System Development with Real-Time Components

Damir Isović, Markus Lindgren and Ivica Crnkovic
Mälardalen Real-Time Research Centre
Mälardalen University, Sweden
{damir.isovic | markus.lindgren | ivica.crnkovic}mdh.se
http://www.mrtc.mdh.se

Abstract

Component-based Software Engineering is a promising approach to improve quality, to 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 analyzed 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.

1 Introduction

Component-based Software Engineering (CBSE) [15] is beneficial to obtain faster development, to keep costs down and to enable reuse of components from other applications. Still CBSE is not widely recognized as an appropriate approach in the development of real-time systems. The question is if it is possible to utilize CBSE for a successful and efficient development process where demands regarding reliability, timing factors, etc. are significantly more important than in many non-real time applications.

Real-time systems are computing systems in which meeting timing constraints are essential to correctness. The correct behaviour of these systems depends not only on the value of the computation but also at which time the results are produced [14]. Real-time systems can be constructed out of sequential programs, but typically they are built of concurrent programs, called tasks.

Embedded real-time systems contain a computer as a part of a larger system and interact directly with external devices. They usually have to meet stringent specification for safety, reliability, limited hardware etc. An analysis of reuse factors for embedded systems design has been presented in [8].

A typical timing constraint on a real-time task is the deadline, i.e., the time before which the task should complete its execution. Depending on the consequences that may occur due to a missed deadline, real-time systems are distinguished into two classes, hard and soft. In hard real-time systems all task deadlines must be met, while in soft real-time systems the deadlines are desirable but not necessary.

In this paper we analyse the possibility of applying CBSE to the development of hard real-time systems. The aim is to identify real-time component attributes and to provide a design method applicable to reuse them. The main focus of the design method is on temporal properties of real-time components. It is an extension of the development process for real-time systems presented in [11], which enables the benefits of CBSE. The method uses a top-down approach with precisely defined real-time requirements at design time. It also assumes a library of well-defined real-time components.

This paper is divided into six sections. Section 2 extends the definition of components to include real-time properties. Section 3 is the core part of the paper. It presents the development method and the structure of the component library. Section 4 deals with reuse and effective replacement of real-time components. In Section 5 we present the "state-of-the-practice" on real-time components. We also suggest how an existing real-time development environment can be extended to support our design method. Finally, Section 6 concludes the paper.

2 CBSE and Real-Time Systems

CBSE systems are built in two stages: *select*, which selects existing components to be used, and *compose*, where the selected components are composed to fulfil customer requirements. Separate from the system development, we have development of components. This approach requires two different kinds of programmers: application programmers who have great knowledge about the system being built and which components that are required, and component programmers who develop the components that are needed in different applications.

In purely functional applications, the components are typically composed without considering timing requirements. In real-time applications, components must collaborate to meet timing constraints. In this section we discuss what are additional demands on CBSE to manage real-time components. We also elaborate how a component can be defined with respect to real-time properties.

2.1 Real-time Challenges

Designing reusable real-time components is more complex than designing components in the normal case [4]. This complexity arises from several aspects of real-time systems that do not appear in non-real-time systems.

In real-time applications, components must collaborate to meet timing constraints, also referred as 'end-to-end' transaction deadlines [16]. Furthermore, in order to keep production cost down, embedded systems resources are usually scarce, yet they must perform within tight deadlines. They must often run continuously for long periods of time.

Concurrent real-time systems are extremely complex to specify and develop because many independent operations can occur at the same time. When systems are large, these interactions make it difficult for developers to understand the implications of their design decisions. Often it is impossible to predict with certainty when particular events will occur, what their order or occurrence will be, and how long they will last. Yet real-time systems must respond to events within a specified, predictable time limit. Similarly, hardware and software failures are usually unpredictable, but the real-time software must be able to handle them in a predictable manner.

The load on a real-time system that comes from its external environment is another source of complexity. Often this means developing a priority-driven system that drops less essential tasks under big load on the system. To be able to address this issue, real-time components must be designed to handle exceptional situations.

