Mälardalen University Doctoral Thesis
No.61

Efficient Memory Utilization in Resource Constrained Real-Time Systems

Kaj Hänninen

June 2008

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-4238
ISBN 978-91-85485-85-7
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

This thesis presents design and run-time techniques for efficient memory utilization in embedded real-time systems. The proposed techniques give developers means to reduce the memory consumption in the systems. Altogether, this gives possibilities to increases the added value of industrial systems, in the sense that more features can be fitted on existing hardware.

The thesis begins by presenting the results of a series of interviews concerning common requirements in development of embedded real-time systems. Based on these results, the thesis presents a novel component model for development of resource constrained real-time systems. The model supports efficient memory usage through stack sharing and is formal enough to enable predictability of the resulting stack usage. To provide run-time support for stack sharing, the thesis presents an integration of the stack sharing strategy in an operating system for the component model. To determine the resulting memory usage under stack sharing, a novel analysis method is presented. In an evaluation, the thesis show that the analysis method is both fast and that it gives tight bounds on the resulting stack usage, which makes it suitable for industrial use. The thesis ends with a presentation showing the integration of the proposed analysis technique in an integrated development environment.

The proposed techniques have been integrated in the commercial tool Rubus-ICE from Arcticus Systems. The techniques will be available for developers in the upcoming release of Rubus-ICE.

To Stissen, Chibi, Krutte, Morris and Azze

Acknowledgements

I wish to thank all my colleagues at Arcticus Systems and Mälardalen Real-Time Research Centre. A special thank you goes out to: Mikael Nolin, Jukka Mäki-Turja, Christer Norström, Markus Bohlin, Jan Carlson, Sasikumar Punnekkat, Kurt-Lennart Lundbäck, John Lundbäck, Hjördis Lundbäck, Mats Lindberg and Staffan Sandberg.

This research has been supported by Arcticus Systems AB[1], Mälardalen Real-Time Research Centre[2] and SAVE-IT[3]. SAVE-IT is an industrial graduate school supported by the KK-foundation[4].

Thank you!

Kaj Hänninen

[1] www.arcticus-systems.com
[2] www.mrtc.mdh.se
[3] www.mrtc.mdh.se/projects/save-it
[4] www.kks.se

List of Publications

Publications included in this thesis

Paper A: Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain*, In Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Potsdam, Germany, March, 2006.

Paper B: Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin, *Efficient Development of Real-Time Systems Using Hybrid Scheduling*, In Proceedings of the 2005 International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Paper C: Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, Kurt-Lennart Lundbäck, *The Rubus Component Model for Resource Constrained Real-Time Systems*, To appear in the Proceedings of the 3rd IEEE International Symposium on Industrial Embedded Systems, Montpellier, France, June, 2008.

Paper D: Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin, *Efficient Event-Triggered Tasks in an RTOS*, In Proceedings of the 2005 International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Paper E: Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson, Mikael Nolin, *Determining Maximum Stack Usage in Preemptive Shared Stack Systems*, In Proceedings of the 27th IEEE Real-Time Systems Symposium, Rio de Janeiro, Brazil, December, 2006.

Paper F: Kaj Hänninen, Jukka Mäki-Turja, Staffan Sandberg, John Lundbäck, Mats Lindberg, Mikael Nolin, Kurt-Lennart Lundbäck, *Introducing a Plug-In Framework for Real-Time Analysis in Rubus-ICE*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-229/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2008. Submitted to the 13th IEEE International Conference on Emerging Technologies and Factory Automation.

Publications relevant to thesis but not included

- Markus Bohlin, Kaj Hänninen, Jukka Mäki-Turja, Jan Carlson, Mikael Nolin, *Bounding Shared-Stack Usage in Systems with Offsets and Precedences*, To appear in the Proceedings of the 20th Euromicro Conference on Real-Time Systems, Prague, Czech Republic, July, 2008.
- Mikael Nolin, Jukka Mäki-Turja, Kaj Hänninen, *Achieving Industrial Strength Timing Predictions of Embedded System Behavior*, To appear in the Proceedings of the 2008 International Conference on Embedded Systems and Applications, Las Vegas, USA, July, 2008.
- Jukka Mäki-Turja, Mikael Nolin, Kaj Hänninen, *Towards Efficient Development of Embedded Real-Time Systems, the Component Based Approach*, In Proceedings of the 2006 International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2006.

Other publications by the author

- Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, Kurt-Lennart Lundbäck, *Supporting Engineering Requirements in the Rubus Component Model*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-223/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2008.
- Markus Bohlin, Kaj Hänninen, Jukka Mäki-Turja, Jan Carlson, Mikael Nolin, *Safe Shared Stack Bounds in Systems with Offsets and Precedences*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-221/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, January, 2008.

- Markus Bohlin, Kaj Hänninen, Jukka Mäki-Turja, *Shared Stack Analysis in Transaction-Based Systems*, Work in Progress Proceedings RTSS'07, Tucson, Arizona, USA, December, 2007.

- Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson, Mikael Nolin, *Determining Maximum Stack Usage in Preemptive Shared Stack Systems*, In Proceedings of the 9th Real-Time in Sweden, Västerås, Sweden, August, 2007.

- Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin, *Efficient Development of Real-Time Systems Using Hybrid Scheduling*, In Proceedings of the 9th Real-Time in Sweden, Västerås, Sweden, August, 2007.

- Kaj Hänninen, *Introducing a Memory Efficient Execution Model in a Tool-Suite for Real-Time Systems*, Licentiate Thesis, Mälardalen Real-Time Research Centre, Mälardalen University, September, 2006.

- Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson, Mikael Nolin, *Analysing Stack Usage in Preemptive Shared Stack Systems*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-202/2006-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, July, 2006.

- Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Investigation of Industrial Requirements in Development of Embedded Real-Time Systems*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-185/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, August, 2005.

- Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Industrial Requirements in Development of Embedded Real-Time Systems -Interviews with Senior Designers*, In Proceedings of the Work-in-Progress Session of the 17th Euromicro Conference on Real-Time Systems, Palma de Mallorca, Spain, July, 2005.

- Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin, *Introducing Resource Efficient Event-Triggered Tasks in an RTOS*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-170/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2005.

- Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin, *Response Times in Hybrid Scheduled Systems*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-169/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2005.
- Kaj Hänninen, Jukka Mäki-Turja, *Component technology in Resource Constrained Embedded Real-Time Systems*, Technical Report, Mälardalen Real-Time Research Centre, Mälardalen University, March, 2004.
- Toni Riutta, Kaj Hänninen, *Optimal Design*, Master's thesis, Mälardalen University, Department of Computer Science and Engineering, February, 2003.

Contents

I

Thesis

Chapter 1

Introduction

Embedded computers play an important role in people's everyday life. Products that used to be purely mechanical have incorporated computers to control features and provide possibilities for more advanced functionality. Nowadays, we can find computers in washing machines, DVD players, cellular phones, cars and in a multitude of other products.

Throughout the years, the number of embedded systems has been increasing and the trend indicates that the number of embedded systems will continue to increase. With every new release of a system, the number of advanced features is increasing either due to customer demands, legislation requirements or simply because manufacturers want to increase the added value of their products by incorporating new features. As a consequence, the complexity in development of embedded systems is increasing.

Component based development is slowly gaining popularity as a development discipline in the embedded domain. The component based development discipline enables system development at different levels of abstraction through hierarchical composition of software units, i.e., components. It has been successfully used in development of complex desktop applications for quite some time, and is now being introduced as an alternative, and promising, way to cope with the complexity in development of embedded systems.

Even though component based development has been introduced for embedded development, many embedded systems are still being realized with considerably simple methods. These methods originates from way back when embedded systems almost exclusively consisted of control features, i.e., systems in which all features have similar, typical hard real-time, requirements.

Today however, the diversity of features, and mixture of real-time requirements in embedded systems, entail that these development methods are adequate only for a subset of the features in a system. For instance, in real-time systems, static scheduling is often used and sometimes even mandated, for realization of safety critical functionality. For reasons, such as, tradition in development, simplicity of use, or lack of proper support in development tools, static scheduling is sometimes used to realize all features. Even though researchers have proposed a large number of alternative models, in practice however, only a few of the proposed models are used and supported in industrial development tools. This implies that developers often force fit functionality with different requirements to be executed under a single execution model, such as the static scheduled model. This force fitting of functionality increases software complexity and leads to poor resource utilization.

Many embedded systems are resource constrained in the sense that they have a limited amount of computational resources. Processors for embedded systems have only a fraction of the processing power of desktop computers. In addition, the amount of available memory in embedded systems is significantly lower than what is common in desktop computers. To improve resource utilization, additional execution models need to be introduced in development. However, not all execution models are suitable for embedded systems. In fact, an execution model must be introduced with care, since different sub-domains within the embedded domain have different requirements that need to be supported by the execution model. Properties such as safety and reliability are considered important and need to be addressed in development of embedded systems. This implies that execution models need to be verifiable, e.g., by formal analysis or manual testing. This in turn, require analysis methods applicable for industrial use, i.e., analysis methods that are easy to use and have low timing complexity.

This thesis will show how software development for embedded real-time systems can be made more efficient, with respect to hardware utilization. Specifically, the thesis will present the integration of a predictable memory efficient execution model, denoted stack sharing, in a development context for component based development of embedded real-time systems. To address the issues outlined in the introduction, the thesis presents a novel component model that support common requirements in development of control systems for ground vehicles. The component model supports multiple execution models, giving developers means to avoid force fitting of functionality. Also, the component model is developed to support predictable and efficient memory usage through stack sharing. To achieve efficient memory usage at run-time, the

thesis presents an integration of stack sharing in an operating system for the proposed component model. In addition, a novel analysis method to predict the memory consumption under stack sharing is presented and integrated in a development environment.

1.1 Thesis problem formulation

To facilitate efficient development of complex industrial embedded systems, suitable development models and efficient execution models for realization of the systems are needed. The models should facilitate development of resource constrained systems and allow features with different real-time requirements to be realized in a single system. Also, the models need to be formal enough for analysis and verification of properties and requirements. At the same time, to gain industrial use, the models need to be adapted to the specific requirements in industrial development and integrated in development tools. Addressing these issues, this thesis investigates the following:

- What are the requirements in development, and properties, of embedded real-time systems?
- How can industrial real-time systems benefit from additional execution models?

Based on these investigations, the thesis then address:

- Component based development of embedded real-time systems, supporting efficient memory usage through stack sharing.
- Integration of stack sharing in a run-time environment.
- Predictability of the memory consumption under stack sharing, through formal analysis methods.
- Integration of an analysis method for stack sharing, in a development environment.

1.2 Outline of thesis

The reminder of this thesis is outlined as follows:

Part I

Chapter 2 presents the background for the work conducted in this thesis. The chapter gives an introduction to real-time systems, development of real-time systems and execution models for real-time systems.

Chapter 3 presents an efficient execution model, called stack sharing, for real-time systems. The chapter outlines issues that need to be considered in integration of stack sharing in a commercial development environment. The chapter also presents how this thesis addresses these issues in integrating the model in a development context.

Chapter 4 presents the overall contribution of the thesis. The chapter also outlines the contribution of the author.

Chapter 5 concludes the work presented in this thesis and presents directions for future work.

Part II

Part II of this thesis contains six papers that constitute the scientific contributions of this thesis.

Chapter 2

Real-time systems

Real-time systems can be found almost everywhere today. Many of us use them on a daily basis. For example, a modern car contains some 40-80 computers, most of which are classified as real-time systems. Most real-time systems are embedded systems that interact with the environment by sensors and actuators. An embedded real-time system in a car, for example, can be responsible of controlling functions such as anti-lock braking, anti-skid, traction control and other safety features as well as entertainment features.

The main distinction between real-time systems and conventional desktop systems stem from the way in which timing is considered. In conventional computer systems such as desktop systems, functional correctness is of main concern, whereas real-time systems emphasize both functional and timing requirements in computing.

The air bag, a safety feature in most cars, is a classical example that demonstrates timing requirements in a computer system. The air bag is an inflatable nylon bag, activated at collision and filled with a gas to protect the passengers. There are several timing aspects that need to be considered for correct functionality of an air bag. Inflating the bag too early, i.e., directly when the collision is detected, might render the bag useless, in the sense that the gas may already have escaped the bag when the passenger hits the bag. Inflating the bag too late, might actually be dangerous for passengers, in the sense that the inflating bag may hit the person with a great force. In essence, to provide proper protection, the bag needs to be inflated within a certain time interval after the collision is detected. The air bag is a typical feature with strict timing requirements and thus can be classified as a real-time system.

2.1 Computational resources

Most embedded real time systems are resource constrained in some sense. The processing power and the amount of memory in the systems are usually only a fraction of what is common in desktop computers. For instance, back in 1990, a processor running at 25MHz was considered as high performance in a desktop system. Nowadays, with processor speeds reaching 4GHz, processors running at 25MHz are of no use in desktop computers. However, in the embedded domain, the use of low performance processor, such as 25MHz processors, is not unusual. In addition, the amount of memory in a typical embedded system is not even comparable to what is common in desktop computers. Typically, the amount of available memory in embedded systems lies in the range of kilo-bytes to mega-bytes. It is not uncommon that the features of a real-time system are realized by, as many as, hundreds of threads that execute and compete for the available resources in the same system. Hence, the limited resources in embedded systems need to be utilized in an efficient way.

2.2 Trends on functionality

Products that used to be purely mechanical have incorporated computers to provide more advanced features and to increase the added value of the products. Incorporating computers in products offers flexibility in the sense that new features can be added in the computer software. The software in modern embedded real-time systems account for an important part of the value growth of the products [HNN08]. Embedded real-time systems have, by tradition, been used to control physical processes. Consequently, the software in these systems consisted almost exclusively of control algorithms with strict timing requirements. With the flexibility offered by software, the number of features introduced in embedded real-time systems has been increasing over the years. Nowadays, embedded real-time systems are often responsible for handling a variety of different type of features with different requirements. For instance, a typical embedded real-time system in a passenger car manages control features, driving assistance, information and entertainment features [SVN07]. This implies that embedded real-time systems need to manage both safety critical features as well as less critical features in the same system. In addition, the trend indicates that the number of features in embedded real-time system will continue to increase.

2.3 Development

The fact that many embedded real-time systems contain a mix of features with requirements ranging from hard real-time to soft, and even non real-time, raises challenges that developers of these systems need to deal with to deliver safe and reliable solutions.

Historically, embedded real-time systems have been developed using low level programming languages to guarantee full control over the system behaviour. Throughout the years however, development of embedded real-time systems has undergone changes to support the diverging requirements on applications. High level programming languages (for example C, C++, and Java) have been adopted as alternatives to the traditional low level programming languages. A recent trend is the introduction of component based development in the embedded community.

2.3.1 Component based development

Component based development is a discipline in which software systems are built by assembling pieces of software units known as components. Many component based development methodologies allow hierarchical composition of components, enabling development of systems at variable levels of abstraction. Abstracting away unnecessary details, allowing developers to focus on the problem at suitable abstraction levels, is in many cases a key in developing complex systems. In the office-/Internet-area component based development has had a tremendous impact. It has shown to be successful in development of complex applications and is slowly gaining popularity in the embedded domain. The main problem to adapt component based development to the embedded domain has been the lack of tools and component models that are able to fulfil domain specific requirements on safety and reliability as well as predictable and efficient resource usage for embedded systems. For this reason, many real-time systems are still developed with considerably simple methods, with main focus on guaranteeing functional correctness and temporal properties.

To support efficient component-based development of embedded real-time systems, at least three viewpoints should be taken in consideration, (i) the developers view, (ii) the analysis view, (iii) the view of the run-time system. These viewpoints need to be jointly approached to find a suitable trade-off between the requirements that need to be fulfilled. For example, a developer (or designer) of a system should have a component model, architectural rules

and constraints at his disposal to develop a high level (abstract away from pure source code) architecture of the application (system). From the analysis viewpoint, the design/architecture must be formal enough so that automated analysis techniques, such as response time and memory utilization analysis, can be performed. Finally, the run-time system should have a small run-time footprint, but still provide sufficient run-time services to the components of the application.

2.4 Execution models

In development of embedded real-time systems, a developer must choose an execution model to realize functionality. In general, an execution model can be seen as a method that provides ways to execute features, i.e., to carry out the functionality in a computer system. Each execution model affects the utilization of the processor as well as the memory consumption. For developers to utilize an execution model, the execution model needs to be supported at design and run-time, e.g., by a component model and by an operating system. Many operating systems only provide a few (often only one) execution models. In addition, very few component models have been explicitly designed to support multiple execution models. This implies that developers have to force fit functionality with diverging requirements to be executed under a single execution model. Force fitting functionality to be executed under a single execution model often results in increasing software complexity and poor resource utilization. Hence, as the need for new features in products increases, better usage of the limited computational resources is needed. This can be achieved by introducing resource efficient and predictable execution models in development.

2.5 Predicting run-time behaviour

To enable predictability of systems run-time behaviour, developers annotate models of real-time systems with temporal properties and requirements. Temporal requirements are often the ones in focus when predictability of real-time systems is addressed. With the increasing amount of functionality in resource constrained real-time systems, the need to verify properties such as timing and memory consumption, becomes more important.

There are a number of ways to verify timing requirements and to derive bounds on memory consumption in real-time systems. In an industrial context,

verification of a system is often performed by executing the system and measuring the properties. To verify whether a system meets its requirements, the measurements are compared with the requirements. It is obvious that such a verification approach seldom gives safe results, i.e., that the requirements are fulfilled, simply because the measurements must represent the worst case behaviour of the properties affecting the verification. Executing a system in a worst case manner with respect to some property, is non trivial and extremely hard to accomplish. Even so, this kind of verification is common in industry. In a more formal context, the requirements on a real-time system can be verified by mathematical analysis methods. Throughout the years, the research community have been active and proposed numerous analysis methods for real-time systems (see for example [ABT+93, BTW95, EHS97, KAS93, Leh90, Pun97, SRL87, SRL90, TB94, TC94, THW94, Tin92]). These methods are however non trivial to understand and often hard to incorporate in an industrial context. Experience shows that analysis methods are not easily transferred from an academic environment to industry [LR03, WAN03]. There are many reasons for this, originating foremost from differences in requirements and assumptions between industry and academia [NMTH08]. In addition, many analysis methods are applicable to small artificial systems, and experience therefore high timing complexity when applied to large real world systems. In essence, to successfully transfer complex analysis methods from academia to industry, the methods need to be incorporated in development tools and adhere to the requirements in the context in which they are integrated.

2.6 Research on predictable execution models

Research on real-time systems has provided a number of different scheduling algorithms, analysis methods and execution models, see, e.g., [ABRW91, But97, Liu00, LL73, SEF98, SSL89, SSRB98, XP90]. Recent results in real-time scheduling theory, e.g., [AB98, Abe98, Bak90, BBLB03, MTN05, PG03], make it possible to combine several execution models in one system (and still achieve predictable timing and efficient resource usage). However, these techniques by themselves, do not remove any problems. In addition, a change in the software development model and tools is needed.

Most of the execution models and analysis techniques proposed by the real-time research community have been developed with timing in focus. The aim of these models has been to guarantee predictability of timing requirements and to improve CPU utilization. Even though it is known that most embedded

real-time systems are resource constrained and that the systems need a certain amount of memory to realize features, relatively little work has been done on execution models and analysis for efficient memory utilization. A common problem that need to be addressed, when introducing means for efficient memory utilization in real-time systems, concerns how analysability of temporal requirements of functionality executed under the model can be guaranteed. Hence, models for efficient memory utilization should be predictable with respect to both the memory usage and timing behavior. There exist however, techniques to improve the memory usage in real-time systems, see for example [Bak90, CMM+04, CRM06, DMT00, GD07, GLN01, Dig, MSB05, Ros06, RRW05]. One of them is stack sharing.

Stack sharing is an execution strategy that allows several tasks to share a common memory area for execution. It is a strategy suitable for many type of systems in the sense that it reduces the overall memory consumption and puts only a few restriction on system development. For stack sharing to be successful, the strategy in itself needs to be integrated and supported by a development environment. Moreover, analysis methods that are able to give tight bounds on the resulting stack usage, under stack sharing, and that are applicable in an industrial setting, are needed.

2.7 Summary

With the increasing amount of diverging functionality in embedded real-time systems, ranging from safety critical functionality to non critical functionality, suitable and efficient execution models should be supported in development. Stack sharing is such a model. It is a predictable model applicable to a wide range of real-time systems.

To gain acceptance in industry, the model needs to be supported in tools and development models, such as component models. Moreover, to provide possibilities for predictability, analysis methods for stack sharing are needed. These methods should be applicable in an industrial setting, i.e., they should be developed to support relevant system models and give tight results whilst having low timing complexity. The analysis methods then need to be integrated in tools and adapted to the requirements in industrial development.

Chapter 3

Efficient memory utilization through stack sharing

3.1 Principle of stack sharing

Computer systems utilize threads and processes to realize features. In real-time systems, the sequential execution of program code is usually performed by tasks. Each task requires a certain amount of memory and execution time to realize a feature, i.e., to execute the code that define the feature. A task is commonly defined by (i) a task control block (ii) the actual task code (program code) and (iii) a stack memory area.

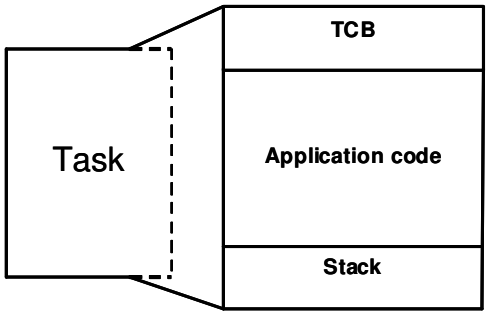

Figure 3.1: Common task structure. Each task has its own stack

The task control block is used to save the internal states of a thread. The task code is the application written by a developer and the stack is a memory area commonly processed in a last-in-first-out manner. The stack is used to store register values, function parameters, local variables and memory addresses. In most computer systems, each task has its own stack area allocated (see Fig. 3.1). Consequently, in systems with a large number of tasks the total amount of memory needed for the run-time stacks can be large. In addition, many systems allow high priority interrupts to utilize the stack of a task, implying that each task also need to have stack space allocated for interrupts.

To reduce the amount of stack memory in systems, stack sharing can be supported among tasks. Stack sharing is an execution strategy that allows several tasks to share a common memory area as a shared run-time stack (Fig. 3.2).

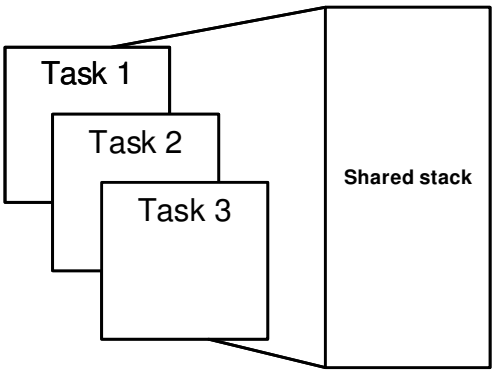

Figure 3.2: Stack sharing among tasks. Several tasks utilize the same stack

The principle of stack sharing is very simple. It is easiest demonstrated by an example. Consider two tasks (task A and task B) sharing a stack. Say that task A executes and pushes data, i.e., adds data to the stack, and later on gets preempted by task B. Then task B simply continues to utilize the stack from the preemption point (see Fig. 3.3). This implies that the stack pointer, i.e., the pointer to the first available memory area in the stack, needs to be shared among task A and task B. The possibility to reduce the total memory usage under stack sharing, originates from the fact that several tasks can actually share the same memory cells at different points in time. In essence, the basic principle is to save resources by sharing them.

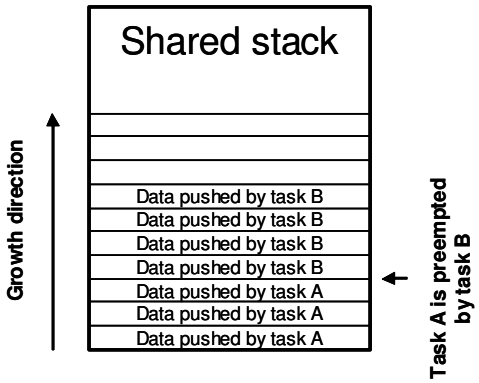

Figure 3.3: Stack sharing between task A and task B. When task A is preempted by task B, task B continues to utilize the stack from the preemption point

Throughout the years, a number of publications, e.g., [Bak90, CMM+04, DMT00, GLN01, GD07, MSB05, RRW05] have addressed stack sharing. Stack sharing has been supported by interrupt-driven systems for quite some time and has also been adopted for pure time-driven and hybrid (static and dynamic) scheduled preemptive systems. Stack sharing is currently supported by a number of operating systems, e.g. Rubus RTOS[Arc], Fusion RTOS[Uni], Erika RTOS[Evi], SMX RTOS[Dig].

3.2 Stack sharing in development

Stack sharing is suitable for many types of systems in the sense that it is easy to use and that it introduces only a few, mainly two, restrictions in development. Under stack sharing, tasks are (i) prohibited to self suspend, (e.g., timed sleep or other delay functions may not be used in the application code of task that participate in stack sharing) and (ii) task synchronization, e.g., access to shared resources, should be done using an early blocking resource access protocol such as the Immediate Priority Ceiling Protocol[BW96] or Stack Resource Policy[Bak90]. By adhering to these restrictions, developers can utilize stack sharing for a number of different types of systems, both preemptive and non-

preemptive systems.

To utilize stack sharing in systems development, tools as well as run-time environments and development methods need to have support for stack sharing. For instance, in a component based development context, stack sharing need to be supported by the component model. This implies that components need to have their run-time semantics specified in such way that they do not violate the above restrictions. Moreover, the component model needs to be formal enough to allow analysis of memory consumption under stack sharing. Paper C in this thesis proposes a novel component model, for resource constrained systems, that support stack sharing. The model enforces a run-to-completion semantics on the software logic to support stack sharing. Furthermore, to enable shared stack analysis, each software item in the model has an execution profile that denotes the individual stack usage of the item. In addition, the run-time environment that provides the execution of the components (often a real-time operating system), also needs to provide support for stack sharing. Introducing support for stack sharing, i.e., an additional execution model, in a real-time operating system, requires careful design to minimize the overhead of the new model and side effects (both temporal and spatial) affecting predictability of the existing models. Paper D, addresses these issues and presents an integration of stack sharing in a real-time operating system with multiple execution models.

3.3 Analysing shared stack usage

When developing systems, with or without stack sharing, the memory that constitutes a stack must be allocated. Hence, a developer needs to calculate or estimate the total amount of memory needed for a stack. This is non-trivial in the sense that the actual stack usage of a task, thus the memory required, depends on many factors, e.g., the number of variables in the application code and the way functions are called from within a task etc. It is imperative that the amount of memory allocated for the stack is enough for a task to realize a feature. Serious errors, such as stack overflow can occur if a task need more stack than it has allocated. Stack errors, such as overflows, are considered extremely hard to detect since overflows are often sporadic.

There are basically two common ways to come up with the total amount of memory required for a stack. In industry, the total amount of memory is often estimated. The stack is then allocated with the estimated size. The system is then simply executed and the stack usage of tasks is monitored. This may

be achieved by filling the stack area with pre-determined values before the system is executed. When the system is terminated, the number of values being overwritten in the pre-determined pattern gives an idea of the stack usage under the particular execution scenario. This is, of course, not a fool proof way of determining the amount of memory required for a stack, however, it is not an uncommon method. A more formal way to determine the stack size needed for a task, may be done by tools such as [Tid, Abs].

In stack sharing, additional properties need to be considered in determining the maximum amount of memory required for a shared stack. Determining the worst-case memory usage, i.e., total amount of memory required for the shared stack in preemptive systems, require:

- Analysis of stack usage of individual tasks (or threads)
- Analysis of possible preemptions among tasks in the system

Hence, under stack sharing, preemption patterns, i.e., the way in which tasks may preempt each other, affects the amount of memory required for the shared stack. A number of publications have addressed shared stack analysis (see for example [DMT00, GLN01, GD07, RRW05, CMM$^+$04]).

