

Optimal Design

*Toni Riutta
Mälardalen university
Västerås, Sweden
tra98002@student.mdh.se*

*Kaj Hänninen
Mälardalen university
Västerås, Sweden
khn98013@student.mdh.se*

*Masters thesis
February 2003.*

*The Department of Computer Science and Engineering,
Mälardalen University, Västerås, Sweden.*

“Beware when the great God lets loose a thinker on this planet”
Ralph Waldo Emerson (1803 - 1882)

Abstract

Volvo Construction Equipments develops different types of vehicle control systems for entrepreneur machines. The systems are based upon the real-time operating system Rubus.

Rubus supports two types of real-time tasks: hard off-line scheduled and soft on-line scheduled tasks. The current systems use off-line schedules with period times equal to 100 ms to limit the ROM usage. The majority of the functionality is modeled as hard real-time tasks.

The resources in the systems are, of course, limited, meaning that the task sets is limited to a certain amount of tasks. Using more of the on-line scheduled tasks could be beneficial in order to decrease the utilization, hence Volvo wants to know if it is possible to move functionality from the hard off-line scheduled to the on-line scheduled part, with preserved requirements. This constitutes the main objective of the thesis.

The thesis present several different solutions to this problem ranging from support to the development process, system modifications and automation tools for off-line analysis.

A case study was performed in order to get familiar with the systems and the development process at Volvo. It was followed by a literature study serving as the base of knowledge for the proposed solutions. During the literature studies no single solution was found applicable without modifications, to the environment at Volvo.

A concept based on hard real-time systems, weakened to include a precisely bounded and predictable distribution of lost deadlines called Weakly Hard Real-Time systems, was adopted to serve as the backbone of the presented solutions. It was modified to suit the development process and architecture at Volvo, including requirements specifications, design and implementation issues etc.

A simple solution of guaranteeing on-line scheduled tasks, in the background of a static cyclic off –line schedule, is also presented as a complement to the Weakly Hard concept.

The usage of the Weakly Hard concept at Volvo would deliver a desired quality of service with decreased utilization i.e. better usage of the system resources with preserved requirements.

Acknowledgements

We would like to thank all the people at the TUE department at Volvo CE in Eskilstuna Sweden, for giving us this rare opportunity to work and suggest improvements to their embedded real-time system. It has been an experience that has learned us a lot about real-time systems in the “real” world.

A special thank you goes out to Nils-Erik Bånkestad and Robert Larsson, our mentors at Volvo, for fruitful discussions and for being kind and understanding that the work burden for masters students can be tough.

We would also like to thank Jukka Mäki-Turja for fruitful discussions and for being our mentor at Mälardalen University.

Last but not least we would like to thank all the researchers that have contributed with valuable information to this thesis, and Arcticus Systems AB for the demo version of Rubus VS.

Toni would also like to thank Nina Surakka for all the encouragement and for being very patient during tough days of life, he will always be in great debt to her. He is also very thankful to his mother for the great number of fantastic and fruitful phone calls and for taking care of him.

Furthermore he is in great debt to Kaj Hänninen for being supportive and for having the energy to push him and wait for him during a long and tough period.

Finally Toni wants to thank all his other friends and loved ones.

Kaj would also like to thank all family members (human and non-human) for their love, and friends, especially Toni Riutta, for all support.

Contents

Abstract.....	3
Acknowledgements.....	4
Contents.....	5
Background.....	7
Objectives and methods.....	8
Introduction to real-time systems	10
1 A Case Study of a Vehicle Control System: The 220 E Wheel Loader.....	15
1.1 Introduction.....	15
1.2 The architecture.....	15
1.3 Rubus OS - A Real-Time Operating System	16
1.3.1 Red services.....	16
1.3.2 Blue services.....	16
1.3.3 Interrupts	17
1.3.4 Additional information.....	17
1.3.5 Task schedule in the VECU.....	17
1.3.6 System partitioning	18
1.3.7 Rubus development tools	18
1.4 Software development process at Volvo	19
1.5 Results from the case study.....	20
1.5.1 Development process.....	20
1.5.2 The communication	21
1.5.3 Converting functionality from Red to Blue	21
1.6 Revisiting the objectives.....	21
2 Literature Study.....	23
2.1 Opening discussion	23
2.2 Requirements specification	24
2.3 Design	26
2.4 Implementation.....	31
2.4.1 Server algorithms	31
2.4.1.1 Fixed-priority servers	31
2.4.1.2 Dynamic priority servers	34
2.4.2 Overload management	36
2.4.2.1 Guarantee-based algorithms	37
2.4.2.2 Best-effort algorithms	51
2.4.2.3 Algorithms based on imprecise computations	53
2.5 Automation tools	54
2.6 Response time analysis	56
2.7 Resource Access Protocols	57
2.8 Performance evaluations and reviews.....	59

3	Findings	61
3.1	Requirements specifications	61
3.2	Design	62
3.2.1	Basic design considerations	63
3.2.2	Timing constraints	64
3.3	Designing different applications	64
3.3.1	Control applications	64
3.3.1.1	All functionality in one task.....	64
3.3.1.2	Split functionality into two tasks	64
3.3.2	Monitoring and logging.....	65
3.3.3	Interaction.....	66
3.4	Examples of techniques to model functionality	66
3.4.1	System model.....	66
3.4.2	Interaction Example	66
3.5	Guaranteeing Blue tasks	67
3.5.1	The simplest approach	69
3.5.1.1	Example	70
3.5.2	Guaranteed Blue TT and ET tasks.....	71
3.5.2.1	Feasibility check of Blue TT and ET tasks	71
3.5.2.2	Response time analysis primary for Blue TT tasks.....	72
3.5.2.3	Application	74
3.5.2.4	System modifications	75
3.5.2.5	Example	78
3.5.3	The Weakly Hard concept.....	80
3.5.3.1	Assumptions	80
3.5.3.2	Finalization time analysis	81
3.5.3.3	Application	82
3.5.3.4	System modifications	83
3.5.3.5	Example	84
3.5.3.6	Improving the suggested concept at Volvo	87
3.6	Support for the weakly hard concept in the development process	88
3.6.1	Managing software requirements for weakly hard systems	88
3.6.2	Designing for weakly hard systems	89
	Future work.....	91
	Conclusions	92

Background

This paper is written as a part of our masters thesis, during spring 2002, at Volvo CE dept. TUE in Eskilstuna.

The department TUE (hereafter called Volvo) develops different types of vehicle control systems for Volvo entrepreneur machines. The control systems are based on Rubus and to a great extent off-line scheduled with hard timing constraints. Rubus also supports non-guaranteed on-line scheduled tasks, which Volvo only uses slightly.

The resources in the system are, of course, limited. Today Volvo uses off-line schedules with period times equal to 100 ms to limit the ROM usage. This restriction forces the developers of the system to set period times to at most 100 ms for the tasks, which make the system over utilized. Additionally the on-line part would be preferred in some situations where flexibility is important.

Volvo wants to know if functionality can be moved from the off-line scheduled part (Red part) to the on-line scheduled part (Blue part), in order to use resources in a more effective way with preserved requirements.

Objectives and methods

Objectives

The main objective of the thesis is to find a way to use the available resources, in the control systems, in a more efficient way with preserved requirements.

To propose solutions, for using system resources in a more efficient way, requires that sub objectives, methods and a main workflow are established. They are needed in order to deal with the large and somehow abstract problem of efficient resource usage. We will start by describing the sub objectives of the thesis.

The following sub objectives are derived from the main objective:

1. Identification of Red and Blue functionality in the system.
2. Finding out how Red and Blue functionality is handled under run-time.
3. Finding functionality in the current system that can be converted from Red to Blue.
4. Finding an optimal partitioning between the Red and Blue parts in order to decrease allocated resources without jeopardizing the requirements.
5. Definition, if possible, of a generic component for Red and Blue tasks.
6. Definition of design rules for Red and Blue functionality.
7. Definition of metrics to evaluate the changes in the design.
8. Validation and verification of the correctness of proposed solutions.

Methods

To achieve the objectives we have used the following methods:

1. A *case study* at Volvo in order to get familiar with the existing control systems and development process. The method includes examination of hardware, software and some requirement specifications.
2. A *literature study* to find out if someone has done researches on similar problems. The method includes searching for relevant literature and state of the art papers, describing how to identify hard and soft functionalities and how to handle them under run-time.
3. A *requirements specification study*.

Case Study

The purpose of the case study is to make appropriate preparations for preceding activities. It is essential to understand the system and the development process in order to fully understand the main objectives and the requirements for the thesis. It is also important to know the prerequisites to success in the preceding phase of the thesis: search of literature.

The case study was done at Volvo with some feedback from the employees. We looked at the development process, i.e. requirement specifications, design, implementation, verification and validation, in order to get an overview of the entire system.

Literature Study

The main goals with the literature study are to find relevant literature describing design issues and requirements specifications of real-time systems. Furthermore, we want to find some algorithms or methods for handling overload situations in real-time systems. We also want to find methods for off-line analysis of systems where soft and/or hard tasks, scheduled according to the fixed priority paradigm, runs in the background of a static off-line schedule.

To find papers, we have searched through the IEEE database and the Internet. Some of the search words we have used on the IEEE database include: real-time, soft, firm, scheduling, overload, QoS, etc. We then scanned through the results by reading the abstracts and the conclusions, if they were of interest and addressed the issues of the thesis then the entire paper was read. Finally if the papers were of interest we wrote down the main ideas presented in them (see chapter 2).

In addition to papers from the web we also collected papers, technical reports and books from the real-time systems design lab at Mälardalen University.

Requirement specifications study

To find functionality that can be converted from the red part to the blue part, in the current system, the requirements specifications must be studied. Functionality that may be converted includes soft functionality or functionality with timing constraints that result in an over-utilized schedule.

When we have searched for functionality that may be converted we have concentrated us on finding groups of functionality, it would be too time consuming to review the possibility of converting every single task.

Introduction to real-time systems

We will start this section by describing some fundamental issues concerning real-time systems, with focus on temporal validity. After that we will briefly comment the objectives to reflect the underlying problems with them.

Requirements

Real-time systems are systems where the correct functionality depends on both logical and temporal results. They are often thought of as fast systems, but a real-time system does not necessarily have to be fast.

Real-time computing systems may be used to support different type of applications. As an example a computing system may be used to interact with a control application consisting of a car, nuclear power plant etc. The real-time system may then be seen as a controller and a car or nuclear power plant etc. as the system to be controlled. Both the controller and the system exist in an environment, meaning that interaction between the controlled system and the environment may occur. How the controller handles the interaction depends on whether the environment or the controller should be responsible for activating events.

One approach is to let all incoming events from the environment activate parts of the controller. An example might be parts of a gearbox in a car. Whenever the driver changes gear, an event is sent to the controller. The controller then activates necessary functions to handle the gear change.

Another approach is to have the controller checking periodically whether any event has happened. An example may be the reading of a temperature sensor. The controller is used to periodically poll for changes in temperature.

Task model

The main task of a real-time system is to deliver correct results within a specified time, often before a so-called deadline (D). A deadline is the latest point in time in which the results should be delivered by a task. Each task is also characterized by its maximum execution time (C). Further attributes for tasks, depending on the task model, may be release time (Rt), period time (T), minimum interarrival time, etc.

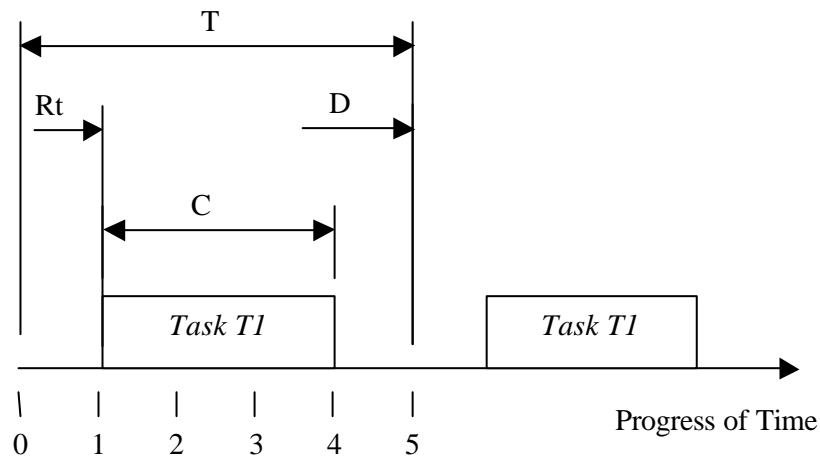


Figure 1 Relations between task attributes

To illustrate task attributes a task called T1 is shown in Figure 1. It has an execution time of 3 units and a deadline at time 5. The period time of T1 is 5. It is released 1 time unit into its period.

There are two fundamental paradigms concerning real-time systems. The first one is called the time-triggered (TT) paradigm. In a time-triggered system all tasks are periodic (defined by T) and driven by progression of time. The second type is called the event-triggered (ET) paradigm. In a system relying on the event-triggered paradigm, all tasks are activated by events e.g. by an interrupt, scheduling decisions are performed at run-time. In event driven systems a minimum interarrival time (MINT) between two consecutive activations of an event is needed in order to calculate response times (the time between an activation of a task instance and the finishing time of it), see the *Schedulability analysis* section for response time analysis.

Time-triggered systems are often easier to test than event-triggered because of their regularity and the absence of synchronization primitives for shared resources. Event driven systems on the other hand are more flexible than time-triggered, and may allocate system resources more appropriately.

Hard and soft systems

Real-time systems are often classified as either hard or soft. The main difference between the two types is that hard systems must guarantee that every deadline will be met. Whereas in soft systems there is usually no such guarantees i.e. tasks may miss their deadlines without serious consequences. However, in soft systems deadline misses are often restricted to a bounded number of times in a window of time.

Schedulability analysis

Both determinism and predictability are important facts in real-time systems. Determinism is important in order to establish an execution order (for testability) of tasks and predictability to avoid unexpected system behavior.

The temporal determinism is especially important and often results in an execution schedule. The schedule may be a table with information about what and when tasks should be executed (dispatched).

To build the schedule several techniques, using the knowledge of tasks importance etc, may be used. One of the most well known scheduling techniques is called Rate Monotonic (RM). In using the RM approach the period times of tasks are used to set their priorities. The lesser the period the higher the priority will be. Another scheduling algorithm, called earliest deadline first (EDF) [37], assigns the task with shortest time left to its deadline, the highest priority.

The two techniques described above may represent two different scheduling strategies namely static and dynamic priority scheduling. The main difference between them is that in static scheduling the order of task execution does not change in time whereas the opposite may go for dynamic scheduling. In dynamic scheduling a task may be activated and receive the highest priority of all hence interfering and changing the execution orders of tasks. I.e. in static scheduling decisions about what task to schedule is static and may be done pre-runtime, whereas in dynamic scheduling decisions about which task to schedule is done dynamically at run-time. Worth noting is that off-line scheduling and static scheduling is not the same thing. An off-line scheduling analysis should always be performed on a task set whether the final run-time algorithm is static or dynamic.

Whether scheduling is static or dynamic the utilization of the system is of great importance. The utilization factor gives information about the amount of processing time used. It is often calculated as the sum of all tasks execution times divided by their period times. Since tasks attributes such as execution times are specified as worst-case execution times, the utilization factor may be seen in two different ways. The first way is to see the utilization as allocated processing time based on worst-case execution times. The second way is to see it as the processing time the tasks actually use. The utilization factor based on worst-case execution times, are often (or always) higher than the actual utilization. This is due to tasks not running for their worst-case execution time at every invocation.

An overloaded system is one which has a utilization factor of greater than 1 hence there is a need for more processing time than the processor can handle. Overloaded systems are usually not preferred because of the difficulties of guaranteeing that tasks meet their deadlines. In an overloaded system one or more task may miss their deadline (at the instant when the overload occurs) due to that time is a limited resource.

In order to guarantee a task set, some kind of off-line analysis must be performed on it regardless of whether the final run-time dispatching is static or dynamic. During the off-line analysis a feasibility check of the task constraints is done in order to check if the set is feasible to schedule.

The first, and most intuitive, step is to check if the utilization (U) exceeds 1 (100%). A task set is for sure not feasible if U is greater than 1. The definition of

U is: $U = \sum_{i=1}^n \frac{C_i}{T_i}$, where C and T is the execution respectively the period time

and n is the number of tasks in the set. This is a necessary condition.

Several scheduling algorithms (e.g. RM and EDF) have so called scheduling bounds that can be checked under certain circumstances. E.g. for RM the task set

is schedulable if: $U \leq n(2^{1/n} - 1)$ [37]. The condition is sufficient but not necessary.

A third feasibility check, worth to mention, is the exact analysis. The exact analysis works on all priority assigned task sets, even on dynamic priority scheduling. In the exact analysis model every task is assumed to be activated simultaneously at the critical instant (a point in time where a task will suffer from maximum interference caused by other tasks in the system). In the analysis the worst-case response time for each task is calculated with an iterative formula:

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

R_i denotes the response time and C_i the worst-case execution time for task i . T_j and C_j denotes the period (or MINT for ET tasks) and the execution time for task j (Observe that the sum equals the execution demand of all higher priority tasks). If the calculated worst-case response time is shorter than or equal to the deadline for each task, the task set is feasible. The exact analysis shown here is only for the basic model and can be extended to contain blocking time, release jitter etc [42].

Run-time

Scheduling algorithms can be applied to static or dynamic scheduling and used off- or on-line. In each case the performance of any algorithm may differ depending on how and when it is used. For example the EDF algorithm is optimal in certain situations when applied to static scheduling but non-optimal in others. In fact, in overloaded systems the EDF does not provide any type of guarantees on which tasks will meet their timing constraints.[12]

Outline of the thesis

The remaining part of the thesis will concentrate on describing the results of the methods, and proposed solutions.

The first chapter was written in the beginning of our thesis as an independent paper, and it should be thought of as one as well when it is read, especially section 1.5 *Results from the case study*.

We will start by describing the architecture at Volvo. After that we give a short introduction to the real-time operating system and software development suite used at Volvo. Section 1.4 describes the software development process at Volvo. The first chapter ends with descriptions of our experiences from the case study.

In chapter 2 we discuss the contents of papers and the possibility of their contribution to this thesis. The most interesting parts of a known concept called *Weakly hard systems* is described in detail under section 2.4.2.1. The concept can be seen as the backbone of this thesis.

Chapter 3 deals with proposed solutions to effective usage of resources. In section 3.5.1 we describe a simple way of improving resource usage at Volvo. It is based on the possibilities of guaranteeing blue tasks in Rubus. The simple approach that

we present is not a part of, nor does it have any connections to, the weakly hard concept.

In chapter 3 we also describe and give examples of how support may be added to the different development phases in order to adapt the weakly hard concept at Volvo.

An important part of the thesis is chapter 2. that contains summaries of all essential papers that we encountered during our literature studies. It may be helpful to read the chapter in order to understand the discussions given further on in this thesis. Chapter 2 may also serve as a reference for those interested in further reading.

Chapter 1

1 A Case Study of a Vehicle Control System: The 220 E Wheel Loader

1.1 Introduction

As mention earlier the case study will, among other things, serve as a support to make appropriate preparations for preceding activities. It is important to know what the development process and the real-time system at Volvo looks like, and what possibilities we have to suggest changes.

The following sections describe the system architecture, development tools and the software development process at Volvo.

1.2 The architecture

The physical architecture in the vehicles at Volvo is embedded distributed real-time control systems. The system consists of several nodes, called ECUs (Electronic Control Units), which are connected through two busses: the CAN and the J1708 bus. The CAN (Control Area Network) is a fast bus and the J1708 bus is a slower bus (9600 bps). Both the CAN bus and the J1708 bus are used for data exchange between the ECUs in the system, the J1708 bus is a redundant bus that makes it possible to control the system with limited functionality if the CAN bus stops functioning. The amount of information sent on the J1708 bus is less than on the CAN. In addition to redundancy the J1708 bus is used by diagnostic service tools.

The particular vehicle we have studied at Volvo is the Wheel Loader. The wheel loader physical architecture consists of three ECUs (see Figure 2):

- Vehicle ECU (VECU) - Responsible for I/O management and controlling/observation of the vehicle, e.g. supervision of transmission oil temperature, etc.
- Instrument ECU (IECU) - The major responsibility of this ECU is the interaction with the driver, e.g. updating the driver display. It also handles the I/O in the driver cabin, e.g. gas pedal.
- Engine ECU (EECU) - Regulates the engine, transmission etc.

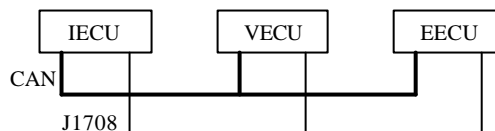


Figure 2 The physical architecture of the Wheel Loader.

We have studied the VECU in the Wheel Loader in more detail than the other two ECUs. Both the VECU and IECU are implemented on hardware platforms supporting three different storage types: EEPROM, Flash and RAM [17]. The memory in an ECU is very limited, e.g. 64 kb of RAM, 512 kb of Flash and 32 kb of EEPROM. The processors used are 20 MHz for the VECU and 16 MHz for the IECU (due to lower system load).

1.3 Rubus OS - A Real-Time Operating System

Rubus OS, used in Volvos entrepreneur machines, is an operating system supporting hard real-time and soft real-time services. Amongst other things the Rubus OS (hereafter called ROS) was designed to minimize utilization of RAM, minimize dispatching overhead and to be a kernel of minimal size.

The hard services in Rubus are referred to as 'Red services' and the soft ones as 'Blue services', interrupts are referred to as 'Green services'. A system running ROS can use either one of the services, or a combination of them.

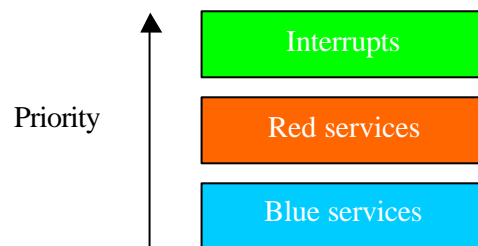


Figure 3 Priorities between services in Rubus

1.3.1 Red services

The Red services are executed according to the time-triggered paradigm and supported in the OS by threads. The Red services are preemptive, but preemption for them is seldom applied.

The following are possible attributes for Red service tasks:

- Release time
- Execution time
- Period time
- Deadline
- Precedence (see Rubus CC)

The threads building up Red services are executed according to an off-line generated schedule created with a formal specification language that is compiled by a tool called Rubus Configuration Compiler (RCC). The Red service of ROS has higher priority than the Blue services; hence Red services are always executed in favor to Blue.

1.3.2 Blue services

Unlike the Red services the Blue services are always pre-emptive and executed according to the event-triggered paradigm (with fixed priorities). The Blue services are priority based and scheduled thereafter. A total of 15 priority levels are supported. Executions of tasks with equal priorities are handled in a First-In-First-Out (FIFO) manner.

Executions of the Blue services are handled in background fashion, meaning that they may execute whenever the Red or Green part is idle.

The following are possible attributes for Blue service tasks:

- Priority
- Stack size

1.3.3 Interrupts

Rubus supports handling of interrupts. Interrupts are assigned highest priority in the system and may pre-empt red and blue serviced tasks. Interrupts are dispatched immediately at arrival. They are served by interrupt service routines which results in them having the highest priority in the system, even higher than the kernel.

1.3.4 Additional information

Tasks in ROS communicate using ports, message queues, semaphores and signals.

ROS supports several system modes e.g. start-up, drive and shutdown mode etc. Since each mode has a dedicated schedule, a schedule switch may occur during mode changes at run-time.