Real-time developers need to ensure they are using available target resources as efficient as possible. Hence, common CBSE technologies (such as JavaBeans, CORBA and COM) are seldom used, due to their processing and memory requirements and unpredictable timing properties.

2.2 Real-time Components

According to [15] a component is a binary element, a unit of independent deployment, a unit of third-party composition and has no persistent state. The definition of a real-time component should also include that its *timing requirements must be met.*

Introducing timing constraints into components implies some difficulties. For example, the timing behavior of a component depends on the target architecture and memory organization. If we obtain a component from a third party company then it is probable that their target differs from ours, and hence the timing behavior on our target could be different.

In hard real-time systems a tasks worst-case execution time (WCET), i.e., the longest possible time it takes to complete the task, is used during schedulability analysis to determine whether the timing requirements can be met or not [5]. If the above component definition is used, we cannot be certain that the timing information we get from the supplier is correct. To overcome this we could try to acquire this information by testing the binary component. However, we can never be certain of finding the WCET (in reasonable time) by testing only [10]. Hence, there is an associated risk of using binary components in real-time systems.

In real-time systems the smallest element that is scheduled is a task. Our definition of a real-time component assumes that *a real-time component should never be smaller than one task.* This level of abstraction enables us to design reusable interfaces, and to assign timing attributes that are relevant to the system.

A component interface should be well defined and well specified to enable good use of it. There are two basic ways of exchanging information between RT components in a system (which define interfaces between the components): These two are *buffered* (message queues) and *unbuffered* (shared memory) communication:

- Buffered data may arrive at any time. When used in hard real-time systems, upper bounds on the number of produced/consumed messages must be determined to enable guarantee of temporal properties.
- Unbuffered data is accessed through shared memory, which always can be read and written to.

Generally it is easier to check systems temporal requirements if unbuffered communication is used. Furthermore, this style of communication is also the most preferred in controller applications. Hence, interfaces of hard real-time components should be unbuffered. We call unbuffered interfaces *ports*.

3 System Development

In this section we present a method for system development with real-time components. This method is an extension of [6, 9], which also is being used for developing real-time systems at a Swedish car industry company. It is a top-down development process where timing and other real-time specific constraints are precisely defined (or better to say are predicted) at design time. Our idea is to implement the same principles, but also taking into consideration features of existing components that might be used in the system. This means that the system is designed not only according to the system requirements, but also in respect to the existing components. This concept assumes that there exists a library of well-defined real-time components.

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.

3.1 Development Process

The development process with real-time components is divided into several stages, as depicted in Figure 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 modules, the designer browses trough the component-library and designs the system having in mind the possible component candidates. 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 new ones have to be developed. The process of component selection and scheduling may have to be repeated several times to refine the design and find proper components. In case that new component must be developed, it should be (when developed and tested) put in the component library. When the system finally meet the requirements from the specification, the timing behavior of the components must be tested on the target platform to verify if they meet the timing constraints defined in the design phase. A detailed description of these steps is given below.

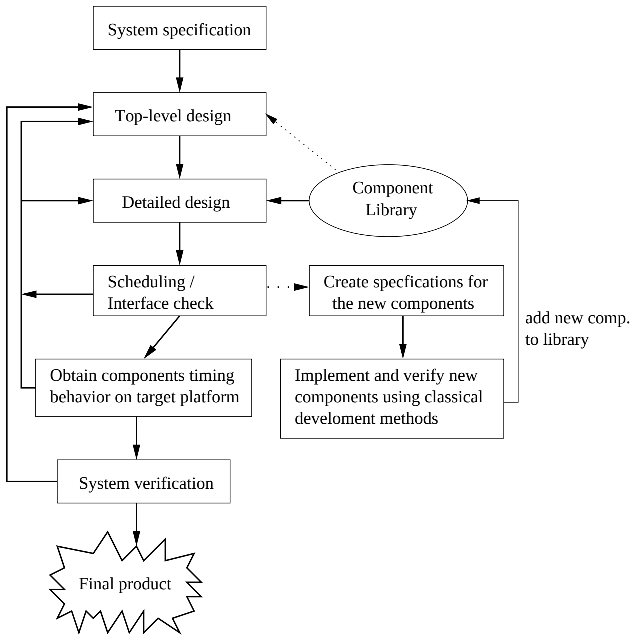
System specification
Top-level design
Detailed design
Component Library
Scheduling / Interface check
Create specfications for the new components
add new comp. to library
Obtain components timing behavior on target platform
Implement and verify new components using classical develoment methods
System verification
Final product