Performing a preemption analysis, i.e., determining the possible preemption patterns, is more or less complex depending on the system model. For example, under priority based execution, priorities are used to dispatch tasks for execution, i.e., a task with high priority may preempt a task with low priority. Hence, the number of unique priorities in the system defines a maximum preemption depth. A commonly used engineering approach to determine the total shared stack usage in priority-based systems is to estimate the maximum stack usage of individual tasks and then compute the sum of maximum stack usage for each priority level. Thus, it is an approach that always assumes fully nested preemption pattern among tasks, and do not consider the actual preemptions that can occur between tasks. From a preemption point of view, this is a safe approach, but it may in many cases be pessimistic and result in rather poor stack utilization. This can be demonstrated with a simple example. Consider the task set in Table 3.1, where P denotes the priority, T denotes the period time an C the worst case execution time of the tasks.

Assume that all tasks in the example share a common stack, and that all tasks are activated at $t = 0$, then the traditional engineering approach would require the shared stack to be 30 bytes (sum of maximum stack usage in each priority level). However, considering the information we have about the system in the example, it is easy to see that no preemption can occur between the tasks (see trace in Fig. 3.4). Hence 10 bytes would be enough for the shared stack.

Table 3.1: An example: Task set in a preemptive system

Task	P	T	C	Stack usage (bytes)
τ_1	High	10	2	10
τ_2	Medium	10	2	10
τ_3	Low	10	2	10

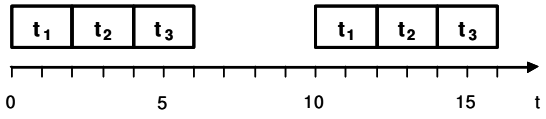

Figure 3.4: Execution trace of task set in Table 3.1

Improving shared stack usage can be achieved by changing the task attributes, which may or may not change the system behaviour, or by improving the analysis method, i.e., reducing the pessimism of the method. For example, the priorities of the tasks in the example could be altered without affecting the temporal behaviour or the actual functionality realized by the tasks. The tasks could all have been assigned high priority and still execute exactly as the original system, i.e., an execution trace would look like in Fig. 3.4. The traditional analysis would now yield in a shared stack usage of 10 bytes. Another approach to improve stack analysis is to improve the preemption analysis, i.e., the preemption analysis should reflect only the preemption patterns that actually can occur. This approach differs from the previous one in the sense that it does not require any modifications of priorities or other task attributes.

The fact that a task may only be preempted during its execution, need to be considered when improving a preemption analysis. Basically this means that a task can only be preempted in a time interval defined by its actual start time and actual finishing time. Considering this interval as a possible time for preemption, would improve shared stack analysis. However, the actual start time and the actual finishing time of a task are dynamic parameters that may change from one execution to another. In many systems however, these dynamic parameters can be bounded by static compile time properties. These bounds can then be used to improve the preemption analysis. Such an approach is presented in Paper E. The dynamic parameters are bounded by offsets and worse case response times. These bounds reflect the possible time interval in which a

task may execute and possibly be preempted. The method presented in paper E, is applicable for industrial systems in the sense that it has reasonably low timing complexity and it is developed for an industrially used system model.

3.4 Supporting shared stack analysis in development

For stack sharing to be accepted in an industrial development context, the analysis method for predictability of the resulting stack usage needs to be supported in development tools, e.g., in an Integrated Development Environment (IDE). As with many analysis methods, stack analysis is complex and hard to implement by non-experts. Hence, means to integrate complex analysis in development environments are needed. The plug-in concept is a promising way to achieve this.

In resent years, plug-in based tools have gained popularity. These tools allow plug-ins to extend the features of the tool. For example, Eclipse [Ecl], an open development platform, allows plug-ins to extend the functionality of the Eclipse platform. Plug-ins can be seen as programs that provide certain functions by interacting with a host application. In essence, plug-ins can be developed in isolation and then integrated with a host application. This makes the concept suitable for integration of complex analysis methods in a development environment. Supporting plug-ins in a development environment allows researchers, the experts on analysis methods, to develop their analysis methods in isolation. These methods can then be plugged-in in a development environment. Paper F in this thesis presents the development of a plug-in framework for integration of real-time analysis methods in Rubus-ICE [Arc]. The paper outlines requirements on the framework as well as the development of application programming interface enabling plug-ins to interact with its host application. Even though a plug-in concept eases the integration of analysis methods in industrial tools, a number of issues still need to be considered for successful integration of analysis methods in an industrial context. For example, the fact that analysis methods are transferred from a controlled research setting to an industrial setting, requires extensive error handling to be introduced in analysis methods. Moreover, even though the implementation of an analysis method may have been verified in a research setting, it often needs to be verified again once integrated. Paper F presents some of the issues that need to be considered from an analysis developer's view.

Chapter 4

Thesis contribution

This thesis presents scientific contributions for development of embedded resource constrained real-time systems. By taking advantage of the novel execution models provided by the research community, the contributions show that efficient memory utilization can achieved in embedded real-time systems.

4.1 Summary of contributions

The contributions of this thesis can be summarized as follows.

The thesis present:

- A summary of properties and common requirements in development of industrial embedded real-time systems.
- A demonstration of how development of embedded real-time systems can be made more efficient, with respect to hardware utilization, by introduction of additional execution models.
- A component model, supporting efficient and predictable stack usage, for embedded real-time systems.
- An implementation of a memory efficient execution model in a real-time operating system.
- An analysis method, suitable for industrial use, for verification of the memory efficient execution model.

- The integration of analysis methods in an IDE for development of embedded real-time systems.

These contributions are presented in the following scientific papers.

4.1.1 Contributions of included papers

Paper A: Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, *Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain*, In Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Potsdam, Germany, March 2006.

Paper A presents common requirements in development of vehicular embedded real-time systems. The requirements were collected in a series of interviews with ten senior designers from four Swedish companies. The paper shows that reliability and safety are the main properties in focus during development. It also shows that the amount of functionality has been increasing in the systems and that requirements are fulfilled using considerably homogenous development methods. The paper also show that there will be even stronger requirements on dependability and control performance and that requirements on soft and resource demanding functionality will continue to increase.

Personal contribution: Kaj was the main author of the paper and has been involved in all parts of the work. He was responsible of preparing the study and establishing the research questions. Furthermore, he coordinated the interviews and compiled the results.

Paper B: Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin, *Efficient Development of Real-Time Systems Using Hybrid Scheduling*, In Proceedings of the 2005 International Conference on Embedded Systems and Applications, Las Vegas, USA, June 2005.

Paper B show how developers can benefit from additional execution models in development. An industrial case study demonstrate how multiple execution models enables more efficient use of computational resources, resulting in a cheaper or more competitive product and that more functionality can be fitted into legacy, resource constrained, hardware.

Personal contribution: Kaj has been involved in the case study part of the paper. He provided the basis, i.e., system information and requirements for the case study.

Paper C: Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, Kurt-Lennart Lundbäck, *The Rubus Component Model for Resource Constrained Real-Time systems*, To appear in the Proceedings of the 3rd IEEE International Symposium on Industrial Embedded Systems, Montpellier, France, June, 2008.

Paper C presents a component model for development of embedded systems for ground vehicles. The industrial requirements presented in paper A were considered for the component model. The model aims at supporting design, analysis as well as efficient and predictable stack usage.

Personal contribution: Kaj was the main author of the paper. He has been involved in all parts of the work.

Paper D: Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin, *Efficient event-Triggered Tasks in an RTOS*, In Proceedings of the 2005 International Conference on Embedded Systems and Applications, Las Vegas, USA, June 2005.

Paper D presents an implementation of a resource efficient execution model in a real-time operating system. The model is suitable for component based control software. The execution model, denoted single-shot execution (SSX), is realized with very simple and resource efficient run-time mechanisms and is highly predictable, hence suitable for use in resource constrained real-time systems. The paper also presents an evaluation, showing that significant memory reductions can be obtained by stack sharing.

Personal contribution: Kaj was the main author of the paper and has been involved in all parts of the work. He was responsible of implementing the execution model in the operating system and evaluating the implementation.

Paper E: Kaj Hänninen, Jukka Mäki-Turja , Markus Bohlin, Jan Carlson, Mikael Nolin, *Determining Maximum Stack Usage in Preemptive Shared Stack Systems*, In Proceedings of the 27th IEEE Real-Time Systems Symposium, Rio de Janeiro, Brazil, December, 2006.

Paper E presents methods to determine the maximum stack memory used in preemptive, shared stack, real-time systems. A general and exact problem formulation applicable for any preemptive system model is presented. The exact formulation is however, not suitable for development of industrial systems, because it based on dynamic properties. The paper also presents an approx-

imate analysis method that is suitable for industrial use. The method safely approximate the total stack usage by using static (compile time) information about the system model and the underlying run-time system. The approximate method supports a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system. Evaluations show that the approximate method significantly reduces the amount of stack memory needed, when compared to a traditional analysis technique.

Personal contribution: Kaj was the main author of the paper. He initiated and coordinated the work and did the background research. He was also responsible for parts of the implementation and did the evaluations.

Paper F: Kaj Hänninen, Jukka Mäki-Turja, Staffan Sandberg, John Lundbäck, Mats Lindberg, Mikael Nolin, Kurt-Lennart Lundbäck, *Introducing a Plug-In Framework for Real-Time Analysis in Rubus-ICE*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-229/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2008. Submitted to the 13th IEEE International Conference on Emerging Technologies and Factory Automation.

Paper F present the development of a plug-in framework for integration of real-time analysis methods in a commercial IDE for embedded real-time systems. The paper also present the integration, of two state of the art analysis techniques (i) response-time analysis for tasks with offsets and (ii) shared stack analysis, in the framework. The paper shows that the framework is well suited for integration of complex analysis methods, however, the paper shows that a considerable amount of modifications of analysis methods are needed to adapt them for industrial use due to differences in requirements and assumptions between industry and academia.

Personal contribution: Kaj was the main author of the paper. He was involved in all parts and responsible for specifying the requirements on the plug-in framework and to realize the analysis methods as plug-ins.

4.2 Impact of contributions

The contributions of this thesis have both industrial and scientific impact.

- From a scientific point of view, the contributions give researchers a picture of issues in transferring research results to industrial use. The thesis show that a considerable amount of work, addressing different issues, is needed to integrate results from academia to industry. In addition, the contributions in Paper E are highly relevant for the research community, in the sense that the paper presents methods to determine the possible preemption patterns in a system.
- From an industrial point of view, the thesis provides contributions on encapsulating novel real-time theory for efficient memory utilization, into methods and software engineering tools. This allows engineers to take advantage of the theories developed by the real-time research community. In general, the thesis provides means for development of predictable, memory efficient and reusable software based products. The results presented in the included papers, are implemented in a commercial tool suite called Rubus. The results will be available for developers with the upcoming release of the tool suite.

Chapter 5

Conclusion and future work

This thesis addresses the introduction of a memory efficient execution strategy, called stack sharing, in development of industrial embedded real-time systems.

The thesis shows, by an investigation of industrial requirements, that industrial embedded real-time systems need to support an increasing number of features with mixed real-time requirements. This requires predictable and efficient use of the computational resources in the systems. The thesis shows that this can be achieved by stack sharing. Stack sharing is a predictable and efficient execution strategy that reduces the memory consumption in systems, hence making it suitable for resource constrained real-time systems.

To achieve industrial use of stack sharing, the thesis presents a component model supporting stack sharing in development of embedded real-time systems with mixed real-time requirements. Furthermore, an integration of stack sharing in a real-time operating system is presented. The integration is presented in detail to highlight common issues in implementing execution models in a real-time operating system. The thesis also presents a novel analysis method for shared stack analysis. In an evaluation, the thesis show that the analysis method is both fast and that it gives tight bounds on the resulting stack usage, which makes it suitable for industrial use. The thesis presents the integration of the analysis method in an integrated development environment.

Altogether, the thesis presents both design and run-time techniques for efficient memory utilization in embedded real-time systems. The proposed techniques have clear scientific impact. Moreover, the techniques have been

implemented in the Rubus tool suite. Rubus is currently used by a number of companies developing embedded real-time systems. With the upcoming release of the tool suite, the scientific results presented in this thesis will be available for developers using Rubus.

Several directions could be taken as future work. For instance:

- Even though it has been shown that stack sharing could theoretically reduce the memory utilization in systems, evaluating the memory reductions in a real-world setting still needs to be done. This would, amongst other things, reveal the actual applicability of stack sharing considering the restrictions (see Section 3.2) introduced by stack sharing.
- Reducing the pessimism of shared stack analysis methods could also be addressed as future work. The analysis method presented in this thesis requires worst case response times as input to the preemption analysis. The response time analysis could for example be extended to represent jitter as well as blocking more accurately. This would result in an improved shared stack analysis.
- Investigating how the preemption analysis, presented in paper E, can be applied in other contexts would be of interest. For example, it might be possible to extend the preemption analysis to support analysis of: (i) memory consumption in systems with dynamic memory allocation, (ii) cache usage, (iii) buffer requirements in synchronisation protocols.

Bibliography

[AB98] L. Abeni and G. Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium (RTSS'98)*, pages 4–13. IEEE Computer Society, Madrid, Spain, December 1998.

[Abe98] L. Abeni. Server Mechanisms for Multimedia Applications. Technical Report RETIS TR98-01, Scuola Superiore S. Anna, Pisa, Italy, 1998.

[ABRW91] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[Abs] Absint. Web page, http://www.absint.com/stackanalyzer/.

[ABT+93] N.C. Audsley, A. Burns, K. Tindell, M.F. Richardson, and A.J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5): 284–292, 1993.

[Arc] Arcticus Systems. Web page, http://www.arcticus-systems.com.

[Bak90] T.P. Baker. A Stack Based Resource Allocation Policy for Real-Time Processes. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*. IEEE, 1990.

[BBLB03] Scott Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time, and Non-Real-Time Processes. In *Proc.* 24^{th} *IEEE*

Real-Time Systems Symposium (RTSS). IEEE Computer Society, December 2003.

[BTW95] A. Burns, K. Tindell, and A Wellings. Effective Analysis for Engineering Real-Time Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 22(5):475–480, May 1995.

[But97] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9994-3.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 13.10.1 Immediate Ceiling Priority Inheritance. Addison-Wesley, second edition, 1996.

[CMM+04] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt-driven programs. *Inf. Comput.*, 194(2):144–174, 2004. ISSN 0890-5401.

[CRM06] A. Crespo, I. Ripoll, and M. Masmano. *Dynamic Memory Management for Embedded Real-Time Systems*, volume 225 of *IFIP International Federation for Information Processing*. Springer, 2006.

[Dig] Micro Digital. Web page, http://www.smxinfo.com/mt.htm.

[DMT00] R. Davis, N. Merriam, and N. Tracey. How Embedded Applications using an RTOS can stay within On-chip Memory Limits. In *Proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[Ecl] Eclipse - an open development platform. Web page, http://www.eclipse.org/.

[EHS97] A. Ermedahl, H. Hansson, and M. Sjödin. Response-Time Guarantees in ATM Networks. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 274–284. IEEE Computer Society Press, December 1997. URL `http://www.docs.uu.se/~mic/papers.html`.

[Evi] Evidence Srl. Web page, http://www.evidence.eu.com.

[GD07] R. Ghattas and A.G. Dean. Preemption Threshold Scheduling:Stack Optimality, Enhancements and Analysis. In *Proceedings of the 13th IEEE REal-Time and Embedded Technology and Applications Symposium*, April 2007.

[GLN01] P. Gai, G. Lipari, and M. Di Natale. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip. In *Proceedings of the 22nd Real-Time Systems Symposium*. London, UK, Dec 2001.

[HNN08] Hans Hansson, Mikael Nolin, and Thomas Nolte. Beating the Automotive Code Complexity Challenge. In *National Workshop on High-Confidence Automotive Cyber-Physical Systems*. Troy, Michigan, USA, April 2008.

[KAS93] D.I. Katcher, H. Arakawa, and J.K. Strosnider. Engineering and Analysis of Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, September 1993.

[Leh90] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proc. 11^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 201–212, December 1990.

[Liu00] J. Liu. *Real-Time Systems*. Prentice Hall, 2000. ISBN 0-13-099651-3.

[LL73] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20 (1):46–61, 1973.

[LR03] R. Lencevicius and A. Ran. Can Fixed Priority Scheduling Work in Practice? In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS)*, page 358, December 2003.

[MSB05] B. Middha, M. Simpson, and R. Barua. MTSS: Multi Task Stack Sharing for Embedded Systems. In *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. San Francisco, CA, Sept 2005.

[MTN05] J. Mäki-Turja and M. Nolin. Fast and Tight Response-Times for Tasks with Offsets. In *Proceedings of the 17^{th} Euromicro Conference on Real-Time Systems*. IEEE Computer Society, July 2005.

[NMTH08] M. Nolin, J. Mäki-Turja, and K. Hänninen. Achieving Industrial Strength Timing Prediction of Embedded System Behavior. In *International conference on Embedded Systems and Applications (ESA)*, July 2008.

[PG03] J.C. Palencia Gutiérrez and M. González Harbour. Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities. In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS)*, December 2003.

[Pun97] S. Punnekkat. *Schedulability Analysis for Fault Tolerant Real-time Systems*. PhD thesis, University of York, June 1997.

[Ros06] C. Del Rosso. The Method, the Tools and Rationales for Assessing Dynamic Memory Efficiency in Embedded Real-Time Systems in Practice. In *Proceedings of the International Conference on Software Engineering Advances*, 2006.

[RRW05] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM Transactions on Embedded Computing Systems*, 4(4):751–778, Nov 2005.

[SEF98] K. Sandström, C. Eriksson, and G. Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, 1998.

[SRL87] L. Sha, R. Rajkumar, and J.P. Lehoczky. Task Scheduling in Distributed Real-Time Systems. In *IEEE Industrial Electronics Conference*, 1987.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: an Approach to Real Time Synchronization . *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[SSL89] B. Sprunt, L. Sha, and J.P. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems Journal*, 1(1), 1989.

[SSRB98] J.A. Stankovic, M. Spuri, K. Ramamritham, and G.C Buttazzo. *Deadline Scheduling for Real-Time Systems, EDF and Related Algorithms*. Kluwer Academic Publishers, 1998. ISBN 0-7923-8269-2.

[SVN07] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded System Design for Automotive Applications. *Computer*, 40(10):42–51, 2007.

[TB94] K. Tindell and A. Burns. Fixed Priority Scheduling of Hard Real-Time Multimedia Disk Traffic. *The Computer Journal*, 37(8): 691–697, 1994.

[TC94] K. Tindell and J. Clark. Holistic Schedulability Analysis For Distributed Hard Real-Time Systems. Technical Report YCS197, Real-Time Systems Research Group, Department of Computer Science, University of York, November 1994. URL ftp://-ftp.cs.york.ac.uk/pub/realtime/papers/YCS197.ps.Z.

[THW94] K. Tindell, H. Hansson, and A. Wellings. Analysing Real-Time Communications: Controller Area Network (CAN). In *Proc. 15^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 259–263. IEEE, IEEE Computer Society Press, December 1994.

[Tid] Tidorum. Web page, http://www.tidorum.fi/bound-t/.

[Tin92] K. Tindell. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. Technical Report YCS189, Dept. of Computer Science, University of York, England, 1992.

[Uni] Unicoi Systems. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.

[WAN03] A. Wall, J. Andersson, and C. Norström. Probabilistic Simulation-based Analysis of Complex Real-Times Systems. In *6th IEEE International Symposium on Object-oriented Real-time distributed Computing*. Hakodate, Hokkaido, Japan, May 2003.

[XP90] J. Xu and D.L Parnas. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. *IEEE Transaction on Software Engineering*, 16(3), 1990.

II

Included Papers

Chapter 6

Paper A: Present and Future Requirements in Developing Industrial Embedded Real-Time Systems - Interviews with Designers in the Vehicle Domain

Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin
In Proceedings of the 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Potsdam, Germany, March, 2006.

Abstract

In this paper, we aim at capturing the industrial viewpoint of todays and future requirements in development of embedded real-time systems. We do this by interviewing ten senior designers at four Swedish companies, developing embedded applications in the vehicle domain. This study shows that reliability and safety are the main properties in focus during development. It also shows that the amount of functionality has been increasing in the examined systems. Still the present requirements are fulfilled using considerably homogenous development methods. The study also shows that, in the future, there will be even stronger requirements on dependability and control performance at the same time as requirements on more softer and resource demanding functionality will continue to increase. Consequently, the complexity will increase, and with diverging requirements, more heterogeneous development methods are called for to fulfil all application specific requirements.

6.1 Introduction

There is an increasing trend towards software solutions in embedded systems. Replacing mechanical functionality with computer-controlled solutions gives opportunities for more advanced and more flexible functionality, e.g., anti-lock braking, traction control etc. Over the years, a large number of publications, e.g., [GLT02, GLT03, kFSC04, Koo96, MFN04, MkFN05, NGS+01, PLC+97] has addressed design issues, embedded application trends or requirements in development of industrial embedded systems. Möller et al [MFN04] present the industrial requirements, both technical as well as process related requirements, on component technologies in the heavy vehicle domain. Åkerholm et al [kFSC04] presents an investigation concerning classification of quality attributes for component technologies in the vehicle industry. The investigation show that dependability characteristics (safety, reliability and predictability) are considered as the most important ones. Koopman [Koo96] presents attributes of four different types of embedded systems (signal processing systems, mission critical and distributed control systems and consumer electronic systems). Koopman addresses requirements, life-cycle support and business models in development of embedded systems. Graaf et al [GLT03] presents an industrial inventory of seven companies developing embedded software products. Their inventory of state of practice addresses requirements engineering and architectural issues such as design and analysis. The inventory covers companies from many different domains, e.g., developers of mobile phones and consumer electronics, distributed data management solutions etc. In this paper, we investigate the industrial requirements in the vehicle domain, especially requirements related to real-time issues on a high overall level, such as safety and reliability requirements of embedded application/products, as well as on a lower technical level, such as choice of operating system (OS) and execution models. The study was performed as a series of interviews with ten senior designers at four Swedish companies. Specifically, we address the following questions:

- Q1. What characterise the embedded applications?
- Q2. What are the designers concern on application properties such as safety, maintainability, testability, reliability, portability and reusability?
- Q3. How are the applications verified/analysed?
- Q4. What are the considerations in choosing an OS, and execution model?

- Q5. What resources are considered as constrained in the systems, and to what degree?
- Q6. What kind of tool support is needed in the development of future systems?
- Q7. What are the designers experiences of software components, i.e., component based development?

The aim of this work is foremost to explore and describe the current and future industrial requirements as perceived by the senior designers. The paper is organised as follows. In Section 6.2, we describe the framework used in the study of the requirements. In Section 6.3, we describe the results of the conducted interviews. In Section 6.4, we address the main questions of the study and discuss our observations of the interviews. In Section 6.5, we conclude our investigation. The paper ends, in Section 6.6, with a discussion concerning verification of the presented results.

6.2 Investigation setup

For this study, we adopted the investigation framework described by Robson [Rob02]. According to the framework, both the purpose of a study and the theory guiding the study should form as guidance when developing the actual research questions. The substance and the form of the research questions form the basis when deciding on suitable investigation method and sampling strategy.
Purpose: The main objective of this study was to investigate the typical set of industrial requirements in development within the embedded control community in the vehicle domain. The results are expected to form a foundation for further research on tool support, design, analysis and synthesis of embedded real-time systems using multiple execution models.
Theories: Our experience is that there has been little effort done to encapsulate novel theories by supporting development tools and techniques in the industrial domain. Traditionally, the development of these systems tends to focus on the safety critical parts, which constitutes a small fraction of the total system functionality. Homogenous development methods are often used for both the safety-critical and the non-critical functionality in the systems. This results in unnecessary complex designs and over utilised systems, where valuable resources such as processing time and memory resources are wasted. We believe

that there is a need for more sophisticated development support compromising of additional tool support and more domain and application specific development platforms, resulting in a more heterogeneous development environment and resource efficient run-time structure. The development platform should aim at handling complexity by relieving the developer of too low details while preserving predictability for core functionality as well as flexibility for less critical functionality in the run-time structure.

Questions: We compromised upon a set of quantitative and qualitative, closed and open-ended questions. The main purpose of the quantitative questions was to facilitate analysis of importance among application properties.

Data collection: Due to the substance and the form of the research questions, the study was conducted as face-to-face interviews using a questionnaire. A pilot study was performed at an OS and development tool vendor. The purpose of the pilot study was to refine the data collection plans and to evaluate the feasibility of the chosen data collection method. The structure of the questionnaire was refined and additional questions were added, as a result of the pilot study.

Sampling: In this study, we use purposive non-probability samples, i.e., the samples are selected as to interest (we do not make generalisation to any population beyond the samples). Four successful and renowned companies in the Swedish vehicle domain were participating in the study. The samples represent both subcontractors as well as own equipment manufacturers. Moreover, the selection is a representative subset of both off-road and road vehicles. The companies range from small and medium-sized enterprises to large corporate groups. The thorough examination of the applications and development processes require us for secrecy reasons to refer the companies as A, B, C and D. Ten software designers with several years of experience from development of control systems for embedded real-time systems participated in the study. For preparation reasons, the questionnaire was mailed in advance to each interviewee.

Analysis: Upon agreement with the interviewees, each interview was tape-recorded. The recordings and notes taken during the interviews were interpreted and analysed both individually and at group basis. We did however not use any specific software package to interpret or analyse the collected data.

Biases: Several factors may introduce unwanted biases in a real-world study. For example, recording interviews may affect the respondent, welcoming or sharing the respondents' views may affect the interview, and so on. To avoid or at least minimise possible biases, we followed recommendations given in [Pet96, Rob02, Yin03] about how to construct questionnaires and conduct face-

to-face interviews in real-world situations. It is our experience that the research questions were easily understood and similarly interpreted by the interviewees, and that the recording had no or very little effect on the respondents and the outcome of the interviews.

6.3 Investigation results

In the following section, we describe: (i) Real-time and functional characteristics of the examined applications. (ii) The interviewees concern on selected application properties. (iii) The currently used resource management policies and available execution models of the examined applications. (iv) The actual resource situation in the examined systems, i.e., availability of computing resources such as CPU time and memory. (v) Some desired support in development tools, as expressed by the interviewees.

6.3.1 Application characteristics

The product volumes of the investigated applications are typically less than 1000 products per year. The applications are mainly used as control applications for various types of vehicles. In addition to control functionality, the applications typically contain functionality for information handling such as logging for diagnostic purposes and presentation of data, i.e., visual interaction with the system operators. The architectures of the examined systems are of distributed character where several nodes, Electronic Control Units (ECUs), perform computations and communicate with each other mainly via CAN buses. Each ECU is usually dedicated to handle specific type of functionality, e.g., an engine controller is mainly responsible for controlling engine specific functionality such as fuel injection, ignition etc. The number of ECUs in the systems has typically been increasing over the years. For example, table 6.1 shows the amount of software and the number of control units in evolution of a single product at one of the investigated companies.

Current characteristics: The examined applications are realized by hard and soft real-time tasks. In several systems, hard real-time tasks are used to model the majority of all functionality. In extreme cases, as much as 95% of the functionality is modelled by hard real-time tasks. In addition, functionality with requirements that are neither hard nor soft, but somewhere in-between, is often modelled as hard. In context to this, the designers stress that development of hard application tasks is considered as more controllable and simpler

Table 6.1: An example of the amount of software and the number of ECUs in a single vehicle, at company A

Year	1991	1997	2002
Lines of code	20000	55000	140000
Files (.c, .h)	50	400	700
ECUs	1	2	3

than development of soft application tasks. In addition, several interviewees consider time-triggered systems to be the most convenient way to model hard real-time functionality. Typical technical requirements in the examined applications include; jitter requirements and precedence relations among tasks. The timing constraints, e.g., deadlines on different functionality, can vary as much as three orders of magnitude in a single application, typically from milliseconds to several seconds. The amount of safety critical functionality varies in the investigated applications. In all of the examined applications, the control functionality is considered as being most safety critical and developed mainly using the time-triggered paradigm. Several interviewees consider their systems being I/O intensive. In some systems, as much as 30% of the available processing time and hundreds of I/O pins is used to handle I/O functionality. The I/O functionality is realised by both time and event-triggered execution models. However, it is most commonly realised using the time-triggered model, i.e., through polling. The information intensity in the investigated applications varies. In some applications, the information originates from logging and diagnostics of the systems operational conditions, whereas other applications receive and process external information that is presented to the users during operation.