Different kernel parts actuate Red and Blue services respectively. The Red services rely on a basic clock with a user specified resolution (1ms at Volvo). The Blue services rely on a dedicated "blue clock" with resolution related to the basic clock (10 at Volvo, meaning that 10 basic clock ticks correspond to one blue tick).

Since Red services very seldom preempt each other, and if they do so they preempt each other in a predictable manner, one shared stack can be used for all of them. The Blue services must, due to unpredictable preemptive behavior, allocate one stack for each task.

1.3.5 Task schedule in the VECU

The number of Blue serviced tasks, in the VECU, is very low compared to the number of Red ones. The utilization of Red services varies in each mode. The drive mode, one of the modes with the highest number of red serviced tasks, has a utilization of approximately 76%, leaving 24% for Blue services. The actual (run-time) utilization of the Red services is considerable lower, because the schedule is based on worst-case execution times.

The total length of the off-line schedule for each mode, used to serve Red tasks, is 100 ms. The assigner (Volvo) wishes to retain the length of the schedule since it is stored in Flash memory. Increasing the length may result in need of more Flash memory.

Finding an execution schedule, for tasks with attributes as explained in previous sections, is a NP-Hard problem. The release times for Red tasks ranges between 0 - 90ms and are given in steps of 10ms e.g. 0, 10, 20 and so on. Given these release times, the tasks will be released in chains with intervals of 10ms. Period times of 10ms, 20ms, 50ms and 100ms are the only ones used for Red serviced tasks.

1.3.6 System partitioning

The functionality of the VECU is divided into categories as:

- *Input* – Handles reading of external data e.g. sensor values.
- *Vehicle* – Includes regulation, supervision, etc. This can be considered as the main functionality.
- *Output* – Produces actuator-data.
- *Logging* – Saves data for analysis.
- *Communication* – Handles communication between ECUs in the system.

Some of the categories have precedence relations between them, e.g. Input precedes all the others while the output executes last in the chains.

1.3.7 Rubus development tools

The complete Rubus product line consists of several tools for development of real-time systems. The following section describes some of the tools we have been using during our case study.

Rubus Visual Studio (RVS)

The RVS is a graphical tool that can be used as a helper to allocate system resources, such as CPU time and memory etc, pre-run-time. It can then be used to visualize the systems run-time behavior and for debugging purposes. Component creation, port connections and communication paths between tasks can be handled visually by ease with the RVS.

Rubus Configuration Compiler (RCC)

The RCC supports the use of components and composites described by a formal language. Each component can be described by a task with certain functionality and communication capabilities through ports. A composite is a component consisting of several components.

The RCC is an application responsible for compiling the formal design language, in order to create an off-line schedule, ports etc. used by ROS. Since ROS supports the use of several different modes, the RCC creates one schedule for each mode.

The Rubus Configuration Analyzer (RCA)

The RCA is an application that can be used to visualize the off-line scheduled of the Red services in different modes. Task executions and information such as release times, execution times and so on, are shown in an understandable manner.

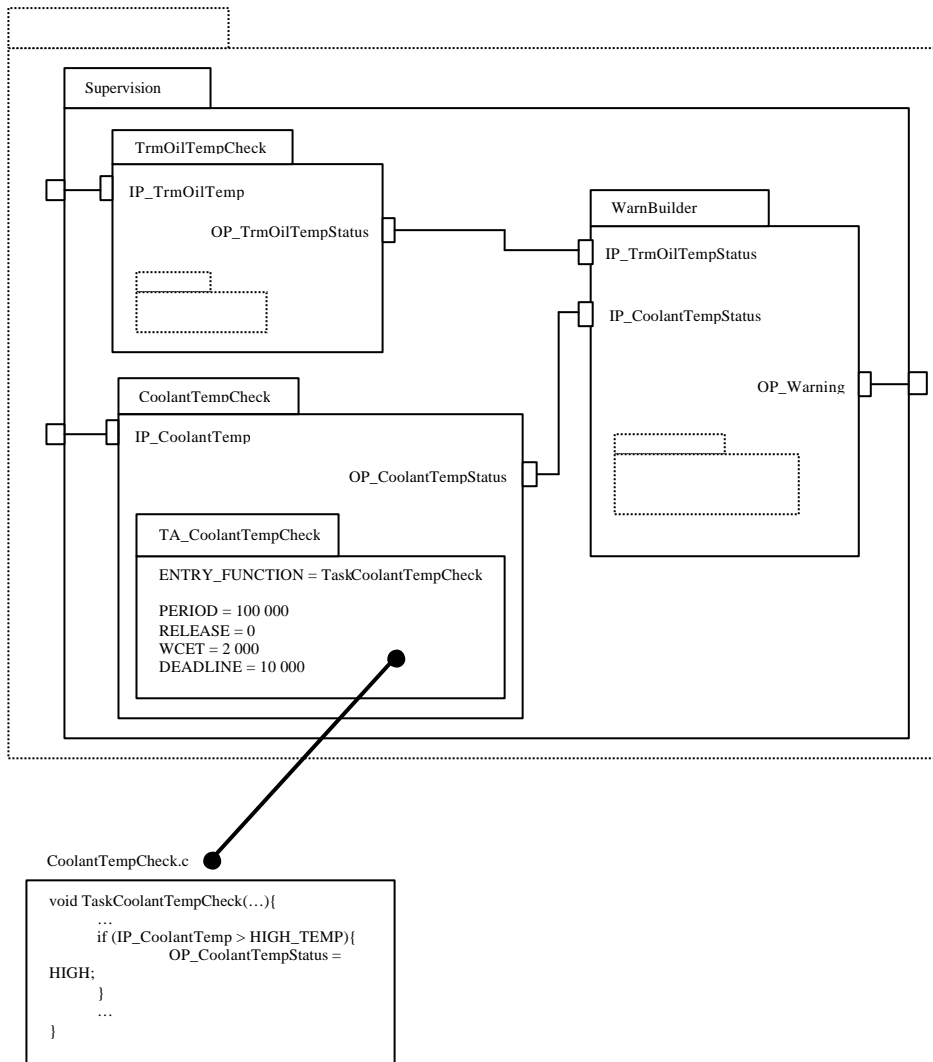


Figure 4 A graphical representation of components

1.4 Software development process at Volvo

The following five steps is a general description of the software development process at Volvo. The activities are performed in parallel (overlapping) whenever possible. Working in parallel often shortens the calendar time it takes to develop a system but the required effort in man-days still remains.

It is important to understand the activities in each step in the process, in order to propose changes with support to them.

Managing software requirements

This is the first step in the process. In this step the requirements specifications are received from the customer. Both documented and verbally given requirements form the inputs for further actions i.e. technical and economical reviews etc. The requirements consist of functional and temporal specifications. The outputs from this step are a list of approved requirements, functional and test descriptions. These outputs form the inputs to the next step in the process.

Design of software

The second step in the process concerns the design of the software. It involves activities such as environment checks (both target and development environment), placing of I/O functionality, identifying communication and modes, task and component identification. The outputs are formed of six parts, e.g system and subsystem design descriptions, communication and warning descriptions etc.

Implementation of Software

This is the third step in the development process. In this step the outputs from both the requirements managing and design steps are used as a base for the implementation activities. There are a total of six activities in this step e.g creating of OS-shell, addition of basic components, creation of functional components and data sets etc.

Verification and validation of software

This step is sometimes referred to as testing. The implemented software is tested both automatic and in interactive modes. The results are compiled into fault reports, deviation reports and test protocols.

Delivering of software

This is the last step in the process. The software package, deviation report and customer approval forms the initial input to activities such as preparation, updating, verifying and creation of the released software.

1.5 Results from the case study

During the case study some ideas and possible solutions to the objectives came up. The following sections briefly describe them.

1.5.1 Development process

We believe that the software development process at Volvo is carefully worked out to suit them. It is well documented since they are ISO certified.

Still, during the case study, we encountered some lacking issues such as:

- Traceability in the requirements documentation. It is hard to find references between documents.
- Some of the temporal requirements from the customer seem too tight for the desired purpose. It seems like the temporal constraints are too tight due to limited knowledge or insight in the real-time application, e.g. timing constraints of 0.5 seconds for visual interaction.
- Some information in documents is duplicated at several places. This may lead to ambiguous information especially when changes are made in them.
- The references between source code and the corresponding documents are sometimes hard to follow.
- Some source codes are insufficiently commented.
- Non re-entrant driver(s), e.g. the J1587 communication driver.
- Some precedence relations are not retained by Rubus CC (bug?).

Although we found some lacking issues, we must mention that the software development process at Volvo in general is very good and maintained very well. Some of the issues above may be due to our inexperience; still they are worth mentioning since most new employees will probably encounter them.

1.5.2 The communication

The J1587 (J1587 is the protocol on the J1708 bus) communication in the VECU is currently handled in software. Due to the high intensity of the interrupt (approximately 35% of the system utilization [15]), a suggestion would be to handle it entirely in hardware. This would save a lot of system resources that today are used on communication.

According to Volvo the J1708 bus will be removed and replaced with an additional CAN bus in the future. The CAN bus is today controlled with hardware, and hence, the problem will be negligible since interrupt interference and communication handling will be largely reduced.

1.5.3 Converting functionality from Red to Blue

One of our tasks is to find functionality that can be moved from the Red part to the Blue part. During the case study we have found some possible solutions to the task.

We have identified the following as convertible from Red to Blue:

- The visual interaction with the driver through the IECU. The timing constraints, given by the customer, seem too tight today. They are not realistic and the criticality of such information is usually not high.
- For tasks with a desired period greater than 100ms. Many tasks are today scheduled with a period of 100ms because it is the longest possible without extending the length of the off-line schedule.
- Tasks with short period times where the majority of the calculations are performed at events that seldom occur, can be broken up into two parts; one red polling task that triggers a blue task performing the calculation. This will decrease the utilization of the system while retaining the desired requirements. This will of course increase the utilization of the blue part, but the reduced WCETs for the polling tasks will decrease the utilization of the red part and thus it benefits the total system.
- Tasks where the desired period will increase the length of the red schedule i.e. non-multiples. Volvo will probably never use this, but we think it is worth mentioning it.

1.6 Revisiting the objectives

The case study gave us a view of the current architecture and development process at Volvo. Looking at the objectives once again, and commenting them with the knowledge we have from the case study, makes it easier to find correct literature.

The main objective of this thesis is to make it possible to increase the actual utilization of the system at Volvo. In doing this, the entire development process has to be considered and may be subject to changes. The system is currently mainly scheduled as a hard real-time system with timing constraints for the majority of tasks. Each task is scheduled with its worst-case execution time. Since resources are limited this approach will eventually lead to a state where no more Red tasks can be added to the system. Even though a time-triggered system can be scheduled close to 100% utilization, the actual utilization may be less than 50% [10], due to tasks not running for the worst-case execution time at every invocation.

To increase the actual utilization one can introduce Blue tasks into the system. This will lead to two possible scenarios concerning processor demand. Either the

demand is less than 100% or greater than 100%. Feasibility for a solution where the demand is less than 100% is pretty straightforward. We have to think about communications between Red and Blue tasks and blocking when using shared resources.

Another possible solution to increase the actual utilization is to schedule the processor(s) for more than 100%. This can only be achieved in the current system at Volvo by introducing parts of the functionality as Blue. Using Blue parts the total scheduled utilization can be greater than 100% and the actual utilization less than 100%. This situation is likely to happen when the system load is low or normal [11]. But the situation where the system load is high must always be considered. The high loaded system may lead to an overloaded situation [4], where not all tasks can meet their deadlines. In worst-case a domino effect may cause all tasks to miss their deadline [12]. In these situations a guaranteed quality of service (QoS) is desirable.

Chapter 2

2 Literature Study

This literature study is done as a part of our masters thesis at Volvo Construction Equipment (Volvo CE) at Eskilstuna. The main goals with the literature study are to find relevant literature describing design issues and requirements specifications of real-time systems. Furthermore, we want to find some algorithms or methods for handling overload situations in real-time systems. We also want to find methods for off-line analysis of systems where soft and/or hard tasks, scheduled according to the fixed priority paradigm, runs in the background of a static off-line schedule.

This chapter briefly describes the most relevant issues in the papers and books we have read during the literature studies. Some of the papers are discussed; advantages and drawbacks are given for the solutions/methods/algorithms in the papers.

We have tried to structure the chapter according to the development process at Volvo. First we describe papers that address the problems and issues of requirements specifications, then we give the papers that handle design issues, and finally we present papers concerning implementation.

2.1 *Opening discussion*

There is no paper that deals or implicit suggest solution of how to fulfill all of the objectives, that is the reason to why several papers are used to form solutions.

Requirements

Finding papers with specific information about requirements for real-time systems was not easy. Most papers contain general information about how to capture or write requirements to avoid misinterpretation. However, several papers describe common characteristics of good requirements such as correct, complete and consistent specifications.

A very important characteristic as we see it is also the realistic-ness of specifications, meaning that each specified requirement must be possible to fulfill. Both logical and temporal requirements may fall under this characteristic.

Design

When designing real-time systems, the classification of task as hard, soft, firm or a combination of them, should be considered. Several papers describe the difference between hard, soft and firm tasks. Also deciding whether to use the time-triggered or event-triggered paradigm, or a combination of them, should be considered with the respective paradigms advantages and disadvantages in mind.

Time-triggered properties include, easy verification of timing requirements, high degree of determinism, possible difficulties in constructing a schedule etc. Event-triggered properties include, easy handling of dynamic events, avoidance of

unnecessary polling, easy introduction of new tasks to the system etc. A time-triggered system may be less affected by faults in dynamic events e.g. a malfunctioning sensor, than an event-triggered one [9]. However the event-triggered paradigm may have higher processor utilization than a time-triggered.

We believe that it is possible to introduce guaranteed event driven tasks into the system at Volvo but we also believe that the advantages of such approach do not outweigh the testability and determinism of time-triggered approaches.

Scheduling algorithms

Several papers deal with the problem of scheduling aperiodic tasks at run-time. Most of the server algorithms, intended to serve aperiodic tasks, concentrate on enhancing the response times of the aperiodic tasks [18]. Since aperiodic tasks are often activated by some event, the server algorithms relies on the event-triggered paradigm of real-time systems.

A number of scheduling algorithms deal with overload. Overload is often assumed to happen due to aperiodic tasks arriving dynamically to the system. Overloaded systems should be avoided because they may cause catastrophic effects. When a system is overloaded some scheduling algorithms such as EDF may result in all tasks missing their deadline i.e. the so-called domino effect. Algorithms that handle overload are categorized into three types: Guarantee based, Best effort algorithms and imprecise computation algorithms.

An interesting concept called weakly hard systems is introduced amongst the guarantee-based algorithms. The concept allows system resources to be small by introducing a predictable and bounded allowance of deadline misses.

Response time analysis

In [22] a method to calculate response times for dynamically scheduled tasks executed in the background of a cyclic static off-line schedule is introduced. It could be used at Volvo to calculate response times for dynamic blue tasks. The method is developed to overcome pessimistic result of ordinary response time calculations in the described systems.

2.2 Requirements specification

Distinctions between requirements specification and design of real-time systems [3]

The authors of the paper describe the distinction between design and requirements specifications for real-time systems. Different views of looking at a real-time system when specifying requirements are described. The different views are divided into three perspectives:

1. Functional requirements, e.g. acquire sensor data.
2. Behavioral requirements, e.g. communication between tasks
3. Timing requirements, e.g. max response time.

The design should model the needs expressed in all of the above views. At design time it is no longer possible to think of the system as one with unlimited amount of resources.

One major drawback with this paper is that it is fairly old. We believe that it is interesting to look at the system as described, especially the fact that infinite system capacities can be thought of in capturing the requirements. Unfortunately we think it is unrealistic now days to do so, since economical matters must be considered.

Writing good requirements: A requirements working group information report [20]

In [20] the author addresses common problems encountered in writing requirements. Several problems, including making bad assumptions to over specifying the requirements, are described and suggestions are given in order to eliminate them. The use of grammar for specifications is also described.

The paper contains a lot of useful information and examples of both good and bad ways of writing requirements. Nothing special is mentioned about requirements concerning real-time systems, so the paper can serve as a source of common guidelines for writing requirements.

System requirements specification for real-time systems [14]

In [14] the author discusses how requirements specifications can be effectively formulated. It is a pretty old paper with a lot of common knowledge about the usage of requirements as an instrument in the development process. One thing that differs this paper from many modern ones, concerning requirements, are the attributes (which back then were about 4 ones, depending on the methodology) that makes up good requirements, especially the one about *machine processibility*. It is stated that good requirements only can be achieved if the management is fully acquainted with the realities of the project. Although it is old we think it is an important statement.

Real-time programming specification [15]

In [15] the author suggests guidelines that may be useful in implementation of real-time systems.

An interesting discussion is given about *programming specifications*. They are a set of documents defining the system in programmable terms by translation of functional requirements to technical descriptions. One of the suggested purposes (except being an aid in system design and providing an organized means of programmer to programmer communication etc) of the programming specifications, is that they should permit effective use of inexperienced personnel. Meaning that they should be written in an understandable language and containing only useful information permitting the inexperienced personnel from making significant system decisions.

Software engineering, theory and practice [19]

[19] is a book about software engineering in which the different step in a development process are described. A chapter about capturing the requirements describes some characteristics of good requirements as:

- *Correctness* – Stated without errors.
- *Consistent* – No conflicting or ambiguous requirements must exist. Two inconsistent requirements cannot be satisfied simultaneously.
- *Complete* – Contains all states, state changes, inputs, constraints and so on.
- *Realistic* – Is it really possible to do the things stated in the requirements?
- *Needed* – Is some feature really necessary.
- *Verifiable* – One must be able to write tests that demonstrate that requirements have been met. Terms such as easy, rapid quick, user-friendly etc. must be avoided since they have different meaning, thus hard to verify, to different persons.
- *Traceable* – Can each function be traced to a set of requirements?

These characteristics are something that we must have in mind if we are going to suggest a change in the requirements at Volvo CE.

2.3 Design

Real-time systems are usually classified according to the importance of missing a deadline. A real-time system can be: *hard, soft or firm* [18][35]:

- In a *hard* system (or task) it is considered a system fault if a task misses its deadline, and hence produces a late result. The violation of the deadline can result in catastrophic consequences, either economic or human.
- A *soft* system (or task) is one in which the response time is important but it is acceptable to miss deadlines occasionally. Producing a late result is acceptable.
- A *firm* system (or task) is one in which deadlines can be missed occasionally, but producing a late result is worthless.

The term occasionally (in the classification of real-time systems above) is so ambiguous that it has no practical meaning for a specification for systems that may tolerate a degree of missed deadlines. The term hard, on the other hand, is very restrictive because no deadlines are allowed to be missed, and the term soft is too relaxed. To overcome these problems the term *weakly hard* is introduced to bridge the gap between the soft and hard term [35]:

A weakly hard system (or task) is one in which deadlines can be missed in a predictable way, the number of and the pattern of allowed deadline misses must be specified. It is hard in the sense that it has to be guaranteed beforehand that the specification will be met.

Examples of hard and soft activities that may be presented in control applications are given below [18].

Hard activities include: Sensory data acquisition, detection of critical conditions, actuator serving, Low-level control of critical system components, and planning sensory-motor actions that tightly interact with the environment.

Soft activities include: The command interpreter of the user interface, handling input data from the keyboard, displaying messages on the screen, representation of system state variables, graphical activities, and saving report data.

When designing a system it is important to not make tasks harder than they actually are (i.e. functionality are often considered to be harder than it is) but to make them as soft has possible. Making tasks harder than they are will waste resources and the available resources will end up earlier than necessary.

The set of rules that determines the order in which tasks are executed is called a scheduling algorithm. A scheduling algorithm can be [18]:

1. *Preemptive* or *Non-preemptive*.
2. *Static* or *dynamic* – Whether the scheduling decisions are based on fixed or dynamic parameters.
3. *Off-line* or *on-line* – Whether the scheduling algorithm is used off-line or on-line.
4. *Optimal* or *Heuristic* – An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. If it tends toward but does not guarantee to find the optimal schedule, it is said to be heuristic.

According to this classification Volvo has:

1. A “non-preemptive static off-line scheduling algorithm” for the red part, and
2. A “Preemptive static on-line scheduling algorithm” for the blue part.

There are some desirable features of real-time system [18]:

1. *Timeliness* – deliver result within predefined time.
2. *Design for peak load* – should not collapse under transient overload situations.
3. *Predictability* – must be able to guarantee the behavior in advance.
4. *Fault tolerance* – single hardware and software failure should not cause the system to crash.
5. *Maintainability* – should be easy to perform modifications.

Application Design Issues [18]

The authors of [18] describe some important issues related to the design and the development of complex real-time applications requiring sensory acquisition, control, and actuation of mechanical components.

They also show how time constraints, such as periods and deadlines, can be derived from the application requirements.

All complex control applications that require the support of a computing system can be characterized by the following components:

1. The *system* to be controlled – A plant, a car, etc.
2. The *controller* – The computing system.
3. The *environment* – The external world.

Depending on the interaction between the controlled system and the environment, three classes of control systems can be distinguished:

1. *Monitoring systems* – Do not modify the environment, only process the sensory data and displays it to the user.
2. *Open-loop control systems* – Interacts with the environment. The actions made do not strictly depend on the current state of the environment.
3. *Feedback control systems* – Have frequent interactions with the environment in both directions, i.e. the actions produces by the actuators are strictly dependent on the changes in the environment.

Interrupt handling

In [18] the authors mention three possible techniques to reduce the interrupt interference on the application tasks and still perform I/O operations with the external world.

Approach A

The most radical solution is to disable all external interrupts. The application tasks are responsible for handling the peripheral devices through polling the registers of the interface boards.

Pros: The direct access to I/O devices allows great programming flexibility and eliminates delays caused by the drivers execution.

Cons: The main disadvantage is low processor efficiency on I/O operations due to the busy-wait of the task while accessing the device registers.

Approach B

All external interrupts are disabled. Unlike the previous solution, the devices are not directly handled by the application tasks but are managed in turn by dedicated kernel routines, periodically activated by the timer.

Pros: The application tasks do not need knowledge of the low-level details. The approach also eliminates the unbounded delays due to execution of interrupt drivers.

Cons: The system is not that efficient during the I/O operations, due to the fact that the interrupts are disabled and the devices must the handled with busy-wait. Since the device handling routines are part of the kernel, it has to be modified when some device is replaced or added.

Approach C

Leave all external interrupts enabled while reducing the drivers to the least possible size. The only purpose of each driver is to activate a proper task that will take care of the device management.

Pros: The major advantage of this approach with respect to the previous ones is to eliminate the busy-wait during I/O operations. The unbounded delays introduced by the drivers during tasks executions are also drastically reduced, so the task execution times become more predictable.

Cons: The system may suffer heavy overload from malfunctioning sensors (not mentioned in the book).

The issue of interrupt handling is very important at Volvo CE, due to that they have fairly high interrupt interference (approximately 35 % of system utilization [15]) caused by the communication handling etc.

The best approach to reduce the interrupt interference at Volvo CE would be to handle the communication entirely in hardware and not in software as it is handled today, e.g. the time consuming stuffing and un-stuffing, that is handled in software today, could be handled entirely in hardware. Then the communication hardware would only need to generate interrupts at package arrival, or it could even be polled to entirely eliminate the interrupt interference. We know that the hardware solution has been a subject for prior thesis that was never finished.

