Research Report 2/96

A Model for a Flexible and Predictable Object-Oriented Real-Time System

by

Jan Bosch, Peter Molin

Department of
Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HKR-RES—96/2—SE

Sub-band Acoustic Echo Cancellation
by Sven Nordholm, Jörgen Nordberg, Sven Nordebo

ISSN 1103-1581
ISRN HKR-RES—97/7—SE

Printed by Psilander Grafiska, Karlskrona 1997

A Model for a Flexible *and* Predictable Object-Oriented Real-Time System

Jan Bosch & Peter Molin
Department of Computer Science and Business Administration
University of Karlskrona/Ronneby
S-372 25, Ronneby, Sweden

E-mail: Jan.Bosch@ide.hk-r.se, Peter.Molin@ide.hk-r.se
URL: http://www.pt.hk-r.se/[~bosch,~()peter]

Abstract

The requirements on real-time systems are changing. Traditionally, reliability and predictability of, especially hard, real-time systems were the main requirements. This lead to systems that were stand-alone, embedded and static. Future real-time systems, but also current systems, still require reliability and predictability, but also distribution of the real-time system, integration with non real-time systems and the ability to dynamically change the components of the system at runtime. Traditional approaches to real-time system development have difficulties in addressing these additional requirements. Therefore, new ways of constructing real-time systems have to be explored. In this article, we develop a real-time object-oriented model that facilitates the requirements of flexibility without sacrificing the predictability, integration and dynamicity aspects.

1 Introduction

Real-time computing systems play an increasingly important role in our society. The number of computer-based systems in general is increasing constantly and, within that, the percentage of systems incorporating some form of real-time constraints is rising.

Traditionally, real-time systems have been stand-alone, embedded systems that operated in a very well defined static environment, generally controlling a relatively small physical system. The traditional techniques defined for real-time systems are directed towards these systems. Especially hard real-time system approaches are primarily suited for application in small-scale, static environments. In addition to the hard real-time systems, there are a large number of systems that incorporates soft real-time constraints mixed with non real-time parts.

Unfortunately, in the construction of real-time systems, in many cases, the developers do not make use of real-time languages, i.e. languages that allow for the specification of real-time constraints. Rather, they make use of a general programming language, choosing a design that makes it plausible that deadlines will be met, and finally apply different analysis methods and testing determining whether the required time constraints are actually met.

Future real-time systems will distinguish themselves on three aspects from their current counterparts:

- **Integration level**: whereas current real-time systems often are embedded in highly specialised applications, future real-time systems will be integrated with the organisation's information and control systems. This means close interaction between the real-time processes and the non real-time part of the system.

- **Local certifiability**: as a result of the real-time applications getting larger and larger it becomes a necessity that any changes to a well-functioning system can be locally verified, both with respect to functionality and to real-time constraints.

- **Flexibility**: applications, also those with real-time parts, can be added and removed dynamically to and from the system. Also applications might change their real-time requirements, depending on the state of the system.

However, despite these changing requirements, it remains critical to offer predictability for the real-time parts of the system that require this. It is necessary to integrate requirements from both the hard real-time application domain and general information system domains into a new object model. This object model should be powerful enough to facilitate general purpose software engineering, provide expressive constructs for real-time behaviour and tools to achieve predictability.

Most software engineers believe that real-time constraints are only useful in highly technical, critical control applications, e.g. nuclear power stations, that require very strict response times and will result in catastrophes when errors occur. This domain, however, represents a minor percentage of the real-time applications. Based on our experience, we believe that the use of time constraints can be useful in many other domains as well. More and more dependable systems are used in society, and such systems have critical real-time constraints, e.g. alarm systems, medical monitoring systems, automatic traffic control systems, and also some business transaction systems. Hard real-time systems could be defined as systems where a missed deadline implies incorrect functioning of the system. Even systems which do not involve life-critical issues could be viewed as hard real-time systems, e.g. a surveillance system must be able to detect annomalities within a specific time in order to function correctly. Another example from a surveillance and control application, could be the case when an operator controls and moves a remotely located video camera in real-time, while watching the corresponding images. Without hard real-time constraints on the control loop it is impossible to guarantee correct and intended system operation.

In this article, a model is developed that preserves the predictability of a hard real-time system and yet incorporates the aspects of flexibility and integration. The model is based on object-oriented principles and assumes that the system as a whole contains non, soft and hard real-time computation. The real-time object-oriented model provides localisation of the processor resource, in such a way that predictable is improved. In the proposed model, it is possible for objects in the system to guarantee certain behaviour *under the condition that it is guaranteed a certain processor capacity.* Each object has its own *object processor*, i.e. a virtual processor with a guaranteed performance. The physical processor is shared amongst the different object processors. The model assumes an underlying scheduler, capable of dividing the processor capacity in a uniform way. The advantage of our approach where each task is assigned a guaranteed processor performance, is that it facilitates the modification of tasks without affecting the other tasks in the system. The increased flexibility is achieved by a higher overhead cost, but the proposed scheduling method, which may be categorised as dynamic and is based on object processors, has a utilisation upper bound of 100 %, compensating for the overhead. Currently, we are developing simulations to evaluate our model. In the future, we hope to construct a prototype system that illustrates the model.

The remainder of this article is organised as follows. In the next section, the problems of real-time systems that we identified are discussed in more detail. Then, in section 3, two example domains are described in which the need for guaranteed but flexible real-time behaviour is clear requirement from the application domain. Section 4 describes the real-time object model that combines the flexibility and predictability requirements. Subsequently, in section 5, some examples of calculations are presented. Section 6 contains a comparison of our model to the related work. Finally, the article is concluded in section 7.

2 Problem Statement

As mentioned in the introduction, hard real-time systems generally are very rigid and separated from other information systems. The reason for this rigidity is that changes to the real-time system have *global* effects. A change to the system that influences the execution order of tasks in some way, i.e. virtually all changes, may affect the ordering of tasks at synchronisation points and therefore cause missed deadlines. This requires, in principle, that each change somewhere in the system requires a global system analysis, e.g. schedulability analysis, in order to determine whether the change does not cause deadline violations.

The fact that in real-time systems local changes have global effects is more and more experienced as problematic because the domain of systems incorporating real-time constraints is constantly growing. However, the requirements on these future, but also current, real-time systems are such that the rigidness and isolation of traditional hard real-time systems are no longer acceptable. When the traditional hard real-time system approaches are insufficient, the designers of the new real-time systems are forced to use general purpose techniques and tools that do not have any provisions at all for dealing with real-time constraints. The danger of this is that the constructed systems will be much less reliable than acceptable in many situations.

