# The SEESCOA Composer Tool: Using Contracts for Component Composition and Run-Time Monitoring

**Stefan Van Baelen, David Urting, Yolande Berbers**

Department of Computer Science
K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium

{ stefan.vanbaelen, david.urting, yolande.berbers}@cs.kuleuven.ac.be

## Abstract

In this paper, our approach for building embedded applications is presented. The approach is based on the composition of reusable components with the addition of a contract principle for modelling non-functional constraints. Non-functional constraints are an important aspect of embedded systems, and are therefore modelled separately in contracts. As such, the component view presented here differs from traditional component based views, where focus is laid on the functional part. The ideas discussed in the paper have been implemented in a tool. This tool enables the construction of embedded software by means of components and contracts. Currently, the generation of runtime mechanisms - that enable runtime monitoring of the contracts - from the component model are being included.

*Keywords*: tool, component, embedded, real-time, contract.

## 1 Introduction

Component software is quite common today in traditional applications. A large software system often consists of multiple interacting components. These components can be seen as large objects with a clear and well-defined task. Different definitions exist of a component (Szyperski 1998); some see objects as components, while others define components as large parts of coherent code, intended to be reusable and highly documented. However, all definitions have one thing in common: they focus on the functional aspect of a component.

The main goal of using components is the ability to reuse them. Reuse of software is currently one of the much-hyped concepts, since it enables one to build applications relatively fast. Using components is seen as a possible way to ensure this reusability and this is one of the reasons why they receive a lot of attention.

In embedded software, one has to consider non-functional and resource constraints when building a system (besides software quality aspects such as reusability). Embedded systems often have limited processing power, storage capacity and network bandwidth. A developer has to cope with these constraints and make sure that the software will be able to run on the constrained system. Often, embedded systems also have timing constraints on their computations. Missing a time constraint can be catastrophic (e.g. late activation of a cooling subsystem in a factory) or annoying (e.g. missing some video frames on a portable video device).

Today, embedded software is becoming complex and it is not easy to build every system from scratch. Therefore, it is important to reuse existing software as much as possible. This will ensure that one can use validated software and in turn results in a smaller development time. To enable reuse, we have chosen for a component-based approach for building embedded systems.

Reusing components in embedded systems is not easy to do. The correct working of such a system is not only dependent on the correct functional working of the component; it is also dependent on its non-functional properties. For example, using a component that consumes large amounts of internal memory in memory-constrained systems is not a good idea. The same occurs for situations where time constraints exist: if a component takes too long to process some data then this may render the complete application useless.

It is clear that some way is needed for the specification and checking of non-functional constraints. This will enable one to safely reuse components in a design, while being sure that the non-functional constraints will be met. In our approach, contracts are used to ensure this.

The paper discusses some core concepts of components and contracts in section 2. Section 3 discusses the tool and the basic constructs for building applications. The usage of these constructs occurs by means of models, which are described in section 4. Section 5 describes the run-time component system and the generation of run-time contract monitoring. In section 76 related work is discussed. Section 7 gives an overview of the current status and intended future work. We conclude in section 8.

A tool (SEESCOA Composer tool) has been implemented and screenshots of this tool will be shown to illustrate the approach. The work presented in the paper is being performed in the scope of the SEESCOA project (Software Engineering for Embedded Systems, using a Component Oriented Approach), funded by the Belgian IWT.

## 2 Core Concepts

This section discusses some techniques and ideas that lie at the basis of the tool. First we (briefly) discuss the *ROOM* methodology [Selic94]. Next the *design by contract* principle is discussed. We also elaborate on how this contract principle can be applied to specify non-functional constraints.

### 2.1 ROOM

ROOM stands for Real-time Object Oriented Modelling. Since then it has received quite some attention from the (real-time) software area. Although initially intended for designing and building telecommunication systems, the ROOM methodology can also be used for the design of other types of embedded systems.