Figure 1: Design model for real-time components

3.1.1 Top-level Design

The first stage of the development process involves decomposition of the system into manageable modules. We need to determine the interfaces between them and to specify the functionality and safety issues associated with each module. At the same time we do the decomposition, we browse the component library to identify a set of *candidate components*, i.e., components that might be useful in our design.

3.1.2 Detailed Design

At this stage a detailed module design is performed, by selecting components to be used in each module 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 have to identify missing components, i.e., components needed for our design but not available in the component library.

Once we have identified all 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 among them.

A standard way of doing the detailed design is use WCET specified for every 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 [6], and shown to be useful in practice. Experienced engineers are often needed to make correct assignments of time budgets.

3.1.3 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 provide a schedule based on temporal requirements of each component.

A scheduler that can handle the mentioned timing attributes has been presented in [6]. It takes a set of components with assigned timing attributes, and creates a static schedule.

If scheduling fails, changes need to be done. It could be sufficient to redo detailed design by refining the temporal requirements or simply replacing components by new ones from the candidate set. Otherwise, we need to go back to top-level design and either choose other components from the library, or specify new ones.

At the same time when scheduling is done, component interfaces are checked to see if input ports are connected and if their types match, i.e., system integration is performed.

3.1.4 WCET Verification

Even if we have a specification of the WCET from a component provider, we must verify it on our target platform. This is absolutely necessary when the system environment is not the same as in the component specification. We can verify WCET by running test cases obtained by component creator, and measuring the execution time. The longest time is assigned as the components WCET. Obtaining WCET for a component is a quite complex process, especially if we do not have the source code and no possibility to do an analysis based on that. For this reason proper information about WCET from the component provider is essential. Subsection 3.2.1 discusses what type of information related to WCET is desirable.

3.1.5 Implementation of New Components

Now we need to implement new components; those that are not in the library. Standard development process for development of software components is used. It may happen that some of the new components fail to meet their assigned time budgets. It's up to designer to either add those to the library for eventual reuse in other projects, or to discard them.

In order to proceed, the target platform must be available at this stage. Once a component is implemented and verified we have to derive its WCET on our target platform. We even need to verify WCET of library components, if not done earlier.

3.1.6 System Build and Test

Finally, we build the system using old and new components. Now we need to verify the obtained system's functional and temporal properties. If the test fails, we need to go back to appropriate stage of the development process and correct the error.

3.2 Component library

The component library is the most central part of any CBSE system, since it contains binaries of components and descriptions of them. When selecting components we examine the available attributes in the library. A component library that contains real-time components should provide the following:

Component identification — A unique name that identifies the component. In many cases it may happen that the component evolves over time. New functions or new non-functional attributes can be added to new component versions. If different component versions are identified by the same name, there is a risk that a system includes a component version that is not tested in the system. For this reason the components must also be identified by versions numbers and information about the changes between the versions are needed [7].

Functional description — Describes the functionality of a component. Given the current state-of-the-art we believe that natural language is still to be preferred. Formal methods might be applicable when those techniques mature.

Interface — Defines the input and output ports of the component (name, type, and description). Should be unbuffered.

Memory requirements — Important information when designing memory restricted systems, and when doing trade-off analysis.