From this, one can conclude that the rigidness of real-time systems is considered a necessity from the real-time systems perspective and unacceptable from the perspective of the application domain. For the discussion in this article, we categorise the problems that we identified with traditional real-time approaches:

- **Global effects of local changes**: As discussed earlier, local changes in real-time systems lead to global effects. This is problematic because, in principle, each local change requires system-wide analysis to determine its effects. Local changes should also have local effects and the analysis of the effects should only be required locally.
- **Lack of flexibility**: The traditional approaches to hard real-time system construction often take the standpoint that all tasks of the system have to be known and frozen at system construction time. The underlying line of reasoning is that if the workload of the system changes dynamically, this might lead to situations where the system might violate its deadlines. As a consequence, the resulting system is required to be static. However, many systems incorporating real-time constraints have as an inherent requirement that the tasks of the system change under run-time. This requires flexibility from the system that cannot be delivered by traditional real-time approaches.
- **Lack of integration**: Traditionally, one can recognise a classification into hard real-time systems and soft real-time systems. Hard real-time systems generally cannot be integrated with soft and non real-time systems. The soft and non real-time tasks may influence the hard real-time tasks in unpredictable ways that do not allow for the guarantees that stand-alone, rigid hard real-time tasks provide. However, todays and especially future

real-time systems cannot accept this lack of integration. The hard real-time system has to be integrated with the other parts of the system.

Traditionally, scheduling has been solved with attributing priorities to the different tasks that constitute the system. The scheduling algorithm have been very simple, execute the task with the highest priority. An improvement was made by allowing for pre-emptive scheduling where a higher priority task interrupts a lower priority task when the higher priority task becomes ready for execution. It has been shown by [Liu & Layland 73] that a set of tasks with static priority is schedulable if it is schedulable with the rate monotonic algorithm. In the same paper a dynamic scheduling algorithm, nearest deadline first, was proposed which allows the system to meet all deadlines with an upper bound of utilisation of 1 [Sha et al. 91]. More and more complex situations has been analyzed and different utilisation levels have been found.

Another problem with static priorities is the priority inversion problem where lower priority tasks may block higher priority task. In fact these problems is of such great importance that it is impossible to use conventional languages and tasking models such as Ada for implementing hard deadline scheduling algorithms, without modifying the semantics of the tasking model [Molin 87].

The earliest deadline first algorithm and most of its succeeders are based on the traditional view that a scheduler runs the higher-priority task, when it is ready for execution. The underlying problem is the shared processor resources that are limited. In order to guarantee the fullfilment of deadlines of the various tasks in the system, the traditional algorithms require the environment to be very rigid.

However, one problem with traditional priority based scheduling is that they do not support local certifiability. If a task of higher priority is defined, or if a task of higher priority prolong its execution time, it may affect the possibility of lower priority tasks to meet their deadlines.

We believe that development of new techniques and models for real-time systems is required, that do not suffer from the problems of rigidness and separation, but still preserves reliability and predictability. We are not the first in identifying these problems. In section 6, we discuss a number of approaches that addressed some of the problems that we described. However, the approach we take to solve the identified problems we believe to be novel.

3 Example Systems

As mentioned in the introduction, the domain of real-time systems, especially hard real-time systems, is changing from stand-alone, embedded and static systems to integrated, open and flexible systems. This change in requirements caused that the traditional hard real-time system techniques do not apply to these systems. In this section, we describe two application domains where we have experienced these changing system requirements.

3.1 Surveillance and Monitoring Systems

Surveillance systems, such as fire alarm systems, intruder alarm systems or access control systems become more and more integrated into each other and also integrated to other information systems of an industry. Such a large complex system consist of a number of hard real-time tasks on different system levels. Examples of such tasks could be controlling special purpose hardware devices, remote video camera control or fire extinguisher control protecting important equipment. At the same time, a multitude of soft real time tasks are required, for example self

monitoring and fault detection. More and more integration is required from such systems, for example a requirement to communicate alarm information to a manufacturing system, inform all office automation users of important alarm information or integrate the access control system with the pay-roll system.

3.2 Integrated Manufacturing Systems

Manufacturing systems have already reached some level of integration. Especially at the level of the production cell, there is a rather high level of integration. The different equipment parts of the production cell communicate frequently and perform tasks together. The equipment in the production cell has hard real-time tasks, such as the control of a robot arm, soft real-time tasks, such as changing a tool in a machine, and non real-time computation, such as the generation of the production figures for the last hour.

The approach taken in todays systems is that the equipment is composed using an *interfacing*, rather than *integrating* approach. Due to that, the differences in underlying hardware and software architectures can be dealt with. However, in the future, manufacturing systems have to become really integrated in their architecture. This is necessary to achieve the flexibility and openness that is required by the new demands on manufacturing systems, such as efficient and cost effective production of small series and individual products.

In such an *integrated* manufacturing system, the production and demands on the system will be constantly changing. This flexibility will also affect the real-time components of the system. Consequently, at run-time, the hard and soft real-time requirements of the various parts of the software system will change dynamically. Traditional techniques for hard real-time systems are unable to deal with the combination of flexibility and hard real-time guarantees. Thus, new approaches are required to provide hard real-time guarantees under the presence of frequently changing real-time requirements.

The flexibility of real-time requirements might, on occasion, indeed lead to the situation that the system is unable to fullfil all the requirements demanded from it. However, this should occur at the time when the hard real-time computation is preallocated at the various parts of the system, rather than when the hard real-time tasks have been started within the system and, for example, a critical task is unable to finish within its bounds. The latter may lead to very unfortunate situations, whereas rejecting a hard real-time task before it is preallocated, in the worst case, will just delay the progress of the manufacturing.

4 System Model

The object model defined in this article is based on the concurrent object-oriented model as, for example, used in Apertos [Yokote & Tokoro 92], DROL [Takashio & Tokoro 92], the layered object model [Bosch 95a, Bosch 95b] and Sina [Aksit et al. 94]. An object consists of a collection of instance variables and a collection of methods. The instance variables are encapsulated by the object and can only be accessed by the methods of the object. The methods of the objects are protected from inappropriate concurrent access through the use of *synchronisation sets*. Each synchronisation set contains a list of methods names that have to run mutually exclusive. Client objects can send messages to the object requesting the execution of a method. A message contains information about its receiver, its sender, the selector, i.e. the name of the requested method and a list of argument objects.

Since the object model operates in a real-time environment, the object can be extended with

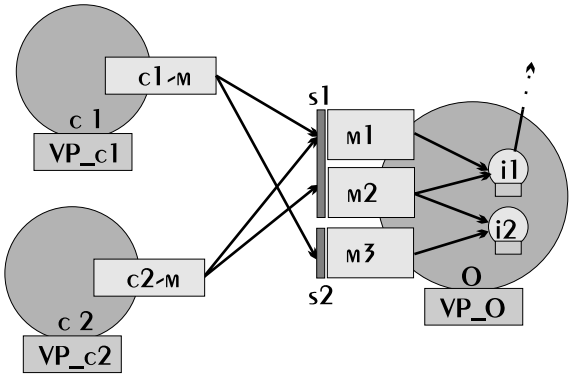

Figure 1: An example object O

real-time constraints. Real-time constraints represent the time interval in which the object has to respond to a message. Real-time constraints can be specified at the server side, but also at the client side. Consequently, the object may receive a message requesting the execution of a certain method of which the deadline is stricter than the interval specified by the server itself. In that case, the object has to adopt the interval specified by the client.

