Advanced Software Technology

INFORMATION INTERFACE TO THE SCHEDULING LEVEL OF A HARD REAL-TIME SYSTEM DESIGN MODEL

Jens Larsson

t91jla@docs.uu.se

Technical Report
ASTEC 97/02
DoCS 97/81

Department of Computer Systems
Uppsala University

February 20, 1997

DoCS

ABSTRACT

This report shows how a general information interface to the scheduling level of a design model of a hard real-time system can be derived. A scheduling approach independent interface is proposed by first determining what information is needed for fixed priority scheduling analysis and then comparing this information with the information needed for static cyclic scheduling. The report also presents a way of handling inter process relations by introducing an object called a communicator. A communicator can be asynchronous or synchronous, as well as local or distributed.

PREFACE

This report is performed in connection to my Master of Science thesis work in engineering physics. The work was performed at the Department of Computer Systems, at Uppsala University, under supervision of Hans Hansson.

The project is performed together with Mecel AB within the ASTEC (Advanced Software TEChnology) centre of excellence supported by NUTEK (Swedish National Board for Technical and Industrial Development). Support has also been provided by Volvo Science Foundation.

This report was written in Microsoft Word 6.0 under Windows 3.11.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Hans Hansson for proof-reading this document. I would also like to thank Sven Larsson, Magnus Lindahl and Mikael Strömberg at Mecel in Göteborg for their work in the project of developing a design model for hard real-time systems.

CONTENTS

1. INTRODUCTION 1

2. MODEL DESCRIPTION 2

2.1 DESIGN MODEL 2
2.1.1 Allocation and Scheduling 3

3. THE SCHEDULING INTERFACE 5

3.1 PROCESSES 5
3.1.1 Type of Processes 6
3.2 FIXED PRIORITY SCHEDULING 6
3.2.1 Basic Processes 6
3.2.2 Processes Including Jitter 7
3.2.3 Processes with Bursty Behaviour 7
3.2.4 Processes with Time Offsets 8
3.2.5 Processes Locking Semaphores 8
3.2.6 Scheduler Overhead 9
3.2.7 Summary of Needed Parameters 9
3.3 INTER PROCESS RELATIONS 10
3.3.1 Synchronous Vs Asynchronous Relations 11
3.3.2 Local Vs Distributed Relations 11
3.3.3 Local Asynchronous Relations 12
3.3.4 Local Synchronous Relations 12
3.3.5 Distributed Synchronous Relations 14
3.3.6 Distributed Asynchronous Relations 14
3.3.7 Branched Local Asynchronous Relations 15
3.3.8 Branched Local Synchronous Relations 16
3.3.9 Branched Distributed Synchronous Relations 17
3.3.10 Branched Distributed Asynchronous Relations 18
3.4 STATIC CYCLIC SCHEDULING 18
3.4.1 BASEMENT 19
3.5 A SCHEDULING ALGORITHM INDEPENDENT INTERFACE 19
3.5.1 Comparison 20

4. SUMMARY AND CONCLUSIONS 23

5. REFERENCES 24

1. INTRODUCTION

Real-time computing systems are becoming more and more important in the technological infrastructure of industrialised nations. Modern telecommunication systems, factories, spacecraft, aircraft and airports cannot operate without them. Real-time computing systems are also introduced in new fields, such as the automotive industry, to optimise performance and costs. For these systems to behave correctly it is necessary for different applications in the system to share resources (e.g. CPUs and networks) in a predictable and cost efficient way. It would therefore be good for the system designers to have a design model that includes methods for verification and testing of the systems. One part of the design work is to predict whether process deadlines will be met before the costly implementation phase of a project begins. For this reason different scheduling methods and theories have been developed, such as static cyclic scheduling, generalised rate monotonic scheduling and fixed priority scheduling. This part belongs to the scheduling level of a proposed design model. To support the work in the scheduling level of the design model it would be good to have a general information interface of the scheduling level that defines what information is needed. The information interface should be general in the meaning that it is independent of the used scheduling approach. The goal of this report is to derive such an interface.

Section 2 outlines a proposed design model that can be used when designing a hard real-time system. Section 3 uses Fixed Priority Scheduling analysis to derive the needed information in different situations. A communicator object is introduced which supports handling of inter process relations in a general and more abstract way. Finally, by comparing Static Cyclic Scheduling with Fixed Priority Scheduling a general information interface is presented.

2. MODEL DESCRIPTION

In the development of hard real-time systems the method of trial and error is often used. Directly from functional requirements the coding work starts. This method of working seldom supports verifications of the system functions - only testing can be used. By testing a system it can be shown that desired functions exist but it cannot be shown that undesired functions do not exist. For this reason we need verification possibilities.

Within ASTEC* the project ART† is aiming at finding a practical working design model of hard real-time systems. The design model should support verification at each level of the design. The work within the project is done by DoCS - Department of Computer Systems at Uppsala University together with Mecel AB, Göteborg. A case study is used with the purpose of developing and testing the design model. The case study is an application example called Powertrain Management, which is an example of a hard real-time system embedded in a car. With four nodes and one communication bus it is suggested how functions such as ABS-brakes, anti spin and an automatic clutch free gearbox could be implemented. The system specification is found in [1].

The design model is under construction, so the current model is just an outline which may be changed or at least refined during the work. The plan of the forth going work is to bring more substance into the outline of the design model and this document is a part of that work. The outline of the design model is described in Section 2.1 below.

2.1 DESIGN MODEL

The outline of the design model is shown in Figure 2-1. The first step is the analysis of the problem. The work in this step is to collect information about the problem and also to formalise the requirements.

* ASTEC - Advanced Software TEChnology competence centre.

† ART - ASTEC Real-Time.

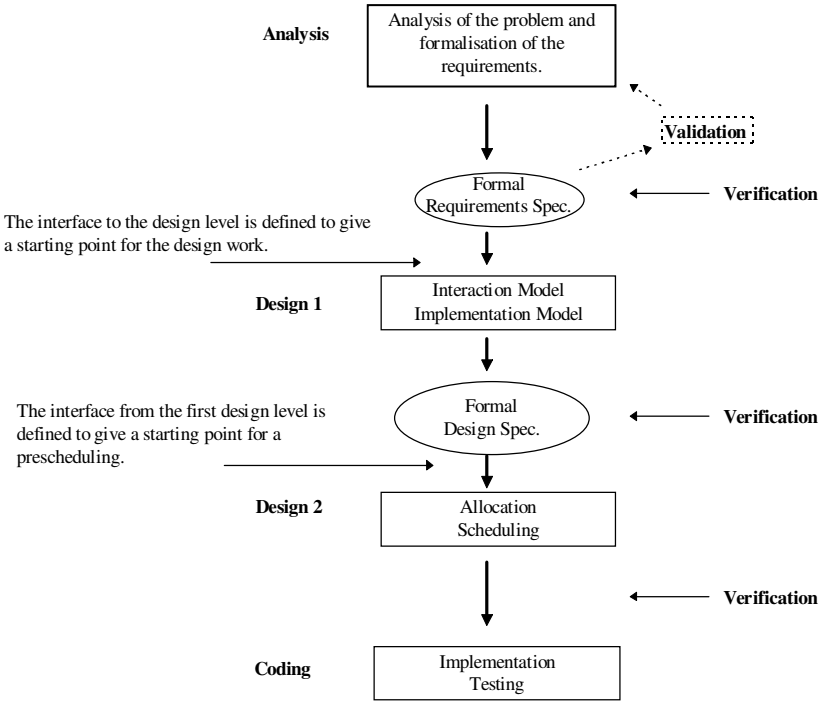

Figure 2-1 The outline of the design model.

