Component technology in Resource Constrained Embedded Real-Time Systems

Kaj Hänninen, Jukka Mäki-Turja
Mälardalen Real-Time Research Centre
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
{kaj.hanninen, jukka.maki-turja}@mdh.se
http://www.mrtc.mdh.se

Abstract: This paper presents a framework to incorporate real-time theory with component based software engineering, in order to achieve predictable systems. The proposed technology is aimed at releasing the developers from analysis aspects, and having a synthesis tool resource efficiently mapping a feasible component based software architecture to a run-time environment. Two component technologies form the base of our proposal, the predictable infrastructure PECT and the real-time component technology AutoComp. By combining properties from both, we achieve a framework suited for resource constrained real-time systems.

1 Introduction

Though diversified in many aspects, different companies all have the common goal of maximizing profit. Different domains and companies within a domain have targeted towards different business goals. For example, a developer using a component technology have different needs from those supplying the infrastructure and those supplying components. Furthermore, a company with few customers per product will benefit more from an efficient development process since development cost must be shared by few customers, an example being the business segment of heavy vehicles [MFN04]. On the other hand, companies with large volumes are more willing to place extra effort in the development phase and reducing the cost of each product, the car industry being an example. Technically the systems in heavy vehicles and cars are closely related, but due to their different business needs, different technologies are used [FSN+03].

Software reuse, through component based software engineering, is often argued to reduce system development and/or maintenance cost [CL02]. However, it is important that the range of commonality is sufficiently large. With this we mean that a component model must be applicable to a wide range of developers in order to see a business need for sub-suppliers to provide and develop components and component frameworks. However, the component model can not be too general in that it will not meet the specific needs (such as scarce resource). This is a difficult trade off where the challenge lies in finding a sufficiently general component model that will meet the specific needs of a wide range of requirements placed on applications of a domain.

The view on embedded real-time system (ERTS) has been a, once developed, monolithic, platform dependent view, which is not constructed for evolution. However, the typical life-cycle of such sys-

tems, in practice, depicts a quite opposite reality. ERTS tends to have a very long life-time, decades in some cases. The effort placed in them can not easily be ignored, therefore such systems tends to become legacy systems that are hard to incorporate into functionality and/or technology shifts [FSN+03].

Component technology offers an opportunity to increase productivity by providing natural units of reuse (components and architectures), by raising the level of abstraction for system construction, and by separating services from their configuration to facilitate evolution [BCC+03]. Embedded real-time systems are often characterized by having scarce resources such as memory, processing power and communication bandwidth. Yet many of these systems need to satisfy requirements on dependability [CL02, MFN04]. The focus of this paper is how development and maintenance of resource constrained embedded real-time systems (RCERTS) can be aided by a component technology and component based development (CBD) methodology.

Developers of RCERTS face the challenge of making safe and easy to maintain applications running on limited resources, without overrunning project budgets. Historically, the development of ERTS has been done using low level programming language to guarantee full control over the system behaviour [SFA04]. However, during recent years a new software engineering discipline, component based software engineering (CBSE), has received attention in the embedded development domain. It has been seen as a promising approach to handle the complexities involved in the development of ERTS. The complexities arise, amongst other things, from constant demands on adding new functionalities in systems, to keep up market shares for the developed products. Besides being a discipline aiding developers to cope with complexity, CBSE is concerned with rapid assembly by reuse of components in different applications (products)[BBB+02, CL02].

Introducing a component based development process for ERTS is associated with challenges. First, a component based development process must be able to handle and satisfy at least the same set of functional requirements as the existing development process. Furthermore, for RCERTS, non-functional requirements such as timing behaviour, safety, memory consumption and so on, must be analysable in the context of the development process. In this paper we will focus on two vital properties for RCERTS: Space and temporal resources expressed by memory usage and schedulability. To introduce a component technology for a domain, such as RCERTS domain, we must be able to show the economical benefit for all parties, ranging from end users of the component technology, suppliers of the component framework (run-time support), to third party component suppliers.

In this paper, we take the view of a component framework provider (CFP). A CFP can be seen as one delivering both development as well as run-time supporting tools to a company (compare this with providers of operating systems enclosing development environments). Furthermore, the company may already have, or may need to develop, their own in-house components. In addition, the company may need to rely on third party components adhering to the component framework standard.

The rest of the paper is organized as follows. In section 2 we describe the terminology used in component based software engineering. Section 3 depicts our ideas for CBSE in the context of RCERTS at a general level. Section 4 describes how our ideas of combining component technology with real-time system theory, relates to work done by others. Section 5 shows how an example application can be developed by using ideas presented in this paper. Finally, in section 6 we conclude the paper and describe some future work.

2 Component technology definitions

In this section we describe basic concepts used in component based software engineering literature. We begin this section with a description of definitions given by different authors. We then conclude this section by describing the terminology that we will use throughout this paper.

Development and engineering

Component based development (CBD) and component based software engineering (CBSE), are two common terms used in the context of software engineering with components. Component based development refers to the building of software systems by assembling developed components ready for integration. The systematic approach focusing on component aspects of development, recognized as a sub discipline of software engineering, is referred to as component based software engineering [CL02].

Brinksma et al [BCC+03] distinguishes between component development and system development with components. They state that the difference lies in the requirements and business goals in the two cases. They argue that the main emphasis on component development is on reusability whereas system development is focused on the identification of reusable entities and relations between them. This is a similar argument as we had in the introduction, i.e., in order for CBSE to be successfully adopted, both suppliers and users of components must be able to see a business benefit.

Components

Component based software engineering is a relative young engineering discipline. Several attempts to define a component have been made. Bachmann et al [BBB+02] describes a component as a software implementation that can be executed on a physical or logical device. Furthermore, they state that a component is an opaque implementation of functionality that may be subject to third-party composition and conformant with a component model. Brinksma et al [BCC+03] follows the notion of Szyperski [Szy98] stating that:

> *"a component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition."*

We agree to this definition if the development time can be seen as the time of deployment.

