints on periods and deadlines for the sub-components. This possibility remains the subject of research and cannot be considered feasible today.

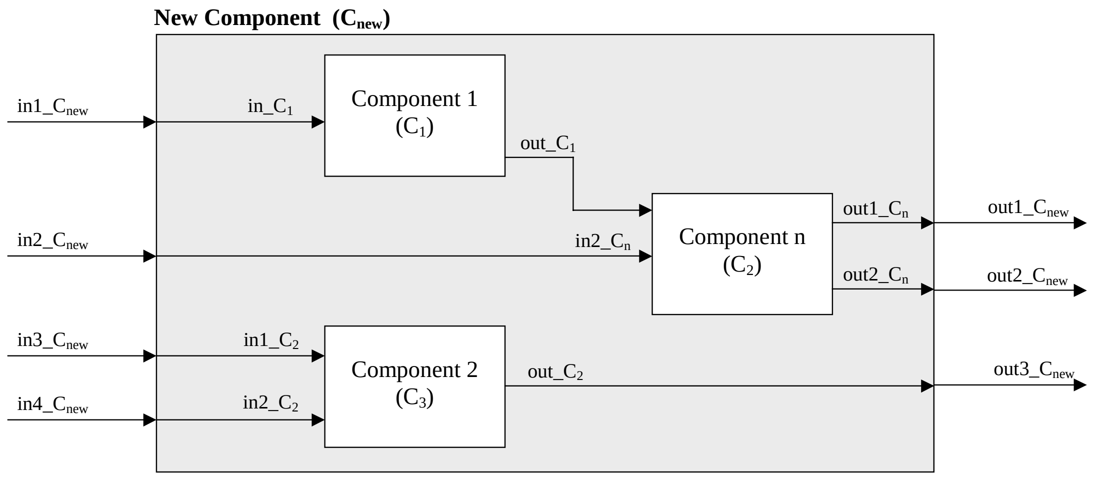

Figure 3-1: Composition of components

Furthermore, we specify virtual timing attributes (period, release time and deadline) of the composed component, which are used to compute the timing attributes of sub-components. For example, if the virtual period is set to *P*, then the period of a sub-component A should be $f_A * P$ and the period of B is f_B * P, where f_A and f_B are constants for the composed component, which are stored in the component library. This enables the specification of timing attributes at the proper abstraction level. The binary of the composed component is not stored in the component library. Instead references to the sub-components are stored, to permit the retrieval of the correct set of binaries.

4 Example: RT components in Rubus OS

Currently there are not so many real-time operating systems that have some concept of components. The Rubus operating system [19] is one of those. In this section we will describe the main features of Rubus, and then present extensions that will make it suitable to use together with our development process. The scheduling theory behind this framework is explained in [14].

4.1 Rubus

Rubus is hybrid operating system, in the sense that it supports both pre-emptive static scheduling and fixed priority scheduling, also referred to as the *red* and *blue* parts of Rubus. The red part deals only with hard real-time and the blue part only with soft. Here we focus on the red part only.

Each task in the red part is periodic and has a set of input and output ports, which are used for unbuffered communication with other tasks. This set also defines a task's interface. A task provides the thread of execution for a component and the interface to other components in the system via the ports. In Figure 4-1 we can see an example of how a task/component interface could look like.