The second step is the first design phase where a general design of the system is created. The system is here divided into sub-functions. Through allocation and scheduling in the second design phase it is determined what resources are needed to realise all sub-functions in the system, i.e. the architecture of the system is designed. In the third step which is the second design phase the idea is to optimise the utilisation of the hardware - we want to use as inexpensive components as possible without any lack of performance. When the second design phase has been iterated some times it is decided what hardware, scheduling approach, operating system and so on to use in the final system. In the coding phase of the design model all sub-functions of the system are implemented and also tested.

After each phase passed in the design it is desirable to be able to do a verification of the system to show that the functional requirements still are met. In ART we evaluate the suitability of using the formal verification tool UPPAAL [2, 3] for verification. The verification after the second design phase is done by scheduling analysis.

Note that the design model is under construction, so forthcoming work could change the structure of the model presented above. At least it will produce a more detailed description of the model.

2.1.1 ALLOCATION AND SCHEDULING

The second design phase in the design model described above includes both allocation and scheduling. We divide this phase into two sub-levels which we call the allocation and the scheduling level. The allocation problem is today often solved manually by the

system designer, but there is also a possibility to use an allocation algorithm such as the simulated annealing based algorithm used in [4].

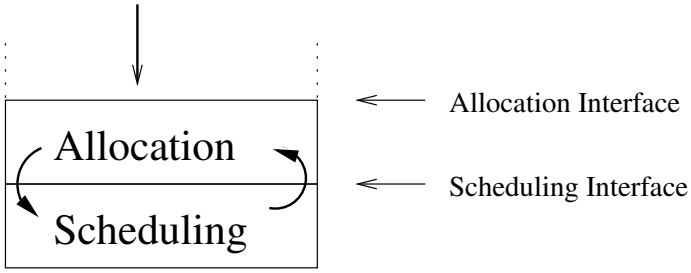

Figure 2-2 The Allocation and the Scheduling level of the design model with information flow.

As we see in Figure 2-2 information has to be passed into the Allocation level from above through the Allocation Interface. (The Allocation Interface corresponds to the interface between the first and the second design phase of the design model in Figure 2-1.) Parts of the information passed into the Allocation Interface and also new derived information is passed into the Scheduling level. We can also see how information has to be passed back from the Scheduling level up to the Allocation level again. This information is needed for any allocation algorithm to find an optimal allocation and cannot just be yes or no about the schedulability of the current system since a "smart" allocation algorithm needs more information than that. The problem is that this information depends on which allocation algorithm is chosen and we cannot define the information interface from the Scheduling level up to the Allocation level in a general way. Therefore this report will focus on the information needed to the Scheduling level, i.e. the Scheduling Interface. In the Scheduling level it is assumed that all processes in the system are allocated to a node. Of course we can still have an interaction with the Allocation level (manually or automatically) where a new allocation is made and again pass it into the Scheduling level. This iteration can continue until a final allocation is decided.

We will now see how the Scheduling Interface could be derived.

3. THE SCHEDULING INTERFACE

In this section the information needed by the Scheduling level of the design model is derived. The Scheduling Interface should be seen as a requirement on information needed to allow scheduling by off-line scheduling algorithms (e.g. Static Cyclic Scheduling) or scheduling analysis for on-line scheduling algorithms (e.g. Fixed Priority Scheduling).

This section is based on information from references 5, 6, 7, 8, 9, 10 and 11 which all are discussing Fixed Priority Scheduling. Therefore this section is influenced by the Fixed Priority Scheduling view of the scheduling problem when we derive the needed information.

In Section 3.1 we will describe what a process is in general. Fixed Priority Scheduling is described in Section 3.2 together with the different types of analysable process and the parameters needed for each type of process. Section 3.3 describes different types of inter process relations and introduces a relation object called a communicator. The Static Cyclic Scheduling approach is presented in Section 3.4. In Section 3.5 we finally try to define an information interface to the scheduling level by comparing the need of information in the Fixed Priority Scheduling and Static Cyclic Scheduling algorithms.

3.1 PROCESSES

A program consists of code (written in ***C*** or any other language) that defines it's functionality. When a program is being executed on a processor it is called a *process*. To make a program executable it first has to be *compiled*, i.e. translated into machine language. On machine level a process usually consists of executable code, data, a stack, a program counter, a stack pointer and some other registers.

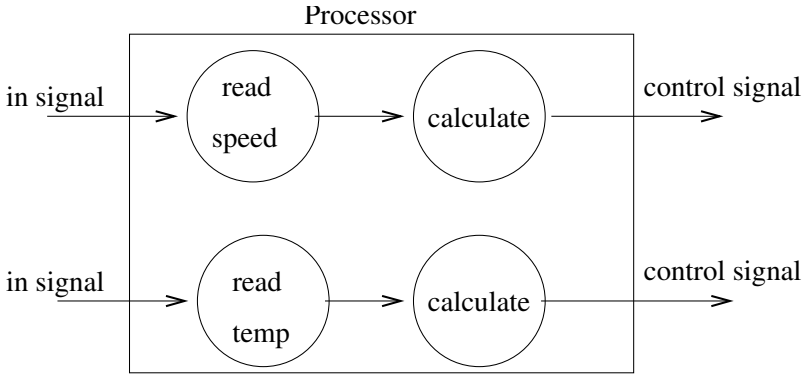

Figure 3-1. A processor with four processes.

A real-time process is a process that has at least one real-time constraint; typically a *deadline* to meet. We make a distinction between hard and soft real-time processes:

- A hard real-time process – must *always* finish its execution before its deadline.
- A soft real-time process – should meet its deadline, but is allowed to miss it sometimes.

A hard real-time process often performs safety critical tasks. For instance, when a pilot controls an aircraft, the actions of the commands must be taken within a couple of milliseconds, not five minutes later. A system implementing safety critical hard real-time processes is called a hard real-time system. Soft real-time processes are those used to maintain an acceptable response time of the system, but it is no catastrophe if the deadlines sometimes are missed. An example is a cash dispenser*.

3.1.1 TYPE OF PROCESSES

Real-time processes are classified according to their timing behaviour.

Three common process types are:

- Periodic processes – arrives with the period time *T*, meaning that a process instance is activated every *T* time units.
- Sporadic processes – are triggered by some event. The time between two successive events has a minimum of *T*.
- Aperiodic process – is triggered by events that may arrive within arbitrary small time intervals.