Interfaces and contracts

The operation and context dependencies of a component is defined by its interfaces. Bachmann et al [BBB+02] shortly describes an interface as means to integrate component into assemblies, which in turn, makes it possible to reason about the assemblies. Brinksma et al [BCC+03] states that interfaces summarizes the component properties that are externally visible to other parts of the system. Furthermore, they describe that so called rich-interfaces may contain extra-functional information (such as execution time etc).

Component specifications may also be described by contracts. Bachmann et al [BBB+02] describes a contract as pattern of interaction rooted on a component. Furthermore, they state that contracts specify the services provided by a component, and the obligations of clients and an environment needed by a component, to provide its services. Brinksma et al [BCC+03] state that a contract specifies functional or non-functional properties of a component, which are observable in a components interface. Crnkovic et al [CL02] describes a contract as a specification focusing on conditions in

which a component interacts with its environment.

Models and frameworks

Brinksma et al [BCC+03] describes a component model and a component framework as two basic prerequisites that enables components to be integrated and work together;. The component model *"specifies the standards and conventions that component must follow to enable proper interaction"* , whereas the framework is *"the design and run-time infrastructure that manages resources for components and supports component interactions".* Bachmann et al [BBB+02] describes a framework as means of managing resources and providing mechanisms that enable communication among components. In this paper we focus on design time infrastructure and synthesis to a minimal, application specific, run-time infrastructure.

In the rest of this paper we will adopt the definitions given by Brinksma et al [BCC+03], which to a large extent is based on work by Szyperski [Szy98].

3 CBSE in development context for RCERTS

It is our belief that the challenges of introducing component technologies into RCERTS can be overcome, and in the long run gain acceptance in the industry. Pretty much the same way compilers have become accepted (even indispensable) tools in the development of such systems. Our view of component technologies (which is shared by others [SFA04, Wal03]) in RCERTS bares many similarities, both at the high level and low level, with compiler technology. The following discussion highlights some of the similarities.

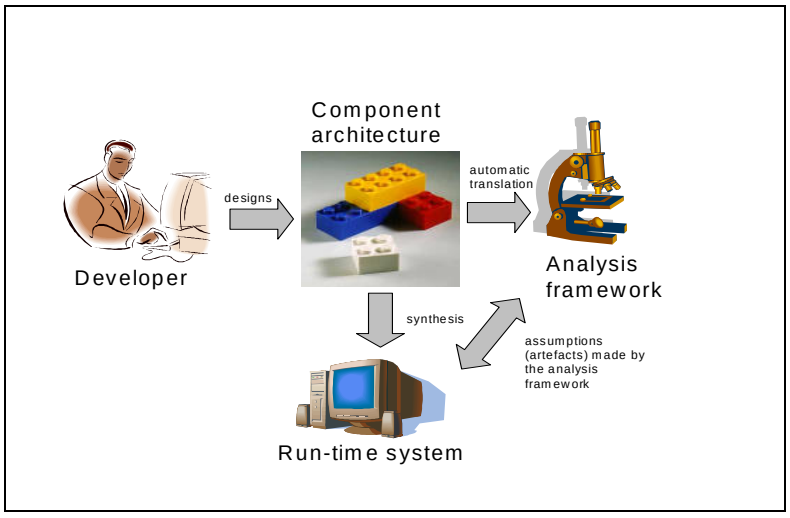

Figure 1: Component technology in the development of RCERTS applications. The arrows indicates process flow in a development scenario

For the following discussion, refer to figure 1. In an ideal situation we envision the developer (or designer) of a system having a component model, architectural rules and constraints at his disposal in order to develop a high level (abstracts away from pure source code) architecture of the application (system). The systems is built up from components supported by the component technology (component framework, constructive rules on composition). The goal during construction is to relive the

designers from the burden of low level details so that they can focus on the problem at hand. This enables the developer to construct a component architecture that is understandable and maintainable. The architecture should also be formal enough for automated analysis and synthesis. The analogy to compiler theory can be seen as a programmer programming in source code (high level of abstraction) instead of assembler (low level of abstraction).

A component architecture for an application should be analysed for certain properties with an analysis framework. The role of an analysis framework is to ensure that a well formed (syntactically correct) architecture can be analysed whether it satisfies certain properties or not. The analogy to compilers could be checking for semantic correctness, e.g., type checking. In this paper we are concerned with memory consumption and temporal feasibility (schedulability analysis) which results in two analysis frameworks, one for memory consumption and one for schedulability analysis. The objective of an analysis framework is to deal with as intricate problems as possible (this is where we want to hide the complexity) with automated tool support, i.e., the analysis framework has the knowledge of the component architecture (design) as well as the constraints and services provided by the component framework (run-time system). An analysis framework aims at satisfying a feasibility property (such as doing a schedulability analysis) or enabling system property analysis derived from individual component properties such as memory usage. Functional properties are not in the scope of this paper, however the component model can support both a testing approach or a model based approach if components functional behaviour is described formally with for example timed automata [UPP].

Another important task for the analysis framework is to provide information of assumptions (artefacts) it had to make in order make a property feasible. Such artefacts typically consists of task model attributes that can not be derived from the design (one example being task priorities). This information together with the component architecture, i.e., an architecture that is both syntactically and semantically correct, may be used by a synthesis tool (analogous to a back-end generator of a compiler) when generating code for the run-time environment. A synthesis tool thus takes the architecture design and possibly some artefacts (such as priorities for a task model) produced by the analysis framework, and maps it to the run-time system (OS, component glue code etc.). The aim is to provide a run-time system that has a small footprint, but still providing sufficient run-time services to the components of the application. Note that this is a degree of freedom; if the application makes use of lot of run-time services these must be provided by the run-time system. However, at one extreme, if the application is purely static, all connections between components can be resolved off-line (by analysis framework and synthesis tool) which result in a static schedule yielding little run-time overhead. With this view one can say that the entire component framework is provided at development time (as opposed to at run-time for component technologies such as e.g. .NET), whereas only the part that are used are mapped down to the run-time system.