The second best solution would be to apply *Approach C* above. (This is almost the solution implemented at Volvo today, but the ISR routines are currently not minimized.) This would lead to less interference from the high priority ISR (Interrupt Service Routine/Driver), handling the interrupt from the communication, and a more predicible system. The time spent handling the communication will of course not vanish, but will be moved to the schedule in the form of an event-triggered task, and hence, it will be considered during the analysis of the system.

This approach can also be modified to a non event-triggered solution (which we think would be preferred at Volvo CE). In this approach the only purpose of the ISR is to store the received packages in a buffer that a periodic task later can read and perform proper actions for each packet.

The design of real-time systems: from specification to implementation and verification [2]

In [2] a design methodology for distributed real-time systems is described. Static scheduling and a globally synchronized time-base are assumed. Characteristics such as predictability, testability, fault tolerance, system design, systematic decomposition, evaluability are explained. Furthermore they explain how timing behavior for dependable tasks (e.g. communication from tasks that read sensors to the tasks that control actuators i.e. transaction) can be handled by: precedence constraints and an upper bound for the completion time of each transaction. They also discuss issues such as estimated and calculated execution times and their relations. They suggest comparing the calculated and estimated execution times to prevent too pessimistic results.

This is an paper with some issues that could be useful for us, especially the parts about timing issues. The design method (differs from the current one used at Volvo CE) in itself, would lead to too large changes at Volvo CE.

Design issues in real-time [7]

In [7] the author discusses issues involved in design of a real-time system. To predict how a system will behave in all possible circumstances, determinism is discussed and defined. In a deterministic system a unique set of outputs and next states of the system can be determined depending on the systems current state and the inputs to it. Determinism is usually divided into two categories: *event determinism* and *temporal determinism*. Event determinism means that the next states and outputs of the system are known for each set of inputs which triggers events. Temporal determinism implies that the response times for each set of outputs are known.

For a hard real-time system it is important to fulfill both event and temporal types of determinism.

Real-time system design: a temporal perspective [13]

A set of techniques that can be used by a designer in choosing artifacts such as period time, deadline and priorities etc, are summarized in [13]. Examples are given for both time and event driven systems. The main approach in [13] is to start with implicit timing constraints called performance specifications (e.g. maximum steady state error), and mapping them into system-level timing constraints (e.g. maximum end to end latency from a sensor to an actuator). The system-level timing constraints are then transformed to task level timing attributes (e.g. constraints, such as period deadline and phases, amenable to schedulability analysis).

This is a way of breaking down timing requirements from a high abstraction level to tasks. Although the paper contains general and sometimes, to us, known information, it might be helpful to consider these facts when end to end constraints are to be handled.

Tolerating sensor timing faults in highly responsive hard real-time systems [9]

In [9] the author discusses hard real-time systems with a combination of time and even triggered tasks. It is stated that time-triggered activations are less affected by sensor timing faults than is event-triggered. However, the processor utilization for time-triggered systems are often low, sometimes unacceptably low, compared to event-triggered ones.

So either you have an time-triggered system which tolerates sensor timing faults fairly well but with a low utilization factor, or an event-triggered systems with high utilization that may be unreliable when sensors fails.

The author introduces a task splitting model (TSM) in order to activate tasks at reliable intervals and preserve good processor utilization. The TSM transforms one task into three parts. The TSM is basically used to determine the correct time slot for activation of tasks, e.g. to avoid peaks of interrupts caused by sensor faults.

We believe that this might be a way of controlling burst arrivals of interrupts in the system at Volvo CE, but it is out of our thesis scope so it will be left as a reminder to the developers of the real-time operating system Rubus.

2.4 Implementation

2.4.1 Server algorithms

Many real-time control applications require both aperiodic and periodic tasks. Periodic tasks are typically time-driven and execute critical control activities with hard timing constraints. Aperiodic tasks are usually event-driven with hard or soft timing constraints depending on the specific application.

When dealing with such hybrid task sets, the main objective of the kernel is to guarantee the schedulability of all critical tasks and provide a good average response time for the soft and non real-time activities. If event-driven aperiodic tasks must be guaranteed, some assumptions on the arrival rate must be made (define a minimum inter-arrival time, *MINT*), to be able to perform the off-line guarantees.

In systems with hybrid task sets, server algorithms may be used to enhance the response time of aperiodic requests through the use of a server task (the server). The main task of the server is to preserve capacity (execution time), which can be used later to serve aperiodic tasks requests with.

There exist two categories of server algorithms: *Fixed-priority servers* and *Dynamic priority servers*. The difference between fixed-priority servers and dynamic priority servers is the current scheduling policy they use, or actually how the priorities are assigned to the tasks in the system. Fixed-priority servers are based on fixed-priority scheduling and dynamic priority servers are based on dynamic priority scheduling. All the dynamic priority servers described here are scheduled by the earliest deadline first algorithm.

2.4.1.1 Fixed-priority servers

Below we give different Fixed-priority server algorithms as described in [18].

Background scheduling

Background scheduling is a very simple method to handle soft aperiodic activities in the presence of periodic tasks. The soft aperiodic tasks are scheduled in the background and executed when there are no periodic instances ready to execute.

Background scheduling can only be adopted when the periodic load is not high and the aperiodic activities do not have stringent timing constraints. For high periodic loads, the response times of the aperiodic requests can be too long for certain applications. The major advantage of the background scheduling is its simplicity; only two queues are needed to implement the scheduling mechanism.

The Blue services in Rubus are executed according to the background scheduling technique.

Polling Server (PS)

The average response times of the aperiodic requests can be improved with respect to Background Scheduling through the use of a server.

The server is a periodic task, whose responsibility is to service aperiodic requests as soon as possible. The server is characterized by a period, T_s , and a computation time, C_s , called the server capacity. In general the server is scheduled with the same algorithm used for the periodic tasks, and when it is active it serves

aperiodic requests within the limit of its server capacity. If an aperiodic request arrives after the server has suspended, it must wait until the beginning of the next polling period.

Deferrable Server (DS)

The Deferrable Server is a service technique introduced to improve the average response time of aperiodic requests with respect to polling service.

As the Polling Server, the DS algorithm creates a periodic task (usually having the highest priority) for servicing aperiodic requests. Unlike polling, DS preserves its capacity if no aperiodic requests are pending upon the invocation of the server. The capacity is maintained and can be used to service aperiodic requests until it is exhausted. At the beginning of any server period, the capacity is replenished at its full value.

DS provides much better aperiodic responsiveness than polling, since it preserves the capacity until it is needed. One disadvantage with DS is that the server can defer (even though it is the highest priority task and ready to run, it does not run until an aperiodic request arrives) its execution when it could execute immediately, and causing lower priority tasks to miss their deadlines even if the task set was schedulable. Such an invasive behavior of the DS results in a lower schedulability bound (~ 65.2 %) for the periodic task set.

Priority Exchange (PE)

The priority exchange algorithm is a technique introduced for servicing a set of soft aperiodic requests along with a set of hard periodic tasks, with a better scheduling bound compared to DS.

Like DS, the PE uses a periodic server (usually at a high priority) for servicing aperiodic requests. However it differs in the manner which the capacity is preserved. Unlike DS, PE preserves its high-priority capacity by exchanging it for execution time of lower-priority periodic tasks, if there are no aperiodic requests pending. Thus, the server capacity is not lost but preserved at a lower priority, which later can be used to service aperiodic requests.

With respect to DS, PE has a slightly worse performance in terms of aperiodic responsiveness but provides a better schedulability bound for the periodic task set. One major drawback with PE is its implementation complexity, this complexity is due to the work required to manage and track the priority exchanges.

Sporadic Server (SS)

The Sporadic Server algorithm enhances the average response time of aperiodic tasks without degrading the utilization bound of the periodic task set.

The SS algorithm creates a high-priority task for servicing aperiodic requests and, like DS, preserves the server capacity at its high-priority level until an aperiodic request occurs. However, SS differs from DS in the way the capacity is replenished. The capacity is not replenished periodically, as in DS, instead it is replenished after it has been consumed by aperiodic task execution.

There are two advantages with this algorithm; Firstly, it has a better utilization bound of the period task set, compared to DS. Secondly, the SS can be considered as a periodic task during the schedulability analysis.

Slack Stealing

The Slack Stealing algorithm is an aperiodic service technique which offers substantial improvements in response time over the previous service methods (PE, DS and SS).

Unlike the earlier mentioned methods, the slack stealing algorithm does not create a periodic server for aperiodic task service. Rather it creates a passive task, referred to as the Slack Stealer, which attempts to make time for servicing aperiodic tasks by “stealing” all the processing time it can from the periodic tasks without causing their deadlines to be missed.

The main idea behind slack stealing is that, typically, there is no benefit in early completion of the periodic tasks. The periodic tasks are normally scheduled with RM. When an aperiodic request arrives, the Slack Stealer steals all the available slack from periodic tasks and uses it to service the request.

There are two major drawbacks with the Slack Stealing algorithm which makes it improper to use in a real-life application. First, the overhead introduced by the large amount of calculations done when recomputing the slack function, may be very large. Finally, the size of the table holding the precomputed slack function can be too large for practical implementations.

E = Excellent, G = Good and P = Poor

Table 1 Evaluation summary of fixed-priority servers.

	Performance	Computational complexity	Memory Requirements	Implementation Complexity
Background Service	P	E	E	E
Polling Server	P	E	E	E
Deferrable Server	G	E	E	E
Priority Exchange	G	G	G	G
Sporadic Server	G	G	G	G
Slack Stealer	E	P	P	P

2.4.1.2 Dynamic priority servers

With respect to fixed-priority assignments, dynamic scheduling algorithms are characterized by higher scheduling bounds, which allow the processor to be better utilized, increase the size of aperiodic servers, and enhance aperiodic responsiveness.

The dynamic Priority exchange servers below are given in [18].

Dynamic Priority Exchange (DPE)

The Dynamic Priority exchange server can be viewed as an extension of the Priority Exchange server, adapted to work with a deadline-based scheduling algorithm.

The main idea of the algorithm is to let the server trade its run-time with the run-time of lower-priority periodic tasks (under EDF this means a longer deadline) in case there are no aperiodic requests pending. In this way, the server run-time is only exchanged with periodic tasks but never wasted (unless there are idle times). It can later be reclaimed when aperiodic requests enter the system.

The DPE server behaves like a periodic task and hence the schedulability analysis is not affected; thus, the periodic task set can be guaranteed using the classical Liu and Layland condition:

$$U_p + U_s \leq 1$$

Where U_p is the utilization factor of the periodic tasks and U_s is the utilization factor of the DPE server.

Dynamic Sporadic Server (DSS)

The Dynamic Sporadic Server extends the Sporadic Server to work under a dynamic EDF scheduler.

The DSS differs from other server algorithms in the way the capacity is replenished; it is not replenished at its full value at the beginning of each server period but only when it has been consumed. The times at which the replenishments occur are chosen according to a replenishment rule, which allows the system to achieve full processor utilization.

The DSS server behaves as a periodic task and the schedulability analysis is not affected. Thus the periodic task set can be guaranteed using the classical Liu and Layland condition, mentioned above under DPE.

One problem with the DSS is; when the server has a long period, the execution of the aperiodic requests can be delayed significantly. One solution to this problem is, of course, to use a Sporadic Server with a shorter period. This solution, however, increases the run-time overhead of the algorithm. A second solution to this problem is addressed in the Total Bandwidth Server.

Total Bandwidth Server (TBS)

The main idea behind Total Bandwidth Server is to assign a possible earlier deadline (compared to the DSS) to each aperiodic request. The earlier deadline assignment reduces the delays of the aperiodic requests that can occur in the DSS, when the server period is long. The assignment must be done in such a way that

the overall processor utilization of the aperiodic load never exceeds a specified maximum value U_s .

When an aperiodic request arrives at time $t = r_k$, it receives a deadline:

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

Once the deadline is assigned, the request is inserted into the ready queue and scheduled by EDF as any other periodic instance. As a consequence, the implementation overhead of this algorithm is practically negligible.

The TBS server, as earlier mentioned dynamic priority servers, behaves as a periodic task and, hence the schedulability analysis is not affected.

Earliest Deadline Late Server (EDL)

The Earliest Deadline Late Server can be viewed as a dynamic version of the Slack Stealing algorithm. The basic principle adopted by the EDL is to use available slack of periodic tasks for advancing the execution of aperiodic request. When there are no aperiodic activities in the system, periodic tasks are scheduled according to the EDF algorithm.

The EDL gives a better performance than the TBS through more complex algorithms. An important property of EDL is that in any interval $[0, t]$ it guarantees the maximum available idle time, i.e. it automatically allocates a bandwidth $1-U_p$ to aperiodic requests. The response times achieved by this method are optimal, so they cannot be reduced further.

The overhead introduced by computational complexity of the algorithm can be too high and, hence, the algorithm may be of little practical interest.

Improved Priority Exchange Server (IPE)

The Improved Priority Exchange adopts the main idea of EDL, to develop a less complex algorithm that still maintains a nearly optimal behavior, and modifies the DPE server using the idle time of an EDL scheduler.

The heavy computations of the idle times can be avoided by using the mechanism of priority exchanges. With this mechanism, in fact, the system can easily keep track of the time advanced to periodic tasks and possibly reclaim it at the right priority level. The idle times of the EDL algorithm can be precomputed off-line and the server can use them to schedule aperiodic requests, when there are any, or to advance the execution of periodic tasks.

There are two major advantages in this approach. First, a far more common efficient replenishment policy is achieved for the server. Second, the resulting server is no longer periodic, and it can always run at the highest priority in the system.

Worth noting is that there are only rare situations in which the optimal EDL server performs slightly better than IPE. Performance evaluations have shown that the performance difference between EDL and IPE is so small that it can reasonably be considered as negligible for any practical application.

Improved TBS (TB*)

The key idea of the Improved TBS is to shorten the deadline assigned by the TBS as much as possible, still maintaining the periodic tasks schedulable.

The process of shortening the deadlines can be applied recursively to each new deadline, until no further improvement is possible or after a maximum number of steps defined by the system designer for bounding the complexity.

E = Excellent, G = Good and P = Poor

Table 2 An evaluation summary of dynamic priority servers.

	Performance	Computational Complexity	Memory Requirements	Implementation Complexity
BKG	P	E	E	E
DPE	G	G	G	G
DSS	G	G	G	G
TB	G	E	E	E
EDL	E	P	P	P
IPE	E	G	P	P
TB*	E	P	E	G

2.4.2 Overload management

An overload condition is a situation in which the computational demand requested by the task set exceeds the time available on the processor, and hence not all tasks can complete within their deadlines.

If the operating system is not designed to handle overload situations, the effect of a transient overload can be catastrophic. There are even cases (for the EDF scheduling algorithm) in which the arrival of a new task can cause all the subsequent tasks to miss their deadlines, in a domino effect. [18]

There are several scheduling algorithms that deal with overloads, in the real-time literature. Most of the algorithms assume that the overload occurs due to aperiodic tasks that arrive dynamically into the system, and hence, the schedule is feasible before run-time. Other algorithms allow the schedule to be infeasible before run-time, to make use of the system resources in a less pessimistic way, and assume that overload occurs due to underestimated WCETs of the periodic tasks.

When tasks can be activated dynamically and overload occurs, it is crucial for the system to distinguish the importance of the task and the time constraints. In general, the importance of a task is not related to its deadline or its period. In order to specify importance of a task, a value is usually assigned to it. The value can then be used by the system to make scheduling decisions.

In the absence of aperiodic tasks that can cause overload in the system, overload can still occur due to underestimated WCETs of the periodic tasks. To handle the overload in such systems one or more periodic task must be skipped. Occasional skips of tasks will usually not degrade the performance of the system too much, if the skips are performed in a predictable way. Most often off-line analysis is performed on the system for these algorithms to guarantee a minimum level of QoS (Quality of Service).

Depending on the strategy used to predict and handle overloads, the overload scheduling algorithms can be divided into three categories [18]:

1. *Guarantee-based algorithms* – The guarantee mechanism is based on WCET assumptions. This means that the guarantee of hard tasks is achieved at the cost of reducing the average performance of the system.
2. *Best-effort algorithms* – A best-effort scheduling algorithm tries to “do its best” to meet deadlines, but there is no guarantee of finding a feasible schedule. This algorithm performs much better than the guarantee-based algorithm in the average case, due to that tasks are only aborted under real overload conditions.
3. *Algorithms based on imprecise computations* – When time and resources are not enough to complete the computations within the deadline, there may still be enough resources to produce approximate results that may at least prevent a catastrophe.

2.4.2.1 Guarantee-based algorithms

As mentioned earlier, the guarantee mechanism in guarantee-based algorithms is based on WCET assumptions. This means that the guarantee of hard tasks is achieved at the cost of reducing the average performance of the system.

Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints [30]

The algorithm deals with a mixed set of tasks: periodic tasks with complex and simple constraints, soft and firm aperiodic, and sporadic tasks.

The algorithm uses an off-line scheduler to reduce the complexity of transforming complex constraints into the EDF model. At run-time an extension of the EDF, two level EDF algorithm, ensures feasible execution of tasks with complex constraints in the presence of additional tasks or overload. Thus this is an integration of off-line and on-line scheduling.

The method presented in the paper is an extension of the Slot Shifting algorithm described in [39]; it is based on the off-line transformation of slot shifting but provides a new run-time algorithm.

The paper assumes that the system is a distributed system and that no task migration takes place under run-time.

The paper also presents an algorithm for handling off-line guaranteed sporadic tasks, to reduce the pessimism about future sporadic arrivals and improve response times and acceptance of firm aperiodic tasks. This is done by using an interference window; “We do not know when a sporadic task will invoke its instances, but once an instance arrives we do know the minimum time until the arrival of the next instance.” This information is used in the acceptance test of firm aperiodic tasks.

This approach is fairly interesting, but it assumes that overload occurs due to aperiodic requests. The idea of handling off-line guaranteed sporadic tasks to reduce the pessimism about future sporadic arrivals, is useful and will be considered when we design our solution at Volvo.

Dual Priority Scheduling [32]

Dual Priority Scheduling is an algorithm for scheduling tasks with soft deadlines in real-time systems containing periodic, sporadic and adaptive tasks with hard deadlines. The main goal of the algorithm is to enhance the responsiveness of the soft tasks while guaranteeing the execution of the hard tasks.

Static priority servers such as Deferrable Server and Priority Exchange algorithm are unable to make spare capacity available as anything other than a background opportunity for soft tasks. Dynamic priority server methods, such as dynamic Slack Stealing, can identify such spare capacity through more complex algorithms with greater computational costs.

The Dual Priority Scheduling is an alternative method of identifying spare capacity. In the Dual Priority Scheduling there is a range of unique priorities split into three bands: Upper, Middle and Lower. Under execution hard tasks execute at either the upper or lower band priority. Upon release each hard task assumes its lower band priority, however, at a fixed time offset from release, the priority of the task is promoted to the upper band. At run-time, other tasks with soft deadlines are assigned priorities in the middle band. Thus soft tasks execute in preference to hard tasks that are yet to undergo priority promotion.

This method has the advantage of low overhead and simple implementation.

Minimizing Aperiodic Response Time in a Firm Real-Time Environment [21]

The paper addresses the problem of scheduling hybrid task sets consisting of firm periodic tasks and soft aperiodic requests, where periodic tasks can occasionally skip one instance once in a while, according to a predefined skip parameter, which controls the quality of service of the task. The authors propose an algorithm that exploits the spare time saved by the skipped instances to minimize the response time of aperiodic requests. Periodic tasks are scheduled according to EDF and, in order to minimize the response times, aperiodic tasks are handled by an optimal server algorithm, the TB* server. Schedulability analysis is performed to derive an off-line feasibility test.

The algorithm can also be used to schedule slightly overloaded systems.

This paper also mentions that the general problem of scheduling firm periodic tasks that allow occasional skips has not received much attention in the real-time literature.

Additionally the authors say (referenced to a paper by Koren and Shasha [34]) that the worst-case for skippable tasks is when the model is *deeply red*: “A system is *deeply red* if all tasks are synchronously activated and the first $s_i - 1$ instances of every task t_i are red.” For this reason, all results in the paper are proved under the assumption above.

This method introduces too much overhead to be considered as interesting to implement at Volvo CE, due to the TB* algorithm.

Overload Management in Real-Time Control Applications Using (m,k)-Firm Guarantee [6]

In [6] the author discusses a way of discarding selected task instances as an overload management technique. It is based on the so called (m,k) firm guarantee model (meeting m deadlines out of k consecutive instances of a task). An instance of a task is classified as either mandatory or optional. For the mandatory instances, deadlines are guaranteed. The optional instances are not guaranteed to meet their deadlines. Whether an instance is classified as mandatory or optional depends on the values m and k .

The author works out a scheduling policy capable of deterministically guaranteeing when and where tasks will miss their deadlines. The information can then be used to carefully discard task instances in order to reduce the effective utilization of the system.

Automobile control applications are given a motivation to the problems considered in the paper. It is stated that most control system can tolerate a few deadline misses. Furthermore, the author describes how an optimal (under the deeply red assumption) control law can be derived and a numerical example is given.

One of the main drawbacks with this method is that there is no way of deciding how the missed instances should be distributed over the window (the k consecutive instances); the method will implicitly evenly distribute the missed instances over the window. Furthermore, the first instance of each task is always considered as mandatory (regardless of the values m and k), and hence, the critical instant will always be at time zero and degrading the schedulability of the task set.

Weakly hard real-time systems [4][36]

This is a summary of a concept called weakly hard real-time systems as described in [36] [4]. The authors describe methods to specify and guarantee tasks that may miss their deadlines a defined number of times. The authors states that a system where some deadlines can be missed (in a predictable way), allows the systems resources to be smaller than it would be for meeting all deadlines. They give three examples of such systems; *computer-driven automatic control system, monitoring system* and *multimedia systems*.

We believe that this concept is the one of most interest and it is also the concept that is most suited to the current architecture at Volvo with respect to the current system. It requires just a few changes to the task model and some support to the development process.

Some parts and definitions are intentionally left out because it is a wide concept where not everything is of our interest. This summary deals only with the most important facts of weakly hard systems, applicable to the system at Volvo.

Introduction

As mentioned in the introduction to this paper, real-time systems are often classified as either hard or soft. The main difference between the two types is that hard systems must guarantee that every deadline will be met, whereas in soft systems there is usually no such guarantees i.e. tasks may miss their deadlines without serious consequences.

Our case study showed that Volvos system, although modeled as hard, also contains soft functionalities. So we would need a way of loosening constraints for certain tasks and still guaranteeing a minimal quality of service.

According to [36] the occasional loss of some deadlines in certain systems can be tolerated, due to negligible or non-serious consequences. A great advantage with not having to meet every deadline is that such systems allow smaller system resources than for meeting all of them. Additionally, such a system is more cost effective whilst guaranteeing a reasonably good level of service.

Furthermore according to [36] there is often a trade-off between two objectives in real-time systems, on one hand meeting all deadlines and on the other maximizing the use of system resources. To reconcile the two objectives, the concept of weakly hard systems is introduced in [36].

Following the definition in [36], a weakly hard system (hereafter called WHS) is formally a concept based on hard real-time systems that are weakened to include a precisely bounded and predictable distribution of lost deadlines. It was originally introduced to bridge the gap between hard and soft systems because the term hard was considered very restrictive and the term soft as too relaxed.

The system at Volvo can adopt the concept of WHS for certain functionalities that does not necessarily have to be modeled as soft nor hard, but somewhere in between.