Test cases — A set of test cases, mainly for regression testing, with input and output values. The test cases are used to make sure that the component delivers the specified functionality. Since unbuffered interfaces are used, we can also use preconditions on input ports and post conditions on output ports to express more input/output combinations.

Component binary — The binary code for the component for a specific processor family.

WCET test cases — Test cases that reveal the components WCET on a particular processor family. See below for a detailed description. Information about WCET for previously used targets should be stored, in order to give a sense of the components processor requirements.

Dependencies — Describes dependencies to other components.

Environment assumptions - Assumptions about the environment in which the component operates, i.e., processor family.

3.2.1 WCET test cases

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

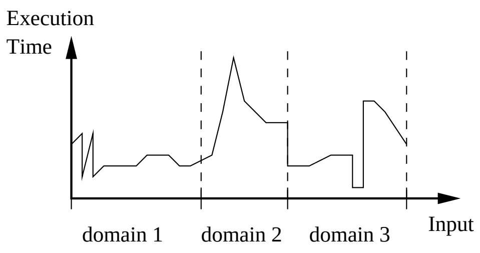

Figure 2: An execution time graph

The execution time shows different values for the different input sub-domains. 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 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, and for others not.) It is not important to give the exact values of the execution time $e(t)$, but find the maximum value within input intervals, i.e.,

$$E_{i-j}(t) \geq e(t) : t \in [t_1, t_2]$$

where t denotes the input value. Example of such presentation is shown in Figure 3.

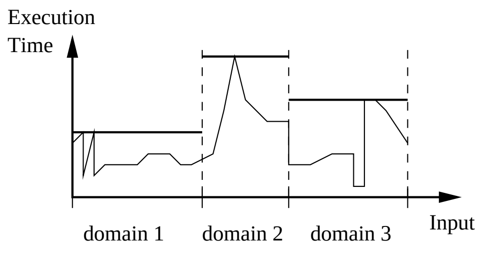

Figure 3: Maximum execution time per sub-domain

When a component is instantiated, the WCET test cases are chosen from the appropriate input sub-domain, i.e., the timing behavior depends on how the component is instantiated. Also, the graph shows the behavior on execution time and may indicate which test cases are of greatest interest.

3.3 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 of the new component and connecting input and output ports of its building blocks, as in Figure 4.

This new kind of component is also stored in the component library, in a similar 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 input and output of the component. End-to-end deadlines are set such that system requirements are fulfilled, similarly as time-budgets

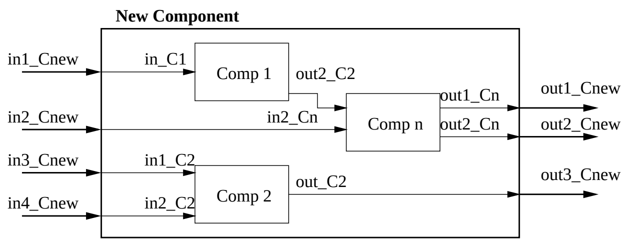

Figure 4: Composition of components

are set. These deadlines should be the input to a tool that can derive constraints on periods and deadlines for the sub-components. However, this topic is still open for research, and cannot be considered feasible today.

Furthermore, we specify virtual timing attributes (period, release time and deadline) of the composed component, which are used to compute timing attributes of sub-components. For example, if the virtual period is set to P then the period of 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 are kept to the sub-components, to enable the retrieval of the correct set of binaries.

4 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 some other important factors that designers of reusable components must consider. They must not only anticipate future design context, but also think about future reuses. They must consider:

- What users need and do not need to know about a reusable design, or how to emphasize relevant and hide irrelevant information.
- What is expected from potential users, and what are their expectations about the reusable design.
- It is desirable, though difficult to implement for 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 carry out 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.

There is an interesting observation about efficient reuse of real-time components, made by engineers at Siemens [8] which says that as a rule of thumb, the overhead to develop a reusable component, including design plus documentation, is recaptured after the *fifth* reuse.