Our vision is to provide the engineer with a powerful and expressive component model for the RCERTS domain. In order to do this, automated tools are needed that handles the complexity of analysis frameworks and code synthesis to the run-time environment. The vision is to get as powerful technology for designing with components as there today is for compilers. Few developers know what goes on inside a compiler (there is a lot of formal theory at work) and they do not have to, it suffices to know what it does and know how it can be used to optimise and synthesise the application requirements. Similarly we envision that analysis frameworks and synthesis tools can aid a designer in the area of RCERTS.

4 Related work and objective

A lot of research has addressed the requirements that must be fulfilled by component technologies, to be accepted by industries developing embedded real-time systems. Möller et al [MFN04, MFN03, MAFN03] presents a set of technical and development requirements that different developers, in the heavy vehicle domain, find important. Among the technical requirements the industry strives for components to be analysable, testable, debuggable, resource constrained and portable. Furthermore, the industry strives for a simple and mature standardized component modelling language. The requirements related to the development process include demands on components to be introducible, reusable, maintainable and understandable. Furthermore, the requirements regarding analysis of components in ERTS include addressing properties such as timeliness, memory consumption, reliability and safety etc. not just for single components, but also for assemblies of components. Hence the assemblies of components should be predictable with regard to functional as well as non-functional requirements.

Sandström et al [SFA04] presents a component technology (AutoComp) for safety critical embedded real-time systems. The technology is aimed at systems with high demands on safety and reliability. This is achieved by mapping a component model with high level real-time constraints, to a real-time model which is analysed and synthesized for predictable execution. The AutoComp model is divided into components, interfaces, compositions, invocation cycles, transactions and system representation. The component interfaces are classified into two categories; data and control interfaces. The data interfaces are used to handle data flow between components and the control interfaces to handle control flow of components.

Wallnau [Wal03] presents a development infrastructure (PECT) enabling critical runtime properties of component assemblies that are objectively analysable and predictable. The logical structure of PECT consists of two distinct frameworks; a construction and one or more reasoning frameworks. The construction framework supports construction activities of CBD, whilst the reasoning framework supports the prediction activities of CBD. The ability to use several independent reasoning frameworks makes PECT interesting for development of RCERTS. In addition, decoupling the analysis and constructive framework increases the usefulness of the constructive component model. Analysis frameworks are used to tailor the application specific requirements. This is an attractive viewpoint and increases the level of commonality of the constructive component model. There are however certain limitations to the component model that may hinder the developers when using PECT. The abstract component technology (ACT) in PECT do not allow the separation of data and control flow, hence transactions including several components that need to execute at different periodicity, which is common in many (multi-rate) control systems [San02], may be difficult to realize.

We find AutoComp being aimed against the embedded systems domain. It is developed using a natural way of guaranteeing temporal predictability of component assemblies. However, the AutoComp model assumes a fixed run-time environment to which all applications must be mapped (synthesised) to (Fixed priority Systems). This will restrict the type of applications that can use the component model to those where FPS run-time support can be afforded. Furthermore, no commercially available FPS OS:es have run-time support for offsets. Also the only reasoning framework in AutoComp is the schedulability analysis, which is integrated in the development method. The PECT theory, on the other hand, feels more natural with respect to assembly predictions, especially since several frameworks may be used to analyse different properties. Unfortunately, it lacks synthesis theory.

In essence we find that combining the view from AutoComp and the view from PECT together with adding stronger and more flexible synthesis theory will contribute in achieving the vision outlined in the previous section. The idea is to combine component technology with real-time system theory and encapsulate them into development and synthesis tools to achieve good quality software for RCERTS. To concretise our ideas, and support our claims, we will show how a simple example application can be developed using the techniques in AutoComp [SFA04], PECT [Wal03], and our idea of synthesis.

5 Example system development

In the following sections we present an example system that uses design, analysis and synthesis properties from both AutoComp [SFA04] and PECT [Wal03].

The challenge is to develop a control system responsible for the positioning of a industrial robot. The main goal of this example is to describe how activities from existing component technologies may be interleaved, resulting in predictable component assemblies suitable for memory constrained ERTS. Our intention is not to develop an optimal control system, in fact, the primary concern is to design the system in a way that clearly illustrates and motivates the activities used throughout the component based development process.

Throughout the example, we assume that the necessary tools to choose components, create assemblies etc, are at the developers disposal.

5.1 Introduction

We believe that one of the main concerns of a designer, in the initial phases of development, is to design the system in terms allowing analysis. Hence the component architecture together with the definition of data and control flow between the components, should be performed as early as possible. It would allow for automatic task allocations and analysis of a system, early in the development process. Figure 2 shows an overview of our envisaged development flow in constructing the control system for the robot. The system requirements form as input to the design process. The output from the design process should be an analysable and synthesizeable architecture. In case the analysis of the architecture reveals unfulfilled requirements, the process need to be iterated with a change in the requirements and/or design. As can be seen in figure 2, the development process flow take different paths depending on whether components are available in a repository or need to be developed. However, the component and application development processes can be separated, allowing them to be performed in parallel [CL02].

5.2 System requirements

The initial activity depicted in figure 2, consists of collecting the system requirements that will be used as input to the component aware design activity. Therefore, we begin the development of the example system by describing the requirements and prerequisites for our robot system example.

The challenge is to develop an control system responsible for the positioning of a robot arm consisting of three axis. The movement of each axis is controlled by separate motors (m1,m2,m3) and