Future characteristics: The interviewees believe that the information intensity and number of control functionality will increase in the future. They state that in the future both legislation and insurance reasons will force development of more sophisticated control algorithms and require an increasing amount of information to be saved for diagnostic reasons. In addition, designers from one company predicts that legislations, especially non-pollution laws, and future trends in development of vehicle engines will require better control precision. This will result in an increased transformation from open to closed loop controlling. Furthermore, some interviewees predict that functionality interacting with the environment will be developed using fewer sensors in the future and that certain conditions/states of the environment will be derived using the re-

maining set of sensors. Classification of functionality in Safety Integrity Levels (SIL) [Com00] is also believed to be an important activity in the future.

6.3.2 Functional application properties

In this section, we present the interviewees concern on the following application properties: safety, maintainability, testability, reliability, portability and reusability.

Safety: Safety is considered as a derived property originating foremost from analysis and testing. In some of the examined systems, redundancy and certain safety properties are solved outside the actual software implementation, by physical cabling etc. The software in these systems can be overridden by mechanics in case the software malfunctions and a safety critical situation occurs.

Maintainability: Some interviewees state that the developers consider and try to facilitate future maintainability of applications. Some interviewees also state that they have very strong requirements (economical and quality) on applications being error free since withdrawing an erroneous application would be very costly due to the product volumes. There seems to be an agreement on that maintainability will have to be considered as a more important property in the future, specifically in the context of upgradeability. The lifespan of the examined systems can be several decades and customers put demand on new features and require hardware replacement parts to be available during the entire lifespan of a system. This requires applications to be well structured and easy to understand for future developers (maintainers).

Testability: Testability is stated as an important and necessary property to achieve reliability and safety. Today testing is the main technique to verify functional requirements.

Reliability: Several interviewees state that a company's reputation is very much dependent on the reliability of the delivered systems; i.e., it is considered as being of utmost importance to develop systems that actually are, and perceived by customers as, reliable. Failure in producing reliable systems is often stated to origin from erroneous requirement specifications, i.e., not from the implementation itself.

Portability: Some interviewees do not consider portability during development, simply because they seldom change hardware or OSs. Other respondents claim that portability is an increasing concern and that it is mainly facilitated by separation of hardware and software dependent functionality.

Reusability: Reusability of both soft- and hardware is an ongoing activity in all of the examined systems. However, the amount of reusable software varies

in the examined systems. Some interviewees' state that reusability of architectures is not achieved until they have undergone several modifications, hence it may takes years before certain parts of architectures are actually reusable. To facilitate reusability among different systems, some of the companies have developed common software platforms. The platforms contain all common functionality and have standardised interfaces. General software components are also mentioned as reusable entities. The components are general in the sense that they are, to a large degree, application independent.

Additional properties: When asked for additional properties that are considered as important for their applications, the interviewees mentioned robustness, scalability and usability. Robustness is defined by the respondents as 'the absence of unexpected behaviour' or as 'an additional degree of reliability'. Scalability is considered in the context of development as the ability to scale systems using the available development tools. Usability of architectures is mentioned as a process related issue. In that context, the usability of architectures is said to be dependent on whether it facilitates understanding and communication between developers. All of the respondents stress the importance of architectural descriptions as means of communication between people, i.e., not only as logical or structural system description.

6.3.3 Temporal application properties

This section describes the interviewees view on the temporal analysability of the applications and verification of functional/temporal behaviour. It also addresses the verification of resource utilisation in the examined applications.

Analysability and verification: Analysis of real-time properties such as response-times, jitter, and precedence relations, are commonly performed in development of the examined applications. In this context, some interviewees stress the desire of better analysis support in development tools and state that analysing a whole system with respect to temporal and spatial attributes is very difficult, sometimes even intractable. Due to the difficulties in analysing a complete system, and for upgradeability reasons, some of the examined systems are intentionally over-dimensioned with respect to processing power and memory resources. The emphasis on verification is foremost on the functional behaviour. Our experience is that the temporal attributes are not serving as direct guiding factors (albeit they are more or less considered) during development.

Functional behaviour: All of the respondents had a unanimous opinion that analysis and verification of the functional behaviour was the most important

activity in the verification and analysis processes (more important than analysis and verification of temporal behaviour). The functional behaviour is mainly verified by manual and automatic module and systems tests. Failure mode and effect analysis (FMEA) are commonly performed both during development and on complete systems. Several interviewees state that source code inspection is performed among the developers and that it serves as analysis/verification of functional behaviour.
Temporal behaviour: The verification of temporal behaviour was said to have lower importance than of functional behaviour. The temporal verification of the examined systems commonly involves verification of precedence relations among functions and verifying that deadlines are met, i.e., that estimated worst-case execution times holds and that calculated worst case response-times are met.
Verification of resource utilisation: Many of the examined systems have been evolving for several years. The amount of resources, e.g., the number of control units, has been increasing over the years. Currently, all of the examined systems have more than enough processing time and available memory to perform the intended computations. Hence, verification of resource utilisation, such as memory consumption, is considered of lower importance. However, some interviewees desire possibilities to analyse memory consumption, mainly to be used when the available resources are running low, i.e., before additional resources (ECUs) have to be added to the system.

6.3.4 Operating systems

In this section, we describe the issues involved in choosing operating system and the execution models used in the examined applications. We describe the main motivations to why these operating systems were chosen and the interviewees expressed experience of the used execution models. When investigating the type of technical considerations that has bearing on the choice of OS for the embedded applications, we discovered several non-technical considerations that are strong motivators to the choice of a specific OS, e.g., requirements on coordination to use a common OS at different departments of a company. These requirements do not directly reflect the technical need in development. The technical requirements are commonly considered later on. However, the requirements on simplicity, i.e., ease of use, is a motivator both when choosing OS and among available execution models The commercial operating systems that are used, or have been used, in the embedded applications by the investigated companies are Rubus [Arc], VxWorks [Win], OSE [Ene], O'Tool [Arc],

RTX [Ard] and WinCE [Mic]. In addition, one of the investigated companies develops their own operating systems, used in a majority of their applications. The main motivation for this is that their own operating systems are claimed to be simpler, more robust and have less run-time footprint (timing and memory overhead) than the commercial OSes. The interviewees' state that the main considerations when choosing a commercial operating system include:

- Cost (royalties, licenses).
- Availability of supported development tools related to the OS.
- The supported execution models in the OS, i.e., its suitability for the application domain.
- Coordination within a corporate group or subsidiaries to use a common OS.
- Recommendations originating from other companies evaluating the OS.
- The popularity of the OS, i.e., to what extent is the OS used by other companies.
- The OSs internal timing and memory overhead.
- Safety classification issues.

6.3.5 Execution models

Both time- and event-triggered execution models are used in all of the examined applications. The time-triggered model is commonly used for control functionality whereas the even-triggered model is used mainly for information handling for diagnostic reasons. The interviewees state that the choice of execution model in development is mainly dependent on: (i) Verification possibilities, both functional and temporal. (ii) Flexibility of adding new functionality. (iii) Required response-time on functionality. (iv) Simplicity of use in development.

6.3.6 Resource limitations

This section describes the current resource situation in the examined systems, as expressed by the interviewees. We investigated whether and to what degree, the amount of processing time, RAM, ROM and communication bandwidth,

were considered constrained in the systems. As described in Section 6.3.3, many of the examined systems are intentionally over-dimensioned; hence, the interviewees did not consider any of the resources as being particularly constrained during software development. However, in case the systems would run out of resources, the interviewees' state that they would most probably consider installing additional hardware resources rather than redesigning the way the applications utilises the resources. This is however, said to be dependent on the urgency of system delivery. In extreme cases, functionality has been removed from the examined systems, when the available resources have been fully utilised.

6.3.7 Desired tool support

In this section, we present the interviewees expressed desire concerning support in development tools and their experiences of software components, i.e., component-based development [CL02]. The expressed wishes, concerning support in development tools, amplify the requirements on verification, safety and reliability aspects. The concise picture seems to be requirements on simulation and verification possibilities of applications on PCs. Moreover, an integrated possibility for model-based development with Matlab and Simulink together with automated code generation is another common desire expressed by the interviewees. The following is a list of desired tool support, as expressed by the interviewees. The desired support addresses both technical and process related issues. The interviewees would like to see:

- Simulation of the embedded applications on PCs.
- Replacing of text based user interfaces with graphical user interfaces.
- Support for model based development with possibilities to exchange information between tools from different vendors.
- Abstractions of graphical models, i.e., visualisation of architectures at different levels and from different views.
- Automatic code generation, e.g., from models to source code.
- Support for formal verification of source code.
- Support for execution time analysis.

- Possibilities to identify or trace the requirement specifications from the source code, and vice versa.

The current support in development tools varies at the companies. For example, one company has extensive support for simulation of embedded applications on a PC, whereas others do not have simulation possibilities at all. However, none of the examined companies has all of the listed support in their development tools.

6.3.8 Software components

Only one of the examined companies explicitly state that they use software components in the development of their applications. The company uses both in-house as well as third party developed components. The reasons to why the other investigated companies do not use software components are related to facts such as difficulties in understanding the concept of component-based development. Furthermore, issues such as modifiability of functionality are stated as a restricting factor for use of software components. However, all of the interviewees' state that the abstraction possibilities that components provide, is one of the main motivators of component based development, simply because it facilitates understanding and communication between developers.

6.4 Discussion - our observations

In this section, we address the main questions of the study and present our own observations and conclusions of the interviews.

Q1. What characterise the embedded applications?

The fact that more and more mechanical solutions are replaced with software, results in an increasing complexity both in size and in diversity. The applications are evolving and contain more heterogeneous functionality that before. In the future, this requires abilities to cope with (i) increasing data handling and (ii) increasing complexity in control functionality. It is common that applications contain a mix of hard and soft real-time tasks. We observed that a surprisingly small fraction (e.g., 25% at company A) of the requirements reflects need of hard real-time tasks. Still, the use of hard real-time tasks is very high (75% at company A). We believe that the high utilisation of hard tasks is mainly related to three reasons: (i) simplicity in development (ii) for verifications/reproducibility reasons (iii) tradition in development. The simplicity in development originates from years of evolving support in development tools

that to large extents is intended for development of safety critical real-time systems. There is also a tradition in using hard real-time tasks for the majority of functionality, simply because developers tend to rely on designs from previous projects, instead of scrutinizing and considering the designs appropriateness for the diverging type of functionality found in today's and in future applications. Hence, the predicted increase in information intensity and diversity of functionality, require use of more suitable development models, i.e., models for diverging strategies that can handle both safety critical functionality as well as more flexible and resource efficient functionality in the same system.

Q2. What are the designers concern on application properties such as safety, maintainability, testability, reliability, portability and reusability?

The future classification of functionality in Safety Integrity Levels (SIL) implies that reliability, safety, analysability and testability will continue to be very important application properties in the future. Moreover, we believe that facilitating maintainability of the applications will be a more important activity to consider due to the increasing complexity, long product life cycles and demand on upgradeability of the applications. However, moving into the area of more maintainable systems, through, e.g., raising the level of abstraction and introducing reusable frameworks, introduces challenges since it must be done without compromising the systems safety or reliability.

Q3. How are the applications verified/analysed?

Functional behaviour is typically verified through testing on the target platform, whereas properties such as temporal behaviour are mainly verified with support of software analysis tools. Worst-case execution times are commonly estimated during development, and later on, verified through measurements on the target platform. The interviewees desire tools for verification of both functional and temporal behaviour of embedded applications on PCs. We believe that for the large fraction of future functionality, predictable and flexible execution models, where combinations of different analysis techniques that focus more on average case behaviour and quality of service rather than on worst-case behaviour, will be significant.

Q4. What are the considerations in choosing an OS, and execution model?

Politics and non-technical aspects are strong motivators in choosing OS. It is obvious that such issues could motivate the use of an OS that is more or less suitable to fulfil the technical issues in an application domain or specific needs within a corporate group. We believe that the increasing complexity in the examined application domain require more focus on technical issues, such as availability of novel tool support related to the OS and possibility to utilise more suitable execution models in the OS. For example, with

increasing demand on safety classification such as SIL, the OS must be able to support the trade off between technical aspects such as verifiability and efficiency. For example, the small core of safety critical functionality should be allowed to use more resources if it must fulfil the SIL classification and be verifiable (testable and analysable), whereas the rest of the functionality (non-safety-critical) should utilise more resource efficient run-time mechanism to implement the functionality.

Q5. What resources are constrained in the systems, and to what degree?

Our investigation revealed that the computational resources are not considered as constrained during software development. We believe there are two possible reasons for this. (i) The investigated companies are already using resource efficient development methods (legacy methods), originating from times when all functionality was homogenously implemented. (ii) The systems are over-dimensioned at the same time as the developers put most effort in implementing complex functionality without having tool support to analyse resource consumption, e.g., memory usage. The increasing number of ECUs reveal that the computational resources are highly utilised from time to time, i.e., before addition of hardware. With an increase in diverging functionality, the current situation where a static schedule is used for the majority of all functionality, will either be intractable or overly resource demanding (ending up in new ECUs being added) in the future. Instead, the future development tools need to support an efficient and verifiable way to allocate resources, so that the developers either can: (i) Continue their efficient way of developing with efficient tool support adapted to the diverging functionality in the application domain, or (ii) Have novel tool support that allows them to begin developing systems using efficient and resource saving models. Some interviewees experience that the quality of software increases when developers do not have to worry about resource consumption. Hence, future support for resource efficient development needs to be automated to as large extents as possible.

Q6. What kind of tool support is needed in the development of future systems?

The view on future requirements is that safety critical functionality needs to be certifiable and the emphasis on less critical functionality will be on more efficient resource usage (e.g., average resource utilisation rather than worst case utilisation). This requires system integration tools with possibilities to take domain specific models that support efficient automatic code generation, reproducibility for the safety critical functionality and efficient resource usage for the rest. In addition, to cope with the increasing complexity developers need tools that lift the level of abstraction, i.e., tools that provide both different

levels of abstraction as well as different views (e.g., temporal and functional) at each level of abstraction. It is imperative that the tools relieve some burden of developers (our study show that simplicity is a strong motivator in development) for example by letting synthesis tools provide details (such as assigning temporal attributes, priorities etc.) so that requirements are met.

Q7. What are the designers experiences of software components, i.e., component based development

There is an ongoing activity at one of the investigated companies concerning reusability of general type (application independent) software components. We believe that general components facilitate development and may increase the software quality since they are often adapted in several applications and being subject to extensive testing. However, to be resource efficient, or predictable for safety critical parts, these type of components need to be efficiently, and/or predictably, synthesised, i.e., become application specific in the run time system. Hence, the components should be general and execution model independent during development, and then mapped to an application specific runtime structure.

6.5 Conclusions

In this paper, we presented some requirements in development of industrial embedded systems in the vehicle domain. The requirements were collected by a number of interviews with ten senior designers at four companies in Sweden. Many of the investigated applications are developed using methods that are adequate for the (relatively small) parts that are safety critical. Less critical parts are adapted to fit into the framework of the critical parts. With the increasing size and complexity of software, this homogenous way of developing applications will, we believe, be inadequate. In the future software development strategies, methods and tools must be able to capture the different diverse requirements of the applications and trends in the application domains. Ranging from a small core part of the application that is safety critical to a larger part of the system focused on, for example, quality of service and average case behaviour. The characteristics of the examined systems and the predicted increase in information intensity and higher precision on control functionality, would allow for more suitable execution models, i.e., resource saving and quality enhancing, to be introduced (one company even expressed their interest in execution models addressing variable quality of service levels). A wide spectrum of different kind of tool support is desired in development of the ap-

plications. For example, tools for model-based development with simulation possibilities and automatic code generation are considered as highly desirable. Furthermore, the use of software components and CBSE in general, provides possibilities for architectural descriptions at a high level. The importance of architectural descriptions as means of communication between developers, i.e., not only as logical or structural system descriptions, implies that a strong motivator to use software components is their ability to serve as descriptive entities, i.e., not only as reusable entities.

6.6 Verification of the investigation results

According to Robson [Rob02] there are no standardised means of assuring complete reliability in a study that use flexible design strategy. We did however follow recommendations in [Rob02] to minimise threats to the reliability of the conducted study by:

- Studying and minimising possible sources of biases.
- Describing the application characteristics, properties etc. (Section 6.3) based on information from notes and tape recordings taken during the interviews.
- Interpreting the respondents answers at a group basis when necessary.
- Verifying the observations (Section 6.4 and Section 6.5), with the help of a senior designer with expertise in vehicular real-time systems.
- Verifying our observations (Section 6.4) with representatives from two of the participating companies.

Bibliography

[Arc] Arcticus Systems. Web page, http://www.arcticus-systems.se.

[Ard] Ardence. Web page, http://www.vci.com.

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

[Com00] International Electrotechnical Commission. *Functional Safety and IEC 61508*, May 2000.

[Ene] Enea Embedded Technology. Web page, http://www.ose.com.

[GLT02] B. Graaf, M. Lormans, and H. Toetenel. Software Technologies for Embedded Systems: An Industry Inventory. In *4th International Conference on Product Focused Software Process Improvement*. Rovaniemi, Finland, 2002.

[GLT03] B. Graaf, M. Lormans, and H. Toetenel. Embedded Software Engineering: The State of the Practice. *IEEE Software*, 20(6), 2003.

[kFSC04] M. Åkerholm, J. Fredriksson, K. Sandström, and I. Crnkovic. Quality Attribute Support in a Component Technology for Vehicular Software. In *Fourth Conference on Software Engineering Research and Practice in Sweden*. Linköping, Sweden, October 2004.

[Koo96] P. Koopman. Embedded Systems Design Issues(the Rest of the Story). In *Proceeding of the International Conference on Computer Design(ICCD)*. Austin, October 1996.

[MFN04] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *International*

Symposium on Component-based Software Engineering(CBSE7). Edingburgh, Scotland, May 2004.

[Mic] Microsoft. Web page, http://msdn.microsoft.com/embedded/-prevver/ce.net/.

[MkFN05] A. Möller, M. Åkerholm, J. Fröberg, and M. Nolin. Industrial Grading of Quality Requirements for Automotive Software Component Technologies. In *Embedded Real-Time Systems Implementation Workshop in conjunction with the 26th IEEE International Real-Time Systems Symposium*. Miami, USA, December 2005.

[NGS+01] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, and N-E. Bånkestad. Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry. In *Eight Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. Washington, US, April 2001.

[Pet96] M G.E. Peterson. User Satisfaction Surveys, What the Engineer Should Know. In *Proceedings of the Ninth IEEE Symposium on Computer-Based Medical Systems*, June 1996.

[PLC+97] P.G. Paulin, C. Liem, M. Cornero, F. Nacabal, and G. Goossens. Embedded Software in Real-Time Signal Processing Systems: Application and Architecture Trends. In *Proceedings of the IEEE, Volume 85, Issue 3*, 1997.

[Rob02] C. Robson. *Real World Research, 2nd edition*. Blackwell Publishing, 2002. ISBN 0-631-21305-8.

[Win] Wind river. Web page, http://www.windriver.com.

[Yin03] R. Yin. *Case Study Research, 3rd edition*. Sage Publications, 2003. ISBN 0-7619-2553-8.

Chapter 7

Paper B: Efficient Development of Real-Time Systems Using Hybrid Scheduling

Jukka Mäki-Turja, Kaj Hänninen, Mikael Nolin
In Proceedings of the 2005 International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Abstract

This paper will show how advanced embedded real-time systems, with functionality ranging from time-triggered control functionality to event-triggered user interaction, can be made more efficient. Efficient with respect to development effort as well as run-time resource utilization. This is achieved by using a hybrid, static and dynamic, scheduling strategy. The approach is applicable even for hard real-time systems since tight response time guarantees can be given by the response time analysis method for tasks with offsets.

An industrial case study will demonstrate how this approach enables more efficient use of computational resources, resulting in a cheaper or more competitive product since more functionality can be fitted into legacy, resource constrained, hardware.

7.1 Introduction

As the complexity of embedded real-time systems keeps growing, both by increases in size and in diversity, the developers are faced with the increasing challenge of modelling, analyzing, implementing and testing both the functional as well as the temporal behavior of these systems. This paper will present ways to simplify some of that complexity by introducing methods to verify the temporal correctness for a larger class of such systems.

Traditionally, one design parameter has been what execution model to choose. Two common and widespread execution models are the static and dynamic execution models:

- **Static scheduling**, where a schedule is produced off-line. The schedule contains all scheduling decisions, such as when to execute each task or to send each message. During run-time a simple dispatcher dispatches tasks according to the schedule. Static scheduling is sometimes referred to as time-triggered scheduling.
- **Dynamic scheduling**, where scheduling decisions are made on-line by a run-time scheduler. Typically some task attribute (such as priority or deadline) is used by the scheduler to decide what task to execute. The scheduler implements some queueing discipline, such as fixed priority scheduling or earliest deadline first. Dynamic scheduling is sometimes referred to as event-triggered scheduling.

Since both models have their pros and cons, the design decision of which one to use is not simple. A few trade-offs when choosing execution model are:

- **Overhead** – Since all scheduling and synchronization decision are made off-line in the static approach, the run-time overhead for scheduling is kept low. In dynamic scheduling these decisions are made on-line, often resulting in a larger overhead.
- **Responsiveness** – Statically scheduled systems are inflexible and have therefore limited possibility in responding to dynamic events, resulting in poor responsiveness. Dynamically scheduled systems, on the other hand, handles dynamic events naturally and can provide high degree of responsiveness.
- **Resource usage** – In order to provide some degree of responsiveness for dynamic events in the environment, statically scheduled systems tend to waste resources on redundant polling, whereas event-triggered dynamic schedulers only handle the actual events, enabling better service to soft or non-real time functionality when events do not occur at their maximum rate.
- **Overload** – In static scheduling the effects of overload are highly predictable.

The exact capacity, e.g. in terms of number of inputs handled, is known and the effect of lost events, e.g. due to slow polling, can be predicted. In dynamic scheduling, no natural overload control is inherent. Instead, ad-hoc mechanisms are used to prevent, e.g., faulty sensors from flooding the systems with interrupts. A dynamically scheduled system which becomes overloaded is unpredictable, it is often difficult to assess which buffer will overflow and thus which tasks will miss their deadlines.

- **Determinism** – A statically scheduled system is highly deterministic, it executes according to the pre-defined schedule each time. A dynamically scheduled system, on the other hand, may exhibit different behavior each time the system is run, due to, e.g., race conditions on shared resources. This has a major impact on reproducibility, and thus also on the functional testability, of the system. Determinism also simplifies the verification process which is a major part when certifying safety critical applications.
- **Complex constraints** – Statically scheduled systems can handle more complicated inter-task relation constraints. For example, control systems, where control performance is important, need to have small (input and/or output) jitter, which is easier to accommodate in a static scheduler than with simpler dynamic scheduling parameters.
- **Adding new functionality** – Once a static schedule has been constructed it can be very hard to add new functionality in the system, a completely new schedule has to be constructed. For a dynamically scheduled system, new functions can be added with a minimum of impact on other parts of the system.

For further discussions on these trade-offs see [XP00] which advocates cyclic scheduling), and [Loc92] which advocates dynamic, fixed priority, scheduling.

As can be seen, both approaches have their virtues and one often wishes to have both approaches available when developing embedded real-time applications. This desire is clearly illustrated by the last few years of development in the area of field busses for automotive applications. The Controller Area Network (CAN) [CAN92] has been predominant in the automotive industry. CAN provides dynamic scheduling (using fixed priority scheduling). However, the automotive industry felt a need for a more dependable and predictable bus architecture. So when Kopetz brought attention to his Time Triggered Protocol (TTP) [KG94], which provides static scheduling, many automotive manufacturers and their sub-contractors embraced the new technology. It was soon recognized that TTP was a bit *too* static. Hence, a consortium of automotive manufacturers and sub-contractors started the development of FlexRay [Flx],

which provides both static and dynamic scheduling. Also, on the operating-system side, products that support both static and dynamic scheduling have emerged. For instance, Arcticus Systems' operating system Rubus [Arc], and the open source real-time operating system Asterix [Ast]. In fact, most priority driven operating systems can implement hybrid static and dynamic scheduling by letting a dispatcher (a time-table) execute at highest priority.

Thus, we see that the need to combine static and dynamic scheduling have led to some practical solutions available today. However, one problem with systems that tries to combine static and dynamic scheduling is that they often consider the dynamic part as non real-time, e.g. [Arc, Flx]. That is, dynamic scheduled tasks/messages are not given any response-time guarantees, only best-effort service is provided. However, in order to fully utilize the potential of combining static and dynamic scheduling in hard real-time systems, both the dynamic and the static parts need to be able to provide response-time guarantees. A recent study of industrial needs recognizes that one of the key issues for embedded systems is analyzability [MFN04].

This paper presents a method to model hybrid, statically and dynamically, scheduled systems with the task model with offsets [MTN04]. With this model, and the corresponding response time analysis, tight response time guarantees can be given also for dynamically scheduled tasks. The modelled system can be realized with commercially available operating systems support. Furthermore, in a case study we show how a legacy system at Volvo Construction Equipment could benefit from this approach by migrating functionality from the resource demanding statically scheduled part to the dynamically scheduled part, freeing system resources while still fulfilling original temporal constraints.

Paper Outline: Next, Section 7.2 describes the type of systems studied in this paper. Section 7.3 shows how these systems can be modelled using the task model with offsets. Section 7.4 discusses related work. Section 7.5 illustrates, through a case study, how this approach can be applied to a legacy system, migrating functions from a static schedule, freeing system resources. Finally, Section 7.6 presents our conclusions.

7.2 System description

In this paper, we address the issue of providing tight response-time guarantees to dynamically scheduled tasks running "in the background" of a static schedule. The system model contains:

- **Interrupts**. There may be multiple interrupt levels, i.e., an interrupt may be preempted by higher level interrupts.

- **A static cyclic schedule**.
  - ○A set of periodic static tasks (functions) are scheduled in the schedule. Each task has a known worst case execution time (WCET).
  - ○The schedule has a length (a duration) that is equal to the LCM (least common multiple) of all statically scheduled function periods. The schedule is constructed off-line by a scheduling tool.
  - ○Each function is scheduled at an offset relative to the start of the schedule. This is also referred to as a function's *release time*.
  - ○The static cyclic scheduler activates each function in the schedule at its release time. When the whole schedule has been executed the schedule is restarted from the beginning.

  Interrupts may preempt the execution of statically scheduled functions.
- **A set dynamically dispatched tasks**. We call each such task a *dynamic task*. These tasks executes in the time slots available between interrupts and statically scheduled functions. Dynamic tasks are scheduled by a fixed priority preemptive scheduler. They are assumed to be periodic or, at least, to have a known minimum time between two invocations.

We assume that a static cyclic schedule has been constructed prior to the analysis of dynamic tasks. Furthermore, we assume that the schedule is valid even if its functions are preempted by interrupts. How a scheduler can generate a feasible schedule, with interfering interrupts, is described in [SEF98].

7.2.1 Example system

Fig. 7.1 shows a static cyclic schedule of length 20, with 4 functions released at times 0, 5, 10 and 15, with WCETs 4, 1, 1 and 3 respectively.

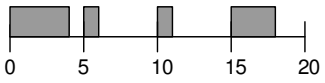

Figure 7.1: Example of static cyclic schedule

In Fig. 7.2 we see an example execution scenario when executing the schedule from Fig. 7.1, with one interfering interrupt source and one dynamically scheduled task (two instances of that task are activated). We make the observation that both interrupts and the static schedule act like higher priority tasks from the dynamic tasks' point of view.