ROOM designs contain primarily actors, ports, bindings and state machines:

- An actor is an autonomous piece of code that plays a specific role in the system. The main distinction between an actor and an object is the fact that an actor behaves autonomously. An advantage of this is that the encapsulation goes a step further: objects only offer data encapsulation while actors offers data and thread encapsulation[1].
- A port is an opening on the encapsulation shell of the actor through which it can send and receive messages. A port has also a specification (protocol) associated with it.
- A binding is a connection between two ports. Actors can exchange messages via bindings. It is only possible to create a binding between ports that have the same protocol.
- A state machine describes the internal workings of an actor. This state machine describes how the actor will respond to (external) messages and the states it will be in.

The ROOM methodology has some new and interesting ideas: it introduces thread encapsulation that hides the internal thread mechanisms. It offers and alternative way of connecting software components by means of bindings. Also, the idea of port protocols is an advantage since it enforces a designer to only connect compatible ports. To conclude, it offers the ability to generate code (by putting code into transitions of the state machines) and model execution[2].

ROOM however lacks a consistent way to annotate time in designs. In general, ROOM has no support for the annotation of non-functional constraints, like memory and bandwidth constraints.

## 2.2 Design by Contract

The design by contract principle is a well-known and interesting principle [Meyer97].

In general, a contract specifies an agreement between two or more parties about a service. This contract principle can be applied to software: since components offer services to other components, the properties of these services can be put in contracts.

There are different ways to use a contract:

- A first approach is to use them in a notational manner; the contract is only informing the designer about particular service properties. In this case, a contract can be seen as extra documentation.
- Contracts can also by used to test an application; the contract is used to perform runtime checking of the contract properties. The contract is thus not only used for annotating the application, it is also used for monitoring the application. However, using contracts for testing does not guarantee that the contract will hold at all times.
- Finally, contracts can be submitted to an analysis process; an algorithm performs a static analysis on the contracts before execution of the application. This static analysis guarantees the correct working of the application at all times. A drawback is that such an algorithm can be highly complex and in some cases even impossible to implement.

## 2.3 Contracts and Non-functional Constraints

Meyer was initially focussing on contracts for the description of pre- and postconditions of operations. These contracts are describing the semantics of applications. A runtime checking mechanism for these semantical contracts has also been implemented in the Eiffel language.

We have taken the same approach (= contract based with runtime checking) for specifying and monitoring non-functional constraints. Non-functional constraints are specified by means of contracts, and a runtime mechanism is responsible for the runtime checking of these contracts.

Every non-functional constraint has its own properties (deadline, duration, period, … for timing constraints and heap usage, stack size, … for memory constraints) and thus different contract types need to be defined. We have currently focussed on one particular type of contracts: timing contracts. These contracts are used for the specification and monitoring of timing constraints.

## 3 Basic Tool Concepts

As was mentioned earlier, our approach is component oriented and based on ROOM. What follows is the definition of a SEESCOA component:

*A SEESCOA component is an object offering a coherent behaviour. Some other component can access this behaviour by sending asynchronously messages to the component. To do so, both components need a port. These ports have to be connected by a connector. Also, these ports have to understand each other: they have to speak the same protocol. The port protocol is described in the type or interface of the port.*

Our approach consists of the following important constructs: component blueprint, port blueprint, component instance, port instance, connector and contract. These constructs are explained in the following subsections.

[1] Thread encapsulation means that two different threads cannot alter the internal state of the object at the same time. This ensures that the data is always left in a consistent state.

[2] Rational RoseRT (www.rational.com) supports code generation and model execution.

## 3.1 Component and Port Blueprint

A component blueprint is a reusable entity and contains the type description and implementation of a component. It is a static construct that has no runtime meaning. Component blueprints have an identifier, a version and can be stored in a catalogue. It is represented as a stereotyped UML class, built as complex composition of all objects and classes that are contained inside the component.