Specifications

In real-time systems there is often an interest in calculating response times for tasks in order to decide whether a task will meet its deadline or not. Moreover it is the maximum response time (R_i) that is of greatest interest and it is often calculated at a critical instant i.e. a point in time where a task will suffer from maximum interference caused by other tasks in the system.

In WHS however, computing the maximum response time R_i is not enough. The response time must be calculated for every invocation of a task in the hyper period. The reason for the need of response time calculations at every invocation is the fact that WHS tolerates a bounded and predictable number of deadline misses using patterns (see next section). At different invocation in the hyper period (least common multiple of the tasks periods) a task will not be invoked simultaneously with all higher priority tasks leading to lesser interference than at a critical instant.

So a task may meet its deadline at different invocations even though it does not do so in the worst-case.

Patterns

WHS characterize tasks with two patterns. The first one is called m -pattern, which is a worst-case execution pattern, and the second is \bar{m} -pattern being an observed execution pattern at run-time. The patterns simply characterize the sequences of missed and met deadlines of tasks.

For example a m -pattern of 10101 shows that a task will meet its deadline 3 times in the worst-case (at first, third and fifth invocation) and miss its deadline two times. Hence a met deadline is denoted by a 1 in the pattern, and a missed by 0.

Constructing the patterns shows the need for response time calculations at different invocations of a task, in fact the patterns are constructed as sequences of every invocation in a hyper period at the level of the task H_i (see section analysis of weakly hard systems) .

WHS temporal constraints

The concept of WHS allows tasks to miss their deadlines in a bounded and predictable way. To support the concept there are four possible types of constraints for tasks. Each of the constraints considers a window of m consecutive task invocations. The four constraints are formally defined as follows:

- Meet any n in m deadlines denoted by $\binom{n}{m}$ i.e. in any window of m consecutive invocations of a task, there are at least n invocations in any order that meet the deadline.
- Meet row n in m deadlines denoted by $\langle n \rangle_m$ i.e. in any window of m consecutive invocations of a task, there are at least n consecutive invocations that meet the deadline.
- Misses any n in m deadlines denoted by $\overline{\binom{n}{m}}$ i.e. in any window of m consecutive invocations of a task, no more than n deadlines are missed.
- Missing row n in m deadlines denoted by $\overline{\langle n \rangle_m}$ i.e. in any window of m consecutive invocations of a task, there are no n consecutive invocations that miss the deadline.

The different constraints are needed because different applications suffer more or less depending on whether deadlines are missed consecutive or spread out over a window of time.

The reason to why the constraints of met or missed deadlines are specified as above and not as any ratio with percentage (e.g. 90 % of deadlines should be met), is that specifications that uses percentage has no bounds to any period of time. Moreover a percentage notation does not say anything about how deadlines may be missed, consecutive or non- consecutive.

As an example meeting 90% of all deadlines could be fulfilled by meeting 90 deadlines out of 100. It would also be fulfilled by meeting 900 deadlines out of 1000. The latter case may meet the first 900 deadlines and miss the last 100. Depending on the application, the scenario could lead to unexpected results which are undesirable in real-time systems.

***k* – Sequences**

In WHS *k*-sequences are used to analyze the properties of *m*-patterns. A *k*-sequence is just a sequence **W** of *k* symbols (in particular, any subsequence of a *m*-pattern is a *k*-sequence). Subsequences are as the name implies, just parts of the whole sequence.

In order to understand the meaning of the used sequences in WHS, the following example shows *k*-sequences and subsequences.

Example:

If a *m*-patterns looks like 0101110011 then a sequence **W** where *k* = 4 could look like 0101 or 1110 and so on.

If we have a sequence **W** = 0101 the sub sequence **W**^{3,2} would be 101 i.e. the subsequence starting at index 2 and of length 3.

In WHS two *k*-sequences are of special interest namely the ones built up of only 1's (called good sequence) and the ones built up of only 0's (called bad sequence). Moreover it is important to consider possible wrap around of *k*-sequences e.g. in cyclic schedules.

k-sequences are often compared and classified as harder or softer than others. The definition in WHS is as follows:

*“If a *k*-sequence **W** has 1's in the same position as another *k*-sequence **W'**, but **W** has additional 1's in the place where **W'** has 0's, then **W** is harder than **W'**”.*

As an example, following the definition above, $\mathbf{W} \preceq \mathbf{W}'$ denotes that **W** is harder than **W'**. It is proven in [36] that if a *k*-sequence satisfies a constraint, any other *k*-sequence that is harder will also satisfy it.

Constraints over an alphabet

In WHS the temporal constraints (as explained earlier) are reformulated in terms of *k*-sequences as follows:

$$A = \binom{n}{m}, \bar{A} = \overline{\binom{n}{m}}, R = \langle \binom{n}{m} \rangle, \bar{R} = \overline{\langle \binom{n}{m} \rangle} \quad ; \quad k \geq m$$

This is done in order to decide whether a *k*-sequence satisfies any of the constraints. The definition is also used to define relationship between different constraints over *k*-sequences.

As an example, the notions harder and weaker are used as relationship over constraints as shown below:

- Given two constraints **I** and **I'**. Then **I** is harder than **I'** (i.e. **I'** is weaker than **I**) if all *k*-sequences that satisfy **I** also satisfy **I'**. The relationship is denoted by $\mathbf{I} \preceq \mathbf{I}'$.

The notion harder and weaker may be used as a support to build up sets of sequences that satisfies a constraint and subsets of constraints that a task satisfies. It is worth noting that according to [36] there exist constraints that are not comparable.

Fragility

When a k -sequence satisfies a constraint, it is interesting to distinguish between fragile and non-fragile k -sequences.

A fragile k -sequence is one that satisfies a constraint, but if a 1 is switch to a 0 it no longer satisfies it. Hence a non-fragile k -sequence satisfies a constraint even though a 1 is switched to a 0.

The distinction between fragile and non-fragile sequences can be used in a run-time mechanism to decide whether a task must be activated or if it can be skipped in order to let other tasks execute.

Characteristics of relationships between constraints

Several relationships between constraints are presented and proved in [36]. The most important ones are summarized below:

- Meeting at least n deadlines out of m is equivalent to not missing more than $m - n$ in m .
- Not missing more than n deadlines in a row is independent of the window size ' m '.
- If a k -sequence satisfies an A constraint, it will also satisfy less demanding constraints i.e. one which requires less 1's or one that extends over a wider window m .

Algorithm

In order to test whether a k -sequence satisfies a constraint, the paper presents an algorithm. The algorithm deals with four sub problems regarding satisfaction of WHS constraints. They are:

1. Given a k -sequence, the algorithm determines if it satisfies a constraint. This is mainly a problem of determining off-line whether a task satisfies a weakly hard constraint. The presented solution is simply a counting algorithm that scans the sub sequences and check whether the constraint is met or not.
2. Given a k -sequence, the algorithm determines the subset of constraints that it satisfies. This part deals with the problem of off-line determining the set of constraints satisfied by a task.

The third and fourth parts relates to problems of on-line checking whether meeting the deadline at next task invocation is required assuming that:

- All future invocations meet the deadline, or
- The future behavior is given by the \mathbf{m} -pattern.

The last parts of the algorithm are performed to evaluate if it is possible to put in better use of the time required for a task. It could for example be used for improving the responsiveness of soft tasks if the algorithm shows that the next deadline for a task can be missed still satisfying the weakly hard constraint.

Analysis of weakly hard systems

The basic process model presented in [36] is a system made up of purely periodic tasks with the following constraints:

- T_i (period time)
- D_i (deadline)
- I_i (weakly hard constraint)
- C_i (worst-case execution time)

Where i denotes a task priority and deadlines are assumed to be equal or lesser than the period.

Given the process model above, it is stated in [36] that the whole pattern of task invocations is repeated at every least common multiple (LCM), referred to as hyper period H .

For a task with priority i the pattern of both release times and response times, is in fact repeated every hyper period at level i (H_i). Analyzing a task within H_i (i.e. at LCM for all tasks with equal or higher priority than i) requires less computation than analyzing it within H .

An important observation given in [36] is that worst-case scenarios for weakly hard constraints do not necessarily happen around a critical instant. It is possible that a task misses more consecutive deadlines at a later point than it does at a critical instant. This fact shows the importance of computing the m -pattern and analyze it for all invocations within H_i .

Since the whole pattern of task invocations is repeated at H there is a possibility that overrunning tasks (i.e. a task executing longer than expected) causes interference with invocations in successive hyper periods. The paper presents a way of determining whether such interference within a level i will occur. The solution is to calculate if a task is bounded i.e. if the utilization given by:

$$U_i = \sum_{t_j \in \text{hep}(t_i)} \frac{C_j}{T_j}$$

is equal or lesser than 1. If a task is bounded then it does not overrun over H_i , so:

“if the utilization at level i of a task is smaller or equal to 1 then the last invocation of the task in H_i finishes before the start of the next H_i ”

The authors of [36] also presents a way of calculating worst-case finalization times for tasks at different invocations. The presented solution uses a so-called idle time at level i , that is the amount of time a processor can be used by tasks with lower priority than i . The idle time at level i is then summed with the total execution times for a task and with the total interference caused by tasks with equal or higher priority than i .

Although the presented solution is a nice one, it is not well suited for the system at Volvo. We suggest (see section 3.5.3.2) using a different formula, developed by us, at Volvo to calculate finalization times.

Extensions to the basic process model

The basic process model is extended in [36] to include arbitrary deadlines i.e. $D_i \geq T_i$ to consider interference from previous invocations. Offsets and blocking times are also considered in the extended model, but the most important extension is the introduction of strongly hard sporadic tasks in the model. The latter model extension corresponds closely to the system at Volvo.

An important observation in the model with sporadic tasks is that the worst-case scenario for a mixed task set does not correspond to the worst-case scenario for mixed weakly hard tasks. Meaning that applying the worst-case scenarios for a model with periodic and sporadic tasks is not enough for a similar model with weakly hard tasks.

The worst-case scenario for weakly hard tasks does not happen when the sporadic tasks start at $t = 0$, neither when they arrive at their maximum rate. The following example is given in [36] to demonstrate the problem.

Table 3 A task set with weakly hard constraints

r_i	T_i	D_i	C_i	I_i
s_1	15	3	3	$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$
t_2	10	6	5	$\begin{pmatrix} 2 \\ 3 \end{pmatrix}$

Consider the task set given in Table 3. In Figure 5 the sporadic task arrive at $t = 0$ and re-arrives at the maximum rate 15. The sporadic task only interferes (causes t_2 to miss a deadline at time 6) with the first invocation of t_2 in every hyper period. Despite the miss, the constraint $\begin{pmatrix} 2 \\ 3 \end{pmatrix}$ for t_2 is satisfied.

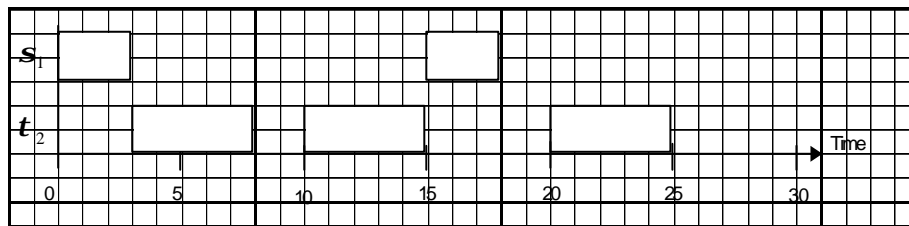


Figure 5 Sporadic arriving at $t=0$ and re-arriving at maximum rate

In Figure 6 the sporadic arrives at $t = 4$ and re arrives at its maximum rate. The sporadic causes t_2 to miss two deadlines i.e. the constraint for t_2 is unsatisfied.

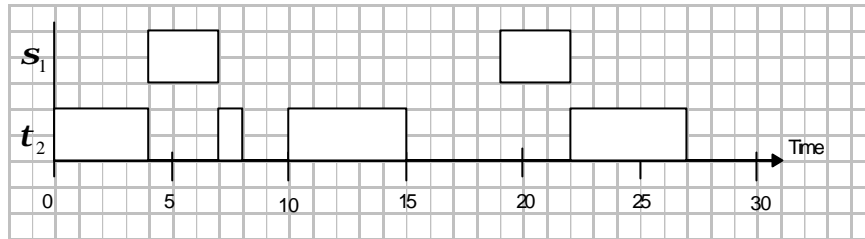


Figure 6 Sporadic arriving at $t=4$ and re-arriving at maximum rate

In Figure 7 the sporadic arrives at $t = 0$ and re arrives at a slower than maximum rate. Again t_2 misses two deadlines because of sporadic interference.

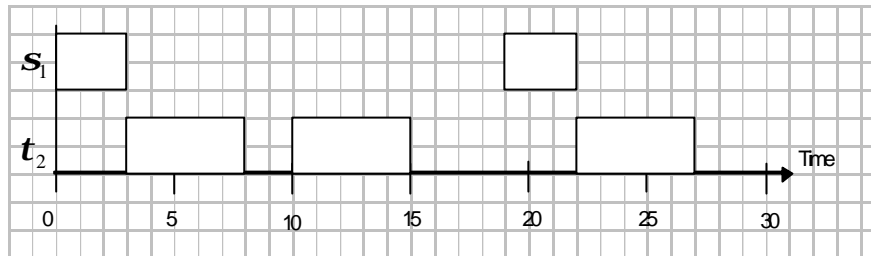


Figure 7 Sporadic arriving at $t=0$ and re-arriving at slower than maximum rate

The example suggest that exact analysis is computationally intractable as it will require to test all possible arrival times for the sporadic tasks and to consider all possible phasing between the sporadic tasks. The author of [36] suggests a method to cope with the problem. It is a sufficient but not necessary method. The worst-case scenario for a task T_i happens when all higher priority tasks (sporadic tasks included) are released together with T_i and when the sporadic tasks arrives at their maximum rate. This consideration has to be applied to each invocation of a task with weakly hard constraints in order to calculate its worst-case response time.

Optimal priority assignment

Neither deadline monotonic priority (DMA) nor rate monotonic priority assignment is optimal (except for strongly hard tasks with $D_i \leq T_i$) for weakly hard systems. In [36] the author presents an optimal priority assignment algorithm that maintains the DMA for strongly hard tasks and swaps priorities of pairs of tasks until the optimal priority assignment is found or until an unschedulable task set is determined.

Using the optimal priority assignment is highly recommended for weakly hard tasks.

On-line mechanism

In [36] Bernat gives two very simple algorithms to use on-line to enhance the performance of the system. When using these on-line algorithms a minimum QoS can be guaranteed to the system. The algorithms can also be used to enhance the responsiveness of soft tasks.

The basic idea behind the described on-line algorithms is very simple: if a task does not have to be invoked for the Weakly Hard constraints to be satisfied, put the time required for the task in a better use.

As mentioned the paper describes two algorithms where the second is an extension of the first one. Below we will give the main concept of the two algorithms where the second one will only be mentioned and discussed briefly.

Next fragile

This problem is introduced to model the case where we have an observed \overline{m} – pattern of a task and, assuming that the task will meet the deadline when it is invoked, we want to determine if the deadline can be missed without violating its Weakly Hard constraints. If the Weakly Hard constraints are not violated the time required for the task could be put in better use, else the task must be scheduled in order to meet its Weakly Hard constraints.

The formulation of the problem is as follows: “Given a k -sequence $?$ that satisfies a constraint $I \in \Gamma$, consider the $k + 1$ -sequence $? = ? \cdot 1$. The question is whether $?(k + 1)$ is a fragile element.” [36]

We will only give a very basic version of the algorithm for checking the $\binom{n}{m}$ constraint, but we encourage the reader to look at the corresponding chapter in [36] for more details and to get the algorithms for the other constraints.

The algorithms are to be used while the k -sequence is being build. Therefore Bernat has divided them into two components: the first component checks, given a state and a k -sequence, if the next element is fragile, the second component updates the state that defines the k -sequence for the following test. It is important to note that the tests can only be applied to k -sequences that already satisfy the constraints, therefore it is assumed that $k = ?.m$.

The first test (check if the next element is fragile) is really simple for the $\binom{n}{m}$ constraint: it is only required to test whether there are $n-1$ 1’s in the last sub- $m-1$ -sequence of $?$. If so, the last element is fragile.

```
BEGIN Next_Fragile_Any
  S = the number of 1's in the sub-m-sequence of ?'.
  IF s = n-1 THEN
    Fragile = TRUE
  ELSE
    Fragile = FALSE
  END IF
END
```

If the element is fragile it is necessary to keep it as a 1 in the k -sequence in order to guarantee the satisfiability of the constraint. However, if it is not fragile we may decide to make it a 0. When this decision has been taken it is required to update the state variables.

BEGIN Update_state_NFA

S = the updated number of 1's (in the sub-m-sequence of ?').

Add the last element to the k -sequence (1 or 0 depending on the decision).

Shift the k -sequence one step to the right.

END

Next fragile with μ -patterns

This algorithm is an extension of the one mentioned above, the *Next fragile* algorithm. In the *Next fragile* algorithm it was assumed that, in the future, all deadlines would be met. However, this may of course not be the case if the \underline{m} -pattern has 0's. Now we consider the case where we have the "observed" \overline{m} -pattern of a task, and its "calculated" \underline{m} -pattern. Now we determine whether the current instance can miss its deadline without violating the Weakly Hard constraint.

The formulation of the problem is as follows: "Given the sequence $\tau, \tau', \tau'' \in \tau^+$ so that $l(\tau') = k$ and $\tau = \tau' \cdot \tau''$, and given any constraint $I \in \Gamma$ such that $\tau' + \tau$. The question is whether $\tau(k+1)$ is a fragile element. In other words, does $\tau^{k+1 \rightarrow 0} + \tau''$ " [36]

In this case τ' corresponds to the "observed" \overline{m} -pattern of the task, and τ'' corresponds to the "calculated" \underline{m} -pattern. τ is then the combination of the observed behavior of the task and the prediction of the future.

This problem is a bit more complex than the previous one, however, the algorithm is still very simple as it follows the same idea as the previous one. In order to check fragility of the $k+1$ element of τ it is required to test the sub- $\tau.m$ -sequences that overlap with the $k-1$ element.

This algorithm has the advantage of better performance than the previous, but there is a slightly larger overhead introduced to the system, which may be a problem.

Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips [34]

The paper addresses the problem of scheduling occasionally skippable periodic tasks in an overloaded real-time system. The authors show that making optimal use of skips is NP-hard. They also give two algorithms, called Skip-Over Algorithms (one for EDF and one for RM), which exploit skips and scheduling bounds for both of them.

In the model each skippable task is divided into instances where each instance occurs during a single period of the task. Every instance of a task can be either red or blue; a red task instance must complete before its deadline, and a blue task instance can be aborted at any time. Each task is also characterized with a skip parameter s , $2 \leq s \leq 8$, which gives the tolerance to missing deadlines. The distance between two consecutive skips must be at least s periods.

The paper gives a necessary condition for the feasibility of a skippable task set based on cumulative processor utilization, and it also shows that it is impossible to give a sufficient condition.

Then the authors give a sufficient condition for the feasibility of a skippable task set scheduled according to EDF, based on the processor demand criteria.

The first algorithm that they present is the Read Task Only (RTO) algorithm. The algorithm is a lazy algorithm in the sense that it never attempts to schedule a blue task instance, i.e. it never does any work unless it absolutely has to. The red tasks are scheduled according to EDF. This algorithm is optimal in the deeply red model, i.e. all instances but the last ones of a task are considered red (see [21] for the definition of a deeply red model).

The second algorithm presented is the Blue When Possible (BWP) algorithm, which is much more flexible than the RTO algorithm. The main goal of the algorithm is to use idle time to good use, and hence, it tries to schedule blue task instances whenever this does not prevent any red task instance from completing. The red task instances are scheduled according to EDF.

Finally the Rate-Monotonic RTO (RM-RTO) is presented which, like RTO, is a lazy algorithm that schedules red task instances only. The difference between the two RTO algorithms is that the RM-RTOs underlying scheduling algorithm is a fixed priority scheduler where priorities are given according to the Rate-Monotonic scheme, whereas RTO is based on the EDF scheduling policy.

One interesting feature with the presented schedulability analysis algorithms in the paper are that they manage to test the schedulability of the task set even when the utilization factor exceeds 100 %.

Combining (n, m)-Hard deadlines and Dual Priority Scheduling [24]

The paper addresses the problem of effectively scheduling soft tasks while guaranteeing the behavior of hard tasks. To achieve this, the authors present an algorithm based on the Dual Priority Scheduling which increases the capacity for soft tasks, and therefore the effectiveness of the real-time system, by assigning (n, m)-hard temporal constraints to the periodic tasks. The approach reduces the gap between dynamic priority and fixed priority scheduling with the goal of reducing the average response time of soft tasks.

The approach described in the paper allows hard tasks to run with (1, 1)-hard performance when there are no soft tasks, but revert to (n, m)-hard in the presence of soft tasks. By doing this, the spare capacity (bandwidth) is preserved until a time at which soft tasks can utilize it.

The paper also mentions different approaches to increase the responsiveness of soft tasks, e.g. two promotion selection strategies, one static and one dynamic. In the static promotion selection (SPS) the scheduler promotes a pre-defined subset

of task invocations, while in the dynamic promotion selection (DPS) the promotion test is computed on-line.

According to the authors, the approaches are easy to implement and they introduces very low scheduler overhead.

Enhanced Fixed-Priority Scheduling with (m, k)-Firm Guarantee [23]

The paper addresses the problem of scheduling task sets with (m, k) constraints under the fixed priority paradigm. In the approach each task is divided into two sets: mandatory and optional (corresponds to red and blue in [4]). The mandatory jobs are scheduled according to their pre-defined priorities, while optional jobs are assigned to the lowest priority.

The method in the paper overcomes the potential problems with the technique proposed in [6], e.g. that the first job of every task is always assigned as mandatory (which leads to a critical instant at time 0 for the tasks), or that the partitioning algorithm depends solely on the ratio of m over k of each task (i.e. the period and execution times does not affect the distribution of each task), and finally, the method in [6] always distributes the mandatory jobs evenly over the window which is not always desirable (in the view of schedulability).

The authors present two methods in the paper:

1. The first algorithm improves evenly distributes (m, k)-patterns by judiciously “rotating” the (m, k)-pattern. The key idea is to reduce the execution interference between tasks.
2. The second algorithm is a GA (Genetic Algorithm) which improves the schedulability of the task set. The authors present a fitness function to minimize the execution interference between tasks. The fitness function is only a estimated metric for the task set schedulability, i.e. best effort, due to that an accurate execution interference analysis would be too computational costly, as the fitness function is calculated very often in a GA.

The authors of the paper prove that their solution space is a super set of that in [6].

2.4.2.2 Best-effort algorithms

As mentioned earlier a best-effort scheduling algorithm tries to “do its best” to meet deadlines, but there is no guarantee of finding a feasible schedule. These algorithms performs much better than the guarantee-based algorithm in the average case, due to that tasks are only aborted under real overload conditions.

Best-effort algorithms are of little interest for us, due to that there is no minimum QoS guarantee. The reason to why they are given here is that some of them present interesting and useful techniques.

Rate Modulation of Soft Real-Time Tasks in Autonomous Robot Control Systems [31]

The paper describes a scheduling approach adapting soft real-time tasks to available capacity during overloads or occasional high-priority computations.

In the paper the authors say that most of the robot control architectures (an architecture with lot of sensors and actuators) are soft real-time, and a single deadline miss will hardly determine a global system malfunctioning. Rather, occasionally deadline misses will simply affect overall quality in robot task execution.