As mentioned in the introduction, real-time systems are supposed to be reliable and predictable in their behaviour. Each server object, therefore, requires knowledge about its clients, such as the methods called by its clients, the time constraints on the calls and the calling frequency. Based on this knowledge, the object can make a decision on whether it will be able to fullfil its requirements. One complicating factor of this decision is that the object can be accessed by multiple client objects simultaneously. In that case, the object is forced to synchronise these requests, thereby delaying the execution of all but one requests. Secondly, the object may need to call other objects in the course of the method execution. Calling these objects also is an unpredictable factor in the method execution.

The remainder of this section is organised as follows. In the next section, the object model is defined. Then, in section 4.2, the notion of the object processor and the scheduler are introduced. Section 4.3 discusses a model for how real-time objects and schedulers can be composed. Finally, in section 4.4, implementation of a basic scheduling is outlined.

4.1 Object Model

An object in our real-time object model is a concurrent entity that can respond to requests in a predictable, real-time manner. The object, in itself, however is not different from the conventional object model, except for its synchronisation constraints and its *object processor*. The synchronisation constraints of an object specify what methods can be executed concurrently and which methods need to be sequentialised. The object processor of an object performs the computation within the object, i.e. the methods of the objects. An object nested in the object has its own object processor that performs its computation.

In figure 1 an example of a real-time object is shown. The object contains two instance variables and two methods. It is called by two clients c_1 and c_2. All objects have their own *object processor* p (discussed in more detail in section 4.2). Object o has two synchronisation sets s_1 and s_2 that synchronise calls by the clients. In this case, method m_1 and m_2 cannot be executed concurrently and m_3 forms an independent synchronisation set.

If one of the clients of the object wants hard real-time guarantees on its messages to the object, it has to pre-allocate execution time at the object. A request for pre-allocated computation consists of the called method m_i, the frequency at which the call maximally can be repeated f and the deadline for each call d. The request is send to the object that can either accept or reject the preallocation request. If it accepts the request, the computation is pre-allocated at the object processor of the object and the client has guaranteed service at the object. The object can also reject the request, in which case the client still can request the service but the object only promises best effort.

An object o can be defined as $o = \{M, I, p, S\}$, where M is the set of methods of o, I is the set of nested objects, p the object processor of o and S the synchronisation constraints. I is defined as $I = \{oi_1, \ldots, oi_p\}$. The structure of each oi is equivalent to o.

M is defined as $M = \{m_1, ..., m_n\}$, where $m = \{l, d, C\}$, with l the number of instructions within o required to execute m, d the server determined deadline on m and C the set of calls to other objects, either internal and external, performed by m. C is defined as $C = \{c_1, \ldots, c_o\}$. $c = \{s, ms, ds\}$, where s is the called object, ms the method called at the server object and ds the worst-case time interval for the call. In the following, we will use w_m to refer to the total waiting time for all calls performed by a method m, i.e. $w_m = \sum_{c \in C_m} ds_c$.

The object processor p is defined as $p = \{e, X, u\}$, where e represents the performance of the object processor in instructions per time unit, X represents the set of preallocated object invocations and u represents the variation of the processor performance. The set of preallocated invocations, X, is defined as $X = \{x_1, \ldots, x_q\}$. An item $x = \{cl, m, f, d\}$, where cl is the client object, m the method of o, f the maximum frequency of a call to m and d is the deadline of the call. The object processor is discussed in more detail in the next section.

The set of synchronisation constraints S is defined as $S = \{s_1, \ldots, s_r\}$. Each element $s \in S$ is defined as $s = \{m_1, \ldots, m_u\}$. The semantics of an element s are that only one method m out of the set specified by s may execute at any point in time. For reasons of simplicity, we assume that each method m is at least synchronised with itself, i.e. a method m cannot be executed concurrently by two threads. A second simplification taken in this paper is that a method m can only appear in one synchronisation set s.

4.2 Object Processor

As described in the previous section, each object o has its own object processor p on which the computation of the object is scheduled and executed. All computation of an object o is performed by the object processor p. The object processor is an abstraction of a physical processor and, since each object has its own object processor, a physical processor implements, in general, several object processors.

The performance of an object processor p is defined as $\#_{instr}/time - unit$. In the situation where n object processors are implemented by a physical processor, the physical processor will divide its computation such that each object processor progresses with its execution at its assigned abstract performance.

Another property of the object processor is the variation in object processor performance, also referred to as the inaccuracy (u). Assuming that p has a performance of e, and that we would like to execute n instructions, the guaranteed time to execute the instructions will be $(n/e) + u$. The inaccuracy u is caused by the fact that each physical processor is serving multiple object processors, as explained below.

Based on the performance of the object processor, the preallocation of computation is calculated and determined. The computation of a method m of object o is expressed in number of instructions. The performance of the object processor therefore is one of the factors influencing the time t required to execute a method, i.e. $t = ((l_m/e_p) + u_p) + w_m$. In the rest of the paper we will use a more accurate waiting time which includes the performance inaccuracy:

$$w_m = \sum_{c \in C_m} (ds_c + u_p).$$

The following calculations of required performance assumes that the object processor can divide its capacity to the required concurrent activities in an optimal way giving each activity a predictable performance. Let $A = \{a_1, ..., a_n\}$ be the set of concurrent activities inside the object, and let $E = \{e_1, ..., e_n\}$ be the corresponding performances.

We assume that invocations will not arrive to the object faster that the deadline ($1/f >= d$), otherwise the object cannot guarantee any deadlines in the long run.

In order to meet the deadlines of all clients of object o, the object processor needs to have a performance sufficient to compute all local computation within the deadlines. Determining the required object processor performance requires some calculation. The first calculation we perform is the required performance for a preallocated invocation set only consisting of methods that do not need to be synchronised within o.

We base our required performance analysis on the theorem that only the *critical instant* needs to be analyzed, and that corresponds to the case where all invocations occur simultaneously [Liu & Layland 73]. We also assume that each invocation is given its required performance capacity. In this case the required object processor performance is:

$$e = \sum_{x \in X} \frac{l_{m_x}}{d_{m_x} - w_{m_x}}$$

In case some methods are synchronised and there exist preallocated executions for these methods, the calculation is slightly more complicated. We first calculate the required performance for each synchronisation constraint set X_s, where $s \in S$ and $X_s = \{x \in X \mid m_x \in s\}$.