The interfaces of a component blueprint are described by means of port blueprints. A port blueprint has a set of messages that can be received or sent. A port blueprint is specified on four levels [Beugnard99]:

- Syntactic level: syntactic description of messages that can be sent and received.
- Semantic level: pre- and postconditions associated to the messages.
- Synchronization level: description of the sequence in which the messages have to occur. This level is specified by means of extended MSC's. This is an MSC[3] extended with constructs for indicating loops, alternative and optional paths.
- QoS level: quality of service description.

Currently, we only use the first three levels. The QoS level is not formalized yet, but this will be formalized in the future.

The port blueprint has a MNOI[4] property; this property indicates how many times the port can be instantiated. We will discuss its use in the next section.

An example of a component blueprint is given in figure 1. It concerns a sampling component, which defines two ports, one for setting the sampling frequency and another port to output the measured values to other components. The read-out port has an unlimited MNOI, which means that an unlimited number of other components can read out the sampling value. The settings port has defined 1 as MNOI, which means that only one other component can determine the settings.

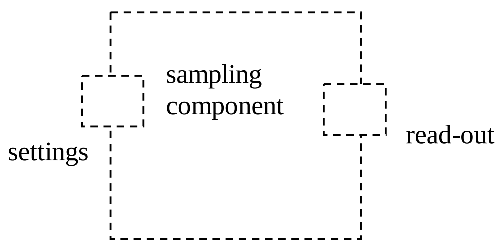

**Figure 1: sampling component blueprint**

## 3.2 Component and Port Instance

A component instance is an instantiation of a component blueprint and has a runtime existence (and state). It is represented as a stereotyped UML object. Communication with other component instances occurs via port instances. These port instances are instantiated from corresponding port blueprints. The maximum number of ports that can be instantiated from a port blueprint is given by its MNOI property. The port blueprint/instances are also represented as stereotyped UML classes/objects, linked to the component with a composition association.

Figure 2 shows an instantiation of the sampling component. The settings port has been instantiated once, while the read-out port has been instantiated three times. This is possible since the read-out port blueprint has an unlimited MNOI.

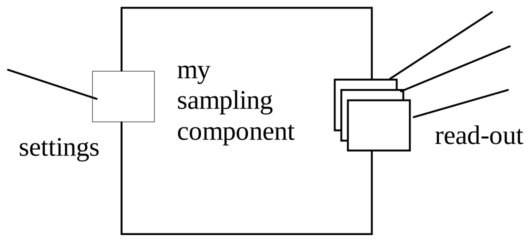

**Figure 2: sampling component instantiation**

## 3.3 Connector

A connector interconnects two or more port instances so that messages can be exchanged between them. A port instance can only be connected once to a connector. A connector is a stereotyped UML association between ports.

It is not possible to connect port instances that have an incompatible protocol: they must be compatible on syntactic, semantic, synchronization and QoS level. This means that port instances have to speak the same 'language' before being able to communicate in a coherent way.

A connector has an MSC associated with it. This MSC can be distilled out of the MSC's of the port instances to which it is connected. The MSC of a connector describes how interaction occurs among the involved port instances.

In figure 3 an example is given of a connector between a sampling component instance and a value display instance. Via this connector, the sampling component sends value updates to the value display. The MSC of this connector is also represented in figure 3.

[3] Message Sequence Chart (ITU-MSC 1996)