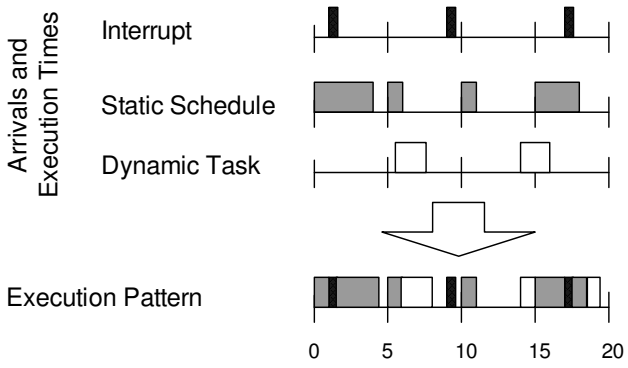

Figure 7.2: Example execution scenario

One of the main objectives of this paper is to enable response-time calculations for dynamic tasks. The goal is to model static schedules (and interrupts) so as to incur as little interference on dynamic tasks execution as possible. Thus, modelling both functions' WCETs as well as their release times as accurately as possible.

7.3 Modelling the system

Classical response-time analysis (see e.g. [ABD+95, BW96, JP86]), assumes that a critical instant[1] occurs when all tasks are released simultaneously. Using this model, the static schedule described in Section 7.2, can be modelled as 4 tasks. These tasks would have a period of 20 and WCETs of 4, 1, 1, and 3 respectively. However, this approach is overly pessimistic since it assumes that all four static tasks can be released for execution at the same time. In our example, assuming no interrupt interference, a dynamic task with a WCET of 1, would have a response time of 10 (4+1+1+3+1). However, looking at Fig. 7.1 one can see that the actual worst possible response-time is 5 (if the dynamic tasks coincides with the static function scheduled at time 0).

In static schedules it is impossible for all static tasks to start at the same time. The task model with offset introduced by [PG98, Tin92] is able to capture the time separation in static schedules, and thus reduce the pessimism.

[1] Point in time, where the task under analysis is released for execution, resulting in the longest possible response-time.

In [MTN04] we further reduced the pessimism in the corresponding response time formulae.

7.3.1 Task model with offsets

The task set, Γ, in [MTN04] consists of a set of k transactions, $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by a periodic sequence of events with period T_i. A transaction Γ_i, contains $|\Gamma_i|$ number of tasks, and each task is activated when a relative time, *offset*, elapses after the arrival of the event.

τ_{ij} is used to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the number of the task within that transaction. A task τ_{ij} is defined by a worst case execution time (C_{ij}), an offset (O_{ij}), a deadline (D_{ij}), maximum jitter (J_{ij}), maximum blocking from lower priority tasks (B_{ij}), and a priority (P_{ij}). The task set Γ is formally expressed as follows:

$$\begin{aligned}\Gamma &:= \{\Gamma_1, \ldots, \Gamma_k\} \\ \Gamma_i &:= \langle \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\}, T_i \rangle \\ \tau_{ij} &:= \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij} \rangle\end{aligned}$$

There are no restrictions placed on offset, deadline or jitter. The maximum blocking time for a task, τ_{ij}, is the maximum time it has to wait for a resource which is locked by a lower priority task. In order to calculate the blocking time for a task, usually, a resource locking protocol like priority ceiling or immediate inheritance is needed. Algorithms to calculate blocking times for different resource locking protocols are presented in [But97]. Priorities can be assigned with any method (e.g. rate monotonic, deadline monotonic, or user defined priorities). One must assume that the load of the task set is less than 100%.[2]

Parameters for an example transaction (Γ_i) with two tasks (τ_{i1} and τ_{i2}) is depicted in Fig. 7.3. The offset denotes the earliest possible release time of a task relative to the start of its transaction and jitter (illustrated by the shaded region) denotes maximum possible variability in the actual release of a task. The upward arrows denotes earliest possible release of a task and the height of the arrow corresponds to the amount of execution released. The end of the shaded region represents the latest possible release of a task.

[2] This can easily be tested, and if not fulfilled some response-times may be infinite; rendering the task set unschedulable.

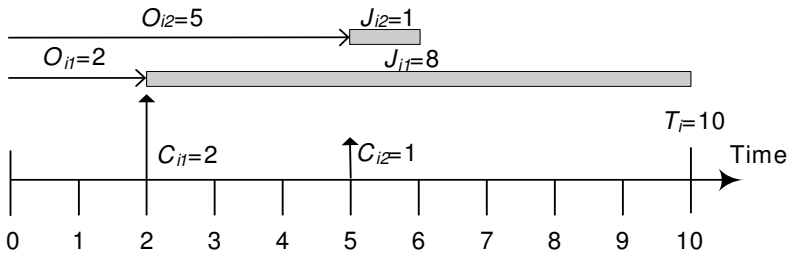

Figure 7.3: Example transaction

7.3.2 System model

The system in Section 7.2 can be modelled, and dynamic tasks subsequently analyzed for response times, with the above task model as follows (subscripts i, s, and d denote a generic interrupt, static, and dynamic transaction respectively):

- **Each interrupt** will be modelled as a transaction, Γ_i, containing one single task (i.e., $|\Gamma_i| = 1$) with T_i set to minimum inter-arrival time of the corresponding interrupt. These interrupt tasks will have the highest priorities in the system. If there are several interrupt levels, priorities are assigned accordingly, i.e., highest priority to highest interrupt level.
- **The static schedule** is modelled as one transaction, Γ_s, where each release time in the schedule is modelled as one task, τ_{sj}, where the offset ,O_{sj}, is set to the corresponding release time. The worst case execution time, C_{sj}, is set to the corresponding functions WCET. The priority, one suffices, for static tasks must be lower than for any interrupt, but higher than those for dynamic tasks.

  Our example schedule of Fig. 7.1 will be modelled as a transaction ($T_s = 20$) with 4 tasks, with offsets 0, 5, 10, 15 and worst case execution time of 4, 1, 1, 3 respectively.
- **Dynamic tasks** will have the most variability on how they are modelled. In the simplest case they are modelled exactly the same way as interrupts but with lower priorities. This situation corresponds to simple periodic (or sporadic) dynamic tasks with no jitter, no time separation (offsets), and no blocking. However depending on the nature of the dynamic tasks their corresponding transaction can be extended by:
  ◦jitter if there is variability in their periodicity,

- by blocking if they share resources and providing the run-time system supports an analyzable resource sharing protocol, and
- offsets if there are temporal dependencies, such as precedence, among dynamic tasks.

Note that dynamic tasks cannot communicate with static tasks, via locked resources, since they must not affect their temporal behavior. However, there exist methods to communicate between these two systems that will not affect the temporal behavior of static tasks, see e.g. [NNT+04].

Assuming the dynamic task of Fig. 7.2 is a sporadic task with minimum inter-arrival time of 10 time units and a release jitter of 3 time units, it is modelled as a transaction with $T_d = 10$ containing one task with $J_{dj} = 3$. The execution time is 2 and since it is the lowest priority task the blocking is zero ($C_{dj} = 2$ and $B_{dj} = 0$).

The formulae to calculate the response times rely on a relaxed critical instant assumption stating that only one task out of every transaction has to coincide with the critical instant. The complete formulae can be found in [MTN04], and would, for our example system of Fig. 7.2, result in a response time of 5 time units for a dynamic task with $C_{dj} = 1$, assuming no interrupt interference.

Since all type of tasks, interrupt, static, and dynamic, can be analyzed for responsiveness, the inability of providing response time guarantees will no longer be a basis for rejecting an execution model for a function, thus making hybrid static and dynamic scheduling suitable even for hard real-time systems.

7.4 Related work

There has been number of research projects addressing the issue of combining several execution models [BBLB03, RS01, SRLK02]. These provide reservation-based guarantees where task characteristics are not fully known in advance. Furthermore, no commercially available real-time operating system support exist for them. Our approach is to model existing systems, supported by commercial RTOSes, where task attributes are fully known at design time. However, [RRW+03] aims at modelling real situations through hierarchically modelling different schedulers. They cover preemptive and non-preemptive priority schedulers and do not model static schedulers. In fact, the work presented in this paper could extend their more general framework with the ability to model also static schedulers.

7.5 Case study

A case study [RH03] conducted at Volvo Construction Equipment (VCE) [VCE], with the objective of finding a way to use available resources in a more efficient way has studied the design trade-offs between static and dynamic scheduling.

VCE has a tradition in statically scheduled systems. This is mainly due to the safety critical nature of their control systems in their heavy machinery, e.g., articulated haulers, trucks, wheel loaders and excavators. Rubus OS by Arcticus [Arc], used by VCE, has run-time support for the system model described in Section 7.2.

Currently at VCE, all safety critical functionality is implemented in the static part and only soft real-time or non real-time activity resides in the dynamic part. In recent interviews (in an ongoing research project) they state that about 20-25% of their applications are considered safety critical, mainly residing in transmission and engine control. However, some operational modes, have static schedule utilization as high as 74%.

The demand on more functionality in next generation machinery is growing. However, the static schedule is getting close to full utilization, leaving little or no room for new functionality. This can either be addressed with new and more expensive hardware or to find a better way of utilizing the current hardware resources.

Demand on responsiveness (i.e. deadlines) for functionality in the static part ranges from a few milliseconds up to several seconds. This could potentially result in very large schedules (with corresponding high memory consumption). VCE's solution to this has been to fix the schedule length at 100ms, which result in waste of computing resources due to redundant polling for any function with a responsiveness demand higher than 100ms (even functions with responsiveness demand within 100ms but associated with events that occur seldom will in this case waste computing resources). A solution that could get rid of this redundant polling, while still guaranteeing the responsiveness and without increasing the schedule length, would be highly desirable.

7.5.1 An example system

Here we will present an example system that can be viewed as a simplified version of one of the systems constructed by VCE. A complete system would consists of several hundreds of tasks [RH03] and would be too complex to present in this paper. We will show how functions currently residing in the static part can be moved to the dynamic part and, by using the response-time

analysis of [MTN04], still guarantee that the function deadlines will be met. Type of functionality that could be moved, according to [RH03], consists of events that by nature are event-triggered, visual interaction with driver, and logging of operational statistics. Another example of functionality that may be moved to the dynamic part is control functionality that is not part of sampling or actuation. Control performance is often sensitive to jitter in sampling and actuation and therefore often placed in a static schedule [Cer99]. However, the control calculation and updating of control state do not have these strict requirements on jitter and their responsiveness requirement is only restricted by the corresponding output action and sampling in the next period respectively. Therefore control and updating control state functionality could be moved to the dynamic part.

Task i	T_i	C_i	D_i	U^{100}	U^T
A	10	2	10	20%	20%
B	20	2	5	10%	10%
C	50	1	2	2%	2%
D	50	6	50	12%	12%
E	100	8	100	8%	8%
F	2000	7	100	7%	0.35%
G	2000	8	100	8%	0.4%
H	2000	8	2000	8%	0.4%

Table 7.1: The set of tasks in the Static system

For our example, the task specification in table 7.1 will be used. (For simplicity we will in this example ignore interrupt interference.) Tasks F and G handle events that may occur once every 2000ms, and with a response time requirement of 100ms. Placing tasks F and G in a static schedule, means that they would have to be polled at the rate of their deadline (100ms) instead of their period (2000ms) (since we do not know exactly when the events are going to occur). Task H, however, could be polled at the rate of its period (2000ms), however, the resulting schedule would become too large and memory consuming (it would have to extend for 2000ms and thus consume over 20kb of ROM). Setting the schedule length to 100ms would be adequate for all tasks except task H. Hence, the schedule length is set to 100ms, and a resulting schedule can be seen in Fig. 7.4 on the facing page.

In table 7.1, U^{100} represents the task utilization when scheduled in a static schedule with a period of 100ms, and U^T represents the utilization when tasks

are scheduled with their period.

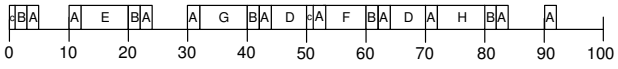

Figure 7.4: Static schedule for table 7.1 task set

The total utilization of the static schedule is 75%. Adding new functionality, requiring some kind of temporal guarantee, to this system can be difficult, there are not many free time-slots in the schedule, especially if there has to be room also for interrupts and non-real-time functionality.

Improving the system

However if tasks F, G, and H could be made event triggered, by placing them in the dynamic part of the Rubus OS, some resources could be freed. The resulting static schedule can be seen in Fig. 7.5. The utilization for the static schedule now becomes 52%. The utilization for the three dynamic tasks are 1,15%, resulting in a total utilization of just above 53%. Thus, by moving these three tasks from the static schedule we free nearly 22%[3] of the CPU resources.

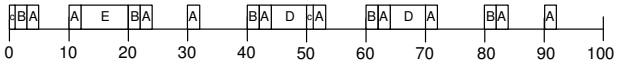

Figure 7.5: Schedule without tasks F, G and H

Now, it remains to see whether the three tasks will meet their deadlines when running as dynamic tasks. To be able to calculate response times for tasks F, G, and H we model the static schedule as a transaction with $T_s = 100$. WCETs and offsets are set as follows:

$$C_{sj} = (5,\ 10,\ 4,\ \ 2,\ 10,\ 3,\ 10,\ \ 2,\ 4,\ 2)$$
$$O_{sj} = (0,\ 10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90)$$

[3]Increase in overhead for tasks F, G, and H as dynamic tasks will be marginal, hence not considered here.

Assuming that F, G, and H have priorities high, medium, and low respectively, we can calculate the response times for the three tasks according to [MTN04]. And the result is:

$$R_F = 26 \quad R_G = 44 \quad R_H = 64$$

We see that all three tasks will meet their deadlines of table 7.1. In fact, their responsiveness is considerably increased compared to being statically scheduled every 100ms. It could be mentioned that by removing tasks F, G and H from the schedule we have enabled shorter response times for other dynamic tasks, that might have existed in the system, as well. The schedule in Fig. 7.4 has a longest busy period of 54ms (between 30–84), whereas the new schedule in Fig. 7.5 has a longest busy period of 14ms (between 10–24). Since any dynamic task (in the worst case) will have to wait for the longest busy period, we now have significantly reduced that time.

With the approach presented in this paper the static schedule could be kept small (with respect to memory consumption as well as utilization). By modelling the static schedule as one transaction, response time analysis for task with offsets can be used to evaluate timeliness for the dynamic part.

Our solution reduce utilization by moving functionality, previously polled excessively, from the static schedule to the dynamic part. Our method also gives a possibility to shrink the static schedule since functions with long periods can be moved from the static schedule. It should be mentioned however, that all tasks in the static schedule share a common stack, whereas moving tasks from the schedule to the dynamic part may require them to have separate stacks, hence increasing the memory consumption for dynamic tasks. However, using a resource locking protocol such as the immediate inheritance allows also dynamic tasks to share a single stack [But97, Nor99].

The possibility to selectively migrate functions from static scheduled legacy systems to dynamic scheduled systems will substantially facilitate for companies to gradually move into the area of dynamic scheduling, and thus, in the long run, help companies to use cheaper hardware for, or fit more functions into, their products. Also the development process becomes easier because event triggered functionality does not have to be force-fitted into a static model.

7.6 Conclusions

As stated in [MFN04] analyzability is one of the major concern for embedded systems development. We have in this paper shown how a hybrid, static and

dynamic, scheduling model can be modeled and dynamic tasks analyzed for responsiveness. The type of system presented can be realized by commercially available OS support, e.g., Rubus OS by Arcticus [Arc]. In fact, any fixed priority OS complemented with an external static scheduler can implement this type of system with the static schedule as a task at highest priority.

A hybrid, static and dynamic, scheduling model simplifies the design trade-offs of which scheduling model to choose. Appropriate scheduling model can be chosen on function level instead of system level. Since temporal guarantees can be provided, this approach will also be applicable for hard real-time systems. Choosing the most appropriate model for each function, instead of force-fitting it to an overall model, not only simplifies the design choices but also gives the possibility to save system resources and improve responsiveness. This is demonstrated in a case study [RH03] at Volvo Construction Equipment using the commercial real-time operating system Rubus by Arcticus [Arc].

Bibliography

[ABD+95] N.C. Audsley, A. Burns, R.I. Davis, K. Tindell, and A.J. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. *Real-Time Systems*, 8(2/3):173–198, 1995.

[Arc] Arcticus Systems. Web page, http://www.arcticus-systems.com.

[Ast] The Asterix Real-Time Kernel. Web page, http://www.mrtc.mdh.-se/projects/asterix/.

[BBLB03] Scott Brandt, Scott Banachowski, Caixue Lin, and Timothy Bisson. Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time, and Non-Real-Time Processes. In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2003.

[But97] G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9994-3.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, second edition, 1996. ISBN 0-201-40365-X.

[CAN92] Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High Speed Communications, February 1992. ISO/DIS 11898.

[Cer99] A. Cervin. Improved Scheduling of Control Tasks. In *Proc. of the 11^{th} Euromicro Workshop of Real-Time Systems*, pages 4 – 10, June 1999.

[Flx] FlexRay. Web page, http://www.flexray-group.org/.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[KG94] H. Kopetz and G. Grünsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, pages 14–23, January 1994.

[Loc92] C.D. Locke. Software Architecture For Hard Real-Time Applications - Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems*, 4:37–53, 1992.

[MFN04] Anders Möller, Joakim Fröberg, and Mikael Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *7th International Symposium on Component-based Software Engineering (CBSE7)*. IEEE Computer Society, May 2004.

[MTN04] Jukka Mäki-Turja and Mikael Nolin. Tighter Response-Times for Tasks with Offsets. In *Proc. of the 10^{th} International conference on Real-Time Computing Systems and Applications (RTCSA'04)*, August 2004.

[NNT+04] Dag Nyström, Mikael Nolin, Aleksandra Tesanovic, Christer Norström, and Jörgen Hansson. Pessimistic Concurrency-Control and Versioning to Support Database Pointers in Real-Time Databases. In *Proc. of the 16^{th} Euromicro Conference on Real-Time Systems*, June 2004.

[Nor99] Northern Real-Time Applications. SSX5 True RTOS, 1999.

[PG98] J.C. Palencia Gutiérrez and M. Gonzáles Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS)*, December 1998.

[RH03] T. Riutta and K. Hänninen. Optimal Design. Master's thesis, Mälardalens Högskola, Dept of Computer Science and Engineering, February 2003.

[RRW+03] J. Regher, A. Reid, K. Webb, M. Parker, and J. Lepreau. Evolving Real-Time Systems Using Hierarchical Scheduling and Concurrency Analysis. In *Proc. 24^{th} IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2003.

[RS01] J. Regher and J.A. Stankovic. HLS: A Framework for Composing Soft Real-Time Schedulers. In *Proc. 22^{th} IEEE Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, December 2001.

[SEF98] Kristian Sandström, Christer Eriksson, and Gerhard Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In *5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 158–165. IEEE Computer Society, Hiroshima, Japan, October 1998.

[SRLK02] S. Saewong, R. Rajkumar, J.P. Lehoczky, and M.H. Klein. Analysis of Hierarchical Fixed Priority Scheduling. In *Proc. of the 14^{th} Euromicro Conference on Real-Time Systems*. IEEE Computer Society, June 2002.

[Tin92] K. Tindell. Using Offset Information to Analyse Static Priority Pre-emptively Scheduled Task Sets. Technical Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[VCE] Volvo construction equipment. Web page, http://www.volvoce.com.

[XP00] J. Xu and D.L. Parnas. Priority Scheduling Versus Pre-Run-Time Scheduling. *The Journal of Real-Time Systems*, 18(1):7–23, January 2000.

Chapter 8

Paper C: The Rubus Component Model for Resource Constrained Real-Time Systems

Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, Kurt-Lennart Lundbäck
To appear in the Proceedings of the 3rd IEEE International Symposium on Industrial Embedded Systems, Montpellier, France, June, 2008.

Abstract

In this paper we present a component model for development of distributed real-time systems. The model is developed to support development of embedded control systems for ground vehicles. The model aims at supporting three important activities in real-time development, (i) design, (ii) analysis and (iii) synthesis. These activities emphasise different and sometimes conflicting requirements that need to be balanced. For example, developers desire freedom in designing to solve complex tasks, analysis tools require the design to be formal enough for analysis and synthesis need to be efficient for low run-time footprint. We have considered industrial requirements for these activities and developed the RubusCMv3 component model. The model has been developed in close cooperation with industrial partners and it is currently being evaluated on real systems.

8.1 Introduction

The industrial requirements on embedded computer systems are constantly evolving. With the flexibility offered by software, the complexity of system designs and the amount of advanced computer controlled functionality in products is increasing. In the automotive domain, for example, systems are typically characterised by having a mix of requirements ranging from hard real-time, soft and even non real-time whilst operating in a resource constrained environment. Historically, developers of embedded real-time systems have used low level programming languages to guarantee full control of the system behaviour. Hence, many embedded real-time systems have become overly complex and hard to manage during functionality or technology shifts. The variety of functionality in today's embedded systems requires development methods and tools that support flexible and efficient development.

In resent years, Component Based Development (CBD), has shown to be successful in development of complex desktop applications. It is an emerging development discipline in which software systems are built by assembling pieces of software units, components, into larger systems. Components are self contained units that provide natural units of reuse by encapsulating functionality into manageable blocks of software. In general, components provide possibilities for high level architectural descriptions, hence, serving as descriptive entities and reusable logic.

Component based engineering has had a tremendous impact in the office-/Internet-area. Today, there exists several commercial component technologies for the desktop- and Internet-market, e.g., COM/DCOM [Mica], Corba [OMG][OMG02], Java Beans/EJB [SUNa][SUNb], .NET [Micb] are readily available and used by developers on a day-to-day basis. However, these technologies are typically not suitable for embedded control systems [MÅFN04]. In the embedded systems domain, CBSE is still only perceived as a promising future technology. Several component models and technologies for embedded systems have been proposed (e.g. Koala [vO02], PECOS [MSZ02][PEC], MetaH [Ves97], VEST [Sta01], the control server [CE03], ReFlex [Wal03] and [DGL+04] etc.). Projects such a Space4U [Spa], its predecessor Robocop [Rob], DECOS [DEC], SAVE [SAV] and PROGRESS [PRO] are targeting CBSE for embedded systems. Ever still, there is an apprehension that current tools and methods for embedded CBSE are lacking one or more key-properties to support industrial requirements such as:

- giving the software developer suitable level of expressiveness and/or abstraction

- enabling development of both safety critical and flexible functionality in a system
- enabling code and architecture reuse
- supporting development of predictable and resource efficient systems

To summarise, we recognise three important aspects of component based software development for resource constrained and predictable real-time systems: (a) the aspects of the developer, (b) the aspects of the analysis framework, and (c) the aspect of the run-time system. These three different aspects emphasise different, and sometimes conflicting, requirements for design, analysis and synthesis. For example, the developers must have sufficient methods and tools to design the overall software architecture. The analysis framework needs the architecture to be formal enough for automated analysis of important properties. And finally, the solution must be able to execute resource efficiently in the run-time platform. Any development strategy, for resource constrained and predictable real-time systems, has to take into account and balance these aspects to gain industrial usefulness.

In this paper we present a novel component model, RubusCMv3, for development of embedded control systems with a mix of hard, soft and non real-time requirements. The model is developed as a joint effort between industrial partners and the MultEx project [Mul] at Mälardalen Real-Time research Centre. The objective of the model is to support and balance common requirements in design, analysis and synthesis of embedded real-time control systems.

Paper outline. The reminder of this paper is organised as follows. In Section 8.2 we outline some common requirements in development of embedded software. In Section 8.3 we describe the main objectives of the RubusCMv3 component model. In Section 8.4 the architectural elements of RubusCMv3 are presented. In Section 8.5 we show a simple design of an oil pressure supervision to illustrate features of RubusCMv3. In Section 8.6 we discuss the key-properties supported by RubusCMv3. Finally, in Section 8.7 we conclude the work described in this paper.

8.2 Engineering requirements on RubusCMv3

The fact that more and more mechanical solutions are replaced with software, results in an increasing system complexity. Today's embedded systems are typically characterised by having a mix of functionality with requirements ranging

from hard real-time, soft and even non real-time. Many of these systems operate in resource constrained environments that need to satisfy requirements on dependability and efficient resource usage. For example, in a recent study [HMTN06], we discovered that vehicular systems are rapidly evolving and contain more heterogeneous functionality than before. Control applications, information handling and entertainment supplies, are nowadays supported in software by a mix of hard and soft real-time requirements. In addition, developers predict an increase in information intensity and a continuing increase in diversity of functionality. Safety and reliability are of utmost importance and considered key properties in development. This implies that development models must be able to support development and analysis of safety critical functionality, as well as development of flexible and resource efficient functionality.

8.3 Objective of RubusCMv3

This section describes the main objectives of the Rubus component model. The objectives are derived from a recent study of industrial requirements in development of embedded systems in the vehicle domain [HMTN06]. One of the key objectives of the RubusCMv3 model is to support an overall descriptive view of the system functionality, i.e., serving as a system description facilitating reasoning about the functionality at a high level. Furthermore, abstraction mechanisms should be supported through hierarchical decomposition. This allows reasoning on different levels of abstraction. Besides having different levels of abstraction a user should be able to see the system through different views, highlighting different aspects. For example, a developer might be interested in the functional view when first designing the system, and later on, focus on the real-time temporal aspects, hiding unnecessary details of the functional aspects. Thus it must be possible to express both real-time requirements and real-time properties of the design. The overall purpose of the component model should be to express the infrastructure of software functions, i.e., the interaction between software functions in terms of data- and control-flow. One important principle is to separate code and infrastructure, i.e., explicit synchronisation or data access should all be visible in the infrastructure level. Separating code and infrastructure facilitates analysis and reuse of components in different contexts.

The component model should have a formal syntax and semantics and thus lend itself to formal analysis at an early stage. This allows timing errors to be

revealed at an early stage in development. Real-time attributes on components can be expressed as budgets or estimates, enabling analysis of, for example, memory consumption and temporal attributes. These budgets can then serve as implementation requirements at a later stage in development.

The resulting architecture must be efficiently mapped to a run-system. With the diverging type of functionality in today's embedded systems, suitable execution models, such as hybrid scheduling [MTHN05], have to be supported. Hence, the user should be able to express events in terms of clocks and internal/external events.

8.4 The RubusCMv3 component model

The component model is developed as a part of the Rubus concept [Arc]. Rubus emanates from Basement [HLS96], and was first introduced for industrial use in 1996. Throughout the years Rubus been used by a number of companies, e.g., [BAE, Kno, Mec, VCE] for development of real-time software.

The RubusCMv3 model is intentionally simple, still giving enough expressiveness for development and analysis of resource constrained systems with mixed real-time requirements. It is intended for development of software architectures expressing data-flow and synchronisation between software entities in single and multi-node systems.

In the following sections, we describe the architectural elements for hierarchical decomposition of software logic and assignment of real-time properties in RubusCMv3. Throughout this paper, we collectively denote software elements as software items (SWIs).

8.4.1 Software logic

Software circuits
Software circuits (SWCs) are the basic unit of hierarchical decomposition in RubusCMv3. The primary purpose of an SWC is to encapsulate functions, hence a SWC can have multiple behaviours, each one represented by a specific entry function serving as the starting point of execution. Each SWC is defined by its behaviour, interface and an internal state data. Interfaces manage interaction between SWCs through ports. Two types of ports are supported, data ports for data flow and trigger ports for control flow. A SWC receives data(D)/triggering(T) on its input ports(I) and produces data/triggering to its output ports(O). Fig. 8.1 shows the graphical notation of an SWC in

RubusCMv3. OTU denotes unconditional triggering, i.e., a trigger signal that is always produced, whereas OTC denotes conditional triggering meaning that the trigger signal on the OTC port may be produced, depending on conditions within the SWC. Data transfer/triggering between two SWCs require that output ports of the transmitting component are connected to input ports on the receiving component. If a data output port and a trigger output port from the same component is connected to one other component, then the run-time environment must guarantee that the data will arrive to the destination before it is triggered by the source.