Invocations in each synchronisation constraint set must be executed sequentially. If the scheduler picks any sequential execution order, all invocations must be executed within the tightest deadline which is expressed by the following constraint:

$$\sum_{x \in X} (\frac{l_{m_x}}{e_{X_s}} + w_{m_x}) < min(d_{m_x})$$

It is possible to calculate the required performance e_{X_s} as follows:

$$e_{X_s} > \frac{\sum_{x \in X} l_{m_x}}{min(d_{m_x}) - \sum_{x \in X} w_{m_x}}$$

The required object processor capacity can now be calculated as $e = \sum_{s \in S} e_{X_s}$. The calculation of e_{X_s} is more complex since the execution of methods in s requires synchronisation with the other methods. Therefore, we have to calculate the worst-case situation where all requests within a synchronisation set arrive at the same time. That situation requires the largest amount of object processor performance. The calculation of e, based on the values of e_s is merely a

simple addition, since the computations in the various synchronisation sets can be performed in parallel.

If a more elaborate scheduler is used the necessary worst case performance can be decreased. The following example calculations are based on the earliest deadline first algorithm. We define $X' =< x'_1, ..., x'_q >$ as an ordered sequence of invocations, where $d_{x'_i} < d_{x'_{i+1}}$.

All invocation must meet their corresponding deadlines, which can be expressed by the following list of constraints:

$$\forall k \in 1..q \sum_{i=1}^{k} \frac{l_{m_{x'_i}}}{e_{X_s}} + w_{m_{x'_k}} < d_{m_{x'_k}}$$

The required performance ensuring that all deadlines will be met is:

$$e_{X_s} = max(e_k, k \in 1..q \mid e_k = \frac{\sum_{i=1}^{k} l_{m_{x'_i}}}{d_{m_{x'_k}} - w_{m_{x'_k}}})$$

In the first case with "any order" scheduler all invocations must have been executed within the shortest deadline, thus no risk of saturation, but with the "shortest deadline first" scheduling algorithm it is necessary that, in the worst case scenario, all invocations must be completely executed before any new invocations arrive. Otherwise, a saturation condition may occur where invocation keeps coming in faster than it is possible to execute them. We define e_f as the necessary performance avoiding saturation effects:

$$e_f = \sum_{x \in X} \frac{l_{m_x}}{1/f_{m_x} - w_{m_x}}$$

Finally, we can correct the formula for the earliest deadline first performance requirement:

$$e_{X_s} = max(e_f, e_k, k \in 1..q \mid e_k = \frac{\sum_{i=1}^{k} l_{m_{x'_i}}}{d_{m_{x'_k}} - w_{m_{x'_k}}})$$

4.2.1 Scheduler Requirements

The object model described so far must be supported the object scheduler. The requirements are centered around the notion of object processor performance $r = \{e, u\}$ where e is the rate of the processor expressed as #instr/time-unit and u is the inaccuracy. The most important property of the object processor is that it guarantees execution times. If n instructions are executed on the object processor with performance $r = \{e, u\}$ it is guaranteed to be executed within $t_p = n/e + u$ time-units. In this subsection the concept of *activity* will be used to denote either an object processor, a method server or anything which need a concurrent execution thread within an object.

The requirements on the scheduler are, first of all, to provide each activity with a predictable performance $r_a = \{e_a, u_a\}$, and, secondly, to make it possible to define a scheduler as an activity. The last requirement serves as a base for the scheduler hierarchy and can also be expressed that a scheduler is also an activity which executes with a specific performance and must provide its activities with specific (lower of course) performance.

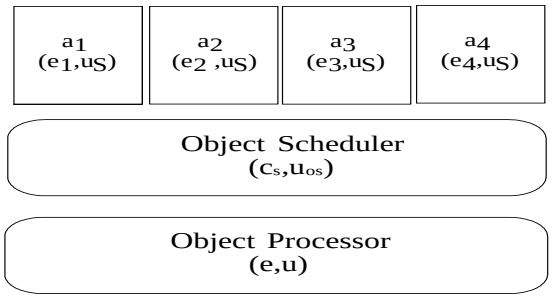

Figure 2: The object scheduler

Figure 2 shows how the object scheduler distributes object processor performance to various object activities. The scheduler is characterised by two properties, the utilisation c which is defined as the sum of maximum available activity performance divided by guaranteed object processor performance and the inaccuracy u_s.

The advantages of this approach is that most scheduler characteristics, such as possible dependencies on the number of activities it handles, are local. A further requirement on the scheduler is that it should be able to run soft deadline activities when there are no more hard deadline activities ready.

4.3 Object and Schedule Composition

The concept of the object processor solves several of the problems identified in section 2. The locality of effects of local changes has been achieved since new computational requirements on an object are dealt with by the object processor. However, the naive use of the object processor also has some disadvantages. One disadvantage is that the number of synchronisation points in one invocation might be rather large because each method called by the originally invoked method needs to synchronise its call. In certain situations this can lead to guarantees that have quite wide bounds, i.e. the guaranteed bound is considerably worse than the actual worst case duration of the invocation.

One approach to address this is discussed in this section. As described in section 4.1, an object can contain nested objects. Each nested object has its own object processor and calls to the nested objects are synchronised by the nested objects. Since the majority of calls by methods are sent to nested objects, the synchronisation of these calls can be an important source of delay. However, in most cases, these inaccuracies can be reduced up to quite some extent by composing the scheduling and synchronisation of an nested object with that of the encapsulating object. As a result of the composition, the object processor of the nested object is removed and its performance and preallocated invocations are integrated with performance and preallocated invocations of the object processor of the encapsulating object. The nested object consequently becomes an passive part of the encapsulating object.

In the next section, we present an algorithm that composes a nested object with its encapsulating object.

4.3.1 Object Composition Algorithm

The object composition algorithm composes an internal object *oi* with its encapsulating object *o*. The composition moves the functionality of the object processor of *oi* to the object processor of *o*, including the performance, the preallocated object invocations on *oi* and part of the synchronisation. The rationale for trying to compose objects is twofold. First, the number of synchronisation points of an invocation on *o* can be reduced by integrating the synchronisation of *oi* with the synchronisation of *o*. This improves the accuracy of the worst-case delay calculation because, in general, several of the synchronisation actions at *oi* could either be avoided or are unnecessary with the appropriate synchronisation of *o*. Secondly, each level of decomposition of the object processor introduces additional overhead or inaccuracy u due to additional context switches, but also because the computation within *o* is unrelated to the computation within *oi*.

The algorithm makes use of the definitions in sections 4.1 and 4.2. Before presenting the algorithm, however, first a few more definitions are required. The set C_{nested} refers to the set of server objects of a method m of object *o* that are nested within the *o*. C_{nested} is defined as $C_{nested} = \{c \in C \mid c \in I_o\}$. Since some of the server objects of m also may be located outside *o*, the set $C_{ext} = C - C_{nested}$ refers to those server objects.