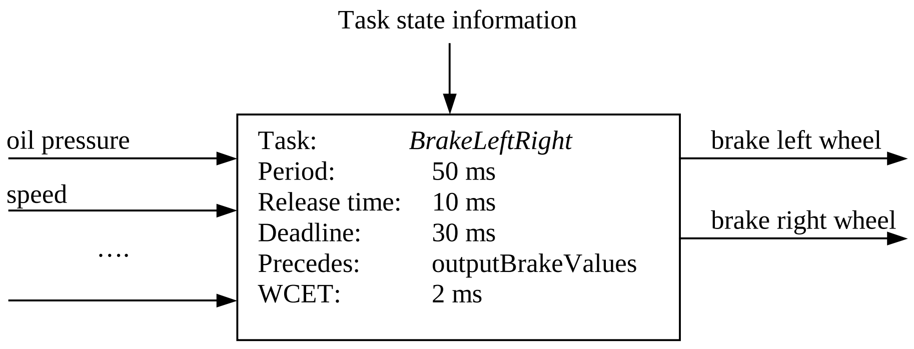

Figure 4-1: A task and its interface in the red model of Rubus

Each tasks has an entry function that which as arguments have input and output ports. The value of the input ports are guaranteed not to change during the execution of the current instance of the task, in order to avoid inconsistency problems. The entry function is re-invoked by the kernel periodically.

The timing requirements of the component/task are shown in Figure 4-1. The timing requirements are specified by release-time, deadline, WCET and period. Besides the timing requirements, it is also possible to specify ordering of tasks using precedence relations, and mutual exclusion. For example the depicted task in is required to execute before the *outputBrakeValues* task, i.e., task *BrakeLeftRight* precedes task *outputBrakeValues*. A system is composed of a set of components/tasks for which the input and output ports have been connected, as depicted in Figure 4-2.

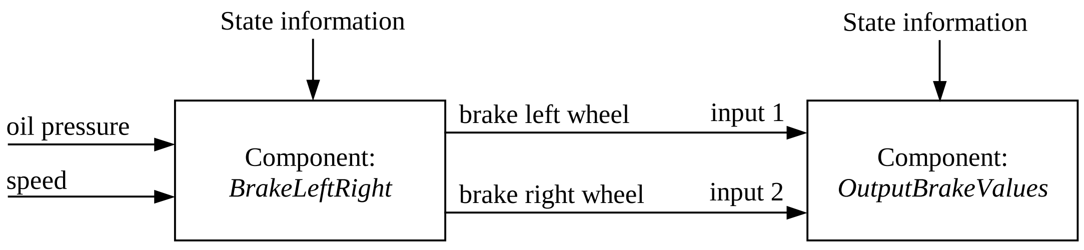

Figure 4-2: A composed system in the red model of Rubus

When the design of a system is finished, a pre run-time scheduler is run to check if the temporal requirements can be fulfilled. If the scheduler succeeds then it also generates a schedule for the design, which is later used by the red kernel to execute the system.

4.2 Extensions for CBSE

Let's see what is missing in Rubus and its supporting tools to make them more suitable for component based development. Firstly, there is currently no support for creating composite components, i.e., components that are built of other components. Secondly, some tool is needed to manage the available components and their associated source files, so that components can be fetched from a library and instantiated into new designs. Besides this there is a lack of real-time tools like: WCET analysis, allocation of tasks to nodes.

Support for composition of components can easily be incorporated into Rubus, since only a front-end tool is needed that can translate component specifications to task descriptions. The front-end tool needs to perform the following for composition:

1. assign a name to the new component
2. specify input and output ports of the composition
3. input and output ports are connected to the tasks/ components within the component, see Figure 4-3.
4. generate task descriptions and port connections for the task within the component.