Similar experience at ABB [3] shows that reusable components are exposed to changes more often than non-reusable parts of software in the beginning of the their life, 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 have to 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 systematic approach in design planning, extensive development and support of a more complex maintenance process.

4.1 Online Upgrades of Components

A method for online upgrades of software in safety-critical real-time systems has been presented in [13, 12]. 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., bringing 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 new component. The real target has to be used for this purpose. However, testing on the real system means that it needs to be shut down, and also introduces a potential risk that the new component could endanger human life or other vital systems.

To overcome these problems it is proposed in [13, 12] that the new component should be monitored to see if its output is within valid ranges. If it is not, then the old component will take over control of the system again. It is assumed that the old component is reliable, but not as good as the new one in some aspect, e.g., the new one provides much better control performance. This technology has been shown to be useful for control applications.

Similar approach can be found in [2] where a component wrapper invokes a specific component version depending on the input values. The timing constraints regarding to the wrapper execution time must be taken into consideration, also such a system must support version management of components.

In our 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 new components WCET 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 infeasible, a new schedule has to be generated and we must take down the system to upgrade it. Using the fault-tolerance method above, we can still do this safely with a short downtime.

A method for online upgrades of software in safety-critical real-time systems has been presented in [13, 12]. 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., bringing down a process plant with high demands on availability can result in big financial losses. It is often not sufficient to simulate the behaviour of the system including new componen. The real target has to be used for this purpose. However, testing on the real system means that it needs to be shut down, and also introduces a potential risk that the new component could endanger human life or other vital systems.

To overcome these problems it is proposed in [13, 12] that the new component should be monitored to see if its output is within valid ranges. If it is not, then the old component will take over control of the system again. It is assumed that the old component is reliable, but not as good as the new one in some aspect, e.g., the new one is much faster. This new technology has been shown to be useful for control applications, e.g. for the inverted pendulum problem.

In our develoment 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 new component's WCET 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 infeasible a new schedule has to be generated and we must take down the to upgrade it. Using the fault-tolerance method above, we can still do this safely with a short downtime.

5 Current status on RT components

Currently there are few real-time operating systems that support static scheduling, and even fewer that have some concept of components. The Rubus operating system [1] 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 [5].

5.1 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 tasks interface. In Figure 5 we see an example of how a task/component interface looks like.

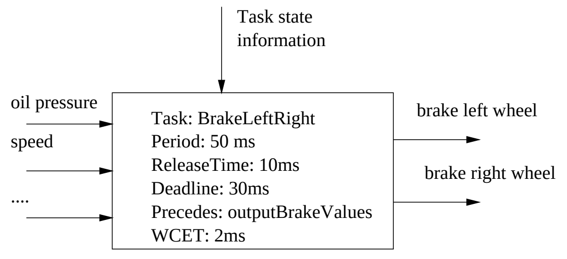

Figure 5: A task in the red model and its interface

Tasks in the red model are implemented as functions, which are passed their input and output ports via parameters to the function. The input to the function is guaranteed not to change during the execution of it, in order to avoid inconsistency problems. The function is re-invoked by the kernel periodically.

The timing requirements of the component/task are shown in Figure 5. 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 task in Figure 5 is required to execute before the *outputBrakeValues* task, i.e., task *BrakeLeftRight* precedes *outputBrakeValues*.

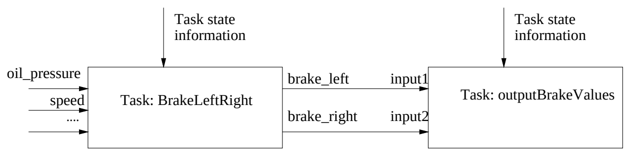

Figure 6: A composed system in the red model

A system is composed of a set of tasks for which the input and output ports have been connected, as in Figure 6. However, this model does not support that components themselves are composed of other components.

When the design of a system is finished, a pre runtime 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.