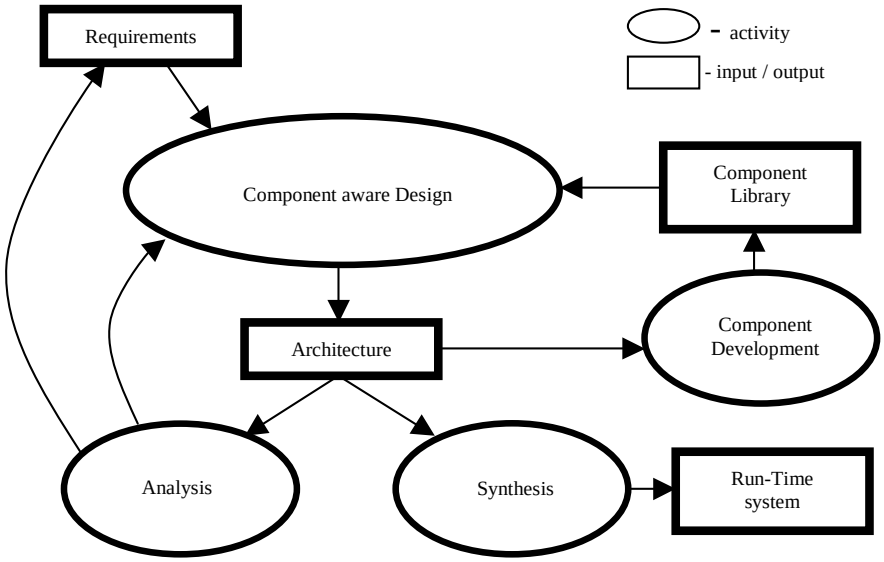

Figure 2: Overview of the Component Based Development process used in the robot system example

indicated by three separate pulse-sensors (s1,s2,s3). In addition, the system consists of an emergency stop button and sensors to calibrate initial axis positions. The requirements on the example system are as follows:

- *Req.1*: Strived positions for the robot arm must be reached with an accuracy of ± 20 pulses
- *Req.2*: A pressed emergency stop button must be detected, and motors stopped, within 100ms
- *Req.3*: A total of 4 kB of random access memory (RAM) are available for the application positioning the robot

The following is a list of system prerequisites and robot characteristics for the example:

- *Pre.1*: The strived positions for the robot, are pre-defined and stored in a vector *v* as coordinates in space. The vector will be available as input to the system.
- *Pre.2*: A sensor *n* detects if an emergency stop button is pushed. The minimum interarrival time between two consecutive activations of the emergency button, is assumed to be infinitely long.
- *Pre.3*: Sensor *s1* generates 3712 pulses per revs
- *Pre.4*: Sensors *s2* and *s3* generates 1280 pulses from end to end positions
- *Pre.5*: Three sensors *e1,e2* and *e3* are used to calibrate axis positions at system start. Each of the sensors generates an event when the desired calibration position is reached. The minimum interarrival time between two consecutive activations of these sensors, is assumed to be infinitely long (indicates that it happens only once).
- *Pre.6*: The maximal velocity of each motor is 100 pulses per second. Each activation of an motor will run the motor for a shorter time (100 - 200ms). Hence there may be a need for several consecutive activations, to reach a specific position. Furthermore, the inertia of each motor may result in a drift of up to 10 pulses at motor deactivation.

- *Pre.7*: The operating system is activated periodically each millisecond by a timer interrupt. A timer interrupt uses approx. 100 bytes of memory at each activation. The timing overhead of the operating system is 30 μs at each timer interrupt. The context switch overhead is approx. 30 bytes.

5.3 Operational modes and implementation architectures

This section describes the design of component architectures for the robot example system.

To begin with, the operational modes [NGS+01] of the robot are identified and realised with a component architecture suited for each mode. The components used in the architectures may either be existing ready to use components, or components in need of development (see figure 2). For the sake of reasoning, we assume in this example, that the functional and non-functional properties are stated with similar notation independently of whether the components are developed in-house or not.

5.3.1 Operational modes

With the requirements at hand, we begin with identifying the operational modes of the system. The purpose is to identify the states of the system and separate the high level functional behaviour. The partitioning into modes gives a high level description of the system, which might facilitate the understanding of the system, especially for new developers. Furthermore, most systems have a concept of modes and making it explicit, rather than implicit in the code, makes the design much more maintainable.

Two modes are identified for the robot; an initial mode and an manoeuvring mode. The initial mode is needed to calibrate the axis to get a pre-defined position (0,0,0). The manoeuvring mode will be used to position the robot axis according to the pre-defined position data in vector v. A simple sketch of possible mode transitions is shown in figure 3.

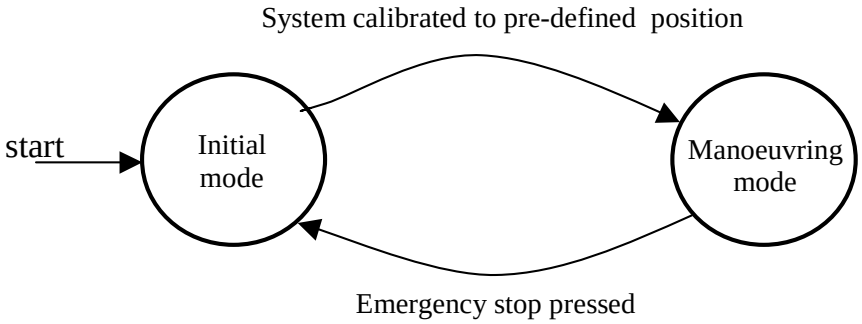

Figure 3: Operational modes for the robot system example

5.3.2 Designing the component architecture

The component architectures for the identified modes may be designed using the following activities:

- Identification of the environment (hardware components)
- Identification the needed functionalities. This should not be confused with identification of program functions. Functionalities are, by our definition, high level system behaviour

- Choosing of components to model the functionalities
- Definition of data flow between components
- Definition of control flow between components

There is however no need to perform the activities in any specific order. The above activities could be performed in the best suitable order for the application in development.

For the sake of simplicity, we will perform the design of the component architecture, in the order depicted above, starting with the identification of system functionality and environment.

5.3.3 Initial mode - identifying functionality and environment

We start by identifying the needed functionalities in the initial mode, beginning with the emergency stop functionality. Two associated functions are identified; the detection of a pressed emergency button and the halting of the motors.

Continuing with the functionality of calibrating the axis, two functionalities are needed; the reading of calibration sensors e1-e3 and the running of the axis motors to move the robot arm to a predefined starting position.

Thus, we can see that the environment consists of the end position sensors (e1,e2,e3), the emergency stop senor (n) and the three motors (m1,m2,m3).

5.3.4 Initial mode - choosing components

We continue by choosing suitable components for the identified functionalities.

The emergency stop functionality could be modelled by using two separate components, a sensor-handler (component N) and an motor controller component (component M).