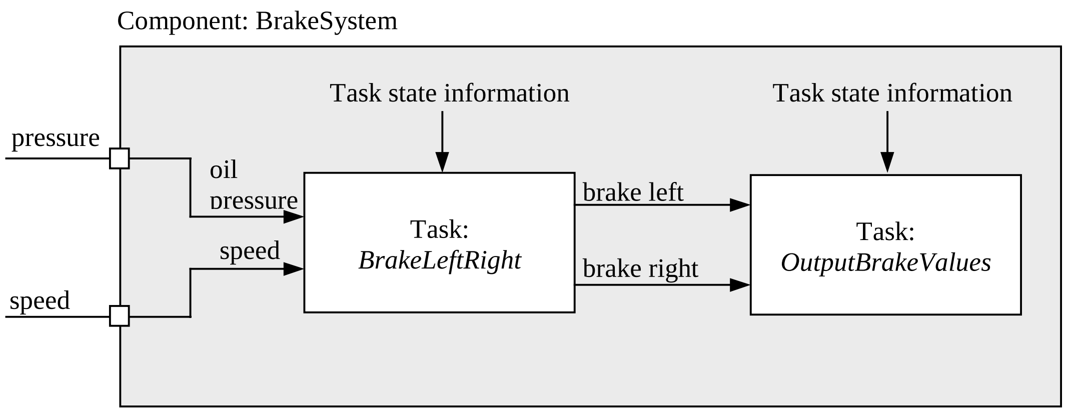

Figure 4-3: Composition of components in Rubus

5 Reuse of RT Components

Design for reuse means that a component from a current project should require a minimum of modification for use in a future project. Abstraction is extremely valuable for reuse. When designing components for reuse, designers should attempt to anticipate as many future applications as possible. Reuse is more successful if designers concentrate on abstract rather than existing uses. The objective should be to minimize the difference between the component's selected and ideal degrees of abstraction. The smaller the variance from the ideal level of abstraction, the more frequently a component will be reused.

There are other important factors which designers of reusable components must consider, they must not only anticipate future design contexts and future reuses. They must consider:

- What users need and do not need to know about a reusable design, or how to emphasize relevant information and conceal that which is irrelevant.
- What is expected from potential users, and what are their expectations about the reusable design.
- That it is desirable, though difficult, to implement binary components, to allow users to instantiate only relevant parts of components. For example, if a user wants to use only some of the available ports of a component, then only the relevant parts should be instantiated.

No designer can actually anticipate all future design contexts, when and in which environment the component will be reused. This means that a reusable component should depend as little as possible on its environment and be able to perform sufficient self-checking. In other words, it should be as independent as possible. Frequency of reuse and utility increase with independence. Thus independence should be another main area of concern when designing reusable components.

An interesting observation about efficient reuse of real-time components, made by engineers at Siemens [15] is that, as a rule of thumb, the overhead cost of developing a reusable component, including design plus documentation, is recovered after the *fifth* reuse. Similar experience at ABB [16] shows that reusable components are exposed to changes more often than non-reusable parts of software at the beginning of their lives, until they reach a stable state.

Designing reusable components for embedded real-time systems is even more complicated due to memory and execution time restrictions. Furthermore, real-time components must be much more carefully tested because of their safety-critical nature.
These examples show that it is not easy to achieve efficient reuse, and that the development of reusable components requires a systematic approach in design planning, extensive development and support of a more complex maintenance process.

5.1 Online Upgrades of Components

A method for online upgrades of software in safety-critical real-time systems has been presented in [17]. It can also be applied to component-based systems when replacing components.

Replacing a component in a safety critical system can result in catastrophic consequences if the new component is faulty. Complete testing of new components is often not economically feasible or even possible, e.g., shutting down a process plant with high demands on availability can result in big financial losses. It is often not sufficient to simulate the behavior of the system including the new component. The real target must be used for this purpose.

However, testing in the real system means that it must be shut down, and there is also a potential risk that the new component could endanger human life or vital systems.
To overcome these problems it is proposed in [17] that the new component should be monitored to check that its output is within valid ranges. If it is not, then the original component will resume control of the system. It is assumed that the old component is reliable, but not as effective as the new component in some respect e.g., the new provides much improved control performance. This technology has been shown to be useful for control applications.

A similar approach can be found in [18] where a component wrapper invokes a specific component version depending on the input values. The timing constraints related to the wrapper execution time must be taken into consideration, and such a system must support version management of components.

In this development model we assume that a static schedule is used at run-time to dispatch the tasks, and since the schedule is static the flexibility is restricted. However, in some cases it is possible to perform online upgrades.

Online upgrade of the system requires that the WCET of the new component is less or equal to the time-budget of the component it replaces. It is also required that it has the same interface and temporal properties, e.g., period and deadline. If this is not feasible, a new schedule must be generated and we must close down the system to upgrade it. Using the fault-tolerance method above, we can still do this safely with a short downtime.