[4] Maximum Number of Instances

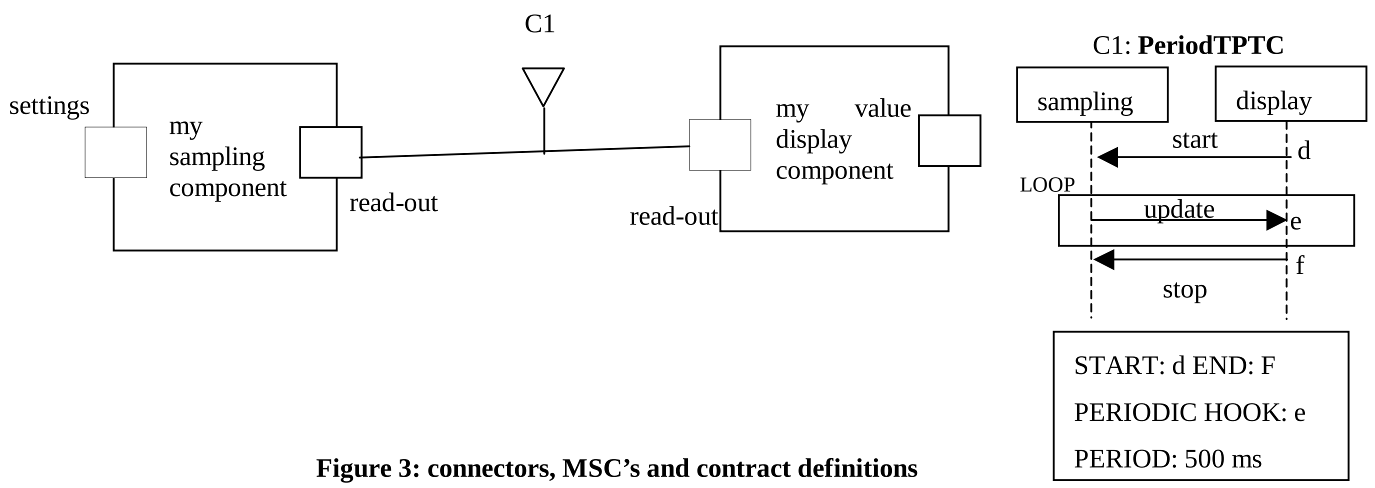

**Figure 3: connectors, MSC’s and contract definitions**

The interface QoS definition level is the most challenging and also the most difficult to specify: it is not straightforward to describe QoS information in a general way without knowing the specific environment and hardware the component will be used in.

There is no single solution to this problem. Nevertheless three approaches can be used to give at least some idea on the QoS:

- QoS parameters can be *measured against a reference setting*: one hardware configuration, one OS configuration and one application type are chosen and then the component is tested in this setting.
- A *formal calculation* of the QoS can be performed. But for this to work, one has to know the dependencies between hardware, OS, etc. Also, one must be able to model these dependencies to create a function that in turn calculates the QoS parameters. This approach is very difficult and is not investigated in our research.
- *Execution and measuring*. This seems to be a valuable and achievable approach. The idea is that a component can be tested on the target platform before it is used. This testing produces measurements that can give an idea of the delivered QoS of the component.

We currently use the execution and measuring approach in SEESCOA

As was mentioned earlier, a contract is used to impose constraints on a design. A contract is attached to one or more participants, where each participant is a component instance, a port instance or a connector. The number and type of participants is of course dependent on the contract type. Currently, only a timing contract has been defined.

A timing contract specifies a time constraint on the interaction between components. As a result, timing contracts can be attached to connectors, since the interaction between components occurs via connectors.

In figure 3, a *TwoPartyTimingContract* (also called a *TPTC*) is shown. It represents the constraint “the value display must be updated at a frequency of 2 Hertz”. The contract that is shown is a periodicity timing contract.

This type of contract makes use of the MSC of the connector between both components. It imposes a periodicity constraint on the occurrence of the update() messages.

A TPTC has following attributes:

- *Start hook*: validity of the contract starts when this hook occurs,
- *End hook*: validity of the contract ends when this hook occurs.

A TPTC has two specific subtypes: a *PeriodTPTC* and a *DurationTPTC*.

A DurationTPTC is a deadline contract, imposing a maximum duration between two particular interaction points. It is specified by:

- *WCD*: the worst-case duration between the occurrence of the end hook and the occurrence of the start hook.

The start and end hook does not have to belong to a single connector. Multiple components and connectors can be included

A PeriodTPTC imposes a periodicity constraint on a particular interaction point. It consists of:

- *Periodic hook*: the hook that should occur periodically,
- *Period*: a value indicating the period length.