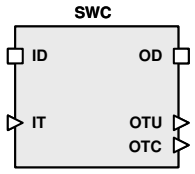

Figure 8.1: Graphical description of a software circuit

A SWC becomes eligible for execution when its trigger condition is true, i.e., when the input trigger port receives a trigger signal. Each SWCs executes with a run to completion semantics as shown in Fig. 8.2, i.e., a SWC is not allowed to have synchronisation primitives defined within its behaviour (i.e., in the source code of the component), meaning that all synchronisations must visible in the design and represented by synchronisation objects. Before an SWC becomes executing it first reads the data on the data input ports. When the SWC terminates, it produces data/trigger on its output ports and reverts to its idle state.

A SWC may preserve its internal state data between executions. The internal state of an SWC must be initialised by a constructor in the SWC and cleaned up by a destructor in the SWC. The constructor must be called by the run-time environment at system start up, and the destructor must be called when the system is shutdown in an orderly fashion.

Assemblies and composites

Assemblies and composites (ASMs/CMPs) provide ways to connect a set of SWCs. They also provide means for hierarchical decomposition of SWCs and their connections. ASMs/CMPs provides no semantics, hence the purpose of ASMs/CMPs is to provide structure and increased abstraction,

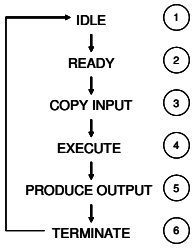

Figure 8.2: Run-cycle of software circuit

i.e., to abstract details of the software architecture. Assemblies and composites communicate through a set in- and output ports, similar to SWCs. Composites differ from assemblies in the sense that a composite object can be divided and parts of it can be deployed on different nodes, whereas assembly objects are un-dividable objects that cannot be split during deployment, i.e., an assembly object can only be deployed as a whole objects to a single node. Fig. 8.3 shows the graphical notation of an ASM in RubusCMv3

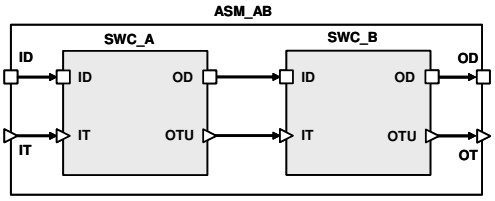

Figure 8.3: Graphical description of an assembly

Modes

Modes are means to distinguish different states or conditions of a system. Each mode describes the functionality that is relevant for that mode. For example, a system may execute a certain type of functionality during start-up, and other type of functionality when in operational mode. Mode transitions are specified in order to show the transitions that are legal in the system. This is illustrated by a high level state diagram describing switches between different system functionality. In RubusCMv3, the mode objects are treated as self contained

applications, realising the operational conditions of a system. Modes are semantically seen as synchronised only within a single node. Fig. 8.4 shows an example of modes and transitions within an Electronic Control Unit (ECU).

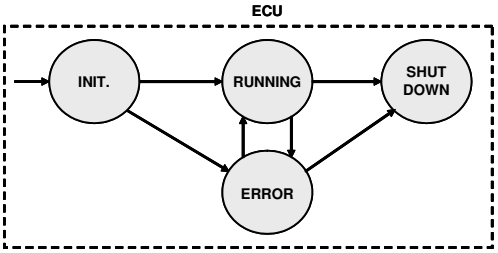

Figure 8.4: Example of modes and mode transitions in an ECU

System
A system is the top level hierarchical entity, describing the software logic and architecture for a complete, possibly distributed, system. A system contains no assignments to platforms.

Logic objects for data and triggering
RubusCMv3 also defines the following software logic items for data and triggering (Fig. 8.5):

- *Source* items are used to, (i) define constant values on data input ports, (ii) indicate an unconnected trigger input port, i.e., a port that will not be triggered.

- *Sink* items are terminators of data- and trigger output ports. The sink object indicates that the data or trigger from the output port will be terminated, i.e., the control or data flow from the port terminates at the sink.

- *Named data* items can be described as collectors of data. The item has blackboard semantics, i.e., any of the architectural element (described above), can write and read from a named data. Moreover, entities that are external to the model may read and write named data.

- *Clock* items define periodic triggering. A clock object has an trigger output port activated with a specified frequency. In addition, a clock object may define a possible delay (offset time) specifying the minimum delay, from that the clock produces a trigger out, until the first logic object connected to the trigger output port on the clock, may be activated for execution.
- *Interrupt and event* items define external interrupts, internal and external events. These items are used to define events generated either by hardware or other software items. Events are specified by a minimum interarrival time (MINT) and priority. MINT specifies the shortest time between two consecutive activation of an event. The priority denotes the priority of the event (for events that correspond to interrupts, the priority denotes the interrupt priority).
- *Down sampling* items may be used to alter the frequency of triggering. For example, consider a control flow between SWCs (SWC_1 and SWC_2), then, without a down sampling object, a trigger signal from an output port of SWC_1 is immediately delivered to the input port of SWC_2. When attaching a down sampling item to the control flow, the delivery of the trigger signal from the output port of SWC_1 to the input port of SWC_2 may be altered in the following way: If DOWNSAMPLE-FACTOR is x then the receiving port (input port on SWC_2 in this case) is triggered no more than every x^{th} time.
- *Precedence* items are used to express precedence within an sequence of connected unconditional output ports. These items have two or more trigger input ports and one trigger output port. Each of the SWIs connected to the input ports of a precedence item must have a single common ancestor in the a graph of preceding trigger ports. This means that a precedence item cannot be used to synchronise between different trigger chains.

8.4.2 Real-time properties in RubusCMv3

The software logic defined by the constituents of RubusCMv3 may be the associated with real-time execution properties defining actions that trigger executions, requirements on execution and timing properties of SWCs. To enable real-time analysis, each SWC is associated with a run-time profile describing the execution-time and memory consumption on different platforms.

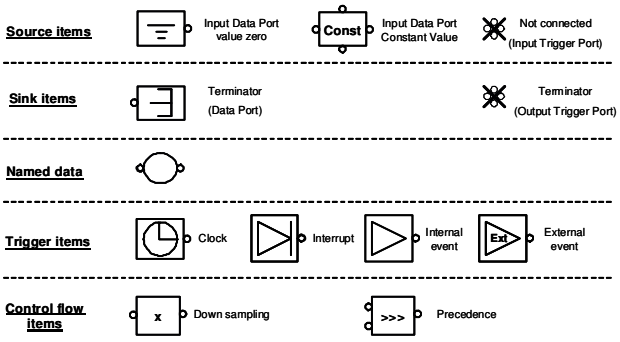

Figure 8.5: Logic items for data and triggering

Three types of real-time requirements are currently supported in RubusCMv3 (i) deadline on completion and (ii) offset and (iii) period jitter.

8.5 System example

In this section, we show a simple design of an oil pressure supervision (see Fig. 8.6) to illustrate some features of RubusCMv3. Assume that the requirements for the example are as follows:

- Correct oil pressure is important for the longevity of an combustion engine, hence the oil pressure of an engine should be regularly monitored. In case of abnormal oil pressure levels, the pressure level should be logged and an alarm, notifying the operator, must be raised within 5 seconds.

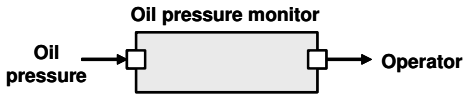

Figure 8.6: Example, oil pressure supervision

A typical development scenario using RubusCMv3 includes identifying and composing the software architecture, deploying the architecture and as-

signing real-time properties to the design.

For this example, we assume that a pressure sensor is used to measure the oil pressure and a LED to inform the operator of abnormal pressure conditions, hence, the software parts in this example are illustrated by the constituents of the oil pressure monitor composite in Fig. 8.7. The monitor consist of an object for supervision of the oil pressure, an object logging abnormal conditions and an alarm object informing the operator of abnormal oil pressure.

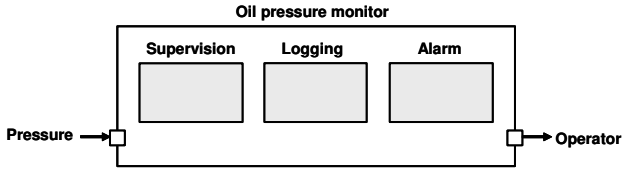

Figure 8.7: Example, oil monitor software parts

We are now ready to create a more detailed design of the oil monitor object and applying the requirements on the software. Recall that in RubusCMv3, boxes represent data, triangles represent activation interfaces and the arrows show direction of data and control flow. A clock is assigned to the supervision object for periodic monitoring of the oil pressure. We assume that abnormal pressure conditions are sporadic, hence the logging and alarming of abnormal pressure is activated by events. The real time requirement, describing the maximum time delay (5 seconds) from the occurrence of abnormal oil pressure until the alarm is raised, is assigned to the design (see Fig. 8.8).

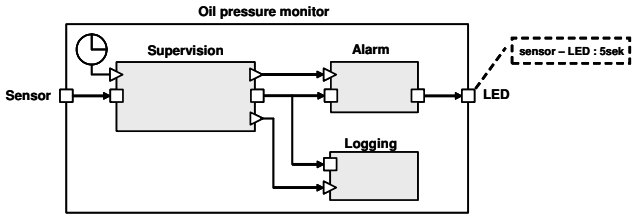

Figure 8.8: Example, oil monitor software logic

Figure 8.8 shows that the data flow and the control flow are separated in the design. Separating data from control flow makes it easier to extract parts from

the design and display only the data flow or only the control flow between object. We can now go further in to details and design the functionality for each of the objects in the oil pressure monitor. For simplicity, we only show an detailed example of the supervision assembly (Fig. 8.9).

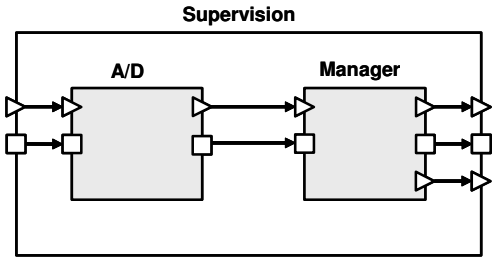

Figure 8.9: Example, supervision software logic

A key feature of RubusCMv3 is to allow developers to focus on functionality on system level, without considering the hardware architecture. To this point, the functionality is designed without hardware architectural concerns. However, in reality, the hardware architecture of a system is often established based on the physical conditions and hence conceptually established at an early point in development. In addition, the placement of sensors and other elements for sensing environmental conditions are guiding the placement of the computational hardware. For the rest of this example we assume that the pressure sensor is connected to an engine ECU and that the LED display is connected to a cabin ECU. The engine ECU is physically placed in the engine compartment and the cabin ECU in the cabin. A CAN bus is used for communication between the ECUs (Fig. 8.10).

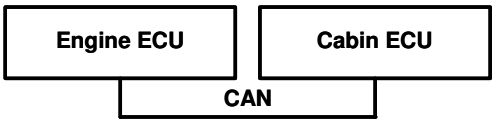

Figure 8.10: Example, ECUs

Objects describing ECUs can have a set of operational modes, i.e., states, and transitions between modes within an ECU. Assuming that we need to

deploy parts of our functionality to the drive mode of the engine ECU and other parts of our functionality to the drive mode of the cabin ECU, we then need to decide how to design the communication between the ECUs. In our example we assume that we need an object that packages messages into frames (CAN send) and an interrupt that activates the transmission of messages. At the receiving end we assign a CAN receive object that unpacks the messages and activates the software that alarms the operator of abnormal oil pressure conditions. Fig. 8.11 shows the software logic of each mode.

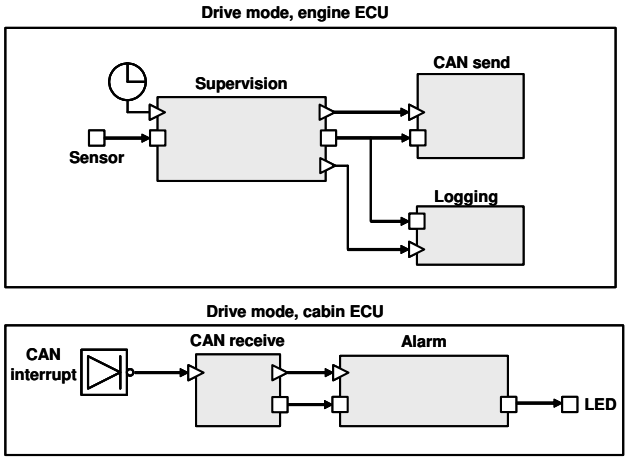

Figure 8.11: Example, mode logic

To enable timing- and memory analysis of our example design, the objects need to be assigned properties such as worst case execution times and maximum stack usage etc. At this point, we may or may not have source code for the objects. We can however, assign timing and memory budgets to the objects and perform analysis. We can then choose to write or retrieve the source code to get more accurate figures on timing and memory consumption. For our example, we also need to consider timing in messages transmission on the CAN bus.

8.6 System development using RubusCMv3

The RubusCMv3 model provides constructs for systems design through hierarchical decomposition of software logic, giving developers the possibility design the logic at different levels of abstraction. By separating data- and control-flow, developers can exploit the benefits of addressing different views, e.g., the logical or the temporal view of a design. The oil pressure example shows the separation of concerns in different phases of development. First, the design of the complete functionality is put in focus, without hardware concerns. Secondly, the data flow and control flow are separated for abstraction reasons. The example also shows different levels of abstraction of a software entity, i.e., the monitor object consist of a supervision object that consists of an A/D converter and an oil manager.

The example also illustrates how a distributed functionality, with hard and non real-time requirements, can be realised by two execution models (EMs). The hard real time requirement, realised by a time-triggered EM, concerns the maximum delay from abnormal pressure until an alarm is raised, whereas the logging of abnormal pressures, realised by the event-triggered EM, lack explicit timing requirements. Industrial systems often contain a mix of different real-time requirements. Still, many of these systems are developed using only the time triggered EM. The time triggered EM is, of course, suitable for periodic control systems and other type of functionality that need to be managed or polled periodically. However, the mixed requirements on systems indicate that the event-triggered execution model could also be used in systems development. For this reason, RubusCMv3 supports two fundamental execution models, the time- and event-triggered models. Hence, developers can choose different EMs for different subsystems, e.g., static scheduling for critical core functions (which is often desired, and sometimes even mandated by safety standards and certification agencies [Com00]) while allowing less critical functions to be executed using a less resource demanding and flexible EM.

The fact that EMs are logical objects defined in the infrastructure, instead of being coded into the components, will facilitated component reuse, since components can be developed independent of the EMs. This, in turn, will in a long term perspective decrease the software development costs. Also, for companies that sustain a product line[1], software reuse is crucial and is an important factor in decreasing the time-to-market for new products.

To formally verify whether the real-time properties of a Rubus design is

[1] A product line is a series of related, but yet distinct, products. Economical benefits are achieved by synergies in the development and maintenance of the products in the product line.

met, response time analysis such as [Red03, MT05, DBBL07, PPE+06] may be used. In addition to formal analysis, verification of the logical functionality may be done by, for example, stimulating data and triggering. We have developed a framework for such verification of SWCs and ASMs/CMPs on PCs, see Fig. 8.12. The framework reads input data from, for example, Matlab. The data is then fed to the simulation process that controls the stimulation of input ports and state variables using probes. The output from the simulation process can be fed back to, for example, Matlab. This gives developers possibilities to test the logical functionality of software elements, even without having the actual hardware at hand. The framework has been successfully tested in development and it is currently used by our industrial partners.

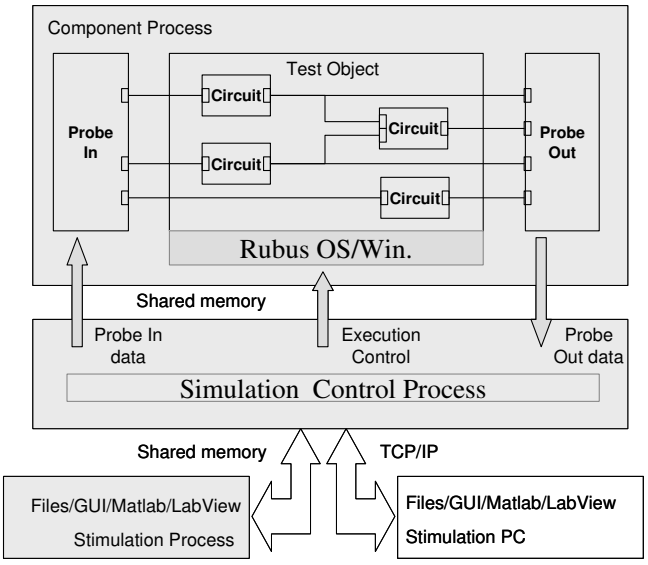

Figure 8.12: Framework for testing of logical functionality of Rubus software elements

For resource constrained embedded systems, the resulting design must be resource efficiently mapped to a run time system. The mapping of logical software items to executable threads can of course be done in several ways. However, the RubusCMv3 model provides possibilities to map several components into one executable thread, minimising contexts switch overheads. This is pos-

sible since the model provides possibilities to realise functionality as transactions, i.e., a sequential execution of components triggered by a common source such as a clock item. In addition, the run-to-completion semantics of SWCs and the fact that synchronisation is done in the infrastructure, enables efficient use of memory by stack sharing. Stack sharing allows several tasks to share, i.e., execute, on a single stack, even in preemptive systems. It has been shown that significant memory savings can be achieved by stack sharing. Today, there exists a number of methods to formally analyse the amount of stack needed in shared stack systems, e.g., [BHMT+08, DMT00, HMTB+06].

A run-time environment is essential to provide the infrastructure services that are needed to execute the logic defined by a component model. Furthermore, a run-time environment must also be able to preserve the semantics and support efficient execution of the logics defined by a component model. In general, RubusCMv3 do not restrict the use of a specific run-time environment, in fact, any run-time environment that endorse the semantics of RubusCMv3 can be used. However, to fully support RubusCMv3 model the run-time environment must be able to realise the complete set of logic with the semantics defined by the model. Currently, we have extended the Rubus-RTOS [Arc] to fully support the RubusCMv3 model.

8.7 Conclusions and future work

Today's embedded systems are typically characterised by having a mix of functionality with requirements ranging from hard real-time, soft and even non real-time. Many of these systems operate in resource constrained environments that need to satisfy requirements on dependability as well as efficient resource usage. Current trends predict a continuing increase in diversity of functionality in systems, resulting in an increasing system complexity. Component Based Development (CBD) is a promising and emerging development discipline for real-time systems, providing many attractive qualities such as encapsulation, high level description and reusable logic.

This paper presented the RubusCMv3 component model, a novel component model for development of resource constrained embedded real-time systems. The model has been developed in close cooperation with industrial partners. It aims at supporting three important activities in real-time development, (i) design, (ii) analysis and (iii) synthesis. These activities emphasise different and sometimes conflicting requirements that need to be balanced. The model provides methods to express the infrastructure of software functions,

i.e., the interaction between software functions in terms of data- and control-flow. The resulting architecture is formal enough for analysis of timing and memory properties. The components and the infrastructure allows for a resource efficient mapping onto a run-time structure.

The model is integrated in the next generation of the Rubus tool suite (RubusICE), the model is currently evaluated on real systems. We have successfully converted a traditionally developed industrial system, into a component based system using RubusCMv3. Future work consists of evaluating the component model in other industrial settings.

Bibliography

[Arc] Arcticus Systems. Web page, http://www.arcticus-systems.com.

[BAE] BAE Systems Hägglunds. Web page, http://www.baesystems.-com/hagglunds.

[BHMT+08] M Bohlin, K Hänninen, J. Mäki-Turja, J Carlson, and M. Nolin. Safe Shared Stack Bounds in Systems with Offsets and Precedences. Technical Report MRTC no. 221, Mälardalen Real-Time Research Centre (MRTC), January 2008.

[CE03] A. Cervin and J. Eker. The Control Server Model: A Computational Model for Real-Time Control Tasks. In *Proc. of the 15th Euromicro Conference on Real-Time Systems*, July 2003.

[Com00] International Electrotechnical Commission. *Functional Safety and IEC 61508*, May 2000.

[DBBL07] R.I. Davis, A. Burns, R.J Bril, and J.J Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[DEC] DECOS - Dependable Embedded Components and Systems. Web page, http://www.decos.at.

[DGL+04] M. Díaz, D. Garrido, L.M. Llopis, F. Rus, and J.M. Troya. Integrating Real-Time Analysis in a Component Model for Embedded Systems. In *Proceedings of the* 30th *Euromicro Conference*, pages 14–21. IEEE Computer Society, Rennes, France, September 2004.

[DMT00] R. Davis, N. Merriam, and N. Tracey. How Embedded Applications using an RTOS can stay within On-chip Memory Limits. In *Proc. of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[HLS96] H. Hansson, H. Lawson, and M. Strömberg. BASEMENT a Distributed Real-Time Architecture for Vehicle Applications. *Journal of Real-Time Systems*, 3(11):223–244, November 1996.

[HMTB+06] K Hänninen, J Mäki-Turja, M Bohlin, J Carlson, and M Nolin. Determining Maximum Stack Usage in Preemptive Shared Stack Systems. In *Proc. 27^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 445–453, December 2006.

[HMTN06] K. Hänninen, J. Mäki-Turja, and M. Nolin. Present and Future Requirements in Developing Industrial Embedded Real-Time Systems -Interviews with Designers in the Vehicle Domain. In *Proceedings of the* 13^{th} *International Conference and Workshop on the Engineering of Computer Based Systems*, pages 139–147. IEEE Computer Society, Potsdam, Germany, March 2006.

[Kno] Knorr-bremse. Web page, http://www.knorr-bremse.com.

[MÅFN04] Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin. Evaluation of Component Technologies with Respect to Industrial Requirements. In *Euromicro Conference, Component-Based Software Engineering Track*, August 2004.

[Mec] Mecel. Web page, http://www.mecel.se/.

[Mica] Microsoft. Microsoft COM Technologies. Web page, http://-www.microsoft.com/com/.

[Micb] Microsoft. .NET. Web page, http://www.microsoft.com/net/.

[MSZ02] P. O. Müller, C. M. Stich, and C. Zeidler. *Building Reliable Component-Based Software Systems*, chapter Component Based Embedded Systems, pages 303–323. Artech House publisher, 2002. ISBN 1-58053-327-2.

[MT05] J. Mäki-Turja. *Engineering Strength Response-Time Analysis, A Timing Analysis Approach for the Development of Real-Time Systems*. PhD thesis, Mälardalen University, Department of Computer Science and Electronics, 2005. 179 pp.

[MTHN05] J. Mäki-Turja, K. Hänninen, and M. Nolin. Efficient Development of Real-Time Systems Using Hybrid Scheduling. In *International conference on Embedded Systems and Applications (ESA)*, June 2005.

[Mul] MultEx project. Web page, http://www.mrtc.mdh.se/projects/-multex/.

[OMG] OMG. CORBA. Web page, http://www.omg.org/corba/.

[OMG02] OMG. CORBA Component Model 3.0, June 2002. Web page, http://www.omg.org/technology/documents/formal/-components.htm.

[PEC] PECOS project. Web page, http://www.pecos-project.org.

[PPE+06] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing Analysis of the FlexRay Communication Protocol. In *Proc. of the 18th Euromicro Conference on Real-Time Systems*, July 2006.

[PRO] PROGRESS. Web page, http://www.mrtc.mdh.se/progress/.

[Red03] O. Redell. *Response Time Analysis for Implementation of Distributed Control Systems*. PhD thesis, KTH, Department of Machine Design, 2003. Series: TRITA-MMK 2003:17.

[Rob] Robocop project. Web page, http://www.extra.research.philips.-com/euprojects/robocop/index.htm.

[SAV] SAVE. SAVE project. Web page, http://www.artes.uu.se/++/-SAVE/.

[Spa] Space4U project. Web page, http://www.extra.research.philips.-com/euprojects/space4u.

[Sta01] John A. Stankovic. VEST — A Toolset for Constructing and Analyzing Component Based Embedded Systems. *Lecture Notes in Computer Science*, 2211:390–402, 2001. URL `citeseer.nj.nec.com/stankovic00vest.html`.

[SUNa] SUN Microsystems. Enterprise Javabeans Technology. Web page, http://java.sun.com/products/ejb/.

[SUNb] SUN Microsystems. Introducing Java Beans. Web page, http://developer.java.sun.com/developer/onlineTraining/-Beans/Beans1/index.html.

[VCE] Volvo construction equipment. Web page, http://www.volvoce.com.

[Ves97] S. Vestal. Support for Real-Time Multi-Processor Avionics. In *Proc. 18^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 11–21, December 1997.

[vO02] Rob van Ommering. *Building Reliable Component-Based Software Systems*, chapter The Koala Component Model, pages 223–236. Artech House Publishers, July 2002. ISBN 1-58053-327-2.

[Wal03] A. Wall. *Architectural Modeling and Analysis of Complex Real-Time Systems*. PhD thesis, Mälardalen University, Dept. of Computer Science and Engineering, September 2003.

Chapter 9

Paper D: Efficient Event-Triggered Tasks in an RTOS

Kaj Hänninen, John Lundbäck, Kurt-Lennart Lundbäck, Jukka Mäki-Turja, Mikael Nolin
In Proceedings of the 2005 International Conference on Embedded Systems and Applications, Las Vegas, USA, June, 2005.

Abstract

In this paper, we add predictable and resource efficient event-triggered tasks in an RTOS. This is done by introducing an execution model suitable for example control software and component-based software. The execution model, denoted single-shot execution (SSX), can be realized with very simple and resource efficient run-time mechanisms and is highly predictable, hence suitable for use in resource constrained real-time systems. In an evaluation, we show that significant memory reductions can be obtained by using the SSX model.

9.1 Introduction

When designing software for embedded systems, resource consumption is often a major concern. Software consumes resources primarily in two domains: the time domain (execution time), and the memory domain (e.g., RAM and flash memory). For systems without real-time requirements, the resource consumption in the time domain may be of less importance. However, most embedded systems are either used to control or monitor some physical process, or used interactively by a human operator. In both of these cases, it is often required that the system responds within fixed time limits. Hence, methods for development of embedded systems need to allow design of both memory and time efficient systems. Moreover, predictable use of the resources are required. Predictions of the amount of resources needed to execute the system are used to dimension the system resources (e.g., selecting CPU and amount of memory). The current trend in development of embedded systems is towards using high-level design tools with a model-based approach. Models are described in tools like Rational Rose, Rhapsody, Simulink, etc. From these models, whole applications or application templates are generated. However, this system generation seldom considers resource consumption. The resulting systems become overly resource consuming and even worse; they exhibit unpredictable resource consumption at run-time. In this paper, we describe and evaluate the integration of a resource efficient and predictable execution model, denoted single shot execution model (SSX), in a commercial real-time operating system. The execution model facilitates stack sharing to reduce memory consumption and priority scheduling to allow timing predictions. The paper is organized as follows. In section 9.2, we describe the properties of the SSX model and the prerequisites needed to utilize the model. In section 9.3, we describe our target platform, the Rubus RTOS. In section 9.4, we describe the integration of SSX in Rubus and in section 9.5, we evaluate the stack usage under the SSX model in different execution scenarios. In section 9.6, we conclude the integration of SSX in Rubus.

9.2 The single shot execution model (SSX)

Throughout the years, research in real-time scheduling and real-time operating systems has resulted in a vast number of different execution models, e.g.,[Bak90, DMT00, Loc92, Nor, XP00], one of them being the single shot execution model in which tasks are considered to terminate at the end of each in-

vocation, i.e., execute to completion (as opposed to indefinitely looping tasks). Baker [Bak90] and Davis et al. [DMT00] shows that the single shot execution model, with an immediate priority ceiling protocol, enables possibilities for efficient resource usage by stack sharing among several tasks. Stack sharing in the SSX model is feasible because higher priority tasks are allowed to pre-empt lower priority tasks and execute to completion (i.e., terminate) before lower priority tasks are allowed to resume their execution. However, the fact that a task must execute to completion (terminate) before any lower priority task is allowed to execute, puts some restrictions on suspensions of tasks in the SSX model:

- To guarantee correct stack access, self-scheduling of SSX tasks, i.e., calling timed sleep or delay functions may not be used in the application code of SSX tasks.
- Task synchronization should be done using the Immediate Priority Ceiling Protocol (IPCP). This ensures that a task will never be allowed to start executing, before it is guaranteed to have access to all resources it needs. Hence, calls for accessing shared resources, such as semaphores, will never result in blocking due to locked resources. Any possible blocking will occur before the task is allowed to start execute.