The calibration of the robot could also be performed using two components; one that handles the input from the calibration sensors and another that controls the motor activations. For the reading of calibration sensors e1-e3 we choose to use a new component (component E).

The calibration of the robot arm requires running and halting of the motors. For this we could use the same component (component M) as we used earlier for the emergency stop functionality. The component M must be continuously activated, according to prerequisite no 6, since the motor will otherwise stop. Some periodicity is therefore required.

5.3.5 Initial mode - defining data and control flow

We continue with defining the dataflow and control flow between the chosen components in the initial mode.

Starting with the emergency stop functionality, the sensor handling component (component N) is activated by a press on the e-stop button. It then processes the event and forwards a halt event to the motor controller component (component M). Hence there would be a need for a control flow between

sensor n, component N and component M.

Continuing with the calibration, component E is activated by sensors e1, e2 or e3 when an axis reaches a desired calibration position. When activated, the component forwards a halt motor event to component M. When component M receives information that all calibration sensors (e1,e2,e3) have reached their end positions (indicates the predefined position (0,0,0)), component M initiates a mode change. Furthermore, during calibration, component M must be activated periodically according to the characteristics. Figure 4 illustrates the notation used in the graphical figures depicting component assemblies. Figure 5 shows the resulting component architecture for the initial mode. Notice that the components in figure 5 have annotations describing their worst case execution times (WCET) and maximum memory consumption (MEM). These are, depending on whether the components are developed or not, either derived specifications for existing components, or budgets given by the application designers to the component developers.

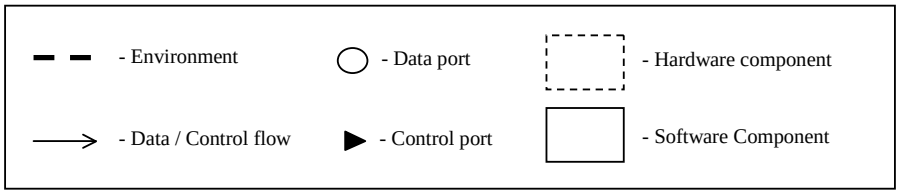

Figure 4: Notation

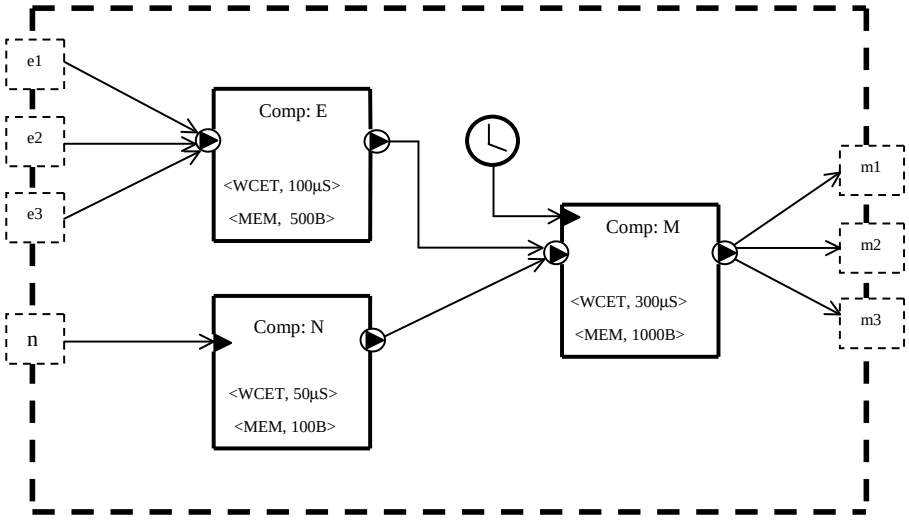

Figure 5: Component architecture for the initial mode

5.3.6 Manoeuvring mode

Using the same design activities as for the initial mode, the component architecture for the manoeuvring mode may be realised by four components. Figure 6 shows the resulting components and their interaction in the manoeuvring mode.

As in the initial mode, component N would be activated by a press on the e-stop button, it then processes the event and forwards a halt event to the motor controller component M. Component P

polls the sensor values s1-s3 for changes in axis positions and calculates the current position of the robot arm. The position is forwarded to component C, which takes vector v as input to decide whether the strived position is reached or not.

For the sake of this example, a required data port from component C is connected to a provided port on component P. The data is used to indicate the direction in which the motors are running. This information is needed since component P only has a pulse as input and has to decide whether to count the position of a robot arm up or down. This exemplifies a case that would not be possible to achieve following the definitions in PECT [Wal03]. The component interactions, as defined in PECT, is handled through sink and source pins. Pins are means of handling sole events, or events bundled with data. There is however, no specified way of interacting through pure data on sink or source pins.

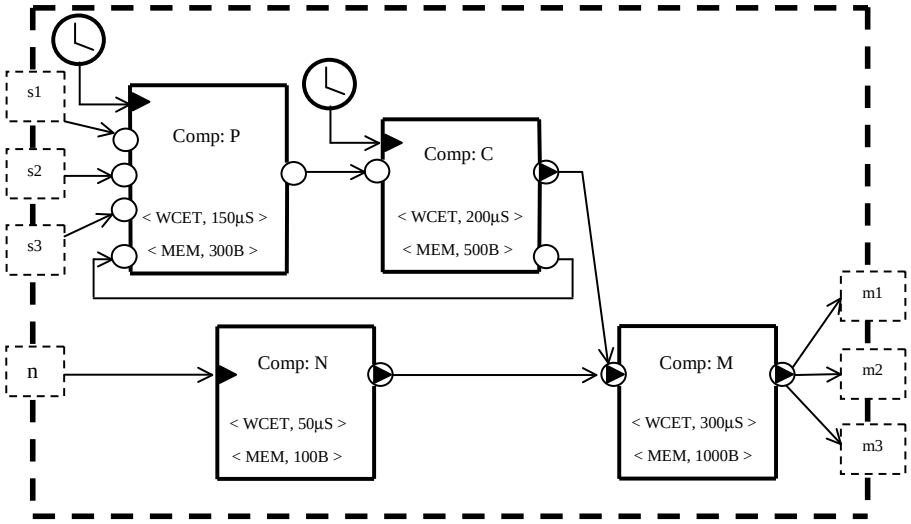

Figure 6: Component architecture for the manoeuvring mode

5.3.7 Component assemblies

In order to deal with complexity issues for the designer, component assemblies must be treated the same way basic components are treated. There are two main reasons for this:

1. Hierarchical decomposition. The developer should be able to hide unnecessary details and view the system from different abstraction levels. A ”flat” structure with all information available at all times clutters the high level abstraction view.