The algorithm presented is a graceful degradation policy. It assumes that the system is divided in two task sets: one hard and one soft. The tasks are then scheduled with the Rate Monotonic Scheduling (RM) and priorities are fixed a-priori.

During overload the algorithm makes a controlled relaxation of the soft-task set utilization factor to manage the situation. In general, the relaxation is achieved by means of a reduction of the rate $f_i = 1/T_i$ for some or all of the soft tasks. To make the relaxation controlled each task must be characterized by a range of admissible rates, this is achieved by applying a range of allowed periods (T_{\min} and T_{\max}) to each soft-task. Several different variations of admission control algorithms are presented.

Value-Driven Multi-Class Overload Management [26]

The paper presents an overload management method for handling hard, soft and aperiodic tasks scheduled under EDF. The method is an extension of the Overload Resolution (OR), presented by the authors in an earlier paper, which is called OR-ULD/BC (Overload resolver – Utility Loss Density driven with Bias Control).

The paper presents a value driven overload management algorithm with a bias control mechanism, which attempts to bias the execution among transaction classes such that minimum completion constraints are satisfied. In addition to the usual constraints for firm transactions a constraint specifying the minimum completion ratio is given, an importance value for each class is also defined. To ensure robustness of the system the bias control treats the transactions classes fairly; if a class has relatively low completion ratios that class importance value is increased, i.e. a weight added to the value is increased as the completion ratio decreases.

Two types of overloads are identified: the first one is due to resource shortage and the second due to lack of a feasible schedule. The first type of overload is

handled by performing admission control and the second type is handled by a bias control mechanism.

The scheduler architecture consists of four components (Dynamic admission Controller, Transaction scheduler, Overload resolver and Dispatcher) and requires a dedicated processing element for scheduling.

The following is also worth mentioning: “Most scheduling algorithm developed for soft and firm real-time systems lack the ability to enforce constraints on the upper limit of misses. Without such enforcement, unbounded consecutive time constraint violations may occur.”

Managing Soft QoS in Distributed Systems [29]

The paper addresses the problem of soft Quality of Service (QoS) requirements for multimedia applications. The authors provide a framework that does not require users or application developers to have detailed knowledge of the resources needed and resource scheduling and allocation techniques in use.

The framework presented in the paper only gives best-effort guarantees, the resources are dynamically adjusted when needed by the application. This requires the ability to detect that a QoS requirement is not being satisfied, determine why it is not being satisfied and then determine correct adaptive steps.

To support the strategy, mentioned above, the authors of the paper have built a specific architecture. The architecture needed is fairly complex and contains e.g. software probes. Even though the architecture is complex, the authors say that experimental results have shown that the introduced overhead is minimal.

Admission Control and QoS Negotiations for Soft-Real-time Applications [33]

The paper presents a new admission control algorithm that exploits the degradability property of soft real-time applications to improve the performance of the system. The algorithm is based on setting aside a portion of the resources as reserves and managing it intelligently so that the total utilization of the system is maximized.

In general admission control represents the problem of deciding if an application can be supported on a resource. Soft real-time applications allow for graceful adaptation of the application QoS and therefore are able to have acceptable performance with reduce resource utilization. This can be used by the admission control process to decide if an application can be admitted even if the resource is congested.

2.4.2.3 Algorithms based on imprecise computations

As mentioned earlier the main idea for these kinds of algorithms is: when time and resources are not enough to complete the computations within the deadline, there may still be enough resources to produce approximate results that may at least prevent a catastrophe.

The idea of imprecise computations can be very useful in control functionality. The mandatory part will produce a sufficient result and the optional part will only improve the result.

An incremental approach to scheduling during overloads in real-time systems [5]

The paper proposes a scheduling framework, based on the EDF priority assignment, for a real-time environment that experience dynamic changes in workload. The dynamic changes in workload are caused by new task arrivals or tasks that leave the system after finishing their execution. The framework is capable of adjusting the system workload in incremental steps under overloaded conditions such that the most critical tasks in the system are always scheduled and the total value of the system is maximized.

Each task in the system is divided into two parts: a mandatory part and an optional part. The timely answer available after the mandatory part can be improved by execution the optional part, the optional part is only executed in non-overload conditions. Furthermore each task is periodic and preemptive, has a value associated to it, independent and has no precedence constraints.

The process of selecting which optional task to discard while maximizing the value of the system requires the exploration of a potentially large number of combinations (the approach is based on an on-line Incremental Server). Since this process is too time-consuming to be computed on-line, an approximate algorithm is executed incrementally whenever the processor would otherwise be idle.

Experimental results have shown that a very few steps of the incremental algorithm is needed to be executed to achieve performance with nearly optimal results.

Scheduling Periodic Jobs That Allow Imprecise Results [35]

The paper addresses the problem of scheduling periodic jobs in hard real-time systems that support imprecise computations. Timing faults are avoided in such systems by making available intermediate, imprecise results of acceptable quality, when results of the desired quality cannot be produced.

Each task is logically decomposed into two parts: a mandatory part followed by an optional part. The mandatory part must be completed to produce an acceptable result. The optional part refines and improves the result produced by the mandatory part. The error in the result is further reduced as the optional part is allowed to execute longer.

Two workload models of imprecise computations for two different types of applications are presented. The two workload models differ from one other on whether the optional parts need ever be completed.

Depending on the kinds of undesirable effects caused by errors, applications are classified as Type *N* or Type *C*.

For Type *N* applications, only the average effect of errors is observable and relevant. Examples of Type *N* applications include image enhancement and speech processing. The paper describes a class of priority-driven, preemptive algorithms that guarantee to produce feasible schedules of Type *N* jobs. These algorithms assign high priorities to all tasks initially and schedule them on the rate-monotone basis. The priority of every task is lowered after it has executed its mandatory part in order to produce an acceptable result. The remaining part of the task, the optional part, is scheduled when the mandatory parts of all tasks have completed.

For Type *C* applications, errors indifferent periods have cumulative effects, making it necessary to generate precise results sometimes. Examples of this type of applications include tracking and control. The authors consider the case when at least one task among every *Q* consecutive tasks in every job is required to complete. The length-monotone algorithm can be used to schedule this type of applications. The paper gives a necessary condition for a set of jobs to be schedulable using the length-monotone algorithm.

One of the most serious disadvantages with the presented algorithms for the scheduling of Type *N* applications is that they may fail to achieve zero average error even when the total utilization factor *U* is equal to or less than 1.

2.5 Automation tools

In the academic world attributes to tasks are usually assigned manually by the engineers of the real-time system. This is possible for smaller systems, however, in the industry systems usually consists of several hundreds of tasks, assigning task attributes manually to these systems is not desirable. Furthermore, there is a great maintenance problem in such large systems.

Having some kind of automation tool, which helps the engineers in assigning task attributes, can attack these problems.

Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling [27]

Off-line scheduling and fixed priority scheduling (FPS) are often considered as complementing and incompatible paradigms. The paper shows how off-line scheduling and FPS run-time scheduling can be combined to get the advantages of both – the capability to cope with complex timing constraints (off-line scheduling) and flexibility (FPS). It presents a method to analyze the off-line schedule and derive an FPS task set with FPS attributes priority, offset and period such that the run-time FPS execution matches the off-line schedule. It assumes that the off-line schedule has been constructed off-line for a set of tasks to meet their complex constraints.

FPS has been widely studied and used in numerous of applications, mostly due to its simple run-time scheduling, small overhead and good flexibility for tasks with incompletely known attributes. However, temporal analysis of FPS algorithms generally focuses on providing guarantees that all deadlines will be met. The actual start and completion times of the tasks are usually not known and depend largely on the run-time events, and hence, compromising predictability.

Off-line scheduling for time-triggered systems, on the other hand, provides determinism as all times for task executions are determined and known in advance. In addition complex constraints can be solved off-line, such as distribution, end-to-end deadlines, precedence relations, jitter and instance separation. As all actions have to be planned before startup, run-time flexibility is lacking.

Instead of only enhancing either FPS or off-line scheduling, the method provides for a combination, such that benefits of either scheme are accessible to the other.

The method presented in this paper can be of great use for our thesis. One of the objectives is to move red functionality to the blue part, i.e. from an off-line schedule to a FPS.

Task Attribute Assignment of Fixed Priority Scheduled Tasks to Reenact Off-line Schedules [28]

This is almost the same paper as [27] and it has the same authors.

The paper presents an algorithm to combine off-line schedule construction with fixed priority run-time scheduling.

The method works by transforming off-line scheduled tasks with their original constraints into tasks with attributes suited for fixed priority scheduling, i.e., periods, deadlines, and offsets, which will reenact the original off-line schedule at run-time. It divides the off-line schedule and its tasks into windows and sequences, sets priorities to ensure execution orders within windows, and determines priorities and offsets to ensure orders and relations between windows.

One of the assumptions for this method is that all the tasks are independent and fully preemptive. All dependencies have been resolved in the off-line schedule.

Managing Complex Temporal Requirements in Real-Time Control Systems [16]

The paper propose a method, which by assigning priorities and offsets to tasks guarantees that complex timing constraints can be met. In addition to complex constraints, the method supports sporadic tasks, shared resources, and varying execution times of tasks.

The heart of the method is a generic algorithm (GA), which finds almost an optimal solution when no solution exists that fulfill the requirements. This is important from the engineering perspective, since the result can be used as input for remodeling of the system.

The paper also mentions that manual assignment of attributes to tasks is possible for smaller systems, but changes in the design would lead to maintenance problems. This problem can be attacked by having some kind of automated tool that helps the engineer to assign the attributes to the tasks.

To make our solution usable, some kind of automation tool is needed. If no automation tool is used, then the solution would be too complex and time consuming to use at Volvo CE and then the solution would not be of any practical interest for them.

2.6 Response time analysis

Rate monotonic has a well known scheduling bound that ensures that a task set is schedulable if the utilization factor of the task set is below the scheduling bound ($\ln 2 \approx 69\%$ for rate monotonic). But if the task set is not schedulable according to the scheduling bound, it still can be schedulable due to that the bound for rate monotonic is only sufficient and not exact.[37]

Exact analysis, on the contrary, is an exact analysis of a task set, which ensures that the task set is schedulable if the analysis says so and that it is not if the analysis comes to that decision.

The advantage of using scheduling bounds is that it does not require as much calculations as exact analysis does. The major drawback is, as mentioned above, that it is not exact and fairly pessimistic.

A Multiframe Model for Real-Time Tasks [25]

The paper addresses the problem of scheduling tasks, when the execution time of task instances may vary greatly but follows a known pattern. The schedulability problem for the model is investigated for the preemptive fixed priority scheduling policy.

The well-known periodic task model and schedulability analysis by Liu and Layland assumes a worst-case execution time bound for every task, this may be too pessimistic if the worst-case execution time of a task is much longer than the average execution time.

The paper propose a multiframe model which takes into account such varying execution patterns, and obtains a significantly much better schedulability bound than for the Liu and Layland task model.

The execution time of each task in the multiframe model is specified with a finite list of numbers. The list is repeated to generate a periodic sequence of numbers such that the execution time of each instance of the task is bounded by the corresponding number in the periodic sequence.

The paper assumes that every task is sporadic instead of periodic, for generality. A periodic task is then defined as a limiting case of the sporadic task whose request arrives at the maximum rate.

We think that this generalization is somewhat strange, probably the authors of the paper do know the difference between the two task sets, but have made this generalization due to that under the schedulability analysis this assumption is correct and usually done (the feasibility test is made at the critical instant, at this instant the two tasks sets can be considered as same).

The method used in this paper is similar to the method used in [22], and the idea of an array of execution times is very useful when we shall construct the finalization analysis algorithm (a response time analysis applied at each invocation of a task).

Response time Analysis for Dynamically and Statically Scheduled Systems [22]

The paper addresses the problem of response time analysis on a set of tasks scheduled by a fixed priority scheduler in the background of a static cyclic schedule, i.e. every task in the static cyclic schedule have higher priorities than the tasks scheduled by the fixed priority scheduler.

The paper presents a method to calculate the worst-case response time for dynamically dispatched tasks (event-triggered) scheduled by a fixed priority scheduler. The method is also able to handle interference from interrupts.

The contribution of the paper is to consider the static cyclic schedule as one high priority task with varying execution times (this is true for the dynamic tasks). To realize this, the execution time of the task is changed to an array of execution times, additionally the number of execution times is specified. The period of the task is set to the minor cycle of the function chain.

To make the analysis more efficient the author also presents an extension of the first mentioned response time analysis.

This report may be very useful for us since it describes how worst-case response times can be calculated for event-triggered (blue) tasks in a system similar to the one at Volvo CE. Although it is useful to know the worst-case response times, we are also interested in calculating response times for Blue tasks at arbitrary activations, not just the worst-case, and hence the method can not be used as it is. We will adopt the ideas of having an array of execution times in the creation of our finalization analysis algorithm (response time at an arbitrary invocation of a task).

2.7 Resource Access Protocols

Below we give a summary of the Resource Access Protocols given in [18].

To ensure consistency of the data structures in exclusive resources (shared resources protected against concurrent access), any concurrent operating system should use appropriate resource access protocols to guarantee a mutual exclusion among competing tasks. Operating systems usually provide a general synchronization tool, called a semaphore, which can be used to achieve such a mutual exclusion. A piece of code executed under mutual exclusion constraints is said to be a critical section.

When concurrent tasks use shared resources, in a uniprocessor system, there may arise some problems. First, there is the priority inversion problem which gives unbounded blocking times. Another problem is deadlock situations which may arise if the user uses the synchronization primitives in an incorrect way.

One solution to the unbounded priority inversion problem is to disallow preemption during the execution of all critical sections. This method, however, is only appropriate for very short critical sections, because it creates unnecessary blocking.

In other approaches, the priority inversion problem is solved through the use of appropriate protocols that control the access to any shared resource. Such protocols are the Priority Inheritance Protocol (PIP) and the Priority Ceiling

Protocol (PCP) which apply to fixed-priority systems, whilst the Stack Resource Policy (SRP) is suitable both for static and dynamic priority systems.

We will not describe these protocols in detail here, but we give an evaluation summary of the mentioned resource protocols below:

Table 4 An evaluation summary of resource access protocols.

	Priority Assignment	Number Of Blocking	Blocking Instant	Transparency	Deadlock Prevention	Implementation	B_i Computation
PIP	Fixed	Min(n,m)	On Resource Access	YES	NO	Hard	Hard
PCP	Fixed	1	On Resource Access	NO	YES	Medium	Easy
SRP	Fixed Or Dynamic	1	On preemption	NO	YES	Easy	Easy

Worth to note is that the PIP, although not so efficient in terms of performance, is the only one that is transparent at the programming level. The other protocols, in fact, require the user to specify the list of resources used by each task, to compute the ceiling values. This feature of the PIP makes it attractive for commercial operating systems, where predictability can be improved without introducing new kernel primitives.

A very useful and interesting feature of the SRP is that it allows the sharing of one stack among tasks in the system, and hence, reducing the amount of memory required for the stacks of all the tasks. This is particularly convenient for those applications consisting of a large number of tasks, dedicated to acquisition, monitoring, and control activities.

For example, consider an application consisting of 100 jobs distributed on 10 preemption levels, with 10 jobs for each level, and suppose that each job needs up to 10 Kbytes of stack space. Using one stack per job, 1000 Kbytes would be required. On the contrary, using a single stack, only 100 Kbytes would be sufficient, since no more than one job per preemption level could be active at one time. This corresponds to a stack reduction or saving of 90% (900 Kbytes). In general, when tasks are distributed on K preemption levels, the space required for a single stack is equal to the sum of the largest request on each level.

The use of one stack for the blue tasks is a very important issue for our thesis. In the systems at Volvo CE, the RAM memory is limited due to cost. If every blue task would have its own stack, there would probably be no benefit in moving tasks from the red to the blue part.

2.8 Performance evaluations and reviews

Value vs. deadline scheduling in overload conditions [1]

In [1] the author gives examples of the performance for different algorithms under different system loads. Four algorithms;

1. *EDF (earliest deadline first)*.
2. *HVF (highest value first)*
3. *HDF (highest density first, i.e. value / computation time)*
4. *MIX (mixed rule where importance value and deadline are considered)*

are simulated in three different versions:

1. *Plain (do not provide any form of guarantee and may be issue of domino effect)*
2. *Guaranteed (characterized by an acceptance test)*
3. *Robust (characterized by a sophisticated rejection a reclaiming strategy).*

Their conclusions are that EDF is optimal in under-loaded conditions, REDF (Robust EDF) provide an effective strategy for a wide range of overload conditions. However RHDF (Robust HDF) performs better in high overloads than REDF.

So if the system at Volvo CE could change scheduling strategy at run-time, EDF would be preferred in under-loaded conditions, REDF for normal overloads and RHDF for high overloads.

Scheduling hard real-time systems: a review [11]

In [11] the author reviews scheduling theories for hard real-time systems. Both static and dynamic priority scheduling algorithms are discussed. It is stated that safety critical real-time systems should be guaranteed before execution, hence using a static scheduling approach. Some advantages for dynamic algorithms are presented in the paper:

- “Dynamic scheduling algorithms are particularly appropriate to soft systems”.
- “They could form part of an error recovery procedure for missed hard deadlines”.

Uniprocessor systems, with and without blocking, and multiprocessor systems are analyzed. It is stated that a system in which all tasks are scheduled using only worst-case execution times, may have an unacceptable low utilization during “normal” execution.

We believe this is an important observation that we certainly must have in mind when proposing our solution to Volvo CE.

Furthermore the paper contains a lot of useful information concerning transient overloads and priority inversions. The author describes common problems that may appear during transient overloads and the reason (e.g. underestimated WCET times) to why they may appear. A way of making the Rate Monotonic algorithm applicable to transient overloads (normally, if using the Rate Monotonic algorithm in overloaded systems, the tasks with the longest periods will miss their deadlines, even though they may be the most important ones) is presented. The presented solution suggests that priorities should be an accurate statement of a tasks importance, this is achieved by transforming their periods

The author also gives information on how priority inversions (the phenomenon that high priority tasks can be blocked by low and medium priority tasks) can be prevented with algorithms such as priority inheritance and ceiling protocols.

When using the priority inheritance protocol a process with m critical sections can be blocked at most m times i.e. each critical section may be blocked by a lower priority process. Some disadvantages with the priority inheritance protocol are discussed. The ones mentioned are:

- The possibility of blocking chains that may occur e.g. P1 being blocked by P2 which is blocked by P3 and so on.
- There is nothing that precludes deadlocks in the priority inheritance protocol.

According to the author Ceiling protocols addresses the above disadvantages. Using *Ceiling protocols*, a high priority process can be blocked at most one time during its execution. Moreover deadlocks and transient blocking is prevented with the ceiling protocol.

Flexible scheduling theory for advanced engine controllers [8]

The background of [8] is a successful introduction of FPS into aircraft engine controllers. The authors describe requirements, especially timing requirements, for safety critical hybrid control systems. Pros (flexibility, efficient scheduling mechanisms and necessary analysis techniques) and cons (release jitter may be worse due to dynamic nature of run-time schedule), for introducing FPS instead of cyclic executive scheduling approaches in such systems, are given. Several timing requirements and the relationship to the timing constraints are described. Priority assignments and timing analysis are briefly described.

Three obstacles to flexible scheduling [10]

In [10] the authors review three problems involved with flexible scheduling. The first one is a question of when to use time slacks, the second of when to run mandatory parts and the third of when to apply acceptance tests.

The conclusion to the obstacles is that neither pushing hard tasks as late as possible nor running mandatory first approaches, are adequate for flexible scheduling. A late acceptance test is more adequate than an early one because the timing window within which the schedulability is made become smaller. Obviously the latest time at which an acceptance test should be made is at the time the task receives its first tick.

Implications of classical scheduling results for real-time systems [12]

In [12] the authors discuss classical scheduling theory results. Theories for both uni- and multi-processor real-time systems are addressed. An interesting fact described is the domino effect that may happen with EDF in overloaded systems, meaning that if a task misses its deadline it may cause all subsequent tasks to do so as well.

The paper deals with a lot of results that may be useful for us to know. Some of which concerns value based algorithms are described below:

- Value based algorithms are often overlooked in real-time system.
- No on-line scheduler can guarantee a cumulative value greater than one-fourth the value obtained by a clairvoyant scheduler.

We will not write about all results presented in the paper because it would be too much. A lot of the results are common knowledge.

Chapter 3

3 Findings

This section gives the results of our thesis. It gives solutions to the main objective of this thesis (the problem of using the resources in the systems at Volvo in a more efficient way with preserved requirements) concentrating on the development process i.e. requirements specifications, design issues and implementation.

We start by discussing requirements specifications. This is followed by design issues and how to design different functionality. We then give three ways of guaranteeing Blue tasks in Rubus by:

- A simple approach intended to calculate response times for blue tasks.
- An extension to the simple approach that guarantees blue TT and ET task
- The weakly hard concept.

The chapter ends with a discussion about how to support the weakly hard concept in the development process.

3.1 Requirements specifications

As mentioned in section 1.4, requirements (including temporal) are captured from the customer. An important issue is to have the customer to think about if the temporal requirements are relevant for the intended functionality. We believe it is easy to fall into a routine where requirements, such as deadlines, are based upon existing deadlines for similar functionality. Although it is an easy way to state temporal requirements in order to try fulfilling the desired functionality, it may be a bad way and eventually lead to a system where several tasks have unnecessary tight timing constraints, and where the limited resources are allocated but not used. So stating temporal requirements based upon existing ones, without considering whether they are realistic or necessary, should be avoided.

We also think it is very important to inform the customer, who is stating the temporal requirements, about the current resource situation. A better understanding between the parties might have the customer to think differently and in fact save resources without any further actions. It might be the case that the customer is simply unaware of the limited resources thus stating requirements as if they were infinite.

Having different priorities on the specified requirements may be used by the project team in order to decide the order in which to fulfill them. A problem that may appear is the classification of the importance of requirements i.e. giving them priorities. Asking the customer about the importance of requirements, will probably lead to a situation where all of them have the highest priority. In order to cope with the problem, requirements could be weighted against each other, meaning that two requirements could be presented with the question “which one is most important”. It may lead to a situation where requirements can be ordered

i.e. sorted by distinct priorities, even though they all had the highest priority initially.

We believe that when stating temporal specifications at Volvo it is important to consider characteristics such as:

- *Consistent and complete* – There are several documents with temporal specifications for the same functionality i.e. temporal specifications for some functionality can be found in several documents. This is a potential source of both inconsistency and incompleteness. We suggest removing redundant information.
- *Realistic* – To state realistic, both operational and economical, requirements is important. The customer(s) does not always have knowledge about what the developers are capable of doing. It is important that both the customer and the developer discuss the possibilities of fulfilling requirements. A better understanding about the customer's needs and developer's capabilities between the two parties, can lead to agreements suitable for all involved.
- *Verifiable* – Some end-to-end requirements at Volvo are very hard to verify because they require static verification methods to be applied to the complex system. If any time functionality is added or removed to the system, the verification procedure must be performed again. Static verification is very time-consuming and costly, so we believe that dynamic verification is the only possible realistic one.

3.2 Design

As most real-time systems have a limited amount of available resources (such as CPU time), the system design can have great effect on the utilization of them. With a bad design, one could end up with a system where the available processor time is fully allocated but much less used.

Consider the case for time-triggered hard real-time systems, in which all tasks are scheduled off-line with their worst-case execution time. The available scheduling time will be allocated to tasks but not necessarily used, due to tasks not running for their worst-case execution time at every invocation. So somewhere in the process of designing real-time systems, the decision whether any functionality should be considered as hard or soft (or maybe a combination of them) should be taken, in order to avoid (or at least decrease the scheduled contention of tasks in a certain window of time) the unnecessary allocation of execution time.