Aperiodic processes are very hard to analyse because of their unpredictable behaviour and therefore are often not allowed in a hard real-time system.

3.2 FIXED PRIORITY SCHEDULING

In Fixed Priority Scheduling each process has an unique and fixed priority. The scheduling is *on-line* and *pre-emptive*. On-line scheduling means that the scheduling is done dynamically at run time in comparison to Static Cyclic Scheduling where a static scheme is generated in advance off-line. Pre-emptive means that an executing process can be interrupted by a higher priority process that arrives. After the high priority process has finished it's execution the low priority process can resume execution The low priority process was *pre-empted* by the high priority process.

We are now going to list the different types of analysable processes and describe them shortly. For a more detailed description of the processes and the equations used in Fixed Priority Scheduling Analysis, see for instance [11]. We will start with the basic process.

3.2.1 BASIC PROCESSES

The basic type of process is a strictly periodic process with period *T*, a known worst-case execution time *C* and a deadline *D*.

Sporadic processes are analysed in the same way as periodic processes, treating them as periodic processes with period equal to the minimum time between two arrivals (*T*).

Needed parameters: *T, C, D*

The parameters δ^{min} and δ^{max} may also be needed for a basic process. We will explain why in Section 3.2.2.

* E.g.: a Swedish Minuten

3.2.2 PROCESSES INCLUDING JITTER

A process can suffer from *release jitter*, which means that there exists variations in the release. The time between two arrivals of a process can sometimes be longer or shorter than *T*, but the average time between two arrivals of a process is still *T*. The definition of jitter is:

> *Release jitter is the difference between the earliest and latest releases of a task relative to the invocation of the task.*

The meaning of *invocation* and *release* are:

- *Invocation* – a task is invoked if it could logically be made runnable at or after that time. Can also be called *arrival* of a task.
- *Release* – a task is released when the scheduler makes it runnable. Usually this means placing the task in a priority-ordered queue of runnable tasks, and executing the task if it is at the head of the queue.

If we denote the maximum delay from invocation to release δ^{max} and the minimum delay δ^{min}, then the release jitter *J* of task *i* (by definition) will be:

$$J_i = \delta_i^{\max} - \delta_i^{\min} \quad (1)$$

Notice that if the delay between invocation and release is constant then the task will experience no release jitter, i.e. if $\delta_i^{\min} = \delta_i^{\max}$ we will have $J_i=0$. The jitter captures the *variations* in time of the release relative to the invocation. Also note that if $\delta^{min}=0$, which is a common assumption, then the jitter is $J_i = \delta_i^{\max}$.

In most of the published fixed priority scheduling analysis it is assumed that $\delta^{min}=0$, but in [11, 12] an adapted analysis handling worst-case response times defined from the invocation of the process ($\delta^{min}>0$) is introduced. If we have $\delta_i^{\min} = \delta_i^{\max}$ the jitter is zero and then we have a basic process but with a constant delay between invocation and release. To handle this case we have to pass information about δ^{min}, δ^{max} even for a basic process.

An example how jitter can occur is when a periodic process *A* at the end of its execution on one processor triggers a sporadic process *B* on another processor. Process *A* can in its execution be pre-empted (interfered) by higher priority processes executing on the same processor as process *A*. Depending on how many higher priority tasks that arrive at a certain execution of process *A*, the time before completion of process *A* varies and hence also the release of process *B*. Process *B inherits* jitter from process *A*.

Additional parameter: *J* or δ^{min}, δ^{max} depending on if analysis for worst-case response times defined from invocation is used or not (see above).

3.2.3 PROCESSES WITH BURSTY BEHAVIOUR

A bursty process has a outer period *T* and a inner period *t*. Every *T* the process executes in a burst with at most *n* invocations in each burst and with the time *t* between each invocation in the burst (see Figure 3-2).

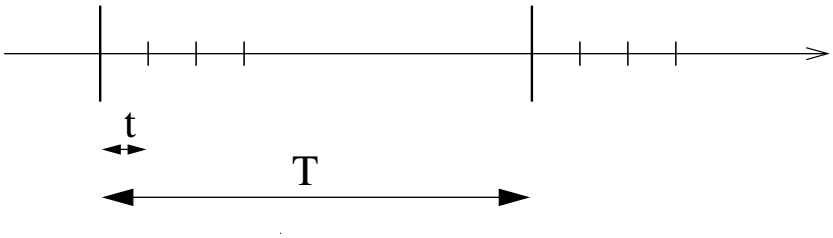

Figure 3-2 A bursty task with outer period T, inner period t and n=3.

The reason of having a bursty task can be for instance when sampling a sensor. The sensor can be unreliable, so the average of *n* readings of the sensor is used to give a more reliable value.

Additional parameters: *t, n*.

3.2.4 PROCESSES WITH TIME OFFSETS

A process can have a time offset relative to other processes grouped together in the same *transaction*. This means that a process with a time offset *O* will start at the time *O* after the start of the transaction. The analysis including time offsets is not yet adapted to handle worst-case response times defined from the invocation of a process, i.e. δ^{min} is assumed to be zero (see Section 3.2.2). Offsets could be used to:

- Reduce jitter.
- Avoid the need of dynamic concurrency control protocols to realise mutual exclusion.
- Expressing complex timing patterns between tasks.
- Realise precedence constraints between tasks in both local and distributed systems.

In the example described in Section 3.2.2 about how jitter can occur a time offset could be used to decrease the inherited jitter of process *B* by waiting until the process *A* producing a value needed by process *B* for sure has finished it's execution.

More about analysis including time offsets can be found in [6,8].

Additional parameter: *O*.

3.2.5 PROCESSES LOCKING SEMAPHORES

When processes are accessing a shared resource, for instance a shared memory buffer, the processes have to be *mutually excluded* so not more than one process can read or write in the shared resource at the same time. This is necessary to prevent inconsistent data in the shared resource.

Mutual exclusion can be implemented by the use of semaphores. A process entering it's *critical section*, i.e. the part of the code where it accessing the shared resource, locks the semaphore guarding that resource. When the process has finished it's critical section it unlocks the semaphore. Let us have the scenario where a process with high priority tries to lock a semaphore already locked by a lower priority process. The high priority process then has to wait until the lower priority process unlocks the semaphore

before it can lock the semaphore and enter it's critical section. This means that a high priority process can get *blocked* by a low priority process. Therefore we want to calculate the *blocking factor* of a process, i.e. the maximum time the process can be blocked by lower priority processes. The blocking factors can be calculated if for instance the *priority ceiling protocol* [13] is used when locking semaphores.

When calculating blocking factors information about processes locking semaphores is needed. For each process and semaphore which the process may lock the pair <*sem*, *cs*> has to be given, where *sem* is the semaphore identifier and *cs* is the maximum time a process locks the semaphore *sem*, i.e. *cs* corresponds to the worst-case execution time of that critical section.