2. Reuse of components. If a component is to be reused it must be packaged, and from the developers view, look exactly as a basic component does. If the developers are forced to figure out the inner workings of the assembly, they would probably be reluctant to use it. It would in many cases be easier for them to construct it by themselves.

The above reasons motivate the black box approach of CBSE. From the user’s perspective, an assembly will behave like a component if the interfaces (provided and required) on the boundary, are passed to the component assembly instead. A special case is control-interfaces, where there is a choice during component construction (or equally during hierarchical decomposition), whether to embed timers and/or hardware components into the component assembly. Both approaches are possible. If left out, a control interface becomes available on the component. And the other way around, if

embedded in the component assembly, the user of the component assembly does not se how the services of the component are activated. As an example, see figure 7 where we have packaged the manoeuvre mode, shown in figure 6, into one component.

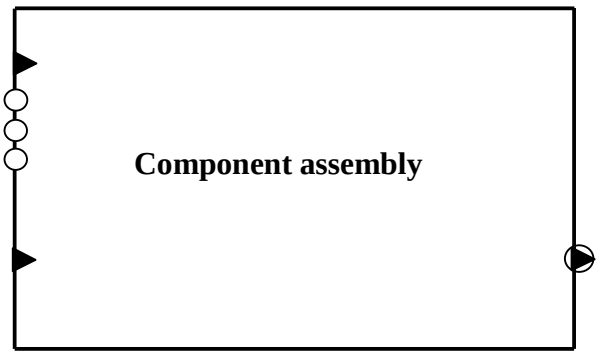

Figure 7: Black box view of a component assembly

In figure 7, we have chosen to embed one timer into the component assembly, whereas the other control interfaces are placed on the border of the component assembly.

5.3.8 Task allocation and attribute assignment

When the initial component architectures are defined, the component assemblies must be partitioned into executable units called tasks. For each mode, every component in the architecture should be automatically mapped to a real-time task. There is a lot of issues involved in the allocation of components into tasks. One of them being the question whether a component should be restricted to deliver one single service or several ones. This issue raises the question of how components should be instantiated. Should we allow several instances of a component or a single instance shared by several tasks. Allowing several instances would consume more memory than a single one, but issues involved in service precedence's might be easier to cope with.

In this example, we allow the instantiation of a component into several tasks. Furthermore, the allocation is performed in a way allowing us to derive WCET:s for each task. Beginning with the components for the initial mode (see figure 5), using some task allocation strategy, might results in three tasks for the mode.

- Task T_M : A periodic task activating motors m1,m2 and m3. Task T_M activates component M. (One can view M as a program function)
- Task T_{NM} : An event triggered task handling emergency stop functionality. Task T_{NM} activates component N followed by M.
- Task T_{EnM} : Three instances (n=1...3) of an event triggered task responsible for the calibration of the robot arm. Each task instance activates component E followed by M.

Continuing with task allocation for the manoeuvring mode (see figure 6), we might end up with the following tasks:

- Task T_P : A periodic task reading pulse sensors s1,s2 and s3. Task T_P activates component P.
- Task T_{CM} : A periodic task controlling the current position of the robot arm. Task T_{CM} activates component C followed by M.

- Task T_{NM} : An event triggered task handling emergency stop functionality. Task T_{NM} activates component N followed by M.

Task attributes, such as period and priority, could be mapped from user specification of components. This information can be specified for software and/or hardware components. For example, a hardware components generating events into the system must describe its temporal characteristics (minimum interarrival time between two consecutive events) and for timers the period time. As for priorities this could be specified on the level of components and mapped to tasks with similar rules as for those used in PECT. Another approach, is to relieve this specification burden from the designer completely, as in AutoComp, and let an analysis tool do priority allocations. This paper does not resolve this issue, any or a combination of the above approaches could be used in our presented context. For the sake of completing our example we assume the following priorities and periods has been derived by some technique:

Initial mode

- Task T_{NM} : P = High
- Task T_{EnM} : P = Medium
- Task T_{M} : P = Low, T = 100ms

Manoeuvring mode

- Task T_{NM} : P = High
- Task T_{P} : P = Medium, T = 5ms
- Task T_{CM} : P = Low, T = 50ms

For the reminder of this example we assume that the resulting tasks sets rely on the single shot execution model which allows several tasks to share a common stack [DMT00, dEg]. Furthermore, we assume that tasks with equal priorities are handled in first come first serve manner (FCFS), i.e., they can not pre-empt each other.

5.3.9 Task allocation for assemblies

For task allocation of assemblies the black box approach described in section 5.3.7 is not sufficient. In order to perform a task allocation the detailed control information is needed, i.e., a structure that is flattened out. From task allocation point of view the control-interfaces can not be black-box, it needs to know the inner workings of each component, i.e., a grey box (look, but don't touch!) approach. Figure 8 shows what information task allocation needs for our component assembly of figure 7.

5.4 Analysing the system design with reasoning frameworks

This section describes how schedulability analysis and prediction of memory usage may be performed for the task sets in our robot system example.