A contract has a specification purpose: it shows the requirements in a formal way. It is also used by the code generator (which is a part of the tool): contracts are compiled to code. As such, these contracts can be monitored and violations can be logged, reported or even caught by the involved components at run-time.

## 4 Application Building

Until now, only the basic constructs for building applications have been discussed. This section introduces the component composer CASE tool[5] that has been built ,

[5] Currently, the tool is not available for external use, but it will be made available in the future via the following

which enables a designer to build an application with components and contracts. The tools supports three model types that are used for building applications: blueprint, instance and scenario models. Every model type has its own purpose, but the key idea is that they provide a way for decomposing a system in coherent parts.

The models consist of multiple instantiated component blueprints. Of course, the corresponding component blueprints need to be available when building these models. These component blueprints can be designed and implemented when needed, or can eventually be reused from a component library. This last aspect is very important, since reuse of existing components will shorten the development cycle.

The tool lets a designer create and design:

- Component blueprints: these component blueprints can then be deployed in the application or stored for later (re)use[6] .
- Applications (or compositions): after having created and loaded component blueprints, the designer is able to create an application by means of:
    - Blueprint models: component blueprints can be loaded into blueprint models.
    - Instance models: component blueprints that have been loaded into blueprint models can be instantiated and connected to each other.
    - Scenario models: constraints on the application are modelled by means of scenario models.

Component blueprints that are being designed can be stored on disk or used in the application. It is not possible to alter a component blueprint after it has been included in the application[7].

### 4.1 Blueprint Model

A blueprint model is a set of component blueprints that are being used in (a part of) the application. An application will often consist of multiple blueprint models, but a particular component blueprint can occur in only one blueprint model.

link: http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/

[6] The support for component blueprint storage is an important aspect if one wants to reuse component blueprints. Additional mechanisms like version control are also important and will be added to the tool in the future.

[7] There is an important distinction between the design of a blueprint and its use: a component blueprint is not necessarily made by the person who is using it.

Component blueprints that are closely related are put in the same model. In fact, a blueprint model is a functional, static decomposition of the application.

### 4.2 Instance Model

An instance model is a collection of interconnected component instances. An application can consist of multiple instance models, and a component instance can be present in more than one instance model.

An instance model represents the runtime situation of the application, so it represents a functional dynamic decomposition of the application.

This model can be compared to an UML object diagram, with the difference that a component instance model is used for an exact modelling of the runtime situation (there is a one to one mapping between design and runtime) while an UML object diagram is a ‘drawing’.

### 4.3 Scenario Model

A scenario model is a collection of interconnected component instances, with contracts attached to them. These contracts are used to specify a non-functional constraint on the application. A scenario model can contain more than one component instance, but the idea is to only add components instances that participate in the constraint. As such, a scenario model specifies a non-functional requirement on the application.

It is not possible to alter the structure of the application in a scenario model. A scenario model is thus used for the non-functional dynamic decomposition of the application.

In a future extension of the tool, we will add a code generator and runtime support for monitoring the contracts specified in scenario models. Scenario models are thus an important part of the overall approach: scenario models will support the monitoring of non-functional constraints imposed on an application.

## 5 Generation of run-time contract monitoring code

The SEESCOA Composer tool generates code for the SEESCOA run-time component system. The component system offers a platform for distributed, asynchronous component communication by intercepting each inter-component call on a component port and forwarding it to the receiver port, based on the present connections.

The generation of the run-time contract monitoring code consists of 2 parts:

- The Event Gathering part: all occurences of relevant events will be intercepted, recorded and timestamped by a probe added to the run-time system
- The Event Processing part: the timestamped events will be used to validate the specified timing constraints.

The SEESCOA Composer Tool generates code for the runtime monitoring of timing contracts. To do this, the timing contracts are mapped on RTL formulas. At runtime, these RTL formulas are checked by a RTL monitoring engine.