The following sections describe useful information about temporal requirements when designing for functionality in a real-time system. We will consider some common applications and describe both time and event-triggered design approaches.

3.2.1 Basic design considerations

To distinguish the process of capturing the requirements from the process of design, one can think of a design as a description or identifier of *how* requirements can be fulfilled. So, at design time we should already know *what* the customer wants.

When designing for logical functionality in a real-time system i.e. choosing what a task should do and when etc, we believe it is important to categorize the underlying functionality that we intend to model, as either hard or soft. To aid the designer, the requirements should contain information about any functionality, such that it can be interpreted as either hard or soft, or as stated above, a combination of hard and soft. Giving this type of information to the designers may be of great importance and use, since system resources may be utilized in a better way over any window of time.

Another category of tasks (besides hard or soft), that can be useful to think about when designing a system, are the ones specified by weakly hard temporal constraints [36]. They are tasks specified to tolerate a well defined degree of missed deadlines in a specified window of time.

Two classes of constraints, either consecutive or non-consecutive, are used in weakly hard systems. Each of the constraints can be used to specify either missed or met deadlines. The motivation for weakly hard real-time systems is the fact that most hard real-time systems can miss some deadlines, provided that it happens in a predictable way. In weakly hard systems the type of constraint specified depends on the application domain. When designing a system with weakly hard tasks, one should keep in mind that certain applications are more or less sensitive to consecutive deadline misses and others to non-consecutive ones. See section 3.6.2 for a more detailed discussion on weakly hard system design

Our experience from the case study shows that designers tend to use hard tasks in a system just because the system has greater support for them, than it has for soft tasks. Another reason, to why systems are modeled with hard tasks, may be that it feels like a safer system than it would if soft tasks were used. In addition developers might feel a bit uncomfortable with the guarantees that dynamic scheduled systems supports. It is known that dynamic scheduling is much more flexible than static, but dynamic scheduling may lead to unexpected system behaviour if calculations or estimations etc. are done wrongly. As an example consider a system involving aperiodic acceptance and guarantee tests. Let's say that calculations are used to check the processor demand in an interval in order to check whether the aperiodic can be accepted and forget to consider the remaining execution times for pre-empted tasks. Then we may end up with an overloaded system.

3.2.2 Timing constraints

When choosing artifacts such as periodicity, deadline etc. for tasks, we believe it is important to consider the possibility of the system to fulfill the chosen constraints, not just at design time but also in the future. It is easy to fall into a routine where timing constraints are chosen as too tight, especially in the beginning of a system design when there seems to be an unlimited amount of resources such as CPU time. This way of thinking will eventually lead to a system where a whole bunch of tasks will have to be scheduled in a window of time i.e. the scheduling process becomes more difficult. Another important issue, concerning the timing constraints, is the economical cost involved in them. When a system reaches a point where all (or close to all) available scheduling time is used, radical changes to the system may be the only solution in order to add further functionalities. This is a cost that may be pushed forward in time, if systems are designed with appropriate constraints in mind.

3.3 Designing different applications

This section describes how different applications may be designed. Several approaches are described and commented. The applications correspond to functionalities found in the system at Volvo.

In each case the underlying challenge is to design functionality that reads sensor data and performs action depending on the data.

3.3.1 Control applications

In a time-triggered hard real-time system a control application may consist of communicating sensor reading tasks and actuator tasks. There are several ways to design such applications. The following are a few examples given with brief descriptions and some comments.

3.3.1.1 All functionality in one task

In the most naive approach both the sensor reading and actuator functionality is modeled in only one task. In doing this there is no way of separating the need of specific periodicity for any of the functionality. Both functionalities in the task must then be modeled with the highest frequency needed for any of the two, in order to preserve the desired dynamics of the system. To overcome the drawbacks the tasks can be modeled as two separate dependable tasks. In doing this, the designer should try to minimize the difference of a task BCET and its WCET. The lesser the difference is the better the utilization becomes, since then the tasks tend to execute most of the allocated time for them. This is actually something one should try to accomplish for every task in the system.

3.3.1.2 Split functionality into two tasks

Instead of having all functionality in one single task, the task can be split up into several parts. The first part will cope with sensor data and the last part will handle the functionality of the actuator. Inbetween them it is possible to have a controller task.

First approach

The first way to decrease allocated resources is to increase the period of the actuator task. This approach will clearly have a slower response or impact on the system. All tasks depending on the actuator will have data delivered at a later time, so the periodicity of the dependable tasks may be subject of change.

A drawback is that it may not be possible to change the periodicity without changing the architecture, in a cyclic scheduled system, especially if the period already is equal to the major cycle period of the system. In that case the weakly hard concept may be considered.

Second approach

The second approach is to remove the periodic behavior of the actuator and rely on the sensor task to activate the actuator when necessary. This approach is likely to be one that saves resources, but in hard systems it needs some way of guaranteeing that the event-triggered actuator task finishes before its deadline. Dangerous situations may arise with this solution e.g. if the sensor task activates the actuator too often due to malfunctioning sensors etc, so the system may be heavily overloaded.

Third approach

Here the sensor task would be event-triggered, i.e. the task is activated by an interrupt. A controller task which would be responsible for most calculations may be used. The sensor task would active to the controller, and the controller would activate the actuator. This approach could be used in order to minimize possible jitter on the sensor and actuator tasks.

Fourth approach

The fourth way is to specify the actuator with weakly hard constraints. In addition with the second and third approach, this approach will also need a way of calculating if the constraint can be fulfilled. A problem that might appear in using weakly hard constraints is for the designer to choose the number of either missed or met deadlines during a window of time. In choosing artifacts such as the maximum number of deadline misses in a window of time, the requirements must be used as a helper. With this we mean that the requirements must contain information to guide the designers in their task of choosing weakly hard constraints. As an example the requirements could mention possible effects of data produced too late.

When using weakly hard constraints in a control system it is important to know that control systems may become unstable if a certain number of deadlines are missed in a row i.e. some control applications are sensitive to consecutive deadline misses while they often can tolerate a small number of non-consecutive deadline misses [4].

3.3.2 Monitoring and logging

Monitoring and logging of information can be useful in a system in which the behavior in time should be possible to overview. In a time-triggered system, it is important to choose appropriate timing constraint such as periodicity for the logging tasks. It is often no use in logging a signals value more often than half of its minimal time between changes (i.e. double frequency sampling according to the Nyquist theorem). This approach is suitable for logging of values where there is a need to know if certain values are reached. When logging values that change proportionally in time (i.e. without trying to estimate if a certain value is reached) the possibility of using of weakly hard tasks can be considered. In this way the behavior of the logged signal can be interpolated even if the logged signal lacks some values i.e. when the weakly hard scheduler has skipped certain activations of the logging task.

When using weakly hard tasks one has to consider which types of deadline misses that are acceptable for the logging and monitoring functions, consecutive or non-consecutive.

3.3.3 Interaction

In a system that occasionally presents some visual information to a user the users ability of reacting to such information should be considered. There is for example no use in specifying temporal constraints too tight if the visual information is intended as confirmation of some otherwise unknown (to the user) action. On the other hand, if the information should confirm an action, too large response times may lead to a system which feels slow. Such systems may be annoying to the users.

An example of interaction might be the reverse siren that confirms the reverse movement of a vehicle. If it beeps within 0.5 s or 1 s after reverse gear is set, does probably not matter a great deal. Of course one has to consider the safety aspect of any functionality. But in the given example chances are that neither the driver nor any person in the nearby area will notice any difference. A way of achieving this behavior is just to specify the periodicity of the siren task double as high. But then it will always respond with the given periodicity. Instead one could give it a weakly hard temporal constraint e.g. 1/2 meaning that at least one of two deadlines must be met, but if possible 2 of 2 will be met. So in doing this the system guarantees a minimum QoS of the intended functionality.

3.4 Examples of techniques to model functionality

In this section we give some examples of different techniques that may be used to model functionalities in a real-time system. Both time and event-triggered approaches are discussed.

3.4.1 System model

The system for which these examples may be applied consists of two nodes communicating via a CAN bus. Each node has a locally scheduled task set consisting of static off-line scheduled (Red tasks) and dynamically dispatched tasks (Blue). The schedule is cyclic with a major cycle of 100 ms. Each major cycle consists of ten minor chains, each of length 10 ms. The maximum communication latencies (say 50 ms) for the bus are known in prior. Each node activates an interrupt when it receives a message on the bus. This is done in order to handle the incoming messages as quickly as possible.

3.4.2 Interaction Example

In this example the requirements are as follows: “*A siren should sound when the vehicle is moving backwards*”. The temporal requirement given from the customer is a maximum latency from the time at which the reverse gear is set until the time at which the siren should be activated. In this example the latency should be max 0.5 s. Furthermore the reverse gear and the siren are handled by different nodes, so there is a need for communication.

Event-triggered approach

In the event-triggered approach of designing for the requirement, the reverse gear may activate an event. As soon as the gear is set, an interrupt may be executed. The interrupt would execute a minimal amount of code in order to activate an event handling task. The event handling task, with WCET say 12 ms, will be responsible of sending an activation signal via the bus to the other node. The task

is also assumed to be responsible for other actions not mentioned here. At the receiving end (the other node) another task will activate the siren.

If we assume that the receiving node will handle all incoming messages within one major cycle, then the time within which the event handling task must finish is $500-100-50 \text{ ms} = 350 \text{ ms}$ from that the reverse gear is set (in the calculation the numbers corresponds to the requirement 500ms, the time it takes to handle an incoming event 100ms and communication latency 50ms). The event handling task must be activated at latest $350 \text{ ms} - 12 \text{ ms} = 338 \text{ ms}$ after the event happens, in order to fulfill the requirements. To see if the task will finish within 338ms after the event happened, the calculations must be performed as if the event happened at the worst point in time, leading to the longest response time for the event handling task i.e. at a critical time instance. To make things worse we have to consider all interrupts that may hit the system within the calculated response time. This is usually achieved by assuming a minimum interarrival time (MINT) for the interrupts. Also when calculating response times for tasks with lower or equal priorities, we have to consider the time that is allocated by tasks with higher priorities. Again this must be done by assuming a MINT for them. In our example we would have to know the minimum time between two consecutive sets of the reverse gear.

Time-triggered approach

If we have a system where the event handling task is time triggered instead of event-triggered, we would at least have to schedule it within a major cycle of the schedule (because the schedule is cyclic). This approach will most certainly allocate too much time for the event handling task, since it will probably not need to execute once per cycle. Consider a case where the driver sets the reverse gear at most ones a minute. Then the allocated time for the event handling tasks will be $60*10*WCET = 7200 \text{ ms}$, but it will only use 12 ms (ca. 0.0017 %) of them under run-time. If we did not have to schedule the tasks within one major cycle the time allocated for the task could be decreased.

3.5 Guaranteeing Blue tasks

As mentioned the main objective of our thesis is to find a way to use the resources in the systems at Volvo in a more effective way, with preserved requirements, by moving functionality from the off-line scheduled Red part to the on-line scheduled Blue part.

The real-time operating system at Volvo has great support for hard tasks by the Red tasks. The main advantages with Red tasks in Rubus are:

- The easy implementation.
- One shared stack.
- Reliability

Although Red tasks have several advantages they have one great disadvantage: lack in flexibility. One problem with the flexibility issue is that the memory is limited and the generated off-line schedule has to be saved in memory during run-time. This implies that the task period times have to be chosen carefully to minimize memory usage. This results in tasks having shorter period times than necessary, which eventually may lead to a system consisting of hard tasks allocating far too much system resources than necessary. An example of this is the systems at Volvo: they generate an off-line schedule of length 100 ms in order to minimize ROM usage. This forces developers to give tasks, handling

functionality with requirements greater than 100 ms, period times of maximum 100 ms.

To make the system more flexible, and hopefully more economical, Volvo should use more of the Blue services. The main advantages with Blue tasks are:

- Flexibility.
- Maintainability.

One great disadvantage with Blue services is that each Blue task is assigned a single private stack.

One of the main problems today is that Rubus lack support for a Blue task model that makes it possible to give any sorts of guarantees, there are even no WCET or deadline attributes for the Blue tasks in the system to use in the feasibility analysis.

The lack of support for guaranteeing Blue tasks in Rubus is probably *the* reason to why the Blue tasks are slightly used by Volvo. Today the Blue tasks run as background tasks with no guarantees and are hence only used in situations or in functionality that really demands the features of the Blue soft services.

The reason to why Volvo would want to use Blue services with guarantees is to use the system resources in a better way, and to be able to give guarantees to functionality demanding Blue services. For example if a functionality has a requirement that results in a task with a period time greater than the length of the off-line schedule, a Red task would be scheduled today with a period time equal to the length of the off-line schedule (100 ms) and hence it would be executed more often than needed. The reason to this is, as mentioned above, that Volvo cannot expand the length of the off-line schedule due to space limitations in the memory. But if a Blue task would be used, it would be possible to assign it almost any period time (even greater than the length of the off-line schedule) that fulfils the requirement. Hence the use of Blue task instead of a Red task may allocate less system resources, in certain situations, and hence increase the available resources and decrease the system utilization.

In the following sections we will give three possible ways of guaranteeing Blue tasks by:

- A simple approach requiring the smallest number changes to Rubus and to the current system. This approach may be preferred if changes in Rubus are difficult to implement or otherwise economically unbeneficial.
- Guaranteeing blue time-triggered and event-triggered tasks in fixed priority manner. The time-triggered approach may be preferred if a high degree of determinism is important hence making it easy to reproduce executions during testing. The event-triggered approach may be preferred in situations where redundant polling is highly unwanted.
- Guaranteeing blue tasks as part of the weakly hard concept. This approach may be preferred in systems where the resources are near to be fully used and where a certain amount of deadline misses is acceptable, e.g. in some control applications, user interaction etc.

Each approach should be seen as an extension to the previous one given. We will also explain problems that may arise when moving tasks from the Red services to Blue services and issues concerning requirements specifications and design. Each section will also explain necessary and wanted changes to Rubus.

3.5.1 The simplest approach

Without any changes to the task model of blue services in Rubus, there is no way of giving temporal guarantees to blue tasks i.e. they may only be used as soft tasks.

However if the blue task model could be extended with:

- worst-case execution time (WCET)
- minimum interarrival time (MINT)

it would be possible to calculate response times for blue tasks. This information gives characteristics of the functionality, which may be used to get an overview of the unguaranteed blue tasks (even though blue tasks lack deadline attributes).

A response time formula for dynamically scheduled tasks that are executed in the background of a static cyclic schedule is described in [22]. The formula overcomes the pessimistic results that may occur by using the response time calculation (described under schedulability analysis in the introduction to real-time systems section). In [22] they begin by introducing an infinite-length array $\hat{C}_i[k]$ formally described below as:

$$\hat{C}_i[k] = \begin{cases} 0 & k = 0 \\ \max_{t \in \{0, \dots, |C_i| - 1\}} \sum_{l=0}^{k-1} C_i[(t+l) \bmod |C_i|] & k > 0 \end{cases}$$

- C_i denotes a vector of execution times for task i such that $C_i = [C_i[0], C_i[1], \dots, C_i[|C_i| - 1]]$
- $|C_i|$ is the number of elements in C_i .
- T_i is the duration of a minor cycle.

Hence $\hat{C}_i[k]$ will contain the maximum total execution time of k successive invocation of task i .

The response time for task i is then calculated with the following formula:

$$R_j = B_j + \hat{C}_j[1] + \sum_{i \in \{x: P_x > P_j\}} \hat{C}_i \left[\left\lceil \frac{t + J_i}{T_i} \right\rceil \right]$$

- R_j denotes the response time for task j .
- B_j denotes the maximum blocking time for task j .
- J_i is the maximum deviation from ideal periodicity for a task i , i.e. its jitter.
- T_i is the period of task i .

So by introducing a WCET and a MINT to the blue services, the described formula could be used to calculate response times for blue tasks at Volvo.

Moreover, for any tasks migrated from red to blue, the worst-case response time could be calculated and compared to the temporal requirements of that task. If the temporal requirements for a migrated task would be unsatisfied, a discussion with

the designers or the customer about temporal changes would be possible. The calculated maximum response time could be presented with the question if a change in the temporal requirements (deadline) could be accepted for the task.

3.5.1.1 Example

The following example uses the explained formula and calculates the worst-case response time for a blue task J with $C_j = 10\text{ms}$, $\text{MINT}_j = 500\text{ms}$, $B_j = 0$ and $J_j = 0$.

A static schedule with a major period of 100ms is divided into 10 minor chains of 10ms each. The static schedule is considered as one task i with higher priority than any blue task.

An array containing the execution times of i in each minor chain is defined as:

$$C_i = [7, 1, 3, 4, 8, 6, 7, 8, 8, 9]$$

Using the formal definition above, the infinite array $\hat{C}_i[k]$ would be:

$$\hat{C}_i[k] = [0, 9, 17, 25, 32, 39, 45, 53, 57, 60, 61, \dots].$$

Applying the response time formula iterative until it converges, gives the maximum response time for the blue task J as:

$$R_j^0 = 0 + 10 + \sum \hat{C}_i \left[\left\lceil \frac{0+0}{10} \right\rceil \right] = 0 + 10 + 0 = 10$$

$$R_j^1 = 0 + 10 + \sum \hat{C}_i \left[\left\lceil \frac{10+0}{10} \right\rceil \right] = 0 + 10 + 9 = 19$$

$$R_j^2 = 0 + 10 + \sum \hat{C}_i \left[\left\lceil \frac{19+0}{10} \right\rceil \right] = 0 + 10 + 17 = 27$$

$$R_j^3 = 0 + 10 + \sum \hat{C}_i \left[\left\lceil \frac{27+0}{10} \right\rceil \right] = 0 + 10 + 25 = 35$$

$$R_j^4 = 0 + 10 + \sum \hat{C}_i \left[\left\lceil \frac{35+0}{10} \right\rceil \right] = 0 + 10 + 32 = 42$$

$$R_j^5 = 0 + 10 + \sum \hat{C}_i \left[\left\lceil \frac{42+0}{10} \right\rceil \right] = 0 + 10 + 39 = 49$$

$$R_j^6 = 0 + 10 + \sum \hat{C}_i \left[\left\lceil \frac{49+0}{10} \right\rceil \right] = 0 + 10 + 39 = 49$$

As can be seen the maximum response time for task J is 49ms, which happens when task J is being activated 60ms into the major period (i.e. at the beginning of the seventh minor chain). A deadline of 50ms would be satisfied by the task.

3.5.2 Guaranteed Blue TT and ET tasks

”Guaranteeing” Blue tasks using the simplest approach may work fine for very small and limited task sets, but most often that approach will not be useful. Instead it would be nice to have a way of guaranteeing Blue tasks without the redundant communication concerning temporal requirements between the developers and the designers or costumers.

Below we will introduce an approach in FPS manner that gives an output that directly tells us if the task set is feasible or not. This approach can be seen as an extension to the “simplest approach” given earlier. The simplest approach is just extended with a deadline attribute.

We will start by discussing the steps to follow during a feasibility analysis of a Blue task set, both event-triggered and time-triggered tasks are handled. Then we give a worst-case analysis method for guaranteeing Blue task running in the background of the Red off-line schedule. We then list necessary and preferred changes to Rubus. Finally an example is given to make things easier to understand.

3.5.2.1 Feasibility check of Blue TT and ET tasks

First of all the utilization of the task set should be checked. If the total utilization of all the tasks in the system exceeds 100% the system is overloaded and hence unschedulable.

If the system utilization is below 100% the feasibility of the task set must be checked, either with an appropriate scheduling bound check or a response time analysis algorithm.

When performing a feasibility analysis, one should always start with the highest priority task in the system and then continue with the next highest and so on. The analysis should be terminated if one task does not meet its deadline. If all tasks meet their deadlines the analysis is terminated with success and the task set is feasible.

As mentioned in the introduction there is a feasibility analysis called exact analysis. Exact analysis is primary made for guaranteeing plain FPS tasks, in this approach the Blue tasks equals FPS tasks. At Volvo the Blue tasks are run in the background of an off-line schedule. Unfortunately, the exact analysis is not made for mixed task sets and hence a new feasibility analysis algorithm must be developed.

We propose two different analysis algorithms for guaranteeing task sets of solely Blue TT tasks respectively Blue ET tasks.

For the pure Blue TT task sets we suggest using a modified response time analysis algorithm assembled primary from [22] but also from [25]. This is described below in section *Response time analysis primary for Blue TT tasks*.

The pure Blue ET tasks set should be guaranteed using the response time analysis described in [22]. In this response time analysis every Blue task runs in the background of a static cyclic off-line schedule. The cyclic schedule is considered as one single high priority task with varying execution times (one for each release chain in the major cycle). At any instant interrupts (modeled as event-triggered tasks) can preempt the Red and Blue tasks and delay their execution. This type of

analysis is shown in *The simplest approach* above, the difference now is that we have a deadline for each task giving feasibility directly and hence the redundant communication concerning temporal requirements can be omitted.

Note that the two analysis methods mentioned above can be used for both Blue ET and Blue TT task sets or a mix of them, but the ET solution is better suited for ET tasks and vice versa. The ET solution is fairly pessimistic when guaranteeing TT task due to that the TT tasks are handled as ET tasks, it only gives sufficient guarantees and not exact guarantees as the TT solution does for TT tasks.

When guaranteeing ET tasks with the TT solution the period time, T , of a TT task should be replaced with the ET tasks INT and vice versa if guaranteeing TT tasks with the ET solution.

Before we start stating our analysis method it is worth to mention a special case that decreases the number of calculations tremendously. In the case where the period of each blue task T_i is harmonic with the major cycle T_{sm} of the static schedule, and where the deadline D_i of a blue task equals its period i.e. $\forall_i (T_i \mid T_i \geq T_{sm} \ \& \ T_i \bmod T_{sm} = 0 \ \& \ D_i = T_i)$, the exact analysis algorithm can be used on the task set; start by creating one high priority task by adding all the execution times of the red off-line tasks, then simply use it as any other task in the exact analysis. Observe that this approach only check for feasibility and may not give the correct response time of a blue task.

3.5.2.2 Response time analysis primary for Blue TT tasks

The analysis method given below is primary assembled from the papers [22] and [25] and the exact analysis formula [41].

Neither one of the algorithms in [22] [25] [41] can be used as is in our model. The response time analysis described in [22] assumes that the “Blue tasks” are event-triggered (ET) which may lead to pessimistic analysis if applied to TT tasks (see discussion below). Furthermore the exact analysis assumes that the task set consist of TT or ET tasks but it cannot handle them running in the background of a static off-line schedule. So in our model we want the analysis to be exact and hence the blue TT tasks should be analyzed by a method better suited.

As stated earlier one can guarantee event-triggered tasks by using their MINT, then the algorithm in [22] could be used as is. We could also use it to guarantee TT tasks. The drawback when guaranteeing TT tasks with the event-triggered analysis is that the tasks will use more of the utilization during the analysis than necessary and hence the analysis would be fairly pessimistic. The analysis for ET tasks in [22] is only sufficient and not exact for TT tasks. Our aim with this solution is to use the system resource in an effective way and hence we want an exact analysis in order to get the most out of the system.

In order to combine the static off-line cyclic schedule (the Red tasks) and the Blue tasks, we have to do some minor changes to the task model for the static tasks. We adapt the concept, of thinking of the cyclic static off-line schedule as one high priority task with varying execution times, from [25]. It is assumed that all minor chains (time between two release chains) are of equal size.