So far in the development process we have mainly used techniques outlined in AutoComp [SFA04]. The contribution of PECT [Wal03], to our robot system example, consists of theories allowing us to reason about our assemblies. In order to analyse our example system, we establish two reasoning frameworks [Wal03], by defining a property theory and an automatic reasoning procedure for each

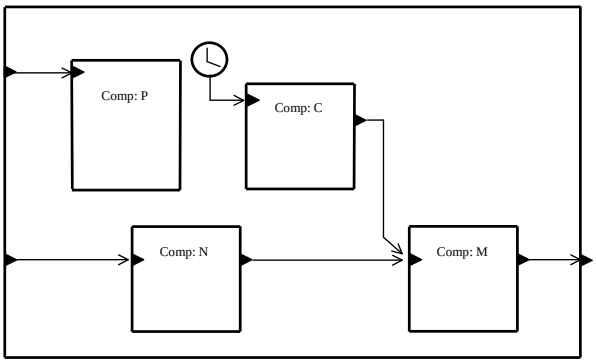

Figure 8: Grey box view of a component assembly

framework. The validation procedure[1] for the analysis of our example system, will however not be address. We assume that the given properties for the components are correctly specified. Furthermore, the reasoning frameworks are based on formally validated theory.

5.4.1 Property theory

The property theory for our example systems consists of two distinct parts; a timing analysis property checking for schedulability, and a memory consumption theory. The property theory for analysing timing behaviours in our example system may be expressed by formula 1 [JP86]. The formula express the maximum response time for a task, taking in consideration the influence imposed by all higher priority tasks. The computed response times may then be used to examine the schedulability of the task sets.

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{1}$$

The maximum memory consumption property for the assembly of the robot system, may be expressed by formula 3. The original formula 3 is presented in [DMT00]. We have slightly modified the formula and added formula 2, to express the memory usage with respect to the FCFS task activation strategy explained in section 5.3.8. The property theory is however a safe estimation of the maximum memory consumption of our robot system example.

$$mode\_mem = \sum_{\forall p \in prio\_levels} \left(\max_{\forall t \in prio(p)} (t^{mem}) + Context \right) \tag{2}$$

$$Total\_memory\_usage = mode\_mem + ISR \tag{3}$$

prio_levels : All priority levels in the system
prio(i) : Set of all tasks with priority level *i*
Context : Memory required to save register data etc., at a task switch
t^{mem} : Memory usage of task *t*
ISR : Memory required by timer interrupt service routine

[1] A reasoning framework in PECT consist of three distinct parts; a property theory, an automated reasoning procedure and a validation procedure. The validation procedure serves to explore the trustworthiness of a reasoning framework

5.4.2 Automated reasoning procedure

The interpretation of the assemblies to the reasoning frameworks is pretty straightforward. In section 5.3.8, components were assigned into sets of tasks with belonging temporal and memory attributes. The assignment makes is possible to automate both timing and memory reasoning of the assemblies i.e., it is possible to map parts of the concrete construction syntax to strings for our property theories.

The following is a list of response times and memory consumptions derived for the tasks in the robot example system. Prerequisite no. 7 lists the OS timing overhead used in the calculations. The response times for each task are calculated with formula 1. The memory consumption of each task is calculated as; the sum of memory used by all components assigned to the task.

Initial mode

- Task T_{NM} : R = 380 μs, Mem = 1100 bytes
- Task T_{EnM} : R = 780 μs per instance, Mem = 1500 bytes per instance
- Task T_{M} : R = 1110 μs, Mem = 1000 bytes

Manoeuvring mode

- Task T_{NM} : R = 380 μs, Mem = 1100 bytes
- Task T_{P} : R = 530 μs, Mem = 300 bytes
- Task T_{CM} : R = 1060 μs, Mem = 1500 bytes

Table 1 shows the calculated memory usage in each mode (see prerequisite no. 7 for the values of ISR and Context).

Table 1: Memory usage in each mode of the robot system example

Priority level	mode_mem (init.)	mode_mem (man.)
High	1100 + 30 = 1130	1100 + 30 = 1130
Medium	1500 + 30 = 1530	300 + 30 = 330
Low	1000 + 30 = 1030	1500 + 30 = 1530
Total (mode_mem + ISR)	3690 + 100 = 3790	2990 + 100 = 3090

The output from the reasoning procedure is then analysed and compared[2] to the requirements.

Req.1: Strived positions are reached with an accuracy of $\pm$17 pulses (max speed: 100 pulses/sec.) in the manoeuvring mode. Hence the requirement of $\pm$20 pulses, is fulfilled.

Req.2: A press on the emergency button is detected and processed within 380μs. The requirement stated that an emergency stop should halt a motor within 100ms upon pressing the button, hence the requirement is fulfilled.

[2]This correspond to the decision procedure element of the automated reasoning procedure in PECT

Req.3: The maximum memory usage is calculated to 3,79 kB, hence the requirement of 4kB, is fulfilled.

5.5 Synthesising the analysed design

In order for the component model and analysis framework(s) to be used in practice, there has to be a resource efficient mapping from the architecture to a resource structure (run-time system). In the synthesis step, tasks are assigned to threads of control and communication between them are solved, mapped and realized by operating system primitives and possibly some glue code. Input from the analysis phase (e.g., task allocation, priorities, schedules) is crucial since the synthesis step must adhere to the assumption made by the analysis framework. Sometimes, the activities of analysis and synthesis are closely related, almost indistinguishable, e.g., when creating a static schedule, which is a proof-by-construction technique.

To show that the mapping to a run-time system is very much a trade-off situation and dependant on which properties are stressed (even in the same system some parts have different needs) we compare the time-triggered (TT) execution paradigm and event-triggered (ET) paradigm.

Time-Triggered:

+ Useful for control functionality, Periodic activation of tasks are not dependant on the environment. This will lead to a robust solution that naturally deals with overload situations, they never occur since the system dictates itself independently of the environment. Another benefit is that it is reproducible and hence testable, which make a testing approach to functional verification much easier.

- Pure time-triggered solutions becomes very strict and inflexible, changes to the schedule (if static schedules are used) by adding or removing tasks, will result in new analysis and synthesis step (for TT task in FPS it may suffice to do the analysis step and just add the task at a feasible priority). Also since the system is designed for the worst case it will always exhibit this behaviour. If events in the environment does not occur at their worst rate, the TT approach is unable to utilise this underload situation for other soft functionality. Futhermore, static schedules can be memory consuming.

Event-Triggered:

+ Events are modelled more naturally, and thus better response times can be obtained for tasks defined at highest priority. The system is well equipped to handle underload situations, since tasks are activated by the environment. Whenever there is a spare capacity, the CPU can use the time for soft functionality such as diagnostics.