We define the function $senders(m)$ which returns all methods m_{sender} that send a message to m during their execution. Similarly, we defined the function $senders(s)$ which returns all synchronisation sets s_{sender} that contain methods that call methods in the synchronisation set s. The function $object_of(m)$ returns the object *o* that contains the method m and the function $synchSet(x)$ returns the synchronisation set containing the method related to invocation x.

As mentioned, the object composition algorithm integrates the *oi* into its encapsulating object *o*. The algorithm takes one synchronisation set s_{oi} per iteration and integrates s_{oi} in *o* by applying on of two possible approaches, i.e. *inlining* or *synchronisation delay minimisation*. In the following, first the composition algorithm is presented and subsequently the algorithms for inlining and synchronisation delay minimisation.

```
Compose( o, oi ∈ I_o)
    p_o = p_o + p_oi, i.e. e_po = e_po + e_poi
    For all s_oi ∈ S_oi do
        X_sio = {x_io ∈ X_io | m_xio}
        If for all x^1_sio, x^2_sio ∈ X_io, cl_x^1_sio = cl_x^2_sio ∧
            synchSet(sender(x^1_sio)) = synchSet(sender(x^1_sio)) Then
            // All senders of s_io are in the same synchronisation set
            // so inlining can take place in a neutral manner
            Inline( o, io, s_io );
        Else
            MinimizeSynchDelay( o, io, s_io );
        End If;
    End For;
    Do PerformanceRecalculations();
End;
```

The algorithm *Compose* recognises two different situations. First, the situation where all clients of a synchronisation set s_{io} are part of one synchronisation set s_o at *o*. Second, the situation where only methods of *o* call methods in s_{io}. Below, the algorithms *Inline* and *MinimizeSynchDelay* are specified.

```
Inline( o, io, s_io )
    s = synchSet(senders(s_io));
```

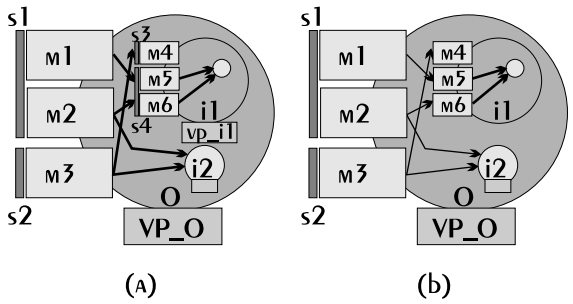

Figure 3: Composing an internal object i_1 with its encapsulating object o

$s = s \cup s_{io}$;
for all methods $m \in senders(s_{io})$ do
$C_m = C_m - \{c \in C_m \mid s_c = oi \wedge ms_c = mx_{oi}\} + C_{m_{oi}}$;
$l_m = l_m + l_{m_{x_{io}}}$;
$d_m = d_m + d_{m_{x_{io}}}$;
End For;
End;

The *Inline* algorithm inlines the methods in s_{io} in the calling methods in o. Since the complete functionality is integrated, the synchronisation set s_{io} and the preallocated invocations $X_{s_{io}}$ do not need to be integrated in the respective sets at o.

MinimizeSynchDelay(o, io, s_{io})
$X_{s_{oi}} = \{x \in X_{oi} \mid m_x \in s_{oi}\}$
While $x^1_{oi}, x^2_{oi} \in X_{s_{oi}}$ exist such that $cl_{x^1_{oi}} = cl_{x^2_{oi}} \wedge synchSet(x^1_{oi}) = synchSet(x^2_{oi}) \wedge (x^1_{oi} \vee x^2_{oi} untagged)$ Then
decrease = $\min(l_{x^1_{oi}} + d_{x^1_{oi}}, l_{x^2_{oi}} + d_{x^2_{oi}})$;
For all $x_{s_{oi}} \in X_{s_{oi}}$ Do
$c = c_{m_{sender}}(x_{s_{oi}})$;
ds_c = ds_c - decrease;
End For;
Tag x^1_{oi}, x^2_{oi};
End While;
$S_o = S_o \cup s_{oi}$;
$X_o = X_o \cup X_{s_{io}}$
End;

The *MinimizeSynchDelay* algorithm searches for pairs of preallocated callers that are in the same synchronisation set, but are both used in the worst case delay calculations. Due to this, the worst case delay is a less accurate estimate and the algorithm removes the unnecessary precalculated delays.

In figure 3, the composition of an internal object i_1 with its encapsulating object o is shown. After analysing the clients of the methods m_4, m_5 and m_6, it shows that both synchronisation sets of i_1, i.e. s_3 and s_4, are called by methods from the same synchronisation sets in o, i.e. s_1 and s_2, respectively. Because of this, the methods of i_1 can safely be inlined without synchronisation in the methods m_1, m_2 and m_3 of object o. In figure 3(a) the situation before the composition is shown. Here, i_1 still has its object processor vp_{i_1} and synchronisation sets s_3 and s_4. In figure 3(b), the situation after the composition is shown. Both the object processor and the synchronisation sets of i_1 have been integrated in o.

4.4 Implementation outline of a predictable real-time scheduler

In this subsection an example scheduler is presented with the objective to demonstrate the possibility to implement such a scheduler. We start by defining the activity class:

```
class Activity

private:
      state = { Active, Suspended };
      performance = (e,u);
      ....
end
```

The state *Active* corresponds to the traditional scheduling states *Executing* or *Ready*. The state denotes that the activity has a certain task to perform and is running on an virtual processor with a specific performance. The state *Suspended* indicates that the object-activity is waiting for external (from its point of view) events to happen. A specific kind of activity is the scheduler, which at this preliminary stage can be described as an activity containing a list of *active* activities. A scheduler also contains methods managing that list.

```
class Scheduler is derived Activity
private
      Active-list list of Activity ;
      Activate ( Activity) ;
      Suspend (Activity ) ;
end
```

The idea is to assign a predictable performance to each activity. The following two examples of simple schedulers, explain the underlying ideas. First, we will show how a simple round-robin scheduler can divide its performance uniformly to a fixed number of activities each given a performance of e_a, u_a.

We define the performance of the physical processor to be v, the number of activities to be n, and that it takes x time-units to perform a context switch, and finally that a context switch occur every m time-units. One cycle where each activity may execute once takes $n * (m + x)$ time-units. Each activity is executing m time-units at a rate of v instructions per time-unit i.e. $m * v$ instructions. Based on this, the performance of an activity is $e_a = (m * v)/(n * (m + x))$. The inaccuracy u_a is one complete period of $n * (m + c)$ time units, i.e. $u_a = n * (m + x)$.