Additional parameters: Sets of pairs <*sem*, *cs*>

3.2.6 SCHEDULER OVERHEAD

To include scheduler overhead in the analysis time parameters of the kernel functions implementing the scheduler must be known. The analysis is depending on the type of scheduler, for instance tick scheduler or event scheduler [12].

Since Fixed Priority Scheduling is pre-emptive, the scheduler has to switch between different process when a higher priority process arrives when a lower priority task is executing. This is called a *context switch*. To include this in the analysis the parameter C_{sw} is needed, which is the computation time of a context switch.

For an event scheduler we also need the parameter C_{timer}, which is the time taken to respond to the timer interrupt causing the events. When a tick scheduler is used we need the parameter C_{queue}, which is the time taken to move a task from the delay queue to the run queue. We also have C_{tick}, which is the time taken to handle the tick interrupt and T_{tick} being the tick period.

Additional parameters: Event scheduler: C_{sw}, C_{timer};
Tick scheduler: C_{sw}, C_{queue}, C_{tick}, T_{tick}.

3.2.7 SUMMARY OF NEEDED PARAMETERS

The parameters needed for scheduling analysis of the different types of processes are listed here. We will first list the existing parameters.

The existing parameters are:

- T – the period time of periodic processes. For sporadic processes T is the minimal time between arrivals.
- t – the inner period of a task with bursty behaviour.
- n – the number of invocations of a bursty task.
- D – the deadline of a task.
- C – the worst-case computation time of a task.
- J – the release jitter of a task.
- δ^{min}, δ^{max} – needed if worst-case response times defined from invocation are used (see Section 3.2.2).
- O – the offset of a process (in a transaction).

e – the every attribute of a process in a transaction means that the process executes every *e* time as the transaction starts.

<*sem*, *cs*> – <semaphore identifier, maximum time the semaphore is hold>

P – is the priority of the task. This parameter could be derived through any priority assignment algorithm.

The parameters for the basic process are always needed, i.e. *T, C, D* (δ^{min}, δ^{max} could also be needed for the basic process as described in Section 3.2.2). Depending on what other types of processes are included in the task set additional parameters are needed according to Table 3-1.

Type of Process Included in the Process Set	**Additional Parameters**
Jitter	*J* or δ^{min}, δ^{max}
Bursty Behaviour	*t, n*
Offset	*O*
Locking Semaphores	<*sem, cs*>
Manual Priority Assignment	*P*

Table 3-1 Needed parameters depending on what types of processes are included in the set of processes.

As an example a process set of two processes where one includes jitter and the other has bursty behaviour, the needed parameters in total would be *T, C, D* (basic parameters), *J* (jitter) and *t, n* (bursty behaviour).

When scheduler overhead is included in the analysis additional parameters are needed according to Table 3-2.

Scheduler Type	**Additional Parameters**
Event Scheduler	C_{sw}, C_{timer}
Tick Scheduler	C_{sw}, C_{queue}, C_{tick}, T_{tick}

Table 3-2 Additional parameters when scheduler overhead is included in the analysis.

3.3 INTER PROCESS RELATIONS

Processes can interact in two ways: synchronous or asynchronous. The interaction can also be local or distributed. From the system designers view it would be good to have some kind of abstract relation object that hides the implementation of the interaction. We will therefore introduce a relation object that we call a communicator and we will see that there exists four types of communicators that can be combined.

We will here explain the terms synchronous and asynchronous (Section 3.3.1), local and distributed (Section 3.3.2) and then continue to describe the different types of interactions for linear and branched inter process relations (Sections 3.3.3 - 3.3.6 and 3.3.7 - 3.3.10 respectively).

3.3.1 SYNCHRONOUS VS ASYNCHRONOUS RELATIONS

Processes can interact in a synchronous or asynchronous way. When processes interact, data is often transferred from one task to another. If this is done synchronously (in a joint action) the processes must have the same period time *T*. Every instance of produced data is consumed by the succeeding process. We could say that the input data propagates through the synchronous chain of processes. The processes in a synchronous chain often share a *precedence constraint*. A precedence relationship between two process means that one has to finish it's execution before the other starts.

When we have an asynchronous interaction between processes, the period times of the processes do not have to be the same. To transfer data between the processes a shared buffer could be used. The process producing the data, updates the data as often as needed to have actual data in the buffer, which of course depends on how often the input data to that process is altered. The process (or processes) that uses the data in the shared buffer reads it as often as needed according to its period time. It is important to maintain mutual exclusion among the tasks accessing the shared buffer (see Section 3.2.5).

If a process relation should be synchronous or asynchronous must be defined in the system design. Examples of synchronous relations are those between the different processes realising a closed control loop. An asynchronous relation could for instance be between the process sampling the vehicle speed from a wheel and the process presenting the vehicle speed to the driver. The display presenting the speed does not have to be updated as often as the sampling rate of the vehicle speed.

3.3.2 LOCAL VS DISTRIBUTED RELATIONS

A distributed system consists typically of two or more processors connected via a communication link (e.g. a bus). A relation between two processes in a distributed system can be either local or distributed. If two interacting processes are located at the same processor their interaction is local. If instead the two processes are located at different processors in the distributed system the relation is distributed. To realise the distributed interaction a message has to be sent over the communication link. The communication link itself is a resource as well as the processors in the system. As any resource in a distributed hard real-time system, the messages sent over the link have to be scheduled.

A process located at a processor (or node) is said to be allocated to that processor. Some processes are statically allocateable and some are dynamically allocateable. A process that needs to access sensors or actuators situated at a certain processor has to be allocated to that processor. Hence the process is statically allocated. On the other hand, if the process is not dependent on resource specific environment it can be allocated at any node and hence it is dynamically allocateable. In the Scheduling level we assume all processes to be statically allocated or that dynamically allocatable processes already are given a fixed allocation.

The relation type between two processes depends on if the processes are statically or dynamically allocateable. If both processes are static allocateable the relation could be either fixed local (i.e. always local) or fixed distributed. With at least one dynamically allocatable process in the relation we have a dynamic relation. Note that in the final system all dynamically allocateable processes have a fix assignment to a node. It is only in the design phase we have the possibility to move dynamically allocateable processes around between different nodes in the system until we found an allocation optimal enough.

3.3.3 LOCAL ASYNCHRONOUS RELATIONS

The local asynchronous relation is typically implemented with a shared buffer or more generally a Protected Shared Object, PSO [10]. All processes accessing the PSO have to be mutually excluded. This can for instance be implemented with semaphores or an atomic update mechanism. In Figure 3-3 we introduce the asynchronous local communicator between the two processes *A* and *B*. We also see the internal structure of the communicator. As said earlier the decision if the communicator should be asynchronous or synchronous depends on the functionality of the process interaction. If it is local or distributed depends on the allocation of the processes in the relation.