The run-time system extension has been designed as such that the *event processing* part of the monitor can be put outside the component system on any node in a network, while the *event gathering* part has to remain in the component system. This is done to minimize the intrusion, so that only the probe introduces a low and predictable overhead. The monitor activity processing can be placed outside the system on a separate node, thus causing no additional intrusion at all. This is illustrated in figure 4.

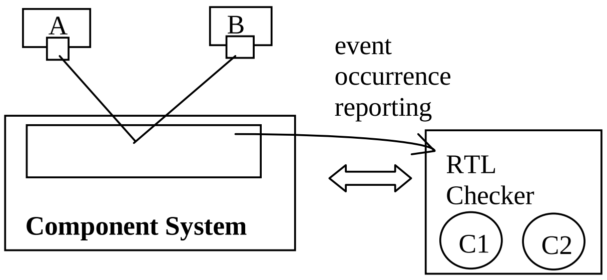

**Figure 4: Implementation of Constraint Monitoring**

However, if timing violations have to be reported to components that are part of the monitored application, then the monitoring system has to reside within the component system. In that case, there is no monitor proxy: all communication between the probe system (event gathering) and monitoring system (event processing) occurs directly.

## 6 Related Work

As mentioned in section 2, some concepts in this paper are based on ROOM [Selic94]. Ports, connectors and components are also present in the ROOM notation and methodology, although named differently. ROOM however lacks a mechanism for annotating and verifying non-functional constraints.

The contract-based approach is based on work done by [Meyer97]. The use of a four-level interface description was introduced by [Beugnard99].

The fourth level in that approach lets one specify QoS constraints. The specification of QoS constraints has received a lot of attention during the last years. Examples of such languages are described by [Frolund98] and [Loyall98]. However, the focus of these languages is not especially on timing constraints.

There is also a lot of research done on the specification of timing constraints. This research is often oriented on static verification of timing constraints. Examples of this are: RTL[8] [Jahanian86], ACSR[9] [Clarke97] and timed MSC's (Ben-Abdallah97]. A classification of RT specification languages can be found in a survey written by [Singhal96]. Little efforts have been done till now concerning the dynamic verification of timing constraints. Initially RTL has been adapted for specifying and monitoring runtime constraints [Raju92, Mok97). RTL is a powerful formalism, but not all RTL formulas can be monitored efficiently at runtime. Our contract approach lets the designer choose from a set of predefined constraint types, which can afterwards be mapped on RTL formulas (see also the next section on future work). As a result, the contract approach has the advantage that only valid timing constraints can be added to a design.

## 7 Status and Future Work

The concepts discussed in this paper have been implemented in a tool. This tool enables the construction of applications by means of blueprint, instance and scenario models. Currently, we have been looking at timing contracts, and we plan to further work out this type of contract, by adding more subtypes.

Next steps will be the addition of other types of contracts, like memory contracts. Research on non-functional constraint specification and verification for real-time and embedded systems is often only focussed on timing constraints. In contrast, we will also consider other embedded software constraints (like memory usage, power consumption, and so on). We believe that a contract-based approach is valuable for specifying and monitoring these constraints.

In the near future, we will investigate topics like evolution of components and component designs, and live updates[10]. Results from this research will also be included in the tool.

Another important area of related research is on record-replay techniques for distributed and embedded software. Research done here is based on the deterministic replay of executions of multithreaded and distributed software. To enable this, a lot of information is needed about task scheduling, interrupt arrival and handling, synchronization, input/output, and so on. This information is logged by means of a monitor that resides inside the kernel. Afterwards the complete application can be replayed deterministically. We are also doing research on record/replay techniques in the SEESCOA project [Ronsse00]. A very interesting source of information on record/replay is [Thane00], which also discusses embedded software monitoring and testing.

## 8 Conclusion

In this paper we have described how component based applications are built by using basic constructs like components, ports and contracts. Next, different types of models have been discussed. These models allow one to subdivide the application into coherent parts.