The attribute C (WCET) for Red tasks is changed to denote a vector, C_s , of varying execution times, where $C_s[0]$ is the first element in the vector. The first element contains the sum of all tasks released at time 0. The period time, T_s , still

denotes the period time of the task, i.e. the minor cycle, e.g. $C_s[n]$ is activated at time $n \cdot T_s$.

The new task model for the Red tasks is hence:

- $C_s = [C_s[0], C_s[1], \dots, C_s[n-1]]$
- T_s

To make the calculations easier we introduce two infinite vectors \hat{C}_s and \hat{C}_{USED} . \hat{C}_s is simply a vector containing an infinite sequence of its corresponding elements in C_s , e.g. if $C_s = [1, 2, 3]$ then $\hat{C}_s = [1, 2, 3, 1, 2, 3, \dots]$. Each element in \hat{C}_{USED} corresponds to the accumulated time used by time-triggered tasks (except the red tasks) at the corresponding release chain, i.e. the accumulated processor demand in the interval of the release chain.

Furthermore, we assume that the release time of each task instance is harmonic with the minor cycle, T_s , of the static schedule. These two assumptions will make the calculations easier and faster, due to that non-harmonic period times could lead to a very large number of task instances to test.

Now we have all the necessary information to give our worst-case response time algorithm. The algorithm is based on a finalization time analysis that is applied to each instant of a task. The finalization time for a task instant is the response time relative to its activation time. The worst-case response time is then simply given by taking the maximum of a tasks finalization times.

$$R_i = \max(F_i(t)) \quad (3.1)$$

The analysis formula below must be applied over the LCM (least common multiple) of the TT tasks iterative until it converges, i.e. when the deadline of a task is exceeded or if the finalization time stabilizes ($F_i^{n+1} = F_i^n$).

$$F_i^{n+1}(t) = C_i + I_{online}(t, t + F_i^n) + I_{offline}(t, t + F_i^n) \quad , \quad F_i^0 = C_i \quad (3.2)$$

I_{online} is the interference caused by on-line scheduled tasks, i.e. all higher priority Blue tasks, interrupts (event-triggered tasks in general). Interrupts are simply modeled as high priority Blue tasks with their MINT (minimum interarrival time) set to the period time of the task. The vector \hat{C}_{USED} is used to get the total execution demand in the interval $[t_s, t_E)$. The index is an integer and hence the division will result in an integer, e.g. 1.6 will be truncated to 1. The $hp(i)$ stands for: every ‘‘event-triggered’’ task with higher priority than the current task i .

$$I_{online}(t_s, t_E) = \sum_{i=\lceil \frac{t_s}{T_s} \rceil}^{\lfloor \frac{t_E-1}{T_s} \rfloor} \hat{C}_{USED}[i] + \sum_{\forall j \in hp(i)} \left\lfloor \frac{t_E - t_s}{T_j} \right\rfloor C_j \quad (3.3)$$

$I_{offline}$ returns the total interference caused by the off-line scheduled static Red tasks in the interval $[t_s, t_E)$ (same discussion as for the \hat{C}_{USED} vector).

$$I_{offline}(t_S, t_E) = \sum_{i=\lfloor \frac{t_S}{T_s} \rfloor}^{\lfloor \frac{t_E-1}{T_s} \rfloor} \hat{C}_s[i] \quad (3.4)$$

When the finalization times for all instances are calculated the \hat{C}_{USED} vector must be updated with the processor demand of the task in the analyzed intervals. At start all the elements in \hat{C}_{USED} equals 0, due to that no processor demand has yet occurred for the TT tasks. During the calculations the processor demand is accumulated as more and more tasks are guaranteed and tested. When a task is guaranteed the \hat{C}_{USED} vector should be updated with the processor demand of the current task, i.e. its WCET. During the update it is necessary to check that the total processor demand in each interval, i , does not exceed T_s , i.e. $\hat{C}[i] + \hat{C}_{USED}[i] \leq T_s$ for all i . If the total processor demand exceeds T_s the processor demand of the current task must be distributed over several intervals in \hat{C}_{USED} .

E.g. if $\hat{C}_s = [8, 5, \dots]$, $\hat{C}_{USED} = [0, 0, \dots]$, $T_s = 10$ ms and the processor demand of a task released at time 0 equals 5 time units, the updated $\hat{C}_{USED} = [2, 3, \dots]$ The reason to why the processor demand is distributed over two intervals (release chains) is that $T_s = 10$ ms and hence the total possible processor demand in each interval is 10 ms, but we already have some processor demand due to the Red tasks in that interval and only the remaining time can be used by the Blue tasks.

As mentioned earlier the number of instances to test during the analysis, for a periodic task t_i , can be very large. Fortunately there is one optimization that reduces the number largely i.e. to only perform the analysis for a task over the hyperperiod at the level of that task.

The *hyperperiod at level i* equals the least common multiple of the periods of the tasks of higher than or equal priority to t_i . It is given by [36]:

$$h_i = lcm\{T_j \mid t_j \in hep(t_i)\} \quad (3.5)$$

Note that the event-triggered blue tasks and interrupts should not be included in the calculation of h_i since the maximum interference of them should be considered at each invocation.

3.5.2.3 Application

In order to successfully use the worst-case response time algorithm at Volvo some kind of automation tool should be developed.

The analysis given above is simple and straightforward to implement. The only problem during the implementation may be the infinite vectors \hat{C}_s and \hat{C}_{USED} , but this should not be a difficult task to solve. It is really not necessary to have an infinite vector during the calculations; it can be limited to a bounded number of elements before the finalization time analysis starts.

The following pseudo code describes a model to use when performing the feasibility check.

```
// Utot = The total system utilization.
// ri = The time the instance of the task ti is released.
// Di = the deadline of task i.

BEGIN feasibility_check
  IF Utot = 1 THEN
    FOR each Blue task ti
      FOR each instance j of ti in hi
        IF Fj(rj) = Di THEN
          Update  $\hat{C}_{USED}$  with Cj
        ELSE
          //Deadline miss so quit
          //the analysis
        END IF
      END FOR
    END FOR
  END IF
END
```

3.5.2.4 System modifications

Below we will describe preferred changes to Rubus and alternatives if no support is given. We start by describing the most important issue: the need of one single shared stack for blue tasks.

One single shared stack for Blue tasks

In a system with pre-emptive tasks there is a need for possible storage of state variables etc. for tasks. If a task is pre-empted by another task, the system must save all relevant data (e.g. the program counter etc) for the pre-empted task in order to continue its execution properly at a later point. The data may be saved on the stack.

This means that all Blue tasks in Rubus must have their own stack memory, i.e. they cannot share a common stack as the Red task may do, unless the pre-emptive behavior is removed or a new semaphore protocol is implemented.

It would require too much memory to give all Blue tasks their own stack and hence no system resources would be saved in moving tasks from Red to Blue services.

Fortunately there is a remedy to this problem: an inheritance protocol, e.g. the stack resource protocol [18].

If tasks share common data, the data must be protected in order to avoid inconsistency etc. the protection is often accomplished with semaphores. If several tasks are waiting for the same semaphore there must be a predictive way to guarantee an upper bound on blocking times for tasks. This is often done by some kind of inheritance protocol.

In Rubus the Priority inheritance protocol (PIP) is used. PIP does not prevent blocking chains, meaning that a single stack may not be possible for tasks using

PIP. There is however protocols allowing one shared stack for all tasks e.g. the stack resource protocol [18].

System modifications with support in Rubus

In order to be able to give any sort of guarantees for the Blue services the task model must be changed. Today the task model for the Blue services only contains priority and the stack size (*Blue services*).

The task model for Blue tasks must be extended to include the following attributes:

- WCET – C
- Period – T
- Deadline – D
- Priority – P

It is not possible to make any off-line analysis of the task set if there is no WCET, period or deadline to check the feasibility with. The Priority attribute already exists and all the scheduling decisions are made based on it.

Furthermore, in order to make the Blue services effective, they should be periodic. If the Blue tasks rely on a relative sleep they may “drift” in time and the periodic behavior may be difficult to achieve.

Today the Blue tasks achieves a pseudo periodic behavior by calling a relative sleep function, this is the reason to the “drift” problem mentioned above. To overcome this problem Rubus should introduce an absolute sleep function.

It would also be preferable to have a larger number of priorities, or even an own priority band, for the guaranteed Blue tasks. Today there are only 15 unique priorities for the Blue “soft” tasks and this may lead to problems when the task set grows.

System modifications without support in Rubus

It is not a problem if Rubus does not extend the task model for Blue tasks to include WCET, Period, Deadline and Priority. The disadvantage is that there is no check for if a task runs over its deadline, and hence no exception is generated and faults may be difficult to find.

The problem with drifting release times may be eliminated by removing the need of relative sleeps by adding a Red “trigger” task acting as a dispatcher for blue TT tasks i.e. activating blue TT tasks. The activation may be performed by signals and the Red “trigger” task will decide which blue task(s) to execute.

Whether the trigger task should be placed in the beginning or the end of each minor cycle, or once per major cycle, depends on the periodicity of the tasks that it is supposed to activate. Placing the trigger task at the beginning of a chain may cause unwanted jitter (especially for the input functionalities, see section 1.3.6). Hence it would be desirable to place it at the end of any chain.

The following code, taken and modified from the Rubus manual [40], shows how tasks may communicate with signals in Rubus.

```
#include <blue/b_signal.h>

void A(void){
    /* This thread waits for the signal 0x0700 */
    if (blueSigTimedWait(0x0700, &rset, NULL) == R_OK)

        if (rset & 0x0100) {
            /* Received a signal from A */
        }
}

void B(void){
    /* Sends a signal to A */
    blueSigSend(&AId, 0x0100);
}
```

Figure 8 Tasks communicating by signals

It is necessary for the trigger task to keep track of when tasks should be activated. This can be achieved with a counter that serves as a dispatch decision variable. The counter may be incremented at each invocation of the trigger task and its value checked and compared to a dispatch table, in order to decide whether any task should be activated. A discussion about possible improvements and skipping strategies for tasks, are presented in section 3.5.3.6.

A question that may arise with the suggested solution is; what happens when the Red dispatcher task sends a signal to a blue task that does not wait for a signal? This scenario happens at the very first time the Red task signals to the blue ones. It happens because the Red task must be located at the beginning of a release chain hence signaling before a blue task has even started to execute. The answer is that the signal becomes pending i.e. it will eventually be received by the blue task.

Without support in Rubus the priority problem is of no great importance. If there exist a red trigger task there is no need for unique Rubus priorities for each task. The trigger task simply follows the execution order defined in the off-line created table.

3.5.2.5 Example

The following is a simple example illustrating how to perform the feasibility check on a task set. The example assumes that support is given in Rubus.

The task set to analyze contains 5 red static tasks, 1 interrupt and 2 blue tasks. The cyclic schedule has a minor cycle of 10 and a major cycle of 50. The interrupt has a MINT of 100. All blue tasks are initially released at $t = 0$ and their deadlines equals their periods.

This gives us the following task set:

Red tasks:

$$\begin{aligned} T_s &= 10 \text{ ms} && \text{(period time/minor cycle)} \\ C_s &= [7, 3, 6, 7, 8] \text{ ms} && \text{(worst-case execution times)} \end{aligned}$$

Interrupt:

$$\begin{aligned} C_i &= 1 \text{ ms} \\ M_{INT} &= 100 \text{ ms} \end{aligned}$$

Blue tasks:

Task	Period (T)	WCET (C)	Deadline (D)	Priority (P)
T ₁	50 ms	7 ms	50 ms	High
T ₂	20 ms	4 ms	20 ms	Low

First of all we must check that the system is bounded so that the finalization formula will converge, i.e. check that the total system utilization does not exceed 1:

$$U_{total} = U_{red} + U_{interrupt} + U_{blue} = \frac{7+3+6+7+8}{5*10} + \frac{1}{100} + \left(\frac{7}{50} + \frac{4}{20}\right) = 0.97$$

Since $U_{total} = 0.97 < 1$ the system is bounded and we can continue with our feasibility check. The next step is to count the finalization times for each instance of each blue task. Begin with the task with the highest priority:

$$\hat{C}_s = [7, 3, 6, 7, 8, 7, 3, 6, 7, 8, \dots]$$

$$\hat{C}_{USED} = [0, 0, \dots]$$

$$F_1^0(0) = C_1 = 7$$

$$F_1^1(0) = 7 + I_{online}(t, t + F_1^0) + I_{offline}(t, t + F_1^0) = 7 + (0 + 1) + 7 = 15$$

$$F_1^2(0) = 7 + (0 + 1) + (7 + 3) = 18$$

$$F_1^3(0) = 18$$

$$F_1^3(0) = F_1^2(0) \Rightarrow F_1(0) = 18$$

Findings

$$F_1^0(50) = C_1 = 7$$

$$F_1^1(50) = 7 + (0 + 1) + 7 = 15$$

$$F_1^2(50) = 7 + (0 + 1) + (7 + 3) = 18$$

$$F_1^3(50) = 18$$

$$F_1^3(50) = F_1^2(50) \Rightarrow F_1(50) = 18$$

Now the worst-case response time is given by taking the maximum of all the finalization times of the task:

$$R_1 = \max(F_1(0), F_1(50)) = 18$$

Now when we have calculated the worst-case response time and checked that the deadline was met, we must update the \hat{C}_{USED} vector. The execution demand of T_1 is 7 ms in the interval $[0, 18]$ and $[50, 68]$.

$$\hat{C}_{USED} = [3, 4, 0, 0, 0, 3, 4, 0, 0, \dots]$$

Each index, i , can at most contain $T_s - \hat{C}_s[i]$ ms of execution demand, that is the reason to why the 7 ms are split over 2 minor cycles (two indexes).

Now we continue with the feasibility check for T_2 :

$$F_2^0(0) = C_2 = 4$$

$$F_2^1(0) = 4 + (3 + 1) + 7 = 15$$

$$F_2^2(0) = 4 + ((3 + 4) + 1) + (7 + 3) = 22$$

$$F_2^3(0) = 4 + ((3 + 4) + 1) + (7 + 3 + 6) = 28$$

$$F_2^4(0) = F_2^3(0) \Rightarrow F_2(0) = 28$$

The instance of T_2 at time 0 does not meet its deadline ($F_2(0) = 28 > 20 = D_2$), hence the task set is unfeasible and the analysis can be stopped.

3.5.3 The Weakly Hard concept

The following sections describe how to apply the weakly hard concept to the current architecture and integrate it in the development process at Volvo.

Weakly hard tasks are simply handled as Blue guaranteed TT tasks with one exception: **the weakly hard constraint**. We encourage the reader to look at the section *Guaranteed Blue TT and ET tasks* before continuing with this chapter.

3.5.3.1 Assumptions

Our intention is to apply the main features of the weakly hard concept as explained in [36] with the following extensions:

- Weakly hard constraints for tasks will be supported for two operating modes; normal and high mode i.e. each weakly hard task will implicitly have a constraint for normal mode and explicitly a constraint for high mode. The modes differ with respect to the amount of processing time needed. At high mode there may be a great amount of tasks executing and using a lot of the available CPU time. At normal mode there may also be a great amount of tasks executing, but the difference is that they use lesser execution time.

Seeing it with a different point of view one could say that ordinary usage of a wheel loader results in a normal mode whilst excessive usage results in high mode:

1. In normal mode all constraints will be considered as strongly hard i.e. $\binom{m}{m}$ and they must be satisfied. To distinguish this approach from ordinary red tasks in Rubus, all necessary calculations for this mode, must be performed with average execution times for tasks. This requires measurements of execution times at the defined system mode. Strongly hard task is handled in the same way as Blue guaranteed tasks, i.e. strongly hard tasks is just another name for Blue guaranteed tasks.
2. In high mode, constraints will be considered as weakly hard i.e. $\binom{n}{m}$.
All necessary calculations will be based on worst-case execution times.

The separation in two modes gives us a system where all deadlines are met in normal mode and where a certain quality of service (QoS) is guaranteed at high mode. The quality of service in high mode can be thought of as a “minimal QoS” since it can be increased with run-time mechanisms as explained in section 3.5.3.6.

In order to apply the concept in an easier manner at Volvo a list of restrictions to the original concept that may be considered are given below:

1. Avoiding shared resources between weakly hard tasks. In doing this the possible blocking factors can be neglected. Also any possible blocking chains will be avoided.
2. Not supporting precedence relations between a weakly hard task and a non weakly hard task. I.e. when a task is converted from Rubus red part to a weakly hard task, any precedence relations may be neglected.
3. Not supporting precedence relations between weakly hard tasks. In doing this there is no need for the designers to apply appropriate priorities or period times for the tasks in the relation in order to keep precedences (although appropriate priorities and period times must obviously be given with respect to the functionalities at Volvo).

In our suggested solution all strongly hard $\binom{1}{1}$ and weakly hard $\binom{n}{m}$ tasks will be modeled as blue tasks in Rubus. Feasibility calculations for the weakly hard constraints will be done off-line.

3.5.3.2 Finalization time analysis

To calculate finalization times for weakly hard tasks the algorithm given in section 3.5.2.2 should be used with some exceptions. The formulas given are listed below:

$$F_i^{n+1}(t) = C_i + I_{online}(t, t + F_i^n) + I_{offline}(t, t + F_i^n) \quad , \quad F_i^0 = C_i \quad (3.2)$$

$$I_{online}(t_S, t_E) = \sum_{i=\lfloor \frac{t_S}{T_S} \rfloor}^{\lfloor \frac{t_E-1}{T_S} \rfloor} \hat{C}_{USED}[i] + \sum_{\forall j \in hp(i)} \left\lfloor \frac{t_E - t_S}{T_j} \right\rfloor C_j \quad (3.3)$$

$$I_{offline}(t_S, t_E) = \sum_{i=\lfloor \frac{t_S}{T_S} \rfloor}^{\lfloor \frac{t_E-1}{T_S} \rfloor} \hat{C}_S[i] \quad (3.4)$$

Table 5 Static task attributes

T_s	Minor chain period time i.e. difference in time between start of two minor chains. Note that this requires that each T_s must be of equal size.
C_s	Vector of execution times starting with $i = 0$. Each element represents the total execution time in each minor chain.
\hat{C}_s	An infinite vector containing the an infinite number of the sequence in C_s .
\hat{C}_{USED}	An infinite vector of execution demands. Each “element” simply corresponds to the requested/used time of time-triggered tasks (except red tasks) in the interval of that release chain.

In order to find a solution the finalization formula must be applied iteratively until it converges. In overloaded systems the formula may not converge, hence careful application of the formula must be considered in those cases.

Time-triggered tasks are handled just as “Blue guaranteed TT tasks” in the formula given above. The main difference between the two cases is when the \hat{C}_{USED} vector should be updated: for the weakly hard tasks it may be the case that an instance of a task is skipped and hence the vector \hat{C}_{USED} should not be updated in that interval.

The reason to why interference by event-triggered tasks and interrupts are considered for each release of a weakly hard task instance is the fact that worst-case scenarios in WHS differs from ordinary worst-case scenarios for mixed task sets. The sufficient but not necessary method to calculate dynamic interference implies that interference must be computed at every invocation and considered to interfere with highest possible frequency.

3.5.3.3 Application

We recommend an application performing necessary finalization time calculations etc. off-line, and another application that tests whether constraints are satisfied for weakly hard tasks. The finalization time application may be responsible for building the μ -patterns also, since building μ -patterns is just a part of checking whether deadlines are met or not.

The following pseudo code describes the model to use when calculating the finalization times for the WHS tasks at Volvo. Note that it is just an extension to the pseudo code given in section 3.5.2.3.

```
// Utot = The total system utilization.
// ri = The time the instance of the task ti is released.
// Di = the deadline of task i.

BEGIN build_u_pattern
  IF Utot = 1 THEN
    FOR each WHS task ti
      FOR each instance j of ti in hi
        IF Fj(rj) = Di THEN
          Add 1 to the μ-pattern
          Update  $\hat{C}_{USED}$ 
        ELSE
          Add 0 to the μ-pattern
        END IF
      END FOR
    END FOR
  END IF
END
```

The method given above assumes that the scheduler is static, and hence no decisions are taken during run-time, instead all is static and the analyzed schedule is the one that will be executed (i.e. the μ -pattern is always followed during run-time). Unfortunately this does not follow the desired concept of minimum QoS. However we can overcome this problem by the use of a dynamic scheduler, which can take decisions about which task to schedule, and hence give the desired minimum QoS. This extension is discussed briefly in section 2.4.2.1.

3.5.3.4 System modifications

The weakly hard concept is an extension to the “Guaranteed Blue tasks”, the same applies to the needed system modifications. Hence the discussion given in section 3.5.2.4 applies for weakly hard tasks, with addition to some extensions given below.

System modification with support in Rubus

To support the WHS concept some changes to the task model has to be done. The tasks that will be subject to weakly hard constraints must have the following attributes:

- WCET – C
- Period – T
- Deadline – D
- Priority – P
- Weakly hard constraint - *I*

System modifications without support in Rubus

If no extension is made to the task model, it will be harder to find faults in the system due to that no control of tasks running over their deadline will be performed.

The activation of weakly hard tasks is handled almost in the same way as Guaranteed Blue TT tasks. The difference is that not all instances of a task are always scheduled for weakly hard tasks, and this must be considered during the table creation and task activations.

3.5.3.5 Example

The following example shows how the algorithm/method above may be used. The static data can be seen as being the current red tasks in the system. The cyclic schedule has a minor cycle of 10 and a major cycle of 50 (5 minor chains according to C_s). The system contains an interrupt with a minimum interarrival time of 100. Two weakly hard tasks will be introduced. All tasks are initially released at $t = 0$ and their deadlines equals their periods. So in our example the interrupt only interferes with the weakly hard tasks.

Example:

Red tasks: $T_s = 10$ ms
 $C_s = [9, 3, 6, 7, 10]$ ms (worst-case execution times)

Interrupt: $C_i = 1$ ms
 MINT = 100 ms

Weakly hard tasks:

Table 6 A task set with weakly hard constraints

Task	Period (T)	WCET (C)	Weakly Hard Constraint	Priority (P)
T_1	20 ms	2 ms	$\begin{pmatrix} 4 \\ 5 \end{pmatrix}$	High
T_2	50 ms	3 ms	$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$	Low

First of all we see that the hyper period $H_1 = \text{LCM}(\{50, 20\}) = 100$, and $H_2 = \text{LCM}(\{50, 20, 50\}) = 100$. Note that the MINT of the interrupt is not included in the calculation of the LCM since it interferes with each invocation of T_1 and T_2 , at its highest rate.

Our extension to the WHS concept requires the above method to be applied twice. First by using worst-case execution times for tasks and finally, when the system is up and running, the average execution times at normal load should be measured and used in the analysis to check the performance.

Let us start by checking that the system is bounded, i.e. check that the total system utilization does not exceed 100%:

$$U_{\text{red}} = \frac{9+3+6+7+10}{50} = 0.70$$

$$U_{\text{interrupt}} = \frac{1}{100} = 0.01$$

$$U_{T_1} = \frac{2}{20} = 0.1$$

$$U_{T_2} = \frac{3}{50} = 0.06$$

$$U_{\text{total}} = 0.70 + 0.01 + 0.1 + 0.06 = 0.87$$

Since $0.87 \leq 1$ the system is bounded.

Now we calculate the finalization time for each invocation of T_1 and T_2 respectively, starting with T_1 .