However, these design restriction also facilitate predictability since the administration of the tasks is left entirely to the operating system. Moreover, it is known that the immediate priority ceiling protocol is deadlock free and exhibits an upper bound on blocking times for tasks sharing resources. This implies that an analysis technique such as response-time analysis [ABD$^+$95] enables analysis of temporal properties of an SSX system. The SSX model is conceptually very simple, at run-time a task can be in one of three states: terminated, ready, or executing. The main difference, from a developers view, is that a conventional RTOS often uses so called self-scheduled tasks. This means that a task is activated once, typically at system start-up, and eventually, after some possible initialization code, ends up in an infinite loop where it self-schedules itself, e.g., using delay calls. An SSX task, on the other hand, when activated by the OS, executes with no delay calls, and terminates upon completion. This means that such tasks have to be re-scheduled by the OS in order to provide a continuous service. Figure 9.1 illustrates the structural difference between a conventional task and an SSX task.

In this paper, we present an integration of the SSX model in the Rubus RTOS. We also present a quantitative evaluation of stack usage under different execution scenarios.

```
taskEntryFunc(){
 while(1){
  //Task code
  sleep(time)
 }
}
```

```
taskEntryFunc(){

 //Task code
}
```

Figure 9.1: Looping task (left). Typical SSX task (right)

9.3 The Rubus operating system

Rubus is a real-time operating system developed by Arcticus Systems [Arc]. Rubus is targeted towards systems that typically require handling of both safety critical functions as well as less critical functions. The emphasis of Rubus is placed on satisfying reliability, safety and temporal verification of applications. It can be seen as a hybrid operating system in the sense that it supports both statically and dynamically scheduled tasks. The key features of Rubus RTOS are:

- Guaranteed real-time service for safety critical applications
- Best-effort service for non-safety critical applications
- Support for time- and even-triggered execution of tasks
- Support for component based applications

Rubus consist of three separate kernels (Figure 9.2). Each kernel supports a specific type of execution model.

The *Red kernel* supports time driven execution of static scheduled 'Red tasks', mainly to be utilized for applications with fixed hard real-time requirements. The static schedule is created off-line by the Rubus Configuration Compiler. Synchronizations of shared resources are handled by time separation in the static schedule. All tasks executed under the Red kernel share a common stack. A Red task is implemented by a C function, and the task is completed when the function returns. The *Blue kernel* administrates event driven execution of dynamic scheduled 'Blue tasks', mainly intended for applications having soft real-time requirements. Task handled by the Blue kernel are scheduled on-line by a fixed priority pre-emptive scheduler. Synchronizations among

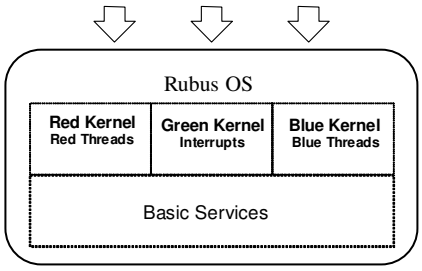

Figure 9.2: Rubus RTOS architecture

Blue tasks are managed by a Priority Ceiling Protocol (PCP)[SRL90]. As opposed to the Red execution model, the Blue execution model does not support stack sharing among Blue tasks. Blue tasks are commonly used as indefinitely looping tasks (see Figure 9.1) periodically reactivated by system calls, e.g., blueSleep, that suspends the execution of Blue tasks for a specified time interval. The *Green kernel* handles external interrupts. The 'Green tasks' are scheduled on-line with a priority based scheduling algorithm dependent of the application hardware, i.e., microprocessor. The Rubus off-line scheduler is guaranteed to generate a static schedule (see Red kernel above) with sufficient slack available to handle interrupts [SEF98]. When a Green task is executed, it may utilize the stack of the currently active Red or Blue task, implying that the active task may need to supply stack space for interrupt handling. Dispatch priorities of the tasks executing under the different kernels are illustrated in Figure 9.3. Tasks managed by the Green kernel have highest priority, and tasks managed by the Blue kernel have lowest priority.

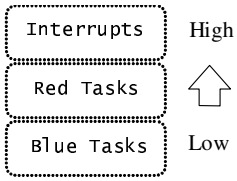

Figure 9.3: Task priorities in Rubus

Rubus supports the possibility to utilize software components for application development. The computational part of the supported software components is realized either by a Green, Red or by a Blue task.

9.4 Integration of SSX in Rubus

Introducing a new execution model in an operating system for resource constrained embedded real-time systems, require careful design to minimize the overhead of the new model and effects (temporal and spatial) on existing models. On one hand, we could minimize the memory overhead imposed by the new execution model, by sharing administrative code in the kernel between the existing execution models and the new execution model. In doing so, we would impose additional timing overhead on the existing models wherever a kernel needs to be able to separate the different models, e.g., at sorting, queuing and error handling etc. On the other hand, we could avoid imposing timing effects on the existing models by separating the models, i.e., modularize, and allow the kernel to administrate the new SSX model in isolation from the existing execution models. This approach would increase the number of administrative functions, thus requiring more memory. In this implementation, we choose to share administrative OS code between the SSX model and the Blue model since the timing overhead imposed by the SSX model, on the Blue model, is very low. A new execution model may be introduced to a system by changing the current scheduling policy or existing task model. In our case, we retain the same scheduling policy for the SSX model as for the Blue model (fixed priority scheduling). However, a new task model is introduced to support the SSX model. Each task in Rubus is defined by its: basic attributes, Task Control Block (TCB), stack/heap memory area and application code. By adding periodicity and deadline attributes to the existing task model, we are able to share all fundamental task structures between SSX tasks and the existing Blue tasks. Administration of SSX tasks is handled entirely by the Blue kernel (Figure 9.4).

The resulting relation of task priorities in Rubus, including SSX, is illustrated in Figure 9.5. The priority assignments and the fact that the administration of SSX tasks are handled entirely by the Blue kernel, makes the temporal attributes of tasks using the SSX model fully analyzable. In systems consisting solely of SSX tasks, the analysis can be performed with [ABD+95]. The SSX tasks can also be analyzed in hybrid systems consisting of Interrupts, Red tasks and SSX tasks with [MTHN05].

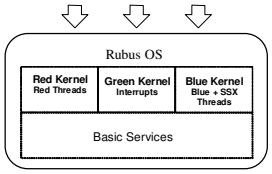

Figure 9.4: Rubus RTOS architecture with SSX model

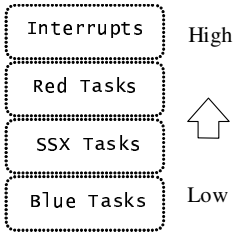

Figure 9.5: Task priorities in Rubus, including SSX

All tasks executing under the SSX model share a common stack (in fact, there is nothing that prevents stack sharing also between Red and SSX tasks). The common stack pointer, for SSX tasks, is globally accessible, hence it does not have to be stored in the TCBs. To support resource sharing in the SSX model, the immediate priority ceiling protocol was implemented. The following is a summary of all major changes made in Rubus to support the SSX execution model:

- Separation of tasks administrated by the Blue kernel and executed under different models
- Modification of administrative functions to support SSX tasks
- Error detection for SSX tasks
- Activation functionality for the SSX tasks
- Introduction of the immediate priority ceiling protocol

The integration of SSX in Rubus allows the execution model to be directly applicable for the Rubus component model. Hence, the possibility to utilize

software component for applications has been extended to include four execution models, the Green, the Red, the Blue and the SSX model. The shared stack in the SSX model can be safely dimensioned, as shown below, by summing the maximum stack usage of all tasks in each priority level, and adding stack-space for interrupts. SSX tasks with equal priorities cannot pre-empt each other in Rubus, hence it suffice to take the maximum stack usage in each priority level.

$$su_{ssx} = su_{\text{int}} + \sum_{\forall i \in P} \max_{\forall \tau_j \in P_i} su_j$$

su_{ssx}, denotes maximum stack usage of all SSX tasks. su_{int}, denotes interrupt stack usage. P_i denotes the set of tasks with priority i. P denotes the set of all priority levels. τ_i denotes task i. $su(\tau_i)$ denotes stack usage, including context switch overhead, of task i. A more accurate dimensioning approach would be to examine possible pre-emptions, and identify the pre-emption(s) resulting in maximum stack usage. Identification of possible pre-emptions in a fixed priority based system is considered in [RG04].

9.5 Evaluation of SSX in Rubus

Stack sharing allows for an efficient memory usage, which may avoid or at least postpone the need for additional RAM in evolving systems. To illustrate how a shared stack affects memory usage, we simulate different execution scenarios where the total stack usage varies a lot depending on the execution model in use. We simulate three different execution scenarios using two different execution models, the Blue and the SSX model, for each scenario. The first scenario is obtained from a flyer promoting the SSX5 RTOS [Nor]. The second scenario models a traditional control application, where a sequence of tasks is used to sense, calculate control parameters, and actuate. These tasks are executed in sequence, hence they do not pre-empt each other. The third scenario illustrates a system with full pre-emption depth, i.e., all tasks are pre-empted. The scenario can be seen as an example where the benefit of SSX is less, e.g., SSX as interrupt handling tasks in systems with multiple interrupt levels.

9.5.1 Evaluation method

The evaluations are performed under a Rubus OS simulator running on a PC. We calculate the stack usage as the maximum number of data pushed onto the

stack, from the dispatching of a task until it has finished execution. In doing so, we are able to include the concealed pushes of stack frames, i.e., stack usage occurring before execution of the actual task code, in the calculations. All stacks are initially filled with a pre determined data pattern. We then calculate the number of overwritten data patterns, i.e., stack usage, by examining the content of the stacks at termination of the system. It is assumed that tasks do not push frames identical to the pre determined data pattern at run time.

9.5.2 Application description

In each of the following execution scenarios, timer interrupts are generated at a frequency of 100Hz. Each timer interrupt activates the Blue kernel task, responsible for time supervision and dispatching of the tasks in the system. The service routine for timer interrupts, having highest priority in the system, execute on the stack of an active task. However, in the following scenarios, the worst-case execution times of the tasks are very short, resulting in all tasks finishing their executions before any consecutive timer interrupt hits the system. The run time model in each of the following scenarios is fixed priority, preemptive scheduling. We denote the priority of a task with Π, and its period, or in the case of sporadic tasks, its minimum interarrival time with T.

Scenario 1

The task set in the following scenario, obtained from a flyer evaluating the overheads of SSX5 [Nor], consists of; seven periodic tasks with periodicities ranging from 10ms to 80ms, and three interrupt handling tasks with a minimum interarrival time of 20ms (see Table 9.1).

Running the system under the SSX model (with one shared stack for tasks τ_1 - τ_{10}), results in a total stack usage of 316 bytes. With the Blue kernel stack included, the total stack usage is 460 bytes. Yet again, we evaluate scenario 1 but with the difference that tasks τ_1 - τ_7 are executed as Blue tasks (Blue execution model), achieving a pseudo periodic behavior by a call to a sleep function. According to the Rubus OS reference [Arc], the suspension (sleep) of the tasks requires two additional local variables, and besides the sleep call, an additional call to a function that converts the suspension time into timer ticks, resulting in increased stack usage (from 72 bytes to approximately 152 bytes) for a Blue task. This results in a total stack usage of 1480 bytes for tasks $\tau 1$ - $\tau 10$. With the Blue kernel stack included, the total stack usage is 1612 bytes. We noticed that the kernel uses 12 bytes less stack under the Blue model, than under the SSX model. This is due to Blue tasks scheduling themselves, instead of being assigned an activation time by the kernel. Table

Table 9.1: Task set, Scenario 1

Task	Π	T(ms)	Stack usage(bytes) SSX/Blue
τ_{KERNEL}	15	10	144/132
τ_1	5	10	72/152
τ_2	5	10	72/152
τ_3	4	20	72/152
τ_4	4	20	72/152
τ_5	3	40	72/152
τ_6	2	80	72/152
τ_7	2	80	72/152
τ_8	5	≥ 20	72/72
τ_9	5	≥ 20	72/72
τ_{10}	5	≥ 20	72/72

9.2 shows the resulting stack usage for scenario 1.

Table 9.2: Stack usage, Scenario 1

Exec. model	Total stack usage (bytes)
SSX	460
Blue	1612
Savings	≈71%

Scenario 2

The following scenario consists of pure periodic tasks with harmonic period times (see Table 9.3). The scenario can be seen as a simplification of a typical vehicular control system, e.g., as described in [RH03].

Table 9.4 shows the resulting stack usage for scenario 2 under the SSX and Blue execution models.

Scenario 3

The previous scenario shows an ideal situation for introducing SSX tasks. However, in applications where most tasks are asynchronous and pre-emptions appear randomly, the gains of SSX tasks is less, Thus, this scenario is prepared to show that the total stack usage, in certain situations, is nearly identical between the SSX and Blue execution model. The task set in this scenario consists of one periodic task τ_4 and three event-triggered tasks τ_1 - τ_3 (see Table

Table 9.3: Task set, Scenario 2

Task	Π	T(ms)	Stack usage(bytes) SSX/Blue
τ_{KERNEL}	15	10	144/132
τ_1	5	10	72/152
τ_2	5	10	72/152
τ_3	4	20	72/152
τ_4	4	20	72/152
τ_5	3	40	72/152
τ_6	2	80	72/152
τ_7	2	80	72/152

Table 9.4: Stack usage, Scenario 2

Exec. model	Total stack usage (bytes)
SSX	216
Blue	1196
Savings	≈82%

9.5). The execution of the task set is prepared to exhibit full pre-emption depth meaning that if a task can be pre-empted it will be so. Each task is assigned a unique priority, thus enabling pre-emption between each pair of tasks.

Table 9.5: Task set, Scenario 3

Task	Π	T(ms)	Stack usage(bytes) SSX/Blue
τ_{KERNEL}	15	10	144/132
τ_1	5	-	72/72
τ_2	4	-	72/72
τ_3	3	-	72/72
τ_4	2	80	72/152

Table 9.6 shows the resulting stack usage for scenario 3 under the SSX and Blue execution models.

Table 9.6: Stack usage, Scenario 3

Exec. model	Total stack usage (bytes)
SSX	612
Blue	708
Savings	≈14%

9.5.3 Results

Simulations have shown that stack memory usage in Rubus OS varies when comparing systems executed under the SSX model and systems executed under the Blue model. The differences in stack usage are mainly dependent on the type of application being realized. The fact that each Blue task is allocated its own stack makes them less memory efficient in all scenarios. In an example system of 7 non-pre-emptable tasks, the difference in stack memory usage is as much as 82% less for SSX tasks than for Blue tasks. Another system derived from a flyer on SSX5, results in a difference of 71% less stack usage for the SSX tasks than for the Blue tasks. However, less difference in stack usage is observed in situations of deeply nested pre-emptions. As the pre-emption depth increases, the difference in stack usage typically decreases. This is shown by our simulations of a system with full pre-emption depth where the difference in stack usage between the SSX model and the Blue model, is relatively low. Hence, the SSX model is specifically suitable for applications where jobs (or transactions) of dependent tasks are modeled without pre-emptions within the jobs, e.g., control systems. On the contrary, the SSX model is less beneficial for applications experiencing large pre-emption depths. However, in any type of application, the SSX model is at least as resource efficient, with respect to stack usage, as the Blue model. This makes the SSX model an attractive choice when developing systems.

9.6 Conclusion and future work

In this paper, we presented the integration of a resource efficient and predictable single shot execution model in the Rubus RTOS. The model allows for efficient stack usage and predictability of temporal attributes. These facts make the model attractive for development of resource constrained real-time systems. The integration has shown that the model can be integrated with very

simple run-time mechanisms. As future work, we are planning to include support for development and analysis (temporal and spatial) of SSX in Rubus Visual Studio (VS), which is an integrated environment for design, simulation and analyzing of embedded real-time applications.

Bibliography

[ABD+95] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed Priority Pre-Emptive Scheduling: An Historical Perspective. In *Real-Time Systems, 8(2/3)*, 1995.

[Arc] Arcticus Systems. Web page, http://www.arcticus-systems.se.

[Bak90] T.P. Baker. A Stack Based Resource Allocation Policy for Real-Time Processes. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*. IEEE, 1990.

[DMT00] R. Davis, N. Merriam, and N. Tracey. How Embedded Applications using an RTOS can stay within On-chip Memory Limits. In *Proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[Loc92] C.D. Locke. Software Architecture For Hard Real-Time Applications - Cyclic Executives vs. Fixed Priority Executives. *The Journal of Real-Time Systems*, 4:37–53, 1992.

[MTHN05] J. Mäki-Turja, K. Hänninen, and M. Nolin. Efficient Development of Real-Time Systems using Hybrid Scheduling. In *Proceedings of the International Conference on Embedded Systems and Applications (ESA)*, June 2005.

[Nor] Northern Real-Time Applications, SSX5 true RTOS. Web page, http://www.ssx5.com/NRTAHome.htm.

[RG04] R.Dobrin and G.Fohler. Reducing the number of preemptions in fixed priority scheduling. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, July 2004.

[RH03] T. Riutta and K. Hänninen. Optimal Design. Master's thesis, Dept. of Computer Science and Engineering, Mälardalen University, 2003.

[SEF98] K. Sandström, C. Eriksson, and G. Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, 1998.

[SRL90] L. Sha, R. Rajkumar, and JP. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In *IEEE Transactions on Computers, Volume: 39, Issue 9*, 1990.

[XP00] J. Xu and D.L. Parnas. Priority Scheduling Versus Pre-Run-Time Scheduling. *The Journal of Real-Time Systems*, 18(1):7–23, January 2000.

Chapter 10

Paper E: Determining Maximum Stack Usage in Preemptive Shared Stack Systems

Kaj Hänninen, Jukka Mäki-Turja, Markus Bohlin, Jan Carlson, Mikael Nolin
In Proceedings of the 27th IEEE Real-Time Systems Symposium, Rio de Janeiro, Brazil, December, 2006.

Abstract

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based on dynamic (run-time) properties.

We also show how to safely approximate the exact stack usage by using static (compile time) information about the system model and the underlying run-time system on a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system.

Comprehensive evaluations show that our technique significantly reduces the amount of stack memory needed compared to existing analysis techniques. For typical task sets a decrease in the order of 70% is typical.

10.1 Introduction

In conventional multitasking systems, each thread of execution (task) has its own allocated execution stack. In systems with a large number of tasks a large number of stacks are required. Hence the total amount of RAM needed for the stacks can grow exceedingly large.

Stack sharing is a memory model in which several tasks share one common run-time stack. It has been shown that stack sharing can result in memory savings [DMT00, HLL+05] compared to the conventional stack model. The shared stack model is applicable to both non-preemptive as well as preemptive systems, and it is especially suitable in resource constrained embedded real-time systems with limited amount of memory. Stack sharing is currently supported by many commercial real-time kernels, e.g., [Arc, Dig, Gro, Uni].

The traditional method to calculate the memory requirements for a shared run-time stack in preemptive systems is to sum the maximum stack usage of tasks in each preemption level and possibly consider additional overheads such as memory used by interrupts and context switches. A major drawback with the traditional calculation method is that it often results in over allocation of stack memory by presuming that all tasks with maximum stack usage in each priority level can preempt each other in a nested fashion during run-time. However, there may, in many cases, be no actual possibility for these tasks to preempt each other (e.g., due to explicit or implicit separation in time). Moreover, the possible preemptions may not be able to occur in a nested fashion.

Taking advantage of the fact that many real-time system exhibit a predictable temporal behavior, it is possible to identify feasible preemption scenarios, i.e., which preemptions can in fact occur, and whether they can occur in a nested fashion or not. Therefore, a more accurate stack analysis can be made. One example of a system that lends itself to such analysis is a hybrid, statically and dynamically, scheduled system. Such a system consists of an off-line scheduler producing the static schedule and a fixed priority scheduler (FPS) that dispatches tasks at run-time. The commercial operating system, Rubus OS by Arcticus Systems AB [Arc], supports such a system model. The Rubus OS is mainly used in resource-constrained embedded real-time systems. For instance, in the vehicular industry, Volvo Construction Equipment (VCE) [VCE], BAE Systems Hägglunds [BAE], and Haldex Traction Systems [Hal] all use the Rubus OS in their vehicles or components.

In this paper, we present the general problem of analyzing a shared system stack for resource constrained preemptive real-time systems. We provide a general and exact problem formulation applicable for preemptive systems

based on dynamic run-time properties. We also present an approximate stack analysis method to derive a safe upper bound on stack usage in static offset based, fixed priority and preemptive systems that use a shared stack. We evaluate and show that the proposed method gives significantly lower upper bounds on stack memory requirements than existing stack dimensioning methods for fixed priority systems.

Paper outline. Section 10.2 describes related work and sets the context for the contributions of this paper. In sections 10.3, 10.4, and 10.5 we present the exact formulation of determining the maximum stack usage and our safe approximation of the stack usage for our target system model. Section 10.6 presents an evaluation of our approximative analysis method, and Section 10.7 concludes the paper.

10.2 Related work

The notion of shared stack has been used in several publications to describe the ability to utilize either a common run-time stack or a pool of run-time stacks. For example, in [Mic], stack sharing is performed by having a pool of available stack areas. When a task starts executing, it fetches a stack from the pool, and returns it at termination. In [MSB05], Middha *et al.* address stack sharing in the sense that the stack of a task can grow into the stack area of another task.

In this paper, we use the notion of stack sharing when several tasks use one common, statically allocated, run-time stack. This type of stack sharing can be efficiently implemented in systems where tasks have run-to-completion semantics, and do not suspend themselves. This type of stack sharing is supported by several commercial real-time operating systems, e.g., [Arc, Gro, Uni].

10.2.1 Stack analysis

In [Bak90], Baker presents the Stack Resource Policy (SRP) that permits stack sharing among processes in static and in some dynamic priority preemptive systems. The basic method to determine the maximum amount of stack usage in SRP is to identify the maximum stack usage for tasks at each priority level (or preemption level) and then to sum up these maximums for each priority level. A safe upper bound (SPL) on the total stack usage using information about priority levels can formally be expressed as:

$$SPL = \sum_{l \in \textit{prio-levels}} \max_{i \in \textit{tasks with prio } l} (S_i) \tag{10.1}$$

where S_i is the maximum stack usage of task i.

Gai *et al.* [GLN01] present SRP with preemption thresholds (SRPT). They present a procedure to minimize shared stack usage, without jeopardizing schedulability, by use of non-preemption groups for tasks using SRPT. They extend the work of Saksena and Wang [SW00] by taking the stack usage of tasks into account when establishing non-preemption groups.

In [DMT00] Davis *et al.* address stack memory requirements by using non-preemption groups to reduce the amount of memory needed for a shared stack. They show that the number of preemption levels required for typical systems can be relatively small, while maintaining schedulability.

Although non-preemption groups can reduce the amount of RAM needed for a shared stack, the use of non-preemption groups affects a system by restricting the occurrences of preemptions, which can have a negative affect on schedulability. Also, the method we present in this paper can further reduce the system stack by performing our analysis after preemption groups have been assigned.

10.2.2 Preemption analysis

A large number of publications address preemption analysis for different reasons, see, e.g., [ARJ97, CRJ05, DF04, LLH+01, Reg02, RM06, SSE05]. For example, in [LLH+01] Lee *et al.* present a technique to bound cache-related preemption delays in fixed-priority preemptive systems. They account for task phasing and nested preemption patterns among tasks to establish an upper bound on the cache timing delay introduced by preemptions. Our work relates to theirs in the sense that we also investigate occurrences of nested preemption patterns. However, our objectives differ in that Lee *et al.* are mainly interested in timing delays caused by cache reloading and preemption patterns whereas we address shared memory requirements as an effect of nested preemption patterns.

In [DF04], Dobrin and Fohler present a method to reduce the number of preemptions in fixed priority based systems. They define three fundamental conditions that have to be satisfied in order for a preemption to occur. The same conditions form the basis of our upper bound method described in Section 10.5.

10.3 Stack analysis of preemptive systems

The primary purpose of an execution stack is to store local data which consists of variables and state registers, parameters to subroutines and return addresses. Real-time systems typically have a separate stack, statically allocated, for each task. However, under certain conditions, tasks can share stack to achieve a lower overall memory footprint of the system.

In this paper we consider systems where a subset of tasks use a common, statically allocated, run-time stack. For this to be possible, we assume that a task only uses the stack between the start time of an instance and the finishing time of that instance, i.e., no data remains on the stack from one instance of a task to the next. Furthermore, we require non-interleaving task execution [Bak90, DMT00]. If υ_j begins executing between the start and finish of υ_i, then υ_i is not allowed to resume execution until υ_j has finished. In practice, this is ensured by not allowing tasks to suspend themselves voluntarily, or to be suspended by blocking once they have started their execution. In practice this means that OS-primitives like `sleep()` and `wait_for_event()` cannot be used, and that any blocking on shared resources must be handled before execution start, e.g., with a semaphore protocol like immediate inheritance protocol [BW96].

We formally define the start and finishing time of a task instance υ_i, as follows:

st_i The absolute time when υ_i actually begins executing.

ft_i The absolute time when υ_i terminates its execution.

At any given point in time, the worst case total stack usage of the system equals the sum of the stack usage for each individual task instance. Thus, with $s_i(t)$ denoting the actual stack usage of υ_i at time t, the maximum stack usage of the system can be expressed as follows:

$$\max_{t \in \textit{time instant}} \sum_{\upsilon_i \in \textit{task instances}} s_i(t) \tag{10.2}$$

This corresponds to the amount of memory that must be statically allocated for the shared stack to ensure the absence of stack overflow errors. For some systems, e.g., non-preemptive, statically scheduled systems with simple task code, it might be possible to directly compute or estimate $s_i(t)$. In general, however, they are not directly computable before the system is executed.

We note that the total stack usage depends on three basic properties:

(i) the stack memory usage of each task instance

(ii) the possible preemptions that may occur between any two instances

(iii) the ways in which preemptions can be nested

Determining the stack memory usage of a single task instance requires knowledge of the possible control-flow paths within the task code [HF05]. In [BDP01] Brylow *et al.* present a static checker for interrupt driven software. The checker is able to calculate the stack size of assembler programs by producing a control-flow graph annotated with information about time, space, safety and liveness.

However, due to the difficulties in determining the exact stack usage at every point in time for a given task instance, shared-stack analysis methods often assume that whenever a task is preempted, it is preempted when it uses its maximum stack depth. We make the same assumption, and use S_i to denote the maximum stack usage for task instance υ_i. Thus, when υ_i and υ_j are instances of the same task, we have $S_i = S_j$. Bounds on maximum stack usage for a given task can be derived by abstract interpretation using tools such as AbsInt [Abs] and Bound-T [Tid].

In order to calculate the maximum stack usage of the full system, we need to account for all possible preemption patterns. Under the assumption of non-interleaving task execution, a task instance, υ_i, is preempted by another task instance, υ_j, if (and only if) the following holds:

$$st_i < st_j < ft_i \tag{10.3}$$

In particular, we are interested in chains of nested preemptions. We define a *preemption chain* to be a set $\{\upsilon_1, \upsilon_2, \ldots, \upsilon_k\}$ of task instances such that

$$st_1 < st_2 < \cdots < st_k < ft_k < ft_{k-1} < \cdots < ft_1 \tag{10.4}$$

Under the assumption that the worst case stack usage of a task occur when the task is preempted, the worst case stack usage SWC for a shared stack preemptive system can be expressed as follows:

$$SWC = \max_{PC \in \textit{preemption chains}} \sum_{\upsilon_i \in PC} S_i \tag{10.5}$$

This formulation, however, cannot be directly used for analyzing and dimensioning the shared system stack since it is based on the dynamic (only available at run-time) properties st_i and ft_i. To be able to statically analyze the system, one has to relate the static (compile-time) properties to these dynamic

properties. This is done by establishing how the system model, scheduling policy, and run-time mechanism constrain the values of the actual start and finishing times. The problem can be viewed as a scheduling problem with the objective of maximizing the total stack usage of the schedule, subject to system constraints on how tasks are ordered in the schedule.