6 Summary

In this paper we presented certain issues related to the use of component technology in the development of real-time systems. We pointed out the challenges introduced by using real-time components, such as guaranteeing the temporal behavior not only of the real-time components but also the entire composed system.

When designing real-time systems with components, the design process must be changed to include timing analysis and especially to permit high-level analysis on an architectural design level. We presented a method for the development of reliable real-time systems using the component-based approach. The method emphasizes the temporal constraints which are estimated in the early design phase of the systems and are matched with the characteristics of existing real-time components. We outlined the information needed when reusing binary components, saved in a real-time component library.

Furthermore, we proposed a method for composing components and how the resulting compositions could be handled when designing real-time systems. We also provided guidelines about what one should be aware of when reusing and online updating real-time components.

References

[1] Stankovic, J. and Ramamritham, K. *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, 1998
[2] D. Kalinsky and J. Ready. Distinctions between requirements specification and design of real-time systems. Conference proceedings on TRI-Ada '88 , 1988, Pages 426 – 432.
[3] Douglas, B.P. Real-Time UML - Developing efficient objects for embedded systems. Addison Wesley Longman, Inc, 1998

[4] D. B. Stewart, R. A. Volpe, P. K. Khosla, Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, IEEE Transactions on Software Engineering, Volume 23, Nr. 12, December 1997

[5] International Electrotechnical Commision, Application and Implementation of IEC 1131-3, May 1995

[6] Wellings, A. and Cornwell, P. *Transaction Integration For Reusable Hard Real-Time Components*. IEEE database, 0-8186-7629-9/97, 1997

[7] Puschner P. and Koza C. Calculating the maximum execution time of real-time programs. Journal of Real-time systems, Kluwer A.P., 1(2):159-176, September, 1989

[8] Audsley N. C. et.al. Fixed Priority Pre-emptive Scheduling: An Historical Perspective. Real-Time Systems, The International Journal of Time-Critical Computing Systems, Vol. 8, Number 2/3, March/May 1995.

[9] Xu Jand Parnas D. L. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. IEEE Transaction on Software Engineering, Vol. 16 No. 3, March 1990.

[10] C Norström, K Sandström, M Gustafsson, J Mäki-Turja, and N-E Bånkestad. Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry. In proceedings of 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS01), Washington, US, April 2001. IEEE Computer Society.

[11] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg Volcano a revolution in on-board communications. Volvo Technology Report. 98-12-10.

[12] K. Ramamritham et.al . Using Windows NT for Real-time Applications: Experimental Observations and Recommendations. Proceedings of the fourth Real-Time Techynology and Applications Symposium, June 3-5, 1998, Denver, Colorado, US.

[13] J. Stankovic. VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems. Technical Report CS-2000-19, Department of Computer Science, University of Virginia, Charlottesville, VA, May 2000.

[14] Eriksson, C., Mäki-Turja, J., Post. K., Gustafsson, M., Gustafsson, J., Sandström, K., and Brorsson, E., *An Overview of RTT: A Design Framework for Real-Time Systems*. Journal of Parallel and Distributed Computing, August, 1996

[15] Mrva, M. *Reuse Factors in Embedded Systems Design*. High-Level Design Techniques Dept. at Siemens AG, Munich, Germany, 1997

[16] Crnkovic, I. and Larsson, M. *A Case Study: Demands on Component-based Development*, Proceedings 22nd International Conference on Software Engineering, Cannes, France, 2000

[17] Sha, L., Dependable System Upgrade, Proceeding of the 20th Real-Time Systems Symposium, Madrid, Spain, 1998

[18] Cook, J.E. and Dage, J.A., *Highly Reliable Upgrading of Components,* Proceedings 21st International Conference on Software Engineering, Los Angeles, USA,1999

[19] Articus Systems, Rubus OS - Reference Manual, 1996