5.2 Extensions for CBSE

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. 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 7.
4. generate task descriptions and port connections for the tasks within the component.

6 Summary

In this paper we presented a method for the development of reliable real-time systems using the component-based approach. The method emphasizes the temporal constraints that are estimated in the early design phase of the systems, and matched with the characteristics of

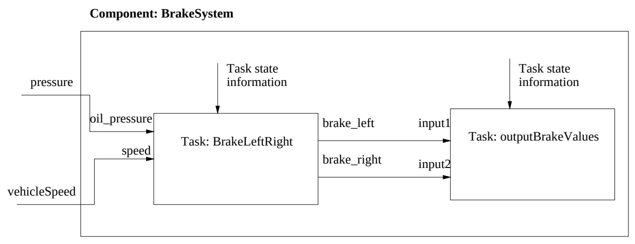

Figure 7: Extending Rubus with composition of components

existing real-time components. We outlined what information is needed when reusing binary components, saved in a real-time component library.

Furthermore, we give a suggestion how components can be composed, and how these compositions can be handled when designing real-time systems. This paper gives several ideas that require further research: Real-time component identification, a component library and related tools implementation and refinement of the development process are some of them.

Acknowledgement

The authors wish to thank Christer Norström for his fruitful comments and stimulating discussions, which helped to improve the quality of the paper.

References

[1] *Rubus OS - Reference Manual.* Articus Systems, 1996.

[2] J.E. Cook and J.A. Dage. Highly Reliable Upgrading of Components. In *Proceedings 21st International conference on Software Engineering*, pages 203–212. Springer, 1999.

[3] Ivica Crnkovic and Magnus Larsson. A Case Study: Demands on Component-based Development. In *Proceedings 22nd International conference on Software Engineering*, 2000. To be published.

[4] Bruce Powel Douglas. *Real-Time UML - Developing efficient objects for embedded systems.* Addison Wesley Longman, Inc, 1998. ISBN 0-201-32579-9.

[5] C. Eriksson, J. Gustavsson, J. Brorson, and M. Gustafsson. An Object Oriented Framework for Designing Hard Real-Time Systems. In *Proc. 5th Euromicro Workshop on Real-Time Systems*, pages 90–97. IEEE Computer Society Press, 1993.

[6] Christer Eriksson, Jukka Mäki-Turja, Kjell Post, Mikael Gustafsson, Jan Gustafsson, Kristian Sandström, and Ellus Brorsson. An Overview of RTT: A Design Framework for Real-Time Systems. *Journal of Parallel and Distributed Computing*, August 1996.

[7] Magnus Larsson and Ivica Crnkovic. New challenges for Software Configuration Management. In *Proceedings 9th Software Configuration Symposium.* Springer, 1999.

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

[9] Christer Norström, Kristian Sandström, Mikael Gustafsson, and Jukka Mäki-Turja. Experiences from using state-of-the-art real-time techniques in an industrial project. Technical report, Mälardalen Real-Timer Research Centre, Sweden, March 2000.

[10] P. Puschner and R. Nossal. Testing the result of static worst-case execution-time analysis. In *Proc. 19th Real-Time Systems Symposium*, pages 134–143. IEEE Computer Society Press, 1998.

[11] Kristian Sandström, Christer Eriksson, and Mikael Gustafsson. RealTimeTalk: A Design Framework for Real-Time Systems — a Case Study. *SNART'97*, August 1997, Sweden.

[12] L. Sha, R. Rajkumar, and M. Gagliardi. Evolving Dependable Real-Time Systems. In *Proc. IEEE Aerospace Applications Conference.* IEEE Computer Society Press, 1996.

[13] Lui Sha. Dependable System Upgrade. In *Proc. 19th Real-Time Systems Symposium*, pages 440–448. IEEE Computer Society Press, 1998.

[14] John Stankovic and Krithi Ramamritham. Tutorial on Hard Real-Time Systems. *IEEE Computer Society Press*, 1988.

[15] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, 1997.

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