Improving Predictability and Resource Utilization in Component-Based Embedded Real-Time Systems

Johan Fredriksson

October 2008

Department of Computer Science and Electronics
Mälardalen University
Västerås, Sweden
Abstract

Embedded real-time systems (ERTS) are integrated into products in many technology areas, e.g., in different kinds of automation systems controlling production and machines. Due to requirements on dependability and low production costs, properties such as predictability and resource utilization are very important.

As the complexity of such software intensive systems grows, more effort is put on producing resource efficient and predictable software. Typical examples of systems where predictability and resource efficiency of ERTS are important are those in which dependability are important, e.g., cars, heart-lung machines and consumer electronics. One important class of requirements for predictability in ERTS systems is real-time requirements because embedded systems typically react on their physical environment and have to respond within a bounded time interval. Developers of such systems want to keep production and development costs to a minimum; however, at the same time more functionality is added for increasing market competitiveness.

Component-Based Software Engineering (CBSE) is a development strategy aiming at systematic reuse of pre-built software components. It is one of the fastest growing development strategies in industry because of its promise to lower development and maintainability costs, and shorten time-to-market. However, CBSE comes with the cost of greater initial investments for creating more general and reusable software components, and the investment in these components are only justified when they are deployed in several environments.

Resource efficiency and predictability are properties heavily depending on the environment and usage of a software component. These properties are contradictory to efficient component reuse. This is because reusability requires generality, and both predictability and resource efficiency are specialized solutions.
To tackle these problems, this thesis presents three novel approaches for improving predictability and utilization of resources in embedded component-based systems. The first approach is a contract-based technique to achieve reuse of worst-case execution times (WCET) predictions in conjunction with reuse of software components. For resource constrained systems, or systems where high degree of predictability is needed, classical techniques for WCET-estimation may result in unacceptable overestimation of the execution-time of reusable software components. Our technique allows different WCETs to be associated with subsets of the component behaviour. The appropriate WCET for any usage context of the component is selected by means of component contracts over the component’s input domain.

The second approach is a method for deriving the input combinations of a software component that causes the WCET. The information resulting from this method can be used, e.g., for guiding measurement-based WCET analysis.

The third approach is a framework for finding resource-efficient allocations from components to real-time tasks. Few component technologies today consider the allocation from components to real-time tasks, while maintaining stipulated real-time constraints. Efficient allocations can reduce memory usage and CPU-overhead considerably.

In addition, this thesis extends the CBSE development process, and demonstrates how the outlines methods are incorporated in the CBSE development process. Furthermore, two tools have implemented and are used for evaluating the methods. The evaluations show that by using the methods outlined in this thesis resource efficiency and predictability can be substantially increased without negative impact on reuse.
Preface and Acknowledgements

When I began my undergraduate studies at Mälardalen University I intended to study for three years, and no more. However, after three years the studies had become so interesting that I decided to continued for a further one and a half years. After this one and a half years I again discovered that my need for more knowledge was unsatisfied. Having been a Ph.D. student for five years I find that the fascination continues and ...

This thesis is the end of the journey, and as E. Hemingway said, “It is good to have an end to journey toward; but it is the journey that matters, in the end”.

.... Prof. Ivica Crnkovic is my main supervisor. Thank you for always believing in me, and for giving me this opportunity to be a PhD student. We have travelled many times together, especially many times to Croatia, where we have had many “piva” and a lot of sladoled. Hvala!

Dr. Kristian Sandström is the assistant supervisor that has been with me the longest. I have always enjoyed our zestful discussion about vital gadgets with. I am amazed by your ability to find the details in my research that noone else finds. I have you to thank for so many things in my research! Tack så mycket!

Dr. Thomas Nolte is my assistant supervisor and friend. We have travelled (and photographed) the world, both within the frame of the research projects and private. Your friendship and guidance have made me not only a better researcher, but also a better person. Grazie Mille!

Prof. Mikael Nolin is my assistant supervisor. Your guidance, both on a personal and professional level have helped me many times. You always believed in me, and backed me up. You are always very enthusiastic and you succeed to spread that to those who work with you. I always enjoy our discussion on gadgets, wine and real-time analysis - the essence of life. Tack så
mycket!

For all help and support with worst-case execution time analysis in general and with SWEET in particular, Dr. Andreas Ermedahl has helped a lot, even in times of moving between houses. We have had countless