10.4 System model for hybrid scheduled systems

The system model we adopt is based on the commercial operating system Rubus OS by Arcticus Systems AB [Arc], which supports the execution of both time triggered and event triggered tasks. The Rubus OS is mainly intended for, and used in dependable resource-constrained embedded real-time systems.

The system model is a hybrid, static and dynamic, scheduled system where a subset of the tasks are dispatched by a static cyclic scheduler (time triggered tasks). The rest of the tasks are dispatched by events in the system (event triggered tasks). The static schedule is constructed off-line and a fixed priority scheduler (FPS) dispatches tasks at run-time. The event-triggered tasks can be categorized in two different classes: (i) event-triggered interrupts which have higher priority than the time-triggered tasks, and (ii) background scheduled event-triggered tasks which have lower priority than the time-triggered tasks.

The time triggered tasks share a common system stack. It is the objective of this paper to analyze, and ultimately dimension this shared system stack efficiently. The time-triggered subsystem is used to host safety critical applications. Hence, to isolate it from any erroneous event-triggered tasks, it uses its own stack.

10.4.1 Formal system model

The system model used in this paper can be seen as an offset based model with static offsets [GH98, MTHN05, MTN05, Tin92], defined as follows: The system, Γ, consists of a set of k transactions $\Gamma_1, \ldots, \Gamma_k$. Each transaction Γ_i is activated by a periodic sequence of events with period T_i. For non-periodic, events T_i denotes the minimum inter-arrival time between two consecutive events. The activating events are mutually independent, i.e., phasing between them is arbitrary.

A transaction, Γ_i, contains $|\Gamma_i|$ tasks, and each task may not be activated (released for execution) until a time, offset, elapses after the arrival of the acti-

vating event.

We use τ_{ij} to denote a task. The first subscript denotes which transaction the task belongs to, and the second subscript denotes the number of the task within the transaction. A task, τ_{ij}, is defined by a worst case execution time (C_{ij}), an offset (O_{ij}), a deadline (D_{ij}), maximum jitter (J_{ij}), maximum blocking from lower priority tasks (B_{ij}), and a priority (P_{ij}). Furthermore, S_{ij} is used to denote the maximum stack usage of τ_{ij}. The system model is formally expressed as:

$$\begin{aligned}\Gamma :=& \{\langle \Gamma_1, T_1\rangle, \ldots, \langle \Gamma_k, T_k\rangle\} \\ \Gamma_i :=& \{\tau_{i1}, \ldots, \tau_{i|\Gamma_i|}\} \\ \tau_{ij} :=& \langle C_{ij}, O_{ij}, D_{ij}, J_{ij}, B_{ij}, P_{ij}, S_{ij}\rangle\end{aligned}$$

We assume that the system is schedulable and that the worst case response-time for each task, (R_{ij}), has been calculated [MTN05].

Due to the non-interleaving criterion for stack sharing, we require that tasks exhibit run-to-completion semantics when activated, i.e., they cannot suspend themselves. An invocation of a task can be viewed as a function call from the operating system, and the invocation terminates when the function call returns. When tasks share the same priority, they are served on a first-come first-served basis.

We assume that if access to shared resources is not handled by the static scheduler by time separation, a resource sharing protocol where blocking is done before start of execution is employed (such as the stack resource protocol [Bak90] or the immediate inheritance protocol [BW96]).

Relating back to Rubus OS, one can view the system as a transaction based system with one transaction, Γ_t, corresponding to the static schedule (time-triggered tasks) and any number of transactions corresponding to higher priority event triggered tasks (interrupts). For the even-triggered transactions there are no restrictions placed on offset, deadline or jitter, i.e., they can each be either smaller or greater than the period. Since Γ_t represents the static schedule, which is cyclical with period T_t, offset, jitter and deadline are less than the period, i.e., $O_{tj}, D_{tj}, J_{tj} \leq T_t$ for the time-triggered transaction. How a scheduler can generate a feasible schedule with interfering interrupts is described in [SEF98, MTHN05].

It is the objective of this paper to find a tight upper bound on the shared system stack for the tasks in the time-triggered transaction Γ_t. Task j belonging to Γ_t we will denote τ_{tj}. The tasks in the transaction can be preempted by other tasks in the transaction and by higher priority event triggered tasks.

10.5 Stack analysis of hybrid scheduled systems

In this section, we describe a polynomial time method to establish a safe upper bound on the shared stack usage for the system model described in Section 10.4. The upper bound is safe in the sense that the run-time stack can never exceed the calculated upper bound.

A safe upper-bound estimate of the exact problem can be found by using offsets and maximum response times as approximations of actual start and finishing times. Generalizing the preemption criteria identified by Dobrin and Fohler [DF04], we form the binary relation $\tau_{ti} \prec \tau_{tj}$ with the interpretation that τ_{ti} may be preempted by τ_{tj}. The relation holds whenever (1) τ_{ti} can become ready before τ_{tj}, (2) τ_{ti} possibly finishes (i.e., has a response time) after the start of τ_{tj}, and (3) τ_{ti} has lower priority than τ_{tj}. The relation can now formally be defined as:

$$\tau_{ti} \prec \tau_{tj} \equiv O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti} \wedge P_{ti} < P_{tj} \tag{10.6}$$

Lemma 1. *The $\prec$ relation is a safe approximation of the possible preemptions between tasks in Γ_t. That is, if τ_{ti} can under any run-time circumstance be preempted by τ_{tj}, then $\tau_{ti} \prec \tau_{tj}$ will hold.*

Proof of Lemma 1. *Suppose that τ_{ti} is preempted by τ_{tj}. We show that this implies (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, (2) $O_{tj} < R_{ti}$, and (3) $P_{ti} < P_{tj}$.*

(3) follows directly from the preemption. Now let t be the time instant when τ_{tj} has finished blocking, which implies $t \leq O_{tj} + J_{tj} + B_{tj}$. Then a possibly empty interval $[t, st_{tj}]$ of execution with higher priority than τ_{tj} follows, in which τ_{ti} cannot execute because $P_{ti} < P_{tj}$. Since τ_{ti} must start before τ_{tj}, we can conclude that $st_{ti} < t$, which together with $O_{ti} \leq st_{ti}$ and $t \leq O_{tj} + J_{tj} + B_{tj}$ gives us $O_{ti} < O_{tj} + J_{tj} + B_{tj}$ and (1). From Equation 10.3 we have $st_{tj} < ft_{ti}$ and this together with $O_{tj} \leq st_{tj}$ and $ft_{ti} \leq R_{ti}$ leads to $O_{tj} < R_{ti}$ and (2), which completes the proof. □

The upper-bound problem can now be informally stated as finding the maximum stack usage of all possible preemption chains in Γ_t. This equals finding the time instant in the schedule which has a maximum stack usage, given the approximation of actual start and finishing times with offsets and response times respectively, and assuming that at all preemptions the preempted task uses its maximal stack.

A sequence Q of tasks is a *possible preemption chain* (PPC) if it holds that $\tau_{ti} \prec \tau_{tj}$ for all τ_{ti}, τ_{tj} in Q where τ_{ti} occurs before τ_{tj} in the sequence. The

stack usage SU_Q of a PPC Q is the sum of the stack usage of the individual tasks in the chain, i.e., $SU_Q = \sum_{\tau_{ti} \in Q} S_{ti}$.

A straightforward computation of a safe upper bound for a set of tasks involves computing the stack usage for all PPCs. However, for a set of n tasks there exist $2^n - 1$ different PPCs in the worst case, which yields an exponential time complexity for an algorithm based on this idea. A more efficient algorithm can be constructed by first finding sets of tasks which all overlap in time without regarding priorities. These sets can then be investigated, in turn, to find a PPC with maximal stack usage.

We let the relation $\tau_{ti} \preceq \tau_{tj}$ hold whenever the semiclosed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$ intersect, or more formally:

$$\tau_{ti} \preceq \tau_{tj} \equiv O_{ti} < R_{tj} \wedge O_{tj} < R_{ti} \tag{10.7}$$

The relation $\preceq$ is a relaxation of the $\prec$ relation. That is, $\tau_{ti} \prec \tau_{tj} \rightarrow \tau_{ti} \preceq \tau_{tj}$. To see this, suppose that $\tau_{ti} \prec \tau_{tj}$ which implies $O_{ti} < O_{tj} + J_{tj} + B_{tj} \wedge O_{tj} < R_{ti}$, according to Equation 10.6. Since $O_{tj} + J_{tj} + B_{tj} \leq R_{tj}$ follows from the notion of response time, we have $O_{ti} < R_{tj} \wedge O_{tj} < R_{ti}$, which also is the definition of $\tau_{ti} \preceq \tau_{tj}$.

We can now define an *overlap set* K_r as a set of tasks where:

$$\forall \tau_{ti}, \tau_{tj} \in K_r : \tau_{ti} \preceq \tau_{tj}$$

The stack usage SU_{K_r} of an overlap set K_r is defined as the maximum stack usage SU_Q of all PPCs Q where $Q \subseteq K_r$:

$$SU_{K_r} = \max_{\forall Q \subseteq K_r : PPC(Q)} (SU_Q) \tag{10.8}$$

K_r is *maximal*, if and only if, there exists no overlap set, K_s, such that $K_r \subset K_s$.

Lemma 2. *For any PPC Q, there exists a maximal overlap set K_r such that $Q \subseteq K_r$.*

Proof of Lemma 2. *From the definitions of a PPC and the $\prec$ and $\preceq$ relations, we know that for all tasks $\tau_{ti} \prec \tau_{tj}$ in Q it also holds that $\tau_{ti} \preceq \tau_{tj}$, and thus Q is an overlap set. Then, either Q is maximal, or it can become maximal by extending it with additional tasks. In either case, the lemma holds.* □

In all, the algorithm for computing the upper bound SUB on the maximum stack usage for a set of tasks Γ_t can be summarized as follows:

1. Find the maximal overlap sets in Γ_t:
   $K = \{K_1, K_2, \ldots, K_k\}$.
2. For each of them, compute SU_{K_r} according to Equation 10.8.
3. The upper bound of the stack usage for Γ_t can now be computed as follows:
$$SUB = \max_{\forall K_r \in K} (SU_{K_r}) \tag{10.9}$$

Informally, we start by finding all sets of tasks that can overlap in time based on their offsets and worst case response times, which safely approximates their actual start and finishing times. For each such set (K_i), we find all possible preemption chains (PPCs) by also taking task priorities and maximal jitter and blocking time into account, and compute the stack usage for each chain. The stack usage of K_i is the maximum stack usage of all its PPCs, and the maximum stack usage (SUB) of the system is then obtained by taking the maximum stack usage of every K_i.

10.5.1 Correctness

In order to claim correctness of our approximate stack analysis method, we have to show that it never underestimates the actual stack usage that can occur during run-time.

Theorem 1. *The value computed by the SUB algorithm is a safe upper bound on the actual worst case stack usage for tasks in Γ_t. Formally, $SWC \leq SUB$.*

Proof. Let $\Psi \subseteq \Gamma_t$ be the sequence of tasks instances participating in the preemption situation which cause the worst case stack usage, that is, $SWC = \sum_{\tau_{ti} \in \Psi} S_{ti}$. According to Lemma 1, we must have $\tau_{ti} \prec \tau_{tj}$ for tasks τ_{ti} and τ_{tj} that occur in that order in Ψ, and thus Ψ is a PPC with $SU_\Psi = SWC$. Then, Lemma 2 ensures that there exists a maximal overlap set K_r such that $\Psi \subseteq K_r$, and we have $SU_\Psi \leq SU_{K_r}$. Thus, $SWC \leq SU_{K_r} \leq SUB$, which concludes the proof. □

10.5.2 Computational complexity

The relaxation of $\prec$ into interval intersection (Equation 10.7) allows us to efficiently compute an upper bound on the stack usage (Equation 10.9) by applying a polynomial longest path algorithm on the linearly-bounded number of maximal overlap sets.

To first see that the set of maximal overlap sets $K = \{K_1, K_2, \ldots, K_k\}$ contain at most n elements, i.e., $k \leq n$, consider the graph (Γ_t, E), where Γ_t is the set of vertices and $E = \{\tau_{ti}\tau_{tj} \mid (\tau_{ti} \preceq \tau_{tj}) \wedge \tau_{ti}, \tau_{tj} \in \Gamma_t\}$ is the set of edges. From Equation 10.7, we have that edges $\tau_{ti}\tau_{tj} \in E$ correspond to intersection of the semi-closed intervals $[O_{ti}, R_{ti})$ and $[O_{tj}, R_{tj})$, and therefore the graph is an *interval graph* [MM99]. Because every interval graph is also *chordal* [MM99], all maximal complete subgraphs in (Γ_t, E), which correspond to all maximal overlap sets, can be found in linear time [RT75]. Furthermore, for chordal graphs there exists at most n such sets, and thus we have at most n overlap sets [MM99].

The problem of finding the worst PPC within a single overlap set K_r is significantly easier than for an arbitrary set of tasks. Since it holds that $\tau_{ti} \preceq \tau_{tj}$ for all tasks $\tau_{ti}, \tau_{tj} \in K_r$, and therefore in particular that $O_{ti} < R_{tj}$ for all tasks in K_r, we need only look for a maximum stack usage chain Q where (1) $O_{ti} < O_{tj} + J_{tj} + B_{tj}$, and (2) $P_{ti} < P_{tj}$ for all tasks τ_{ti} and τ_{tj} in that order in Q to find the worst PPC. A directed graph consisting of tasks in K_r and arcs corresponding to properties (1) and (2) is acyclic, and for such graphs a longest-path type algorithm can be used to find the worst PPC [CLRS01]. There exist longest-path algorithms with a time complexity of $O(n + m)$, where n is the number of tasks and m is the number of possible preemptions, of which there are at most $n(n-1)/2$. Taking the maximum of a maximal PPC in each set, K_r, of which there are at most n, we will, therefore, find a maximum stack size PPC in at most $O(n^3)$ time.

10.6 Evaluation

We evaluate the efficiency of our proposed method to establish a safe upper bound on shared stack usage by randomly generating realistic sized task sets. The size, load and stack usage of the task sets are derived from a wheel-loader application by Volvo Construction Equipment [VCE]. We use three different methods to calculate the shared system stack usage:

SPL The traditional method to dimension a shared system stack by summing up the maximum stack usage in each priority level.

SUB The safe upper bound on the shared stack usage presented in Section 10.5

SLB A lower bound on on the shared stack usage, for each task set.

The lower bound is established using simple heuristics that tries to maximize shared stack usage by generating only feasible preemption scenarios for the task set, and thus, represents scenarios that definitely can occur. From all PPCs, the heuristic selects a sample set of roughly 500 chains. For each of them, the method tries to construct a feasible arrival pattern for the ET tasks and actual execution time values that cause an actual preemption between the tasks in the chain. The quality of this heuristic method degrades as the length of the chains or the total number of PPCs increases, which can be seen in the figures.

By establishing a safe upper bound and a feasible lower bound, we know that the actual worst case stack usage is bounded by SUB and SLB. The reason for including SLB is to give an indication on the maximum amount of improvement there might be for SUB.

10.6.1 Simulation setup

In our simulator we generate random task sets as input to the stack analysis application. The task generator takes the following input parameters:

- Total number of TT (time triggered) tasks (default = 250)
- Total load of TT tasks (default = 60%)
- Minimum and maximum priorities of TT tasks (default = 1 and 32)
- Minimum and maximum stack usage of TT tasks (default = 128 and 2048)
- Total number of ET (event triggered) tasks (default = 8)
- Total load of ET tasks (default = 20%)
- The shortest possible minimum inter-arrival time of an ET task (default = 1.000)

The generated schedule for TT tasks is always 10.000 time units. All ET tasks have higher priority than TT tasks. The default values for the input parameters represent a base configuration derived from a real application [VCE].

Using these parameters a task set with the following characteristics is generated:

- Each TT offset (O_{ti}) is randomly and uniformly distributed between 0 and 10.000.

- Worst case execution times for TT tasks, C_{ti}, are initially randomly assigned between 1 and 1000 time units. The execution times get adjusted by multiplying all C_{ti} by a fraction, so that the the TT load (as defined by the input parameter) is obtained.
- TT priorities are assigned randomly between minimum and maximum value with a uniform distribution.

10.6.2 Results

Each diagram shows three graphs corresponding to the stack usage calculated by the three methods: SPL, SUB, and SLB. Each point in the graphs represents the mean value of 100 generated task sets. We also measured the 95% confidence interval for the mean values. These are not shown because of their small size (less than 7% of the y-value for each point). We also measured the CPU time to calculate an upper bound on shared stack usage for each generated task set. Using the method described in Section 10.5, the calculations took less than 63ms per task set, on an Intel Pentium 4, 2.8GHz with 512MB of RAM.

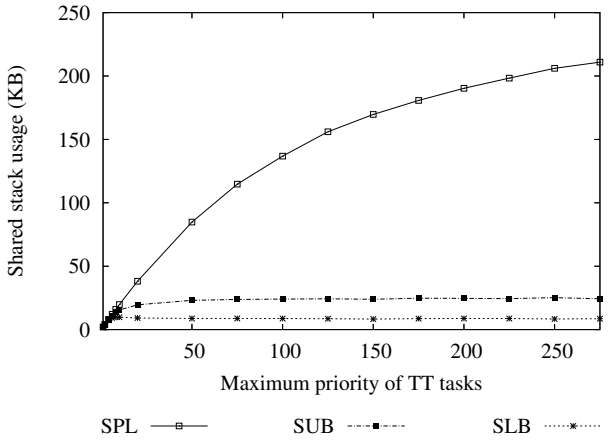

Figure 10.1: Varying the number of priority levels of TT tasks

In Fig. 10.1, we vary the maximum priority for TT tasks between 1 and 300, keeping the minimum priority at 1. This gives a distribution of possible priorities (priority levels), from 1 to n, where n is indicated by the x-axis. We

see, in Fig. 10.2 which zooms in on Fig. 10.1, for maximum priorities up to 10, that the difference in stack usage between SPL and SUB is less noticeable with a low number of priority levels. However, for larger numbers of priority levels the difference is significant. SPL is not expected to flatten out before all tasks actually have unique priorities, whereas our method (SUB) flattens out significantly earlier. We conclude that the maximum number of tasks in any preemption chain is increasing very slowly (or not at all) when the number of TT tasks increases above a certain value, since the system load is constant.

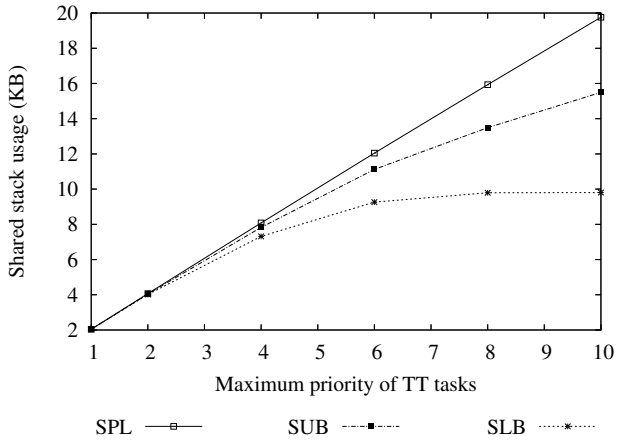

Figure 10.2: Varying the number of priority levels of TT tasks (zoom of Fig. 10.1)

In Fig. 10.3, we vary the maximum stack usage of each TT task between 128 bytes and 4096 bytes. We do this by assigning an initial stack of 128 bytes for each TT task, i.e., initially the stack size variation is zero. We then vary the stack size between 128 and 512 bytes, 128 and 1024 bytes, and so on. The diagram shows that SUB gives significantly lower values on shared stack usage than the traditional SPL. We also notice that an increase in stack variation scales up, linearly, the differences between SPL and SUB. The linearity is expected, since an increase in stack variation does not affect occurrences of possible preemptions in the system, i.e., possible nested preemptions are retained.

In Fig. 10.4 we vary the maximum number of TT tasks between 5 and 275. We see that the shared stack usage of the traditional SPL is dramatically in-

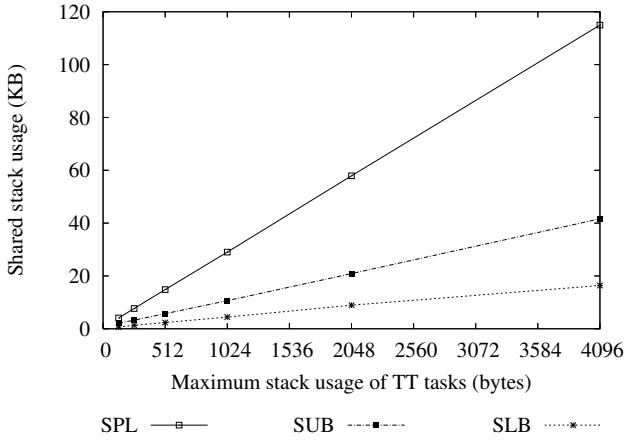

Figure 10.3: Varying stack usage of TT tasks

creasing in the beginning. This is due to the fact that when the number of TT tasks is lower than the maximum priority of TT tasks (32), most TT tasks have unique priorities. SUB, on the other hand, increases much slower than SPL because the maximum number of tasks involved in any preemption chain is slowly increasing. SUB is expected to further approach SPL since increasing the number of tasks will increase the likelihood of larger number of tasks involved in the preemption chains.

In Fig. 10.5, we vary the total load of TT tasks between 10% (0.1) and 70% (0.7). The figure shows that the shared stack usage of SPL is constant, whereas, SUB is slowly increasing. SPL is expected to be constant since it is only affected by the number of priority levels and unaffected by the actual preemptions that can occur in a system. The increase of SUB is due to increasing response-times of TT tasks when the TT load increases, which will increase the likelihood of larger number of tasks involved in nested preemptions.

10.7 Conclusions and future work

This paper presents a novel method to determine the maximum stack memory used in preemptive, shared stack, real-time systems. We provide a general and exact problem formulation applicable for any preemptive system model based

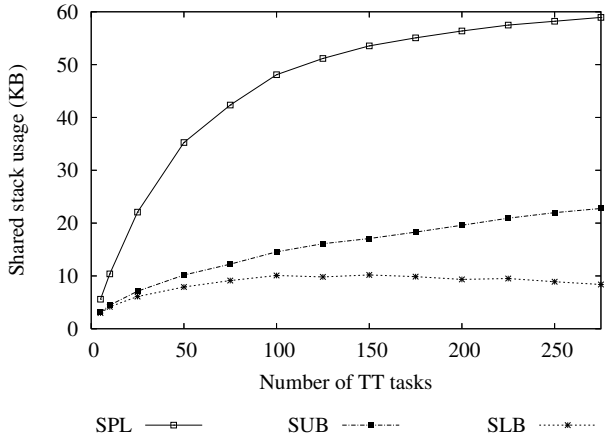

Figure 10.4: Varying the number of TT tasks

on dynamic (run-time) properties.

By approximating these run-time properties, together with information about the underlying run-time system, we present a method to safely approximate the maximum system stack usage at compile time. We do this for a relevant and commercially available system model: A hybrid, statically and dynamically, scheduled system. Such a system model provides lot of static information that we can use to estimate the dynamic start- and finishing-times. Our method finds the nested preemption pattern that results in the maximum shared stack usage. We prove that our method is a safe upper bound of the exact system stack usage and show that our method has a polynomial time complexity.

In a comprehensive simulation study, we evaluated our technique and compared it to the traditional method to estimate stack usage. We find that our method significantly reduces the amount of stack memory needed. For realistically sized task sets, a decrease in the order of 70% is typical.

In this paper, we focused on a system model for a given commercial real-time operating system. In the future, we plan to extend our approximation method to a more general system model, to incorporate all the features of the general model for tasks with offsets [GH98]. Such an extension would make the presented analysis technique applicable to a wider range of systems.

Our current method could also be extended to account for other types of

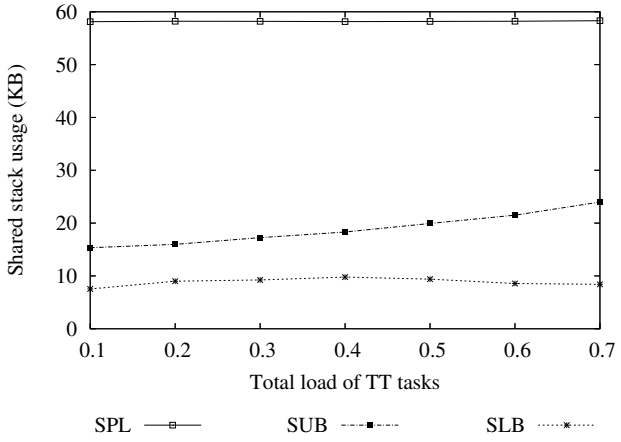

Figure 10.5: Varying the load of TT tasks

information that can further limit the number of possible preemptions. We currently only account for separation in time (offsets and response-times) between tasks. However, in many systems other types of information, such as precedence and mutual-exclusion relations may exists between tasks, thus limiting the possible preemptions.

The method presented here could also be used in synthesis and configuration tools that generate optimized systems from given application constraint. In this case, the results from our analysis can be used to guide optimization or heuristic techniques that try to map application functionality to run-time objects.

Bibliography

[Abs] Absint. Web page, http://www.absint.com/stackanalyzer/.

[Arc] Arcticus Systems. Web page, http://www.arcticus-systems.se.

[ARJ97] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-Time Computing with Lock-Free Shared Objects. *ACM Transactions on Computing Systems*, 15(2):134–165, May 1997.

[BAE] Bae systems hägglunds. Web page, http://www.haggve.se.

[Bak90] T.P. Baker. A Stack Based Resource Allocation Policy for Real-Time Processes. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*. IEEE, 1990.

[BDP01] D. Brylow, N. Damgaard, and J. Palsberg. Static Checking of Interrupt-Driven Software. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[BW96] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*, chapter 13.10.1 Immediate Ceiling Priority Inheritance. Addison-Wesley, second edition, 1996.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, second edition, 2001. ISBN 0-262-03293-7.

[CRJ05] H. Cho, B. Ravindran, and E. D. Jensen. A Space-Optimal Wait-Free Real-Time Synchronization Protocol. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, July 2005.

[DF04] R. Dobrin and G. Fohler. Reducing the Number of Preemptions in Fixed Priority Scheduling. In *16th Euromicro Conference on Real-time Systems*. Catania, Sicily, Italy, July 2004.

[Dig] Micro Digital. Web page, http://www.smxinfo.com/mt.htm.

[DMT00] R. Davis, N. Merriam, and N. Tracey. How Embedded Applications using an RTOS can stay within On-chip Memory Limits. In *Proceedings of the WiP and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[GH98] J. C. Palencia Gutierrez and M. Gonzalez Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the 19th Real-Time Systems Symposium*, Dec 1998.

[GLN01] P. Gai, G. Lipari, and M. Di Natale. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip. In *Proceedings of the 22nd Real-Time Systems Symposium*. London, UK, Dec 2001.

[Gro] Live Devices ETAS Group. Web page, http://en.etasgroup.com/products/rta/.

[Hal] Haldex traction systems. Web page, http://www.haldex-traction.-com/.

[HF05] R. Heckmann and C. Ferdinand. Verifying Safety-Critical Timing and Memory-Usage Properties of Embedded Software by Abstract Interpretation. In *Proceedings of the Design, Automation and Test in Europe*, March 2005.

[HLL+05] K. Hänninen, J. Lundbäck, K.-L. Lundbäck, J. Mäki-Turja, and M. Nolin. Efficient Event-Triggered Tasks in an RTOS. In *Proceedings of the International Conference on Embedded Systems and Applications*, June 2005.

[LLH+01] C. G. Lee, K. Lee, J. Hahn, Y. M. Seo, S. Lyul Min, R. Ha, S. Hong, C. Yun Park, M. Lee, and C. Sang Kim. Bounding Cache-Related Preemption Delay for Real-Time Systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, Sept 2001.

[Mic] Micro Digital Inc. *smx Features and Architecture*.

[MM99] T. A. McKee and F.R. McMorris. *Topics in intersection graph theory*. SIAM Monographs on Discrete Mathematics and Applications #2. Society for Industrial and Applied Mathematics (SIAM), 1999.