- Overload situation may collapse the system, e.g., if a minimum inter-arrival time is incorrectly specified or a sensor is faulty (generating bursty events). Also, the behaviour in a ET system is very hard to reproduce, thus verification by testing gets difficult and time-consuming.

Communication: Generating code for communication between ports is dependant on whether the ports are allocated to the same task or different tasks. Generally, code to copy data from in- to out ports needs to be generated. To guarantee mutual exclusion three cases have to be considered:

- Among TT : Solved in the analysis by time separation

- Among ET : Semaphores are added by tools, response time equation is extended with blocking factor.
- Between ET & TT : In schedule construction (analysis/synthesis tools), blocking of a TT task must be taken into account.

The synthesis part is the work that is mostly future work and where little has been done. There is a commercial concept that realises this step, Rubus [AB]. However the Rubus component model is restricted to statically allocated tasks in pre-defined schedules. Also the analysis and synthesis parts are only concerned with these so called ”red” tasks. The Rubus OS also supports interrupts and FPS tasks. The lack of higher level tool support result in heavy use of the red part for companies using Rubus. We have started a project called MultEx that will look into which execution paradigms should be supported by an OS and how one can provide support for multiple execution paradigms, all the way from construction (component and architecture) via analysis to synthesis.

6 Conclusion and Future Work

How can CBSE aid in the development of RCERTS? Our belief is that connecting CBSE and RTS theory by methods (processes) and automated tools can provide a big step towards this. Our vision of the development process is that:

- Developers design by (re)using components, and thus producing a component architecture at the ”appropriate” level of abstraction. Appropriate means that the components and the component model are expressive enough so that the application requirements can be expressed and fulfilled as naturally as possible. Note also that too expressive component model lets the developers ”stray” and use solutions that are hard to understand and reuse (hacker solutions instead of engineering solutions).
- The produced component architectures should be analysed for certain properties, responsiveness and memory consumption are treated in this paper. We take the PECT view on this where a constructive framework corresponds to the point above. If no analysis framework is used, the developer can utilise the full expressiveness of the model. If an analysis framework is used, it may place some restrictions on architectural constructions or even on components. You can compare this for a real-time system that uses C as a programming language, most probably constructs such as dynamic memory allocation, recursion, unbounded loops etc, are not allowed. However, different applications have different demands of what properties are important to analyse, therefore the plug and play approach of analysis frameworks is attractive instead of hardwiring it to method and tools.
- Last but not least, in order for the component model and analysis framework(s) to be used in practice, there has to be a resource efficient mapping from the architecture to a resource structure (run-time system). Traditionally CBSE is pushing complexity away from the designer, ending up with the complexity in the run-time system. However this is not acceptable for RCERTS. For RCERTS there is an additional push: from the run-time system (saying: ”we have no resources to do all that”) up to the development layer.

So where should the complexity end up, if the user, nor the run-time system can cope with it? Extensive constructive, analysis, and synthesis, methods and tools! Compare to a compiler where lot of theory, analysis, and synthesis techniques are hidden from the user, still a programmer knows how to produce resource efficient code by using the compiler without knowing all the details of it.

References

[AB] Arcticus Systems AB. Home page. http://www.arcticus.se.

[BBB+02] F Bachmann, L Bass, C Buhman, S Comella-Dorda, F Long, J Robert, R Seacord, and K.C Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, September 2002.

[BCC+03] E Brinksma, G Coulson, I Crnkovic, A Evans, S Gérard, S Graf, H Hermanns, B Jonsson, A Ravn, P Schnoebelen, F Terrier, A Votintseva, and J.M Jézéquel. Component-based design and integration platforms. Roadmap, Advanced Real-Time Systems Information Society Technologies (ARTIST), May 2003.

[CL02] I Crnkovic and M Larsson. *Building Reliable Component-Based Software Systems*. Artech House Publishers, 2002. ISBN 1-58053-327-2.

[dEg] Live devices ETAS group. Home page. http://www.ssx5.com.

[DMT00] R Davis, N Merriam, and N Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[FSN+03] J Fröberg, K Sandström, C Norström, H Hansson, J Axelsson, and B Villing. Correlating bussines needs and network architectures in automotive applications - a comparative case study. In *proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FET)*, Aveiro, Portugal, July 2003.

[JP86] M Joseph and P Pandya. Finding response times in a real-time system. *Comput.J*, 29(5), 1986.

[MAFN03] A Möller, M Åkerholm, J Fredriksson, and M Nolin. Software component technologies for real-time systems -an industrial perspective-. In *proceedings of the WiP session of the 24th IEEE Real-Time System Symposium*, Cancun, Mexico, December 2003.

[MFN03] A Möller, J Fröberg, and M Nolin. What are the need for components in vehicular systems? -an industrial perspective. In *proceedings of the WiP session of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003.

[MFN04] A Möller, J Fröberg, and M Nolin. Industrial requirements on component technologies for embedded systems. In *International Symposium on Component-based Software Engineering (CBSE7)*, Edinburgh, Scotland, May 2004.

[NGS+01] C Norström, M Gustafsson, K Sandström, J Mäki-Turja, and N-E Bånkestad. Experiences from introducing state-of-the-art real-time techniques in the automotive industry. In *8th IEEE International conference and workshop on the Engineering of Computer-Based Systems*, Washington, USA, April 2001.

[San02] K Sandström. *Enforcing Temporal Constraints in Embedded Control Systems*. PhD thesis, Royal Institute of Technology, April 2002.

[SFA04] K Sandström, J Fredriksson, and M Åkerholm. Introducing a component technology for safety critical embedded real-time systems. In *International Symposium on Component-based Software Engineering (CBSE7)*, Edinburgh, Scotland, May 2004.

[Szy98] C Szyperski. *Component software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York. N.Y, 1998.

[UPP] UPPAAL. Home page. http://www.uppaal.com.

[Wal03] K.C Wallnau. Volume iii: A technology for predictable assembly from certifiable components. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, April 2003.