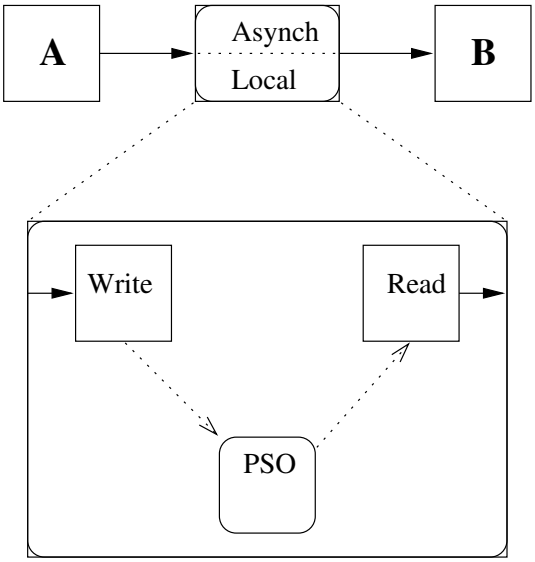

Figure 3-3 A local asynchronous inter process relation.

In Figure 3-3 we see that the communicator consists of a `Write` and `Read` process together with the PSO storing the data to be transferred. The `Write` process has the same period time as process *A*, T_A, and the `Read` process has the same period time as process *B*, T_B. An asynchronous relation means that T_A and T_B do not have to be equal.

3.3.4 LOCAL SYNCHRONOUS RELATIONS

The local synchronous communicator works in the same way as the local asynchronous communicator except that there is flow connection between the `Write` and `Read` process (see Figure 3-4). This means that the period times *T* of process *A*, *B*, `Write` and `Read` are all the same.

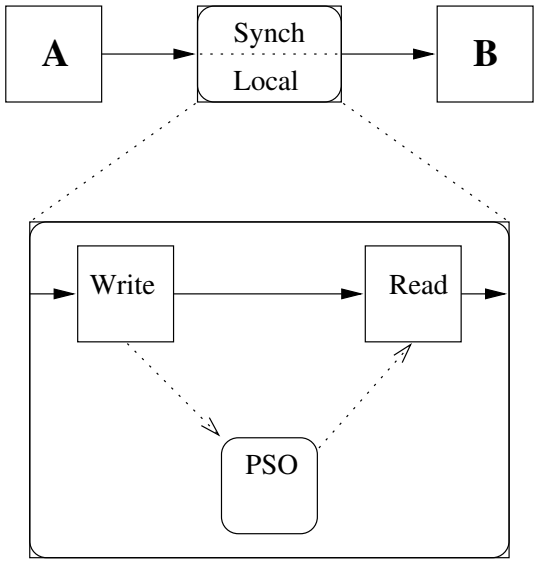

Figure 3-4 A local synchronous inter process relation.

Since the `Write` and `Read` process has a precedence relationship they cannot access the PSO simultaneously and hence there is no need to use a semaphore to guard the PSO. This is valid if no other processes are accessing the PSO and the deadline D of the precedence chain is less than or equal to the period time T of the chain ($D \leq T$). If $D>T$ a new instance of the same process may interfere with the previous one that has not yet finished. By using FIFO-execution among the invocations we will not have any problem with interference between instances of the same process.

From Section 3.2.5 we remember that the part of the code where a process is accessing a shared resource is called the critical section. The `Write` and `Read` process corresponds the critical section of process *A* and process *B* respectively. When using communicators we have to require that the critical sections of process *A* and *B* can be extracted and put into the communicator as `Write` and `Read` respectively. This should be possible, provided that they are the last and the first operations of *A* and *B*. This requirement is also found in the concept used in [14].

The advantage of making an abstraction of the inter process relation by using a communicator is that the implementation dependent problems could be solved within the communicator. With this in mind it can be worth meeting the requirement above.

The coupling between process *A* and `Write` (`Read` and *B*) is synchronous and of course local. In spite of that we cannot here use the local synchronous communicator in Figure 3-4 since we then have to implement the connection between process *A* and the `Write` process in that new communicator and so on. Therefore the communication between process *A* and `Write` has to be solved in some other way. One solution is to have a buffer shared by only *A* and `Write`. This buffer does not have to be protected since we have a precedence constraint between *A* and `Write`, i.e. *A* and `Write` cannot execute at the same time. In this way there is no need of using a semaphore to protect the buffer and hence no critical sections are involved. Another not so structured and general solution is that we in the final implementation again make `Write` and *A* to one process by appending the `Write` process to process *A*. The communication is then solved locally within that single process. Unfortunately we are in this way loosing the

pure communicator structure, but the communicator can anyway be a helpful structuring of the work during the design phase.

3.3.5 DISTRIBUTED SYNCHRONOUS RELATIONS

For distributed relations a message has to be sent over the communication link. A message has to be sent by one process and received by another. Those two processes can be server process doing nothing else. The two processes and the message are then always synchronous. This means that a distributed asynchronous relation has a part that is synchronous. Therefore we start to describe the synchronous distributed relation.

In Figure 3-5 we see a distributed synchronous relation consisting of a process *A*, a distributed synchronous communicator and a process *B*. The distributed synchronous communicator consists of a process sending the message *M*, the message *M* and a process receiving the message. Compared to the local synchronous communicator the `Write` and `Read` processes are replaced by the `Send_M` and the `Rec_M` processes respectively in the distributed synchronous communicator. The PSO is replaced by the message *M*.

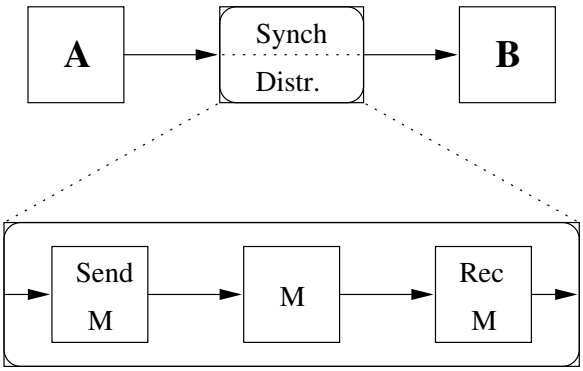

Figure 3-5 A distributed synchronous inter process relation.

The period time is the same for all processes and messages in a synchronous chain.

3.3.6 DISTRIBUTED ASYNCHRONOUS RELATIONS

A distributed asynchronous relation has to send a message over the communication link and hence includes a synchronous part and of course a asynchronous part (see Figure 3-6). The synchronous part is the same as the distributed synchronous communicator described in Section 3.3.5. The asynchronous part is the same as the local asynchronous communicator described in Section 3.3.3. The asynchronous point could be placed at the same node as process *B* or at the same node as process *A*.