[MSB05] B. Middha, M. Simpson, and R. Barua. MTSS: Multi Task Stack Sharing for Embedded Systems. In *Proceedings of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. San Francisco, CA, Sept 2005.

[MTHN05] J. Mäki-Turja, K. Hänninen, and M. Nolin. Efficient Development of Real-Time Systems Using Hybrid Scheduling. In *International conference on Embedded Systems and Applications (ESA)*, June 2005.

[MTN05] J. Mäki-Turja and M. Nolin. Fast and Tight Response-Times for Tasks with Offsets. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*. IEEE Computer Society, July 2005.

[Reg02] J. Regehr. Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, Dec 2002.

[RM06] H. Ramaprasad and F. Mueller. Bounding Preemption Delay within Data Cache Reference Patterns for Real-Time Tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2006.

[RT75] D. J. Rose and R. E. Tarjan. Algorithmic Aspects of Vertex Elimination. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 245–254. ACM Press, New York, NY, USA, 1975.

[SEF98] Kristian Sandström, Christer Eriksson, and Gerhard Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In *5th International Workshop on Real-Time Computing Systems and Applications (RTCSA '98)*, pages 158–165. IEEE Computer Society, Hiroshima, Japan, October 1998.

[SSE05] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling Analysis of real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, July 2005.

[SW00] M. Saksena and Y. Wang. Scalable Real-Time System Design using Preemption Thresholds. In *Proceedings of the 21st Real-Time System Symposium*, Dec 2000.

[Tid] Tidorum. Web page, http://www.tidorum.fi/bound-t/.

[Tin92] K. Tindell. Using Offset Information to Analyse Static Priority Pre-emptively Scheduled Task Sets. Technical Report YCS-182, Dept. of Computer Science, University of York, England, 1992.

[Uni] Unicoi Systems. Web page, http://www.unicoi.com/fusion_rtos/fusion_rtos.htm.

[VCE] Volvo construction equipment. Web page, http://www.volvoce.com.

Chapter 11

Paper F: Introducing a Plug-In Framework for Real-Time Analysis in Rubus-ICE

Kaj Hänninen, Jukka Mäki-Turja, Staffan Sandberg, John Lundbäck, Mats Lindberg, Mikael Nolin, Kurt-Lennart Lundbäck
MRTC report ISSN 1404-3041 ISRN MDH-MRTC-229/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2008. Submitted to the 13th IEEE International Conference on Emerging Technologies and Factory Automation.

Abstract

In this paper, we present the development of a plug-in framework for integration of real-time analysis methods in the Rubus Integrated Component Environment (Rubus-ICE). We also present the implementation, and evaluate the integration, of two state of the art analysis techniques (i) response-time analysis for tasks with offsets and (ii) shared stack analysis, as plug-ins, in the Rubus-ICE framework.

The paper shows that the proposed framework is well suited for integration of complex analysis methods. However, experience also show that analysis methods are not easily transferred from an academic environment to industry. The main reason for this, we believe, originates from differences in requirements and assumptions between industry and academia.

11.1 Introduction

Throughout the years, research on analysis and scheduling has been a significant area within the real-time community. A large number of analysis techniques have been proposed for verification of real-time properties in real-time systems (see for example [ABT+93, BTW95, EHS97, JP86, KAS93, Leh90, Pun97, SAr+04, SRL87, SRL90, TB94, TC94, THW94, Tin92]). However, many of these techniques are state of the art and non-trivial to understand and even more complex to integrate in an industrial development context. In industrial development, a number of tools are used for design, implementation, analysis and verification. These tools are often manufactured by different vendors. The challenge then becomes to integrate state of the art analysis techniques in an existing tool-suite with tools from different vendors. These difficulties are often hard to overcome; hence many useful analysis techniques never find their way to practical use.

In recent years, plug-in based tools, e.g., Eclipse [Ecl] and JDeveloper [Ora] etc. have gained popularity. The plug-in concept has several properties that eases the integration of research results in a development environment, for example, (i) allowing the extension of the functionality of a host application by add-on applications (ii) allowing development of add-on plug-ins in isolation, meaning that developers do not need to compile the source code of the plug in with the source code of its host application.

In this paper, we describe the development of a plug-in framework for integration of real-time analysis methods in the Rubus Integrated Component Environment (Rubus-ICE)[Arc]. We present the Rubus-ICE environment, an IDE targeted for component based development of real-time systems. We then describe the implementation of two novel analysis methods as plug-ins and highlight issues in integration of the plug-ins in a case study.

The contributions of this paper include a proof of concept implementation where state-of-the-art academic results can be deployed in an existing commercial industrial environment. We also report on experiences from transferring academic result to industrial environments and the issues that needs to be dealt with in order to successfully achieve such transfer.

Paper outline. The reminder of this paper is organised as follows. In Section 11.2 we present the Rubus development environment. Section 11.3 presents the development of the plug-in framework for Rubus-ICE. In Section 11.4 we presents the development of two novel analysis methods as plug-ins, for integration in Rubus-ICE. Section Section 11.5 presents a brief case study on the integration of the analysis plug-ins in the proposed framework.

Section Section 11.6 summarises our experiences in developing the framework and introducing novel research results for industrial use. In Section 11.7 we conclude the work and outline some future work.

11.2 Rubus

Rubus is a collection of tools for development of embedded real-time systems. Rubus was introduced for industrial use in 1996. Throughout the years, Rubus has been used by a number of companies, e.g., [BAE, Hal, Kno, VCE] for development of safety critical as well as less critical vehicular software. Although successfully used by real-time developers, the tools in the Rubus framework have by tradition been tightly coupled with each other, making it difficult to integrate additional analysis methods in the framework. Over the years, the tools-suite has evolved to support novel requirements in development. The current version of the tool-suite, Rubus-ICE, is aimed to be plug-in based to facilitate integration of third party applications, such as real-time analysis methods, in the framework.

11.2.1 Rubus-ICE

Rubus-ICE is an IDE consisting of set of tools for systems engineering, design and analysis of component-based real-time systems. The four core elements of Rubus-ICE are as follows:

- Designer: A graphical design tool for component based modelling of systems.
- Compiler: A tool that verifies syntax of the model data created with the designer.
- Builder: A tool that passes design-data in sequence to any number of user specified plug-in modules.
- Coder: A tool that generates the RTOS specific requirements defined by the user.

To exemplify the steps involved in using Rubus-ICE, assume that a developer initially creates a component-based design using Designer. The Designer saves the design in XML format. The compiler then parses the design representation and verifies the syntax of the design. The compiler creates an intermediate representation (ICCM file) of the design. The ICCM file is then used

by the builder. Figure 11.1 shows the sequence in which the core elements of Rubus-ICE are executed.

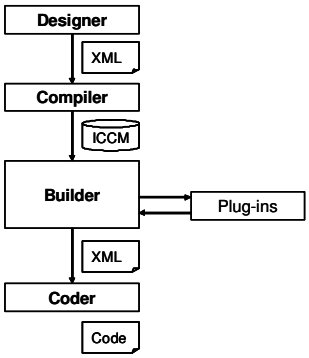

Figure 11.1: Rubus-ICE with a plug-in framework

11.3 Plug-in framework for Rubus-ICE

In this section we describe the development of a plug-in framework for Rubus-ICE. The aim of the framework is to enable integration of novel real-time analysis methods, as plug-ins, in the IDE. Facilitating integration of third-party developed analysis methods is of specific interest. Hence, the framework should allow a plug-in developer to implement plug-ins in isolation, i.e., without having the Rubus tool suite at hand, and deliver the plug-ins as binaries or source code. In the following, we describe the overall requirements on the plug-in framework. We then outline the requirements that need to be fulfilled for plug-ins to be included in the framework. We also present the development of an application programming interface (API) for development of Rubus-ICE plug-ins.

11.3.1 Requirements on the plug-in framework

We start by stating the high level requirements on the plug-in framework. From a developer and a user perspective the following requirements should be fulfilled by the framework:

- A plug-in should be developed as stand alone applications performing a specific task.

- A user should be able to choose, by configuring a build, to execute any available plug-in during the build. Hence, the plug-ins should interface the Builder tool in Rubus-ICE.

- Plug-ins should execute in sequence, meaning that a plug-in should execute to completion and terminate before the next plug-in is executed.

- A user should be able view the progress of a plug-in and to abort the execution of a plug-in if needed.

Figure 11.1 shows the sequence in which the core elements of Rubus-ICE, including plug-ins, are to be executed. The requirements on the framework are motivated by the following. For example, a user might be interested in analysing only temporal aspects of a design, during certain phases in development. Later on, the user might be interested in analysing both temporal and spatial aspects. Hence it should be possible to enable and disable the execution of plug-ins between the builds. The requirement that each plug-in should perform one specific task is required to prevent several plug-ins to perform similar, possibly time consuming, analysis during build. For example, if several analysis algorithms require response times to be calculated as input to the algorithms, then the response-time analysis should be developed as a separate plug-in that is executed only once. The fact that plug-ins are required to execute in sequence facilitate the possibility of several plug-ins to collaborate and solve a larger task. Moreover, in the research community, several analysis methods are proposed as extensions of previously published methods. Requiring the plug-ins to execute in a sequence eases the integration of the extensions in the IDE. User interaction and the possibility to abort the execution of a plug-in is motivated by the fact that the timing complexity of an analysis method may increase dramatically if a system is changed between two builds. For example, analysis methods with exponential complexity may actually perform well, i.e., deliver results in reasonably amount of time, for a certain system. However, for such algorithms, even small changes in system parameters may dramatically increase the time to obtain results. In these type of situations, the analysis may be unusable and aborted by the user. In addition, providing feedback to users when a plug-in fails or when the results of a plug-in differs from what is expected, is important.

11.3.2 Requirements on Rubus-ICE plug-ins

Many analysis methods developed in a research context assume that certain assumptions are fulfilled for the analysis to be valid. For example, in the research community, an analysis algorithm is generally developed for a specific system model, i.e., the results of the analysis is only valid if the correct system model is used. Thus, for each plug-in, the supported system model, i.e., properties and attributes of the supported system, must be specified.

To simplify verification of plug-ins, we require plug-ins to adhere to the following execution sequence: (i) reading required system attributes, (ii) executing the functionality of the plug in and (iii) writing results to the ICCM file. Hence, each plug-in should have a required interface, an internal behaviour and a provided interface. Accessing system attributes should be done by service request provided by an Application Programming Interface.

Each plug-in must have its error handling routines specified. This includes specifying (i) the type of error that the plug-in handles, and (ii) how these error are handled. In addition, in an industrial context, interaction with the user is imperative. Hence, each plug-in needs to define an interface against the user. This interface should provide, e.g., information of the progress of the plug-ins.

The fact that a user should be able to choose, by configuring a build, to load and execute any available plug-in during the build implies that plug-in can be delivered as Dynamic Link Libraries (DLLs) or as source code (C or C++ code).

In essence, for each Rubus-ICE plug-in, the following should specified: (i) the system model supported by the plug-in, (ii) the type of data required by the plug-in, (iii) the type of data produced by the plug-in, (iv) the type of errors handled by the plug-in and (v) user interface.

11.3.3 Defining an API for plug-ins

To support implementation of plug-ins, we defined an API (Application Programming Interface). The API specifies and provides a uniform way to access services that may be needed by plug-ins. Currently, the API supports common services for the system model defined by the Rubus Component Model (RubusCM) [HMTN+08]. In defining the API, we considered common assumptions made by researcher developing analysis algorithms. We also considered common attributes and properties that need to be available for analysis algorithms. For instance, the API provide possibilities to extract transactions, tasks, task attributes, task dependencies, execution policies, execution sched-

ules, memory-models and so on, from the design. In addition, the API provide services to append the results, produced by a plug-in, in the ICCM file. These results may then be used by subsequent plug-ins.

11.4 Developing analysis plug-ins

In this section, we describe the development and integration of two analysis plug-ins for Rubus-ICE. The plug-ins are intended to be stand alone applications computing: (i) the worst case response-time (RTA)[MTN04] of tasks and (ii) the maximum shared stack usage (SSA)[HMTB+06] of the system. Both the RTA and SSA algorithms have originally been implemented for research purposes, e.g., for evaluating the efficiency of the RTA and SSA algorithms. In a research context, the algorithms have been part of a larger application consisting of a task generator, a package for statistics an a graph generator (see Fig. 11.2).

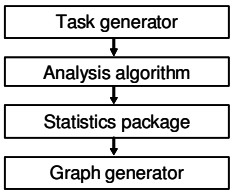

Figure 11.2: Analysis method in a research context

As preparation, the functionality belonging to the algorithms were extracted from the application. These functions constitute the core of the RTA and the core of the SSA plug-ins. According to the requirements specified in Section 11.3.2, the system information is accessed by the provided API. Fig. 11.3 shows a conceptual structure of the RTA and SSA plug-ins.

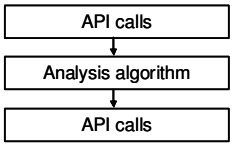

Figure 11.3: Analysis method in a plug-in context

The RTA plug-in is based on work by Mäki-Turja and Nolin [MTN04]. The plug-in will compute the worst case response-time of tasks. The SSA plug-in is based on work by Hänninen *et al.* [HMTB+06] and will compute an upper bound on shared system stack. We start by specifying the system model for the RTA and SSA plug-ins. We then define the required and provided interfaces as well as the error handling and user interface of each plug-in.

The system model in both [MTN04] and [HMTB+06] is an offset based model with transactions (a transactions is defined by a set of tasks with timing dependencies). Each transaction consist of one or more tasks. Tasks, in turn, have common real-time attributes such as worst case execution times, deadlines, priorities etc. For the RTA and SSA plug-ins, this implies that we need to extract transactions and task attributes from the design, execute the analysis and store the analysis results. Furthermore, from [MTN04] we know that the RTA interface should support the following:

- The RTA plug-in require, (i) Transactions with specified Period time (or MINT), (ii) the WCET, Offset and the Priority of the tasks in each transaction.
- The RTA provides the worst case response time (WCRT), relative to the activation of the transaction, of each task.

From [HMTB+06] we know that the SSA interface should support the following:

- The SSA plug-in require, (i) Transactions participating in stack sharing, (ii) the WCRT, Offset, Stack usage and the Priority of the tasks in each transaction.
- The SSA provides an upper bound on shared stack usage of the transactions.

Recall that since plug-ins are executed in succession in Rubus-ICE, each plug in must specify the data it needs and the data it produces. This is required for correct execution sequence of the plug-ins. For example, the above shows that the SSA plug in require worst case response-times for stack analysis, i.e., the RTA plug-in must be executed before the SSA plug-in, showing that analysis methods may have intricate dependencies that need to be considered when establishing the execution sequence of plug-ins.

When designing the error handling of the plug-ins, we observed that if something fails during analysis, the plug-in must be able to handle and isolate

the fault. The plug-in must also inform the host application of the fault. This is needed to isolate, i.e., to prevent the fault or erroneous values to propagate. For example, if the system is overloaded, i.e., the processor utilisation exceeds 100%, the response time analysis, being an iterative method, may never terminate. An even worse scenarios could occur if a variable overflows, then the RTA might actually terminate producing erroneous results. In a controlled research setting, this might not be a problem, since task generators are often configured to produce schedulable task sets as input to an analysis method. In an industrial setting, this assumption no longer hold. It is obvious that conditions such as overloads, variable overflows etc. need to be handled and dealt with properly. We also defined the actions that should be taken if, during analysis, the response-time of a task is larger than its deadline, i.e., the task is missing its deadline. The question then is, should we continue or abort the analysis. Although, this kind of situation might not be considered as an error, however, it might affect the execution of subsequent plug-ins. For instance, the algorithm in the RTA plug-in put no restrictions on response-times, i.e., response-time may be larger than the deadlines without affecting the correctness of the analysis. The SSA algorithm, on the other hand, require that a response-time is smaller (or equal) than the deadline.

The following list the error handling that need to be supported by the RTA and SSA plug-ins:

- The values of the read task attributes need to be checked.
- Overload conditions need to be checked during analysis.
- Variable overflow need to be checked during analysis.

For both the RTA and SSA plug-in, we define a simple interface against the user. The interface provides information of the progress of the plug-in and a summary of the analysis results. Fig. 11.4 shows the complete structure of the RTA and SSA plug-ins.

11.5 Adding plug-ins to Rubus-ICE - A case study

The plug-in framework, described in Section 11.3, allows users of Rubus-ICE to include third part developed plug-ins to the Rubus-ICE framework. To point out issues that may occur when adding plug-ins to the framework, we integrated the RTA and SSA plug-ins in Rubus-ICE. We believe that the RTA and SSA plug-ins represents typical analysis proposed by researchers and that the

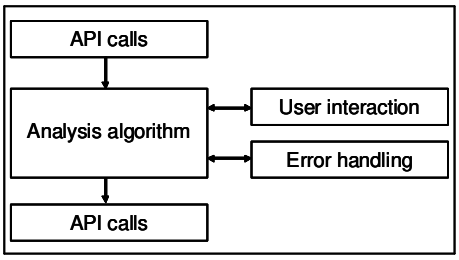

Figure 11.4: Analysis plug-in in an industrial context

integration of RTA and SSA plug-ins relieve issues that may be encountered when integrating other type of real-time analysis methods in Rubus-ICE.

The prerequisites for the integration was as follows:

- The plug-ins were implemented, adhering to the requirements in Section 11.3.2, and delivered as source code to the integrator.
- Both the RTA and SSA plug-ins should be integrated in Rubus-ICE.

The plug-ins were integrated in Rubus-ICE by a developer at Arcticus Systems. The developer had no previous experience of integrating real-time analysis methods, but was familiar with the overall objective of the plug-in framework and had been involved in specifying the requirements on the framework (Section 11.3.1). During integration, the developer was asked to note all issues that occurred during the integration. Integrating the plug-ins include (i) enabling configuration of a build sequence, (ii) establishing a correct execution sequence of plug-ins, (iii) verifying the functionality of the plug-ins and (iv) verifying the error handling of each plug-in. Even though the functionality, including error handling, of the RTA and SSA plug-ins have been verified in isolation, the integration in-itself may introduce unexpected errors, hence the plug-ins need to be verified once integrated. The following summarises the experiences of the integration as encountered by the integrator:

- Integration of the RTA and SSA plug-ins in Builder, i.e., enabling configuration of the build sequence to include the RTA ans SSA plug-ins, proceeded without notable difficulties.
- Establishing the execution sequence of the plug-ins was eased by the interface specifications included with the plug-ins.

- Verification of the functionality of each plug-in was experienced as troublesome. The integrator needed to consult the creators of the plug-ins to perform the verification. To verify the functionality of, for example, the RTA plug-in, example systems with only a few tasks were created. The results of the plug-in then needed to be verified by hand. It is obvious that larger systems are intractable to analyse by hand. Hence only small systems, with varying architecture, were possible to analyse.

- Certain type of error handling was difficult to verify. For example, verifying that variable overflows was handled properly was difficult, simply because it was difficult to create systems, or modify the attributes of the system, in such way that it resulted in variable overflows at the same time as the results of the plug-in seemed valid (see error handling in Section 11.4).

- When verifying the functionality of the RTA plug-in, the integrator discovered that the RTA plug-in sometimes produced pessimistic, albeit safe, response-times. When investigating the reason to this, we discovered that the API service extracting transactions from the design needed to be modified. The extraction, although being correct in a sense, failed to exploit the benefits of the analysis. Specifically, since the analysis is developed for an offset based system (offsets represents timing dependencies), these dependencies must be extracted from the design, and the better the extraction can represent the timing dependencies, the tighter the results from the analysis.

Altogether, the integration of the RTA and SSA plug-in in Rubus-ICE was experienced as fairly easy. We believe that the fact that both plug-ins were developed according to the requirements outlined in Section 11.3.2, facilitated the integration. Verification of the functionality was experienced as the most difficult task and the most time consuming activity. A continuous communication between the integrator and the plug-in developer was needed during the integration. This clearly demonstrates that the work of a plug-in developer do not end with the delivery of a plug-in. Even though the plug-in concept allows integration of third-party developed software, such as novel analysis methods, in a larger context, we believe that when transferring research results for industrial use, especially for use in development of safety critical systems, collaboration between the integrator and the plug-in developer is needed throughout the whole process for a successful end result.

11.6 Experiences summarised

We initiated this work with the aim of developing a plug-in framework to enable integration of third-party developed software. We showed, by integrating two plug-ins in the framework, that it is possible to add complex real-time analysis methods to Rubus-ICE. However, we discovered that a considerable amount of work is needed to prepare and integrate research results for industrial use. The main reason for this, we believe, is that the requirements and assumptions on analysis methods, in an industrial context, differs from the requirements/assumptions in a research context. For example, in an industrial context many analysis methods are used in developing safety critical software, implying that stringent error handling as well as thorough verification of the functionality is needed before analysis methods are accepted for industrial use. We also noticed that verification of the plug-ins, after being integrated in Rubus-ICE, required the help of the plug-in developer, since analysis methods are often very hard to understand and too difficult to verify by non-experts. When defining the user interface of the plug-ins, we discovered that it was non-trivial to provide constructive feedback to the users (compare to finding 9 in [NGS+01]). For example, since the RTA plug-in can be used to verify the schedulability of a system, the plug-in may occasionally discover that a system is unschedulable. The question then is, should we suggest modifications (e.g., changing priorities) of the attributes in the system, i.e., guiding the developer to possibly end up with a schedulable system. In many cases there are a large number of possible reasons why a system is unschedulable, hence suggesting constructive modifications is non-trivial.

Although we managed to integrate two novel analysis methods (the RTA and SSA plug-ins) in the framework, several issues still remain to addressed, for example:

- How can we guarantee that a plug-in only does what it is meant to do. For example, in the current framework it is possible for a plug-in to overwrite any values in the ICCM. These value may later be used by subsequent plug-ins and result in erroneous results.

- How should the output results from a plug-in be named in the ICCM. For example, if two plug-ins collaborate to solve a larger task, i.e., a plug-in reads the results of the other plug-in, then the plug-in reading the values produced by another plug-in need to have knowledge of the naming, simply to be able to fetch the values from the ICCM. Currently, there is no standard notation within the real-time community on how to

name the results produced by an analysis.

- How should we aggregate several plug-ins into a single assembly. Consider, for instance, priority assignment (assigning the dispatching priority to tasks). In real-time engineering, the arduous work of priority assignment is sometimes performed by hand. With the assigned priorities at hand, response time analysis can then be performed to verify schedulability of a system. In case the system turns out to be un-schedulable, the priorities are modified and response-time analysis is performed again. The process is basically an iteration of priority assignment and schedulability check by response-time analysis. The iteration is often performed until a schedulable system is found. This could be an automated procedure using two plug ins in the proposed framework, one that assign priorities and one that performs response-time analysis (e.g., the RTA plug-in). However, iterating the execution of these plug-ins, require them to be grouped in a single plug-in. Currently, there is no other way of iterating the execution of two or more plug-ins in the framework.

11.7 Conclusions and future work

In recent years, plug-in based development tools has increased in popularity. The plug-in concept enable third-party developed software to extend the functionality of a host application by add-ons, giving tool manufacturers a considerable simple way of adding new features and enhancing the value of their products. Developers can also benefit from the plug-in concept in the sense that plug-ins can be developed in isolation and do not require deep knowledge of the host application utilising the plug-in. These facts make the plug-in concept especially interesting when transferring complex research result for industrial use. Most of the published results from the research community are far too difficult to understand and even harder to implement for laymen, implying that many useful results never find their way to practical use.

In this paper we presented the development of a plug-in framework for Rubus Integrated Component Environment, an IDE for development of embedded real-time systems. The plug-in framework aims at facilitating integration of novel analysis methods in the framework. We presented the framework and implemented two novel analysis methods as plug-ins. The plug-ins were integrated in the framework by a developer without previous expertise in analysis methods for real-time systems. We showed that the framework is well suited for integration of complex analysis methods, however, we also showed that a

considerable amount of modifications of analysis methods are needed to adapt them for industrial use. In addition, a continuous communication between the researchers developing the plug-ins and the industrial representative integrating the plug-ins, was needed throughout the process.

As future work, we plan to extend the application programming interface to support other type of system models. Currently the Rubus system model is the only one supported. We also intend to define a way to aggregate several plug-ins into a larger one (an assembly consisting of several separate plug-ins). We believe that many engineering activities, such as priority assignment, schedule creation and task allocation etc. could be automated by aggregated plug-ins.

Bibliography

[ABT+93] N.C. Audsley, A. Burns, K. Tindell, M.F. Richardson, and A.J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5): 284–292, 1993.

[Arc] Arcticus Systems. Web page, http://www.arcticus-systems.se.

[BAE] BAE Systems Hägglunds. Web page, http://www.baesystems.-com/hagglunds.

[BTW95] A. Burns, K. Tindell, and A Wellings. Effective Analysis for Engineering Real-Time Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 22(5):475–480, May 1995.

[Ecl] Eclipse - an open development platform. Web page, http://-www.eclipse.org/.

[EHS97] A. Ermedahl, H. Hansson, and M. Sjödin. Response-Time Guarantees in ATM Networks. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 274–284. IEEE Computer Society Press, December 1997. URL `http://www.docs.uu.se/~mic/papers.html`.

[Hal] Haldex traction systems. Web page, http://www.haldex-traction.com/.

[HMTB+06] K. Hänninen, J. Mäki-Turja, M. Bohlin, J. Carlsson, and M. Nolin. Determining Maximum Stack Usage in Preemptive Shared Stack Systems. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, Dec 2006.

[HMTN+08] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. Supporting Engineering Requirements in the Rubus Component Model. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-223/2008-1-SE, Mälardalen University, February 2008.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[KAS93] D.I. Katcher, H. Arakawa, and J.K. Strosnider. Engineering and Analysis of Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 19(9):920–934, September 1993.

[Kno] Knorr-bremse. Web page, http://www.knorr-bremse.com.

[Leh90] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proc. 11^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 201–212, December 1990.

[MTN04] J. Mäki-Turja and M. Nolin. Tighter Response-Times for Tasks with Offsets. In *Proc. of the 10^{th} International conference on Real-Time Computing Systems and Applications (RTCSA'04)*. Springer-Verlag, Göteborg, Sweden, August 2004.

[NGS+01] C. Norström, M. Gustafsson, K. Sandström, J. Mäki-Turja, and N. E. Bånkestad. Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry. In *Eigth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. IEEE Computer Society, April 2001.

[Ora] Oracle JDeveloper Overview, An Oracle White Paper. Web page, http://www.oracle.com/technology/products/jdev/-collateral/papers/1013/jdev1013_overview.pdf.

[Pun97] S. Punnekkat. *Schedulability Analysis for Fault Tolerant Real-time Systems*. PhD thesis, University of York, June 1997.

[SAr+04] L. Sha, T. Abdelzaher, K-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2/3):101–155, 2004.

[SRL87] L. Sha, R. Rajkumar, and J.P. Lehoczky. Task Scheduling in Distributed Real-Time Systems. In *IEEE Industrial Electronics Conference*, 1987.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: an Approach to Real Time Synchronization . *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[TB94] K. Tindell and A. Burns. Fixed Priority Scheduling of Hard Real-Time Multimedia Disk Traffic. *The Computer Journal*, 37 (8):691–697, 1994.

[TC94] K. Tindell and J. Clark. Holistic Schedulability Analysis For Distributed Hard Real-Time Systems. Technical Report YCS197, Real-Time Systems Research Group, Department of Computer Science, University of York, November 1994. URL ftp://ftp.cs.york.ac.uk/pub/realtime/papers/YCS197.ps.Z.

[THW94] K. Tindell, H. Hansson, and A. Wellings. Analysing Real-Time Communications: Controller Area Network (CAN). In *Proc. 15^{th} IEEE Real-Time Systems Symposium (RTSS)*, pages 259–263. IEEE, IEEE Computer Society Press, December 1994.

[Tin92] K. Tindell. An Extendible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. Technical Report YCS189, Dept. of Computer Science, University of York, England, 1992.

[VCE] Volvo construction equipment. Web page, http://www.volvoce.com.