We construct the two infinite vectors:

$$\hat{C}_s = [9, 3, 6, 7, 10, 9, 3, 6, 7, 10, \dots]$$

$$\hat{C}_{USED} = [0, 0, \dots]$$

Table 7 Finalization time calculations for T_1

$$F_1^0(0) = C_1 = 2$$

$$F_1^1(0) = 2 + (0+1) + 9 = 12$$

$$F_1^2(0) = 2 + 1 + 12 = 15$$

$$F_1^3(0) = 15$$

$$F_1(0) = 15$$

$$F_1^0(20) = C_1 = 2$$

$$F_1^1(20) = 2 + (0+1) + 6 = 9$$

$$F_1^2(20) = 9$$

$$F_1(20) = 9$$

$$F_1^0(40) = C_1 = 2$$

$$F_1^1(40) = 2 + (0+1) + 10 = 13$$

$$F_1^2(40) = 2 + 1 + 19 = 22$$

$$F_1^3(40) = 2 + 1 + 22 = 25$$

$$F_1^4(40) = 25$$

$$F_1(40) = 25$$

$$F_1^0(60) = C_1 = 2$$

$$F_1^1(60) = 2 + (0+1) + 3 = 6$$

$$F_1^2(60) = 6$$

$$F_1(60) = 6$$

$$F_1^0(80) = C_1 = 2$$

$$F_1^1(80) = 2 + (0+1) + 7 = 10$$

$$F_1^2(80) = 10$$

$$F_1(80) = 10$$

As can be seen T_1 will miss its deadline at its third invocation (at 40). So T_1 's \mathbf{m} -pattern would be 11011 hence the sub sequences that we have to check (remembering that the pattern is repeated) are {11011, 10111, 01111, 11110, 11101}.

Checking T_1 's weakly hard constraint $\begin{pmatrix} 4 \\ 5 \end{pmatrix}$ shows that it is satisfied, despite the miss. This step can easily be implemented as an automation tool, see [36] for more details.

Now we have to update \hat{C}_{USED} with the calculated execution demand of T_1 . Since T_1 misses one deadline, but still satisfies its weakly hard constraint, the invocation where the miss appears will be skipped i.e. not added into the vector \hat{C}_{USED} .

The updated $\hat{C}_{USED} = [1, 1, 2, 0, 0, 0, 2, 0, 2, 0, 0, \dots]$

We continue with calculating finalization times for T_2 using the updated \hat{C}_{USED} :

Table 8 Finalization time calculations for T_2

$F_2^0(0) = C_2 = 3$	$F_2^0(50) = C_2 = 3$
$F_2^1(0) = 3 + (1 + 1) + 9 = 14$	$F_2^1(50) = 3 + (0 + 1) + 9 = 13$
$F_2^2(0) = 3 + (2 + 1) + 12 = 18$	$F_2^2(50) = 3 + (2 + 1) + 12 = 18$
$F_2^3(0) = 18$	$F_2^3(50) = 3 + 3 + 12 = 18$
$F_2(0) = 18$	$F_2(50) = 18$

Today must be our lucky day, since T_2 meets all of its deadlines. The \mathbf{m} -pattern for T_2 would be 11 and the updated $\hat{C}_{USED} = [1, 1, 5, 0, 0, 1, 4, 0, 2, 0, 0, \dots]$. The weakly hard constraint $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ for T_2 is of course satisfied.

A quick check shows that the updated \hat{C}_{USED} , \hat{C}_s and the interrupt gives an utilization of 0.85:

$$U_{total} = \frac{1+1+5+1+4+2}{100} + \frac{9+3+6+7+10}{50} + \frac{1}{100} = 0.14 + 0.70 + 0.01 = 0.85$$

That is less than our calculations earlier (0.87). It is lesser because we skipped an invocation of T_2 .

The example shows a scenario where the tasks T_1 and T_2 would not be schedulable as Red tasks in Rubus. The task set would be unschedulable since all Red tasks must meet all their deadlines. Using the weakly hard concept shows that the tasks are schedulable. Even though T_1 misses a deadline, it still delivers a defined quality of service.

3.5.3.6 Improving the suggested concept at Volvo

This following sections deal with possible improvements to our suggested solution.

Off-line improvements

When calculating \mathbf{m} -patterns for tasks it may be the case that the weakly hard task with highest priority will meet all of its deadlines (when using worst-case execution times) in a window of time i.e. it satisfies $\binom{m}{m}$ in both normal and high mode. In doing this the task will allocate much more resources than necessary, especially if the requirements in high mode are $\binom{n}{m}$ and n is much lesser than m . To avoid this situation the \mathbf{m} -pattern must be loosened to satisfy $\binom{n}{m}$ instead of $\binom{m}{m}$ in high mode. This gives lower priority weakly hard tasks greater possibilities to satisfy their constraints. For example, in the finalization calculation example above (section 3.5.3.5), the \mathbf{m} -pattern for task T_2 is 11, it could be lightened up to 10 still satisfying the weakly hard constraint of $\binom{1}{2}$.

The updating of \hat{C}_{USED} must be done corresponding to the loosened \mathbf{m} -pattern i.e. wherever there is a zero in the \mathbf{m} -pattern, the execution demand should not be added to \hat{C}_{USED} .

This problem can be avoided by adding optimal priorities to the tasks. This is done by switching pairs of tasks until the optimal priority assignment is found or until it is determined that the task set is not schedulable. The algorithm exploits the fact that strongly hard tasks should retain the deadline priority assignment between them, and that weakly hard tasks that are not strongly hard are interleaved between the strongly hard tasks. [36]

If any given weakly hard constraint remains unsatisfied there is a possibility to calculate and create a set of constraints that may be satisfied instead.

Consider the \mathbf{m} -pattern 1101011, it is then possible to calculate all $\binom{n}{m}$ that will be satisfied by the pattern. In [36] the author gives an example of how this should be performed. It is suggested to find the smallest m that will satisfy a constraint.

For example, one could start with testing if $\binom{1}{1}$ is satisfied. If it is unsatisfied then continue to check if $m+1$ will be satisfied.

Given the \mathbf{m} -pattern 1101011, the smallest m that will be satisfied for $n = 1$ is 2.

So $\binom{1}{2}$ is satisfied. According to corollary 39 in [36] a k -sequence will also satisfy less demanding constraints that is, one that requires less 1's or one that extends over a wider window.

Each time a constraint is found as satisfied n could be increased by $n + 1$ in order to find the complete set of constraints that will be satisfied.

On-line mechanism

As we mentioned earlier the Weakly Hard concept guarantees a minimum QoS to enhance the actual system utilization. The approach we have mentioned so far always skip the activation of task instances when they have a 0 in the m -pattern. This approach however always gives *the* minimum QoS (i.e. nor less or more) to the system, and hence it does not enhance the actual system utilization when possible.

To overcome this problem dynamic scheduling should be used, e.g. some decisions about which task to dispatch should be performed on-line during run-time. In [36] Bernat gives two very simple algorithms to use on-line to enhance the performance of the system. When using these on-line algorithms a minimum QoS (which may be enhanced at run-time) can be guaranteed to the system. The algorithms can also be used to enhance the responsiveness of soft tasks.

See section 2.4 Weakly Hard systems for more information about the on-line mechanism.

The basic idea behind the described on-line algorithms is very simple: if a task does not have to be invoked for the Weakly Hard constraints to be satisfied, it is possible to put the time required for the task in a better use.

3.6 Support for the weakly hard concept in the development process

The following sections describe the minimal amount of changes needed to the development process in order to apply the concept of weakly hard systems as explained above. The descriptions are sectioned to follow the development process at Volvo because we want to apply support for the concept to each phase in the process.

3.6.1 Managing software requirements for weakly hard systems

As stated earlier, a requirement is a feature of the system or a description of something the system is capable of doing, in order to fulfill the systems purpose. To distinguish the requirements from the design, one can think of requirements as identifiers of *what* the system is suppose to do, and the design as *how*.

At Volvo the customer gives requirements with temporal specifications for all functionality. The deadlines are stated without further explanations on what they are based upon. This makes it difficult for the software designers to decide whether any functionality can be thought of as weakly hard or not. The difficulties arise because of the complexity of the system. It is difficult, if not impossible, for the software designers to have a complete view of all functionalities and possible consequences of missed deadlines in the system. So if the temporal requirements for weakly hard tasks are to be received from a customer in the future, a change in their notations will be necessary.

Also, using weakly hard constraints for tasks leads to a question of who should state the maximum number of allowed deadline misses for a task, the customer or the software designer.

We suggest staying to the way timing requirements are given today, but with explanations about possible consequences of missed deadline(s). This will give the software designers a possibility to consider if functionalities can be implemented as weakly hard tasks, depending on the given consequences. In addition, an explanation of what the deadline(s) is based upon, and a short notice of how they were derived, would be nice and very useful for the same reason.

In doing this both the customer and the developers could communicate and agree about the realism of the requirements. Hopefully the communication with the customer will lead to a better understanding of problems involved in designing and implementing of functionality in a real-time system.

The reason to why we believe this approach is the most suitable is the fact that it will not require the customer to know anything about weakly hard systems. Consider the case where the customer should state weakly hard constraints. It requires the customer to be familiar with relationships between weakly hard constraints, in order to decide whether any constraint will be considered harder than others by the designers e.g. is $\begin{pmatrix} 3 \\ 5 \end{pmatrix}$ harder than $\begin{pmatrix} 4 \\ 6 \end{pmatrix}$? As a consequence the priorities between specifications may be harder to establish for the customer.

There is however some side effects in specifying the requirements as suggested above. We believe that if two designers should interpret the specifications and decide for a suitable weakly hard constraints, they would probably suggest different ones. This fact shows the importance of specifying requirements in a way that reduces misinterpretation. There is probably no way of getting around this problem other than relying on the routine of the designers in choosing weakly hard constraints.

3.6.2 Designing for weakly hard systems

All weakly hard task in the system at Volvo, are to be implemented as blue tasks in Rubus, so when designing it might be helpful to have that in mind. The described attributes for the “*Guaranteed blue ET and TT tasks*” solution see section 3.5.2 holds for designing together with our suggestion of weakly hard concept i.e. tasks with larger period times than the major cycle may be used in the design.

When using weakly hard tasks in any system there is a need for a way of calculating if weakly hard constraints can be satisfied. The suggested applications can be seen as automation tools in order to analyze if weakly hard constraints are satisfied. If constraints can not be satisfied, they may be weakened according to suggestions given in earlier section “*characteristics of relationships between constraints*” about weakly hard systems. This can also be implemented in an automation tool, i.e. it would be great if the automation tool responsible for guaranteeing the WHS task set return a constraint that can be satisfied for that task.

A problem that might appear, when using weakly hard constraints, is for the designer to choose the number of either missed or met deadlines during a window of time. As help in choosing artifacts such as the maximum number of deadline misses in a window of time, the requirements specifications must serve as a source of support. Since WHS distinguishes between different types of deadline

misses, it is important for the designers to know the appropriate types tolerated in a system. Consecutive deadline misses could for example be tolerated in some parts of a system whereas non-consecutive misses may be the only acceptable ones in other parts of the same system.

As an example, when using weakly hard constraints in a control system it is important to know that control systems may become unstable if a certain number of deadlines are missed in a row [4].

Functionalities responsible for logging any behavior could probably be a part that tolerates non-consecutive deadline misses only. At Volvo it does not probably matter if some information is missed now and then, but missing several pieces of information in a row could make it difficult to interpret the log.

It is worth mentioning that choosing constraints should also be done with the number of invocations per hyper period for a task, in mind. E.g. if a task is invoked 10 times in H_i , then choosing the m in the constraint as a multiple of 10, makes calculations easier.

Furthermore the designer should choose parts of the system that may be modeled as weakly hard. Clearly not all functionalities are suitable for the WHS concept. In choosing what may be designed as weakly hard, one could think of functionalities that do not have to be hard nor soft, but something in between.

The suggested parts of the system that may be subject to WHS at Volvo are:

- Most of the visual interaction with the driver, since the criticality of such information is usually not high.
- For tasks with a desired period greater than 100ms. These tasks may also be modeled, not only by the easy approach, but also as weakly hard tasks.
- Parts of control applications, as described in earlier section about general design considerations.
- Some logging functionalities, especially where the nature of the changes in the logged data is proportional.

As explained earlier, the finalization calculations that must be performed for weakly hard tasks, computes the total interference from all equal or higher priority tasks. If a weakly hard task should have a real Rubus priority level of 10, and an existing blue task in the system should have a higher priority, the blue task would interfere with the weakly hard task. But since the blue task lacks any specified worst-case execution time, there would be no way of knowing how much interference the blue task would cause. So we suggest using highest available Rubus priorities for weakly hard tasks in order to avoid unexpected interferences from blue background tasks.

Using non multiple period times for weakly hard tasks does avoid conjunction at certain release chains, but it may increase the length of the hyper period i.e. increasing the amount of computations required for weakly hard feasibility checks.

Future work

As future work we suggest changes to Rubus development kit to support the concept of weakly hard systems. The task model could include weakly hard constraints and attributes necessary to guarantee blue tasks (strongly hard tasks in the weakly hard concept). Integrated automation tools for checking satisfaction of weakly hard constraints etc. would also be appreciated.

An interesting thing to look more at would be to change the on-line skipping strategy of tasks in weakly hard systems. Instead of skipping tasks that does not meet their deadlines one could pre-empt them and let them continue their execution at the next invocation of the task. This might be possible since the tasks TCB (task control block) holds and preserves the value of the program counter between task activations. The advantages would be a very simple on-line mechanism which results in low overhead and the performance would be equal to the on-line approach of the WHS. The disadvantages would be that computations may be performed on old data, however in most cases this might not a problem.

Furthermore it would be of great interest to have different bands of priorities for the different task sets in the system. Interrupts should belong to the “band” of highest priority and then weakly hard tasks and finally soft background tasks would have the lowest priority. This would make it much easier to understand and use the system. It would also prevent mixing of task priorities between different task types, e.g. prevent a typo to give a soft task higher priority than a weakly hard task.

Conclusions

We have given some suggestions on how to increase the actual utilization in the real-time systems at Volvo. In doing this we have considered the main parts of the development process, and applied support for the suggestions to each major activity.

The main suggestion introduces a new way of using Blue tasks, namely as weakly hard. The known concept of weakly hard systems is based upon a formal theory for allowing an upper bound of predictable deadline misses. By applying the weakly hard concept we have theoretically shown that unschedulable tasks sets may in fact be schedulable with an acceptable bound of deadline misses, still guaranteeing a specified quality of service at Volvo.

We have also shown an example of an easier implementation for increasing the use of system resources at Volvo. The suggested solution allows tasks, with larger period times than the cyclic schedules major cycle, to be scheduled.

We believe that the solutions are well suited to the architecture at Volvo. Moreover they are actually implementable implying that they may be used without too much effort.

The methods we used to complete the objectives were well defined and performed in an order making it easy to complete the objectives.

References

- [1] Buttazzo. G, Spuri. M, Sensini, F, “*Value vs. deadline scheduling in overload conditions*”, Real-Time Systems Symposium, 1995. Proceedings, 16th IEEE, 1995
Page(s): 90 –99
- [2] Kopetz. H, Zainlinger. R, Fohler. G, Kantz. H, Puschner. P, Schutz. W, “*The design of real-time systems: from specification to implementation and verification*”, Software Engineering Journal, Volume: 6 Issue: 3, May 1991
Page(s): 72 –82
- [3] Kalinsky. D, Ready. J, “*Distinctions between requirements specification and design of real-time systems*”, Software Engineering for Real Time Systems, 1989., Second International Conference on , 1989
Page(s): 26 –30
- [4] Bernat. G, Burns. A, Liamsi. A, “*Weakly hard real-time systems*”, Computers, IEEE Transactions on, Volume: 50 Issue: 4, April 2001
Page(s): 308 –321
- [5] Mejia-Alvarez. P, Melhem. R, Mosse. D, “*An incremental approach to scheduling during overloads in real-time systems*”, Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE, 2000
Page(s): 283 –293
- [6] Ramanathan. P, “*Overload management in real-time control applications using (m, k)-firm guarantee*”, Parallel and Distributed Systems, IEEE Transactions on , Volume: 10 Issue: 6 , June 1999
Page(s): 549 –559
- [7] Laplante. P.A, “*Design issues in real-time*”, IEEE Potentials, Volume: 12 Issue: 1 , Feb. 1993
Page(s): 24 –26
- [8] Audsley. N.C, Bate. I.J, Burns, A, “*Flexible scheduling theory for advanced engine controllers*”, Hybrid Control for Real-Time Systems (Digest No: 1996/256), IEE Colloquium on, 1996
Page(s): 8/1 -8/3
- [9] Poledna. S, “*Tolerating sensor timing faults in highly responsive hard real-time systems*”, Computers, IEEE Transactions on, Volume: 44 Issue: 2, Feb. 1995
Page(s): 181 –191

References

- [10] Bernat. G, Burns. A, “*Three obstacles to flexible scheduling*”, Real-Time Systems, 13th Euromicro Conference on, 2001. , 2001
Page(s): 11 –18
- [11] Burns. A, “*Scheduling hard real-time systems: a review*”, Software Engineering Journal, Volume: 6 Issue: 3, May 1991
Page(s): 116 –128
- [12] Stankovic. J.A, Spuri. M, Di Natale. M, Buttazzo. G.C, “*Implications of classical scheduling results for real-time systems*”, Computer, Volume: 28 Issue: 6, June 1995
Page(s): 16 –25
- [13] Saksena. M, “*Real-time system design: a temporal perspective*”, Electrical and Computer Engineering, 1998. IEEE Canadian Conference on, Volume: 1, 1998
Page(s): 405 -408 vol.1
- [14] Uhrig. J.L, “*System requirements specification for real-time systems*”, Computer Software and Applications Conference, 1978. COMPSAC '78. The IEEE Computer Society's Second International, 1978
Page(s): 241 –245
- [15] Norstrom. C, Gustafsson. M, Sandstrom. K, Maki-Turja. J, Bankestad. N.-E, “*Experiences from introducing state-of-the-art real-time techniques in the automotive industry*”, Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the, 2001
Page(s): 111 –118
- [16] Sandström. K, Norström. C, ”*Managing complex temporal requirements in real-time control systems*”, Engineering of Computer-Based Systems, Proceedings. 9th Annual IEEE International Conference and Workshop, 2002
Page(s): 103-109
- [17] Nyström. D, Tesanovic. A, Norström. C, Hansson. J, Bånkestad. N-E, “*Data Management Issues in Vehicle Control Systems: a Case Study*”, Euromicro Real-Time Conference 2002, June 2002
- [18] Buttazzo. G, “*Hard real-time computing systems: Predicable scheduling algorithms and applications*”, Kluwer academic publishers, 1997.
- [19] Lawrence Pfleeger. S, “*Software engineering, theory and practice*”, International edition, Prentice Hall Inc, 1998.
- [20] Hooks. I, “*Writing good requirements: A requirements working group information report*”, Proceeding of the third international symposium of the NCOSE, Volume: 2, 1993.
- [21] Buttazzo. G, Caccamo. M., “*Minimizing aperiodic response times in a firm real-time environment*”, Software Engineering, IEEE Transactions on, Volume: 25 Issue: 1, Jan.-Feb. 1999
Page(s): 22-32

References

- [22] Mäki-Turja, J., Sjödin, M., “*Response-Time Analysis for Dynamically and Statically Scheduled Systems*”, MRTC Report no. 71, October 2002
- [23] Quan, G., Hu, X., “*Enhanced Fixed-Priority Scheduling with (m-k)-Firm Guarantees*”, Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE, 2000
Page(s): 79 –88
- [24] Bernat, G., Burns, A., “*Cobmining (n, m)-Hard deadlines and Dual Priority Scheduling*”, Real-Time Systems Symposium, 1997. Proceedings, The 18th IEEE, 1997
Page(s): 46 -57
- [25] Mok, A.K., Chen, D., “*A multiframe model for real-time tasks*”, Software Engineering, IEEE Transactions on, Volume: 23 Issue: 10, Oct. 1997
Page(s): 635 -645
- [26] Hansson, J., Andler, S. F., Son, S. H., “*Value-Driven Multi-Class Overload Management*”, Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on, 1999
Page(s): 21 -28
- [27] Dobrin, R., Fohler, G., Puschner, P., “*Translating Off-line Schedules into Task Attributes for Fixed Priority Scheduling*”, Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE, 2001
Page(s): 225 -234
- [28] Dobrin, R., Özdemir, Y., Fohler, G., “*Task Attribute Assignment of Fixed Priority Scheduled Tasks to Reenact Off-line Schedules*”, Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on, 2000
Page(s): 150 -154
- [29] Molenkamp, G., Katchabaw, M., Lutfiyya, H., Bauer, M., “*Managing Soft QoS Requirements in Distributed Systems*”, Parallel Processing, 2000. Proceedings. 2000 International Workshops on, 2000
Page(s): 461 -468
- [30] Isovich, D., Fohler, G., “*Efficient Scheduling of Sporadic, Aperiodic, and Periodic Tasks with Complex Constraints*”, Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE, 2000
Page(s): 207 -216
- [31] Beccari, G., Caselli, S., Reggiani, M., Zanichelli, F., “*Rate Modulation of Soft Real-Time Tasks in Autonomous Robot Control Systems*”, Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on, 1999
Page(s): 21 -28
- [32] Davis, R., Wellings, A., “*Dual Priority Scheduling*”, Real-Time Systems Symposium, 1995. Proceedings, 16th IEEE, 1995
Page(s): 100 -109

References

- [33] Lee. W, Sabata. B, “*Admission Control and QoS Negotiations for Soft-Real Time Applications*”, Multimedia Computing and Systems, 1999. IEEE International Conference on, Volume: 1, 1999
Page(s): 147 -152 vol.1
- [34] Koren. G, Shasha. D, “*Skip-Over: Algorithms and Complexity for Overloaded Systems that Allow Skips*”, Real-Time Systems Symposium, 1995. Proceedings, 16th IEEE, 1995
Page(s): 110 -117
- [35] Chung. J.-Y., Liu. J.W.S., Lin. K.-J, “*Scheduling periodic jobs that allow imprecise results*”, Computers, IEEE Transactions on, Volume: 39 Issue: 9, Sept. 1990,
Page(s): 1156 –1174
- [36] Bernat. G, “*Specification and Analysis of Weakly Hard Real-Time Systems*”, PhD thesis, Dep. Ciències Matemàtiques I Informàtica. Universitat de les Illes Balears. Spain, Jan. 1998.
- [37] Liu. C. L., Layland. J.W., “*Scheduling algorithms for multiprogramming in a hard real-time environment*”, Journal of the association for computing machinery, Volume: 20, no: 1, Jan. 1973
Page(s): 46-61
- [38] Audsley. N. C., et.al., “*Fixed priority pre-emptive scheduling: An historical perspective*”, The international journal of time-critical computing systems, Volume: 8, no: 2/3, March/May 1995.
- [39] Fohler. G., “*Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems*”, Real-Time Systems Symposium, 1995. Proceedings, The 16th IEEE , Dec. 1995
Page(s): 152-161
- [40] Arcticus systems, The Rubus operating system. <http://www.arcticus.se>
- [41] Joseph. M, Pandya. P, “*Finding response times in a real-time system*”, Comput. J., Volume 29, no: 5, 1986
Page(s): 390 – 395
- [42] Audsley. N, Burns. A, Richardson. M, Tindell. K, Wellings. A.J, “*Applying new scheduling theory to static priority pre-emptive scheduling*”, Software Engineering Journal, Volume: 8 Issue: 5, Sep 1993
Page(s): 284 -292