Secondly, a more elaborate scheduler can be defined, where the activities can be assigned different capacity. We assume that each activity is supposed to execute at a rate of e_a instructions per time-unit. We define the capacity of the physical processor to be v, the number of activities to be n, and that it takes x time-units to perform a context switch, and finally that the scheduler can execute an activity for a specific time interval. Furthermore, we define the scheduling cycle, c as the time interval within which all activities should execute its amount of time. Each activity should execute $x_a = (e_a/v) * c$ within one scheduling cycle. The guaranteed performance for the activity will be $\{e_a, u_a\}$ where $u_a = c$. The scheduling overhead, such as context switching, can be modelled as a utilisation limit ul, where the sum of object-activity performance is less than the total available capacity.

$$ul = (c - n * (x + 2 * \delta))/c.$$

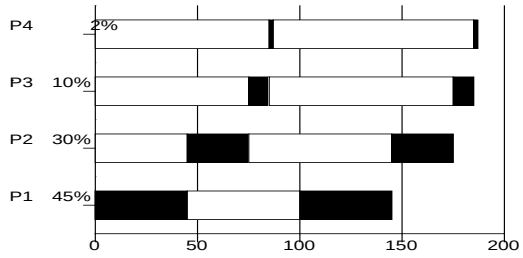

Figure 4: An example scheduler

In figure 4 an example of a schedule is shown during two scheduling cycles for four activities. Individual activity processing performances are expressed in percentage for simplicity.

After these two examples we will return to the description of the scheduler implementation outline, which will be based on the same principles as the second example scheduler. Each scheduler defines its cycle time and during that time, each active activity will be executed once with a variable time giving the activity its required performance. The principle of executing each activity only once during the cycle period makes it possible to include timer inaccuracies as part of context switch execution time simplifying the following computations. Context switch overhead will be modelled as certain number of instructions, and the required timer will be replaced by a mechanism producing an interrupt after a specific number of executed instructions.

Every scheduler relies on its own actual performance in order to perform the necessary schedules. The scheduler must have access to an underlying scheduler providing a one-shot timer mechanism. It is necessary to define a timeout method which is called by the underlying scheduler when the requested time has elapsed, or, more accurately, when the requested number of instructions has been executed. The context switch requires each object to be able to save its execution environment state (registers, stack-pointer etc..) and to restore it later, when dispatching. The extended scheduler description looks in this stage as follows:

```
class Activity
        ...
        Execution-environment-state ;
        Slice ; // Number of instructions to execute
        ...
        Dispatch()
                Execution-evironment-state.Restore;
        end
        Save()
                Execution-environment-state.Save;
        end

class Scheduler is derived Activity
        ...
        Base-scheduler Scheduler ;
        Cycle-time ;
        ...
        Set-timer (instructions); // synchronous

        Expired-time (Object-Activity Current) // asynchronous
```

```
            Current.Save();
            Current = next in Active-list;

            Base-scheduler.Set-timer(Current.slice);
            Current.Dispatch;
        end
end
```

Some computations can be made at this stage in the presentation of the scheduler. We assume that the Base-scheduler has a performance of $\{e_b, u_b\}$ and that each object-activity is supposed to execute at a rate of e_{o_1} instructions per time-unit. The maximum number of object-activities in the list is assumed to be n, and that it takes x instructions to perform a context switch. Furthermore, we define the scheduling cycle to be c time-units. For each activity we define its slice s as the number of instruction to execute within each cycle, i.e. $s = e_{o_i} * c$. These instructions will be executed during $(s/e_b) + u_b$ time-units, but since we must accept a worst case delay on the scheduler level of c time-units, the actual guaranteed time is $((s/e_b) + u_b) + c$. The guaranteed performance for the ctivity will be e_{o_i}, u_{o_i} where $u_o = c + u_b$.

The scheduling overhead, such as context switching, can be modelled as a utilisation limit ul, where the sum of object-activity performance is less than the total available capacity. $ul = (c - (n * x/e_b) - u_b)/c$.

The next step in the description is to outline how the one-shot timer may be implemented. The scheduler has two main responsibilities, the first is to request the underlying scheduler to asynchronously call a timer-callback routine when the timer has expired, or more precisely, the requested number of instructions has been executed. On the other hand, scheduler can make a context switch without notifying its scheduled activities.

The second responsibility is to implement a corresponding timer routine for its scheduled activities. In that case the scheduler needs to be alerted when the required number of instructions has been executed, and interrupt the execution of the scheduled activity, calling the timer expiration method in the activity. It is evident that the scheduler needs to know the actual number of executed instructions.

When an activity is dispatched it shall execute either until its slice timeout is expired, or until an activity timeout has expired. We need to know the remaining number of instructions from the slice whenever a dispatch is made. The next time the scheduler is invoked, either by a new call from one of the activities or when the expiration routine is called it is necessary to know the number of instructions executed since dispatch time. The following extended model incorporate these concepts.

```
class Activity is
    ...
    Slice , Slice-left : instructions;
    Timeout, Timeout-left : instructions;
    Counter : instructions;
end

class Scheduler is derived of Activity
    Start-counting()
        Enter();
        Current.Counter = 0 ;
        Prepare-dispatch();
    end
```

```
    Get-counting()
        Enter();
        return Current.counter;
        Dispatch();
    end

    Set-timer (nr-of-instr)
        Current.Save();
        Timeout, Timeout-left = nr-of-instr;
        Prepare-dispatch();
        Current.Restore();
    end;

    Prepare-dispatch()
        if Current.Slice-left = 0 then Current.Slice-left = Current.Slice;
        Base-Scheduler.Set-timer(min(Current.Slice-left,Timeout-left));
        Base-Scheduler.Start-counting();
    end

    Enter()
        Duration = Base-Scheduler.Stop-counting();
        Current.counter += Duration;
        Timeout-left -= Duration;
        Slice-left -= Duration;
    end
    ...
end
```

Details, such as guaranteeing mutual exclusion between scheduler methods, and testing if no time-out is required, deciding of how to count instructions executed in the scheduling operations, have been left out of the outline.

Finally, it must be mentioned how the scheduler closest to the physical processor will be implemented. The number of instructions must be translated to a specific time interval so that a normal timer can be used.

5 Examples

A simple example consists of two clients (C_1 and C_2) each requesting service from one server object s_o, calling methods m_1 and m_2 respectively. The method m_1 requires an execution of $l_1 = 200000$ instructions and m_2 requires $l_2 = 300000$ instructions. C_1 calls the server with a maximum frequency of once every 4000 time-units and C_2 calls m_2 with a maximum frequency of once every 3000 time-units. The first client has a deadline of 2500 time-units and the second client has a deadline of 2000 time-units after a server call. The problem is to calculate the required performance of the object processor for server S. The object processor needs to share its capacity between invocations $x_1 = \{C_1, m_1, 1/4000, 3000\}$ and $x_2 = \{C_2, m_2, 1/3000, 2000\}$. In this example we assume that both methods can be executed concurrently, thus the synchronisation set is empty.