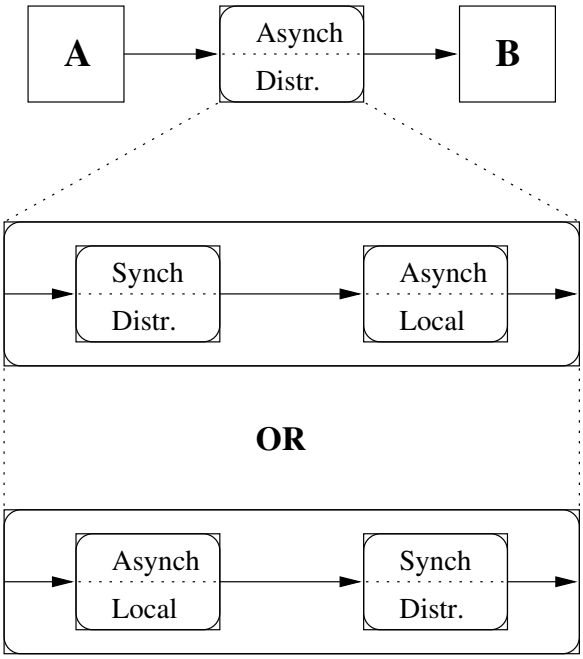

Figure 3-6 A distributed asynchronous inter process relation.

If the asynchronous point is placed at the node of process *B*, the processes and the message in the synchronous distributed communicator have the period time of process *A*. Also the `Write` process of the asynchronous local communicator has this period time, but the `Read` process has the same period time as process *B*. If the asynchronous point instead is placed at the same node as process *A*, the period time of the `Write` process of the local asynchronous communicator is the same as for process *A*. All other processes in the relation have the period time of the process *B*. It is more general to put the asynchronous point at the node of the second process (process *B*) in the relation. We will return to this in Section 3.3.9.

3.3.7 BRANCHED LOCAL ASYNCHRONOUS RELATIONS

A general branched asynchronous relation is shown in Figure 3-7 where a dashed arrow represents a general asynchronous relation.

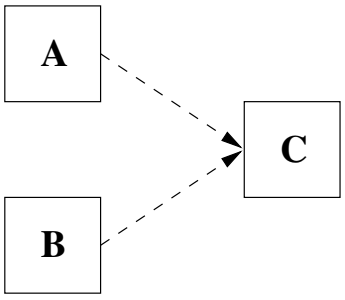

Figure 3-7 A general branched asynchronous inter process relation.

We do now want to represent the relation in Figure 3-7 as a local asynchronous branched relation by using communicators. This is easily done by using the local asynchronous communicator described in Section 3.3.3 and the result is shown in Figure 3-8.

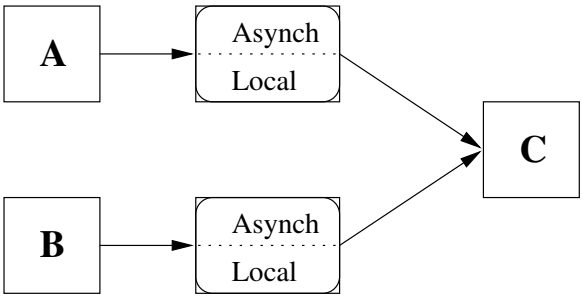

Figure 3-8 A branched local asynchronous inter process relation.

All the processes *A*, *B*, and *C* may have different period times.

3.3.8 BRANCHED LOCAL SYNCHRONOUS RELATIONS

In a general branched synchronous (local or distributed) relation we can allow only one connection to be synchronous, the rest must be asynchronous (see Figure 3-9 where the dashed arrows represent asynchronous relations and the solid line arrows represent the synchronous relation). The reason of this is that process *C* can be triggered by one and only one process. We will return to this problem in the distributed case.

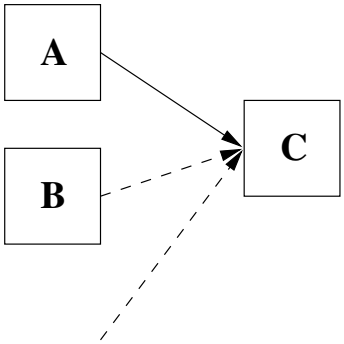

Figure 3-9 A general branched synchronous inter process relation.

The branched local synchronous case realised with communicators is shown in Figure 3-10.

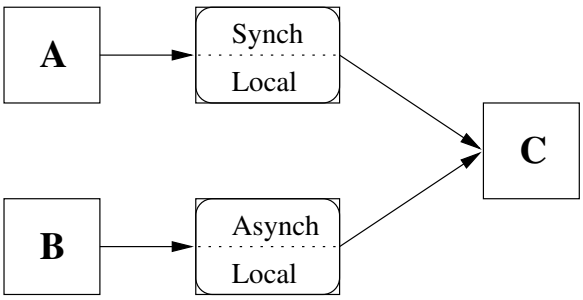

Figure 3-10 A branched local synchronous inter process relation.

The period times T_A and T_C are equal while T_B may have any value. The period time of the `Read` process of the asynchronous local communicator is though equal to T_C, i.e. process *C* reads the data produced by process *B* every time it is triggered by process *A*.

In the local (single processor) case processes cannot execute in parallel. When we have the local situation of Figure 3-9 we could solve the triggering problem by letting the last finishing process trigger process *C* or having a time offset of process *C*.

If a process *A* has inter process relations with more than one process as in Figure 3-11, it is allowed to have more than one synchronous relation, i.e. a process can trigger more than one process (but cannot be triggered by more than one process).

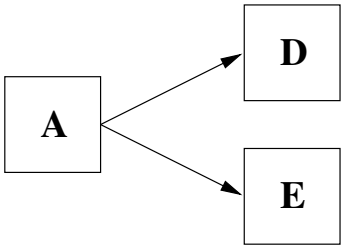

Figure 3-11 A general synchronous inter process relation with a splitting branch.

The relations *A* to *D* and *A* to *E* can either be local or distributed and realised with a local or distributed synchronous communicator respectively. Since process *A* triggers both process *D* and *E* the period times of *D* and *E* are equal to the period time of process *A* ($T_D=T_E=T_A$).

3.3.9 BRANCHED DISTRIBUTED SYNCHRONOUS RELATIONS

The branched distributed synchronous case is seen in Figure 3-12. The asynchronous distributed communicator must have it's asynchronous point at the same node where process *C* is allocated, since process *C* is already triggered by the synchronous communicator from process *A*. Process *C* inherits jitter from the message originating from process *A*. If process *C* is triggered by more than one process we cannot decide from where the process should inherit the jitter. This would for instance be the case if the asynchronous point of the asynchronous distributed communicator was placed at the node of process *B*. By the reason of uniquely inheriting jitter and also the period time we are allowed to have only one synchronous relation at a branching point where relations are merged. Hence the asynchronous point of all asynchronous distributed communicators must be at the node where the relations are merged (the node of process *C*).

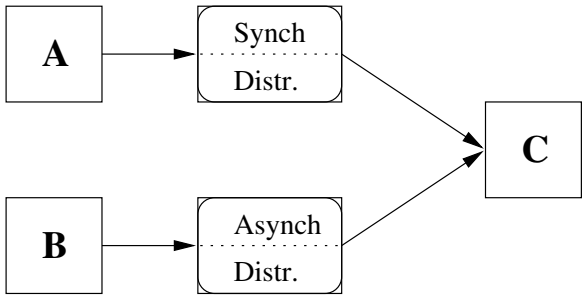

Figure 3-12 A branched distributed synchronous inter process relation.