[8] Real-Time Logic

[9] Algebra of Communicating Shared Resources

[10] Live updates are updates applied to a running application. This is of particular importance for devices that cannot be switched off.

An explicit construct (a contract) for annotating non-functional constraints has been defined. Contracts have a specification and a runtime meaning: they are used to annotate constraints at design time, and to monitor these at runtime. This runtime monitoring of contracts is valuable if one wants to detect non-functional failures at runtime. Of course, a contract-based approach does not prove the correctness of the application. This requires the use of static verification methods.

Tool support for run-time contract checking has been presented, by transforming contracts to RTL expressions. A probe system generates timestamped event information with only limited intrusion on the application to be monitored. The constraint validation based on the gathered information can be performed on a separate node.

The approach that was presented in this paper is based on some well-known principles: components, constraint specification languages, contracts, and runtime monitoring support. We have combined and extended these basic principles, to enable the construction of embedded software with support for the specification and runtime verification of non-functional constraints.

## 9 References

BEN-ABDALLAH, H., LEUE, S. (1997): Expressing and Analyzing Timing Constraints in Message Sequence Chart Specifications. Technical Report 97-04, Department of electrical and Computer engineering, University of Waterloo, Canada.

BEUGNARD, A., JEZEQUEL, J.M., PLOUZEAU, N., WATKINS, D. (1999): Making components contract aware. *Computer IEEE 32*(7): 38-45.

CLARKE, D., LEE, I. (1997): Automatic Specification-Based Testing of Real-Time Properties, *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems, IEEE*, California, USA.

FROLUND, S., KOISTINEN, J. (1998): *QML: a Language for Quality of Service Specification*. Hewlett-Packard Laboratories, Palo Alto, California, USA.

ITU-MSC (1996): *Recommendation Z.120*. ITU-T Telecommunication Standardization Sector, Geneva.

JAHANIAN, F., MOK, A.K., STUART, D.A. (1988): Formal Specification of Real-Time Systems. Technical Report UTCS-TR-88-25, Department of Computer Sciences, the University of Texas at Austin, USA.

LOYALL, J.P., SCHANTZ, R.E., ZINKY, J.A., BAKKEN, D.E. (1998): Specifying and Measuring Quality of Service in Distributed Object Systems. *IEEE Proceedings of ISORC'98*, Japan.

MEYER, B. (1997): *Object oriented software construction 2nd edition*. Englewood Cliffs NJ, Prentice Hall.

MOK, A.K., LIU, G. (1997): Efficient Run-Time Monitoring of Timing Constraints. *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, Montreal, Canada.

RAJU, S.C.V., RAJKUMAR, R., JAHANIAN, F. (1992): Monitoring Timing Constraints in Distributed Real-Time Systems. *IEEE Real-Time Systems Symposium*, Arizona, USA: 57-67.

RONSSE, M., DE BOSSCHERE, K., CHASSIN DE KERGOMMEAUX, J. (2000): Execution replay and debugging. *Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG2000)*, TUM/IRISA, Munchen, Germany.

SCHMIDT, D.C., LEVINE, D.L., MUNGEE, S. (1998): The design and performance of real-time object request brokers. *Computer Communications*, 21(4): 294-324.

SELIC,B., GULLEKSON, G., WARD, P.T. (1994): *Real-time object oriented modelling*. New York, John Wiley & Sons.

SINGHAL, A. (1997): *Real Time Systems: A Survey*. Computer Science Department, University of Rochester, New York, USA.

SZYPERSKI, C (1998): *Component Software: Beyond object-oriented programming.* New York, Addison-Wesley.

THANE, H. (2000): Monitoring, Testing and Debugging of Distributed Real-Time Systems. Ph.D. Thesis, Mechatronics Laboratory, Department of Machine Design, Royal Institute of Technology, Sweden.

URTING, D., VAN BAELEN, S., HOLVOET, T., BERBERS, Y. (2001): Embedded Software Development: Components and Contracts. *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems,* Anaheim, USA: 685-690, ACTA Press.