First we do the calculations without considering the inaccuracy or the maximum utilisation due to scheduler overhead. The required performance for invocation 1 is $e_{x_1} = \frac{200000}{2500}$ and the required performance for the second invocation 2 is $e_{x_2} = \frac{300000}{2000}$ making the required performance for the object processor $e_{S_o} = 80 + 150 = 230$.

Taking inaccuracies and scheduler overhead into consideration, makes it necessary to define the inaccuracy of the provided performance as u_1 and the introduced inaccuracy of the object scheduler as u_2 and the utilisation level caused by scheduler overhead as c. We assume that $u_1 = u_2 = 100$ time-units and that $c = 0.9$. In that case we have the following equation:

$$e_{S_o} = \frac{1}{0.9}(\frac{200000}{2500 - (100 + 100)} + \frac{300000}{2000 - (100 + 100)}) = 257$$

We will next investigate the possibilities to execute the invocation sequentially and still meet all deadlines. Since there are only two invocations we can simply calculate the necessary performance making it possible to execute both invocations within the shortest deadline, namely 2000. In this situation there is no introduced overhead but we have to consider the provided performance inaccuracy $u_1 = 100$.

$$e_{S_o} = \frac{200000 + 300000}{2000 - (100 + 100)} = 277$$

If the two methods belong to the same synchronisation set it is necessary to require sequential execution, making the second computation a requirement and not an option.

If we suppose that both the methods m_1 and m_2 calls an external object with a maximum wait-time of 500 time-units the required performance, assuming concurrent execution will be:

$$e_{S_o} = \frac{1}{0.9}(\frac{200000}{2500 - 500 - (100 + 100)} + \frac{300000}{2000 - 500 - (100 + 100)}) = 342$$

The requirement for sequential execution is:

$$e_{S_o} = (\frac{200000 + 300000}{2000 - (500 + 500) - (100 + 100)}) = 625$$

6 Evaluation and Related Work

The existing work on real-time systems is very large and comparing the proposed object model to all research in the field is difficult. However, one can categorise the related work into three categories, i.e. *language models*, *operating systems* and *scheduling*. In the following, we discuss each category in more detail.

6.1 Language Models

Several real-time object-oriented languages have been defined. One category of languages are the languages based on C++ [Stroustrup 91] and extended with constructs for real-time, concurrency and synchronisation. Examples of these languages are DROL [Takashio & Tokoro 92], FLEX [Lin et al. 91], Maruti Programming Language MPL [Nirkhe et al. 90, Saksena et al. 95] and RTC++ [Ishikawa et al. 90]. A second category of real-time object-oriented languages is based on Smalltalk-80 [Goldberg & Robson 89]. However, this category currently only has one instance, i.e. RealTimeTalk [Brorsson et al. 92, Eriksson et al. 95]. The third category makes use of reflective techniques for representing, among others, real-time and concurrency constraints. Two instances of this category are Sina [Aksit et al. 94] and the layered object

model (LAYOM) [Bosch 95a, Bosch 95b], but DROL, although it is based on C++, makes use of meta-objects and therefore actually is an instance of both categories.

In [Aksit et al. 94], it was identified that the languages in the first and second category suffer from so-called real-time specification inheritance anomalies, i.e. it is often difficult to reuse classes containing real-time constraints. These inheritance anomalies have been avoided by Sina and LAYOM.

When comparing these real-time object-oriented languages with the real-time model proposed in this paper, one can conclude that the proposed model is well suited to be used as the underlying computation model of these programming languages. Thus, the real-time object model can be combined with one of these languages to provide a computation model that provides both the predictability of a hard real-time system and the flexibility of a general system.

6.2 Operating Systems

During the last years, a number of real-time operating systems have been developed. Examples of these operating systems are Spring [Ramamritham & Stankovic 91], CHAOS [Gheith & Schwan 93] and Apertos [Yokote & Tokoro 92]. Spring and CHAOS are based on traditional operating systems, whereas Apertos is based on reflective object-oriented principles. Objects in the Apertos operating system can have meta-object associated with them that provide the functionality of traditional operating systems, such as memory management and communication.

Stankovic and Ramamritham [Stankovic & Ramamritham 95] discussed the functionality of the existing real-time operating systems and identified several problems, among others, the rigid interface to scheduling, fault tolerance and the lack of flexibility in combination with predictable real-time behaviour. Due to this, the design, maintenance, analysis and understandability of real-time systems is difficult. Although the object model proposed in this article does not solve all problems identified by Stankovic and Ramamritham, we consider the proposed model an important contribution to a solution of the described problems. The analysis of the real-time system is simplified considerably due to the local verifiability. For the same reason, design and maintenance is simplified because one can take a modular approach and integrate the various system components rather than having to design and maintain the system as a whole.

6.3 Scheduling

Scheduling of real-time tasks has been an active field of research for many years, as is also illustrated by the number of publications in the field. For an overview of scheduling of real-time tasks we refer to [Bettati et al. 91, Halang & Stoyenko 91, Allen 83, Stoyenko & Marlowe 91, Dasarathy 85, Cheng et al. 88, Schwan & Zhou 92]. All these scheduling approaches, whether they take a dynamic or a static approach, take a global perspective at the level of the physical processor. Thus, all tasks that have to be executed on the physical processor are scheduled by a one scheduler.

As we discussed in the introduction, the global scheduling approach has the unfortunate consequence that any change to one of the tasks scheduled by the scheduler in general affects the other tasks of the scheduler. This removes the possibility of local verifiability which was one of our requirements, next to predictability and flexibility. The hierarchical (or recursive) approach to scheduling provides a solution to all three requirements.

7 Conclusion and Future Work

The domain of real-time systems is changing from the traditional stand-alone, static, embedded systems to systems that are integrated in larger IT systems and that can be changed dynamically at run-time. Traditional techniques for the construction of real-time systems are unable to deal with the requirements of this new category of real-time systems because these techniques tend to take a global system approach. In a global system approach all aspects of the complete system are used as the input for an algorithm that generates, for example, a schedule for the complete system. Although this approach previously might have been considered necessary in order to be able to guarantee the temporal behaviour of the system, the problem with a global system approach is that local changes have global effects. Thus, a change in the requirements of one task might affect the complete system, due to, for example, changed arrival of tasks at a synchronisation point. To solve this problem, *local certifiability* of parts of the system is required. Thus, if a collection of modules is individually certified, the composition of the modules is, by definition, certified.