The problem having a process that receives more than one message is discussed in [7] and the solution as follows is also presented. Multiple messages sent to the same process can be permitted using the following mechanism: a server process is created

for each message destined for process *i*, which then simply writes the messages to a shared protected object. Process *i* can poll the protected object and any messages. The idea used in the communicator is similar where the `Write` process of the local asynchronous part of the distributed asynchronous communicator corresponds to the server process of the received message.

3.3.10 BRANCHED DISTRIBUTED ASYNCHRONOUS RELATIONS

In Figure 3-13 we see a branched distributed asynchronous relation. Each asynchronous distributed communicator must have it's asynchronous point at the same node as process *C* is allocated. Otherwise we cannot determine the period time of process *C* if process *A* and *B* have different period times.

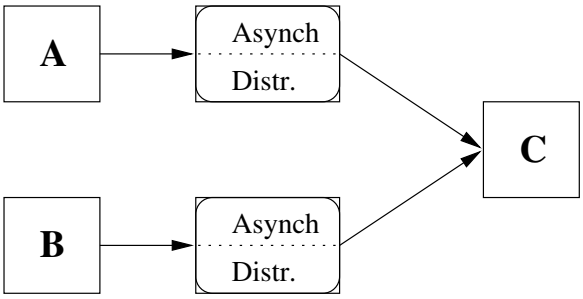

Figure 3-13 A branched distributed asynchronous inter process relation.

3.4 STATIC CYCLIC SCHEDULING

In Static Cyclic Scheduling (SCS) the scheduling is done off-line. Off-line scheduling implies that all processes in the system are scheduled before the system is executed. In Static Cyclic Scheduling a static schedule is generated either by hand or by a tool. The principles of SCS are simple, but for a complex system it can be very hard to find an optimal schedule (it is in general an NP-hard problem).

A drawback with Static Cyclic Scheduling is that outer events must be polled, i.e. it cannot handle sporadic processes. On the other hand it offers a simple method for mutual exclusion when accessing shared resources. Pre-emption can be used in SCS. In that case the exclusion constraints has to be given so critical sections of processes can be scheduled so they do not overlap.

Interrupts can in SCS be polled by a periodic interrup server process. The resolution how oft an interrupt can be detected depends on the periodtime of the server process. Another way to handle interrupt could be to add an interrupt time budget to each instance in the schedule. When an interrupt occurs the schedule is temporarily left and the interrupt handled. After the interrupt handler has finished it's execution the control is resumed by the schedule. Since there is a intterrupt time budget we do not have t the schedule can after the interrupt resumes the control without causing any missed deadlines. The number of interrupts during a time interval has to be limited so that the interrupt time budget is not exceeded.

Sporadic processes could in the way described above be handled by interrupts.

There are tools that generate static cyclic schedules for a given system. One of these tools is introduced below.

3.4.1 BASEMENT

The BASEMENT* concept [14] is based on Static Cyclic Scheduling. Within BASEMENT a scheduling tool has been developed. A description of this tool is found in [15]. The tool also includes an allocation algorithm.

A BASEMENT system consists of a set of nodes interconnected with a communication network. A node can be viewed as a computer with a network interface and, possibly, a set of input/output devices, e.g. sensors/actuators.

The basis for the BASEMENT software architecture is the fundamental difference between hard and soft applications. In BASEMENT, the colour *red* is associated to hard applications, and *blue* to soft applications. Each *red* application is handled by one or more red processes and each blue applications handled by one or more blue processes.

The scheduling for the red system is based on a cyclic, static, off-line paradigm, guaranteeing timing constraints to be satisfied. Scheduling decisions for the blue system are made at run-time.

The communication service is a time multiplexed service. All applications in a BASEMENT system share the communication media. For red processes, BASEMENT communication provides error free transmission with bounded and predictable delays. For blue processes the communication is error free, but no delay guarantees are given.

Further, for the application developer communication is transparent. No considerations of the actual location of different parts in the system need to be taken.

BASEMENT uses something called a Software Curcuit (SC) that is a non pre-emptable part of the code implementing a basic function. Processes are then built of SCs.

A real-time kernel called Rubus has been developed using the BASEMENT concept. A tutorial of Rubus can be found in [16].

3.5 A SCHEDULING ALGORITHM INDEPENDENT INTERFACE

In this section we will start to investigate the differences in need of information for different types of scheduling algorithms. The investigation will be realised through a comparison between the Fixed Priority Scheduling (FPS) algorithm and the Static Cyclic Scheduling (SCS) algorithm. As said before the FPS algorithm is on-line and dynamic while SCS is off-line and static, so the scheduling strategies differs a lot. This prevents our comparison to be just a trivial case and does in this way increase the generality of the information interface.

We will derive the interface as the intersection of the set of information needed for Fixed Priority Scheduling analysis and the set of information needed for Static Cyclic Scheduling.

* BASEMENT is a registered trademark of Mecel AB.

3.5.1 COMPARISON

In Table 3-3 we list the different properties for Fixed Priority Scheduling, Static Cyclic Scheduling and the Basement Scheduling tool. The numbers in the table cells are note numbers. The notes are found below the table. The notations in the table are the following: X = can be handled, - = cannot be handled, (X) = is implicitly handled.

Type of Process:	**Fixed Priority Scheduling (FPS)**	**Static Cyclic Scheduling (SCS)**	**Basement Scheduling Tool**
Periodic	X	X	X
Sporadic	X	- (1)	- (1)
Aperiodic	-	-	-
Basic	X	X	X
Jitter	X	- (2)	- (2)
Bursty	X	X (3)	- (3)
Offset	X (4)	(X) (5)	(X) (5)
Mutual exclusion	X (6)	X (7)	X (8)
Precedence	X (9)	X (10)	X (11)

Table 3-3 Comparison between different scheduling approaches. X = Handles; - = Does not handle.

Notes to Table 3-3:

1. Sporadic processes could be polled and made periodic since all inputs have to be polled. (Interrupt could be used to capture sporadic external signals (see Section 3.4)).
2. Jitter does not occur in a static scheme and hence it is not necessary to handle jitter. (Jitter in external signals is removed since they are polled.)
3. A bursty process could be scheduled in a static scheme but is not handled by the Basement scheduling tool.
4. The OS kernel has to support offsets.
5. There is no need for offsets to reduce jitter in SCS (see Note 2). In a sense meaning a static scheme could be said to be built by the use of offsets.
6. In FPS some mechanism for mutual exclusion is needed. Semaphores is one method and offsets is another. With time offsets a “static” scheme could be created where processes in a transaction are scheduled so they exclude each other mutually.
7. In SCS mutual exclusion is realised by generating a schedule where the critical sections of processes are not overlapping.

8. The Basement scheduling tool uses something called a Programmed OPeration (POP) which is non pre-emptive. If the critical section is defined in a POP the mutual exclusion constraint will be met.
9. In FPS precedence could be realised by the use of the priority mechanism. If processes with precedence constraints are released simultaneously and the first process has the highest priority and the priorities are decreasing monotonically with the processes order in the precedence chain then the precedence constraints will be met.
10. In SCS the precedence constraints are met if the processes are scheduled in the right order in the static scheme.
11. Information about where precedence constraints between processes must be given to the scheduling tool.

Considering the comparison of the different scheduling approaches in Table 3-3 we will now try to derive the common properties and see what requirements we must have on a general information interface to the scheduling level.

According to the table we see that sporadic processes are handled by FPS but not by SCS and therefore we should require that no sporadic processes exist in the system. The FPS analysis of sporadic processes is identical with the analysis of a periodic process. Hence we do not have to exclude sporadic processes since it does not affect the information interface.

Since SCS does not use the jitter parameter it would be more general if we can reduce the jitter parameter from the interface. This means that we cannot allow any input jitter. If we poll a sporadic input then we can take away input jitter. In FPS analysis we can still include inherited jitter in the analysis since those values are derived within the analysis.

As mentioned in the notes above bursty processes could in principle be handled by SCS. Note that the Basement scheduling tool does not handle bursty processes.

Offsets could in FPS for instance be used to reduce inherited jitter. This does just increase the schedulability of the system and is not needed for the FPS in general. Since SCS does not need the offset parameter it will be excluded from the general interface.

When mutual excluded processes are scheduled some extra information is needed. In SCS mutual exclusion is implemented by making a schedule realising the mutual exclusion. We do then need to know what piece of code that corresponds to the critical section so that part of the processes is not pre-empted. The worst-case execution time of the critical section is also needed. For FPS we need the worst-case execution time of the critical section, but instead of the knowledge about what piece of code that corresponds to the critical section we need to now what semaphore is held during the critical section (see Section 3.2.5). To achieve a general interface in this case we have to pass some redundant information. The information what piece of code that corresponds to the critical section is redundant for FPS and the semaphore identifier is redundant when SCS is used. If we use the communicators described in Section 3.3 we will reduce the problem with redundant information since the `Write` and `Read` processes are identical to the critical section of the processes they belong to.

The semaphore identifier needed for FPS could in the SCS case be seen as a identifier of the shared memory buffer. Other information needed when implementing the system is the requirement of the buffer size. This information is though not used during the scheduling phase.

Information about precedence constraints has to be given for both SCS and FPS.

4. SUMMARY AND CONCLUSIONS

In Section 2 the outline of the design model under development within ASTEC was given. The different types of analysable processes within Fixed Priority Scheduling and their needed parameters were presented in Section 3. Different types of inter process relations were presented. A relation object called a communicator was introduced. The purpose of the communicator is to make an abstraction of inter process relation. By presenting the Static Cyclic Scheduling approach and comparing it with Fixed Priority Scheduling a scheduling approach independent information interface was derived.

What this report shows is that we have to restrict the type of processes that can be accepted in the system if we want to have a general information interface to the scheduling level. This limits the degree of freedom for the system designer. There could be two possibilities of increasing the flexibility. One way is to define an interface where we have some redundant information needed for a specific choice of scheduling approach. In this way we could achieve desired possibilities for a certain approach, but we do also have to supply the extra information even if the approach will not be used. A problem is that some functionality's can be implemented with one approach but not with the other. A second way of increasing the flexibility is to make the decision of what scheduling approach to use in an earlier phase of the design work. In this way the possibilities of the chosen approach could be more optimally used.

The communicators have in this report been developed from the FPS approach. The classification of the communicators, i.e. synchronous or asynchronous and local or distributed, is general and is not FPS specific. This classification could (should?) be done with any scheduling approach. The implementation of the communicator may then differ. An idea similar to the communicator is used in Basement for SCS and is called a connector.

5. REFERENCES

[1] Sven Larsson and Magnus Lindahl, "*System Specification - ASTEC Application Example*", MEG-8033-DOC-001SS, Mecel AB, Göteborg.

[2] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi, "*UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems*", In Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 1995.

[3] Johan Bengtsson, David W. O. Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi, "*Verification of an Audio Protocol with Bus Collision Using UPPAAL*", In Proceedings of the 8th International Conference on Computer-Aided Verification, New Brunswick, New Jersey, USA, LNCS 1102, p. 244-256, R. Alur and T.A. Henzinger (Eds.), August 1996.

[4] Tindell, K., Burns, A. and Wellings A.J., "*Allocating Real-Time Tasks (An NP-hard Problem Made Easy)*", Journal of Real-Time Systems, Vol. 4 No. 2, p. 145-165, June 1992.

[5] Tindell, K., "*Using Offset Information to Analyse Static Priority Pre-emptively Scheduled Task Sets,*" YCS 182, Department of Computer Science, University of York (1992).

[6] Tindell, K., "*Fixed Priority Scheduling of Hard Real-Time Systems*", PhD Thesis, Department of Computer Science, University of York (1994).

[7] Ken Tindell and John Clark, "*Holistic Schedulability Analysis For Distributed Hard Real-Time Systems",* Technical Report YCS197, Real-Time Systems Research Group, Department of Computer Science, University of York, 1993.

[8] Ken Tindell, *"Adding Time Offsets to Schedulability Analysis"*, Technical Report YCS221, Department of Computer Science, University of York, November 1994.

[9] Allan Burns and Andy Wellings, "*Real-Time Systems and Their Programming Languages*", Second Edition, Addison-Wesley Publishing Company, 1996. To be published.

[10] Mathai, Joseph (editor), "*Real-Time Systems - Specification, Verification and Analysis*", Chapter 3: Burns, A., Wellings, A. (authors), "*A Computational Model For Fixed Priority Scheduling*", Prentice Hall, 1996, ISBN 0-13-455297-0.

[11] Jens Larsson, "*ScheduLite - A Fixed Priority Scheduling Analysis Tool*", Master of Science Thesis DoCS 96/73, ASTEC 96/01, Department of Computer Systems, Uppsala University, 1996.

[12] Tindell, K., "*Real Time Systems by Fixed Priority Scheduling*", Course Book, Department of Computer Systems, Uppsala University, 1996

[13] Sha, L., Rajkumar, R. and Lehoczky, J., "*Priority Inheritance Protocols: An Approach to Real-Time Synchronisation,*" IEEE Transactions on Computers 39(**9**) (September 1990) pp.1175-1185.

[14] Hans A. Hansson, Harold Lawson, Mikael Strömberg and Sven Larsson, "*Basement: A Distributed Real-Time Architecture for Vehicle*

Applications..", Journal of Real-Time Systems, Kluwer Academic Publishers, Vol. 11, No. 3, p 223 - 244, November 1996.

[15] Erik Larsson, "*Vehicle Internal Architecture BASEMENT Real-Time System: The Scheduling Tool*", Master of Science Thesis DoCS 95/56, ProVIA-94204, Department of Computer Systems, Uppsala University 1995.

[16] C. Eriksson and K.-L. Lundbäck, *"Rubus OS Real-Time Operating System, Tutorial"*, Technical Report, Articus Systems AB, 1996.