In this article, we have proposed a model that uniformly combines the predictability of hard real-time systems with the flexibility of non real-time systems. The model is based on concurrent object-oriented concepts and each object in the real-time object-oriented system consists of *methods*, *internal objects*, *synchronisation sets* and an *object processor*. Clients of an object can preallocate invocations at the object. Preallocated invocations are guaranteed by the object to return within the agreed deadline. The object processor is a virtual processor that executes at a guaranteed performance level and based on this performance level, the object processor can calculate the execution time of its methods and guarantee deadlines. The object processors form a processor hierarchy with, at the top of the hierarchy, a number of physical processors that divide performance among the object processors at the level below. These object processors, in turn, divide their performance to the object processors at the next level. Using this model each object in the system can be guaranteed an amount of processor performance, allowing the object to guarantee certain behaviour to its clients.

The real-time object-oriented model that we proposed solves the problems identified in section 2. Due to the object processors, local changes have local effects, i.e. on the local object processor. Other objects are not affected by the changes. The model facilitates flexibility and dynamic changes of the system, because new clients can preallocate invocations at objects, even at run-time. Preallocated invocations can also be removed removed at a later stage. The model integrates hard, soft and non real-time computation in a uniform model. Objects can always invoke on each other, but only preallocated invocations are guaranteed.

However, we are aware of the fact that the model we proposed still needs considerable work. Currently, we are preparing simulations on the model that will give us information on issues such as processor utilisation and accuracy of worst case deadlines. After these simulations, we plan at implementing the model on an actual platform.

References

[Aksit et al. 94] M. Aksit, J. Bosch, W. vd Sterren, L. Bergmans, "Real-time specification inheritance anomalies and real-time filters," *ECOOP '94,* pp. 386-407, 1994.

[Allen 83] J. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM,* Vol. 26(110), pp. 832-843, ACM,

1983.

[Bettati et al. 91] R. Bettati, D. Gillies, C.C. Han, K.J. Lin, C.L. Liu, J.W.S. Liu, W.K. Shih, "Recent Results in Real-Time Scheduling," in [Tilborg & Koob 91a], pp. 91-127, 1995.

[Bosch 95a] J. Bosch, "Abstract Object State in Real-Time Control," *Proceedings of the IFAC Workshop on Architectures and Algorithms in Real-Time Control* (AARTC'95), May 1995.

[Bosch 95b] J. Bosch, "Layered Object Model - investigating paradigm extensibility," *Ph.D dissertation,* Department of Computer Science, Lund University, November 1995.

[Brorsson et al. 92] E. Brorsson, C. Eriksson, J. Gustafsson, "RealTimeTalk: An Object-Oriented Language for Hard Real-Time Systems," *Proceedings of IFAC International Workshop on Real-Time Programming,* 1992.

[Cheng et al. 88] S.C. Cheng, J.A. Stankovic, K. Ramamritham, "Scheduling algorithms for Hard Real-Time Systems" (eds. Stankovic, Ramamritham), *IEEE Computer Society,* pp. 150-173, 1988.

[Dasarathy 85] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods for Validating Them," *IEEE Transactions on Software Engineering,* 11(1), pp. 80-86, IEEE, 1985.

[Eriksson et al. 95] C. Eriksson, K-L Lundbäck, H. Lawson, "A Real-Time Kernel Integrated with an Off-Line Scheduler," *AARTC '95,* pp. 169-175, 1995.

[Gheith & Schwan 93] A. Gheith, K. Schwan, "CHAOS: Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-Time Multiprocessor Applications," *ACM Transactions on Computer Systems,* Vol. 11, No. 1, pp. 33-72, February 1993.

[Goldberg & Robson 89] A. Goldberg, D. Robson, *Smalltalk-80 - The Language,* Addison-Wesley, 1989.

[Halang & Stoyenko 91] W.A. Halang, A.D. Stoyenko, *Constructing Predictable Real-Time Systems,* Kluwer Academic Publishers, Dordrecht-Hingham, 1991.

[Ishikawa et al. 90] Y. Ishikawa, H. Tokuda, C.W. Clifford, *Object-Oriented Real-Time Language Design,* Carnegie Mellon University, USA, 1990.

[Lin et al. 91] K.J. Lin, J.W.S. Liu, K.B. Kenny, S. Natarajan, "FLEX: A Language for Programming Flexible Real-Time Systems," in [Tilborg & Koob 91b], pp. 251-290, 1991.

[Liu & Layland 73] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM,* Vol. 20, pp. 46-61, 1973.

[Molin 87] P. Molin, "Implementation of an Ada Run-Time System Supporting Hard Deadline Scheduling", *2nd International Workshop on Real-time Ada Issues* , University of York, 1987.

[Nirkhe et al. 90] V.M. Nirkhe, S.K. Tripathi, A.K. Agrawal, "Language Support for the Maruti Real-Time System," *Proc. 1990 Real-Time Systems Symposium,* pp. 257-266, IEEE Computer Society Press, 1990.

[Ramamritham & Stankovic 91] K. Ramamritham, J.A. Stankovic, "Scheduling Strategies Adopted in Spring: An Overview," in [Tilborg & Koob 91a], pp. 277-305, 1991.

[Saksena et al. 95] M. Saksena, J. da Silva, A. Agrawala, "Design and Implementation of Maruti-II," in [Son 95].

[Sha et al. 91] L. Sha, M. H. Klein, J. B. Goodenough, "Rate Monotonic Analysis for Real-Time Systems," in [Tilborg & Koob 91a], pp. 129-155, 1991.

[Schwan & Zhou 92] K. Schwan, H. Zhou, "Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads," *IEEE Transactions on Software Engineering,* 18(8), pp. 736-748, 1992.

[Son 95] S.H. Son, *Advances in Real-Time Systems,* Prentice Hall, 1995.

[Stankovic & Ramamritham 88] J.A. Stankovic, K. Ramamritham, "Tutorial Hard Real-Time Systems," *IEEE Computer Society Press,* 1988.

[Stankovic & Ramamritham 95] J.A. Stankovic, K. Ramaritham, "A Reflective Architecture for Real-Time Operating Systems," in [Son 95].

[Stoyenko & Marlowe 91] A.D. Stoyenko, T.J. Marlowe, "Polynomial-Time Transformations and Schedulability Analysis of Parallel Real-Time Programs with Restricted Resource Contention," *Research Report: CIS-91-18,* New Jersey Institute of Technology.

[Stroustrup 91] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1991.

[Takashio & Tokoro 92] K. Takashio, M. Tokoro, "DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems," *Proceedings of OOPSLA'92,* pp. 276-294, ACM Press, 1992.

[Tilborg & Koob 91a] A.M. van Tilborg, G.M. Koob, *Foundations of Real-Time Computing - Scheduling and Resource Management,* Kluwer Academic Publishers, 1991.

[Tilborg & Koob 91b] A.M. van Tilborg, G.M. Koob, *Foundations of Real-Time Computing - Formal Specifications and Methods,* Kluwer Academic Publishers, 1991.

[Yokote & Tokoro 92] Y. Yokote, M. Tokoro, "The Apertos Reflective Operating System: The Concept and its Implementation," *Proceedings OOPSLA'92,* pp.414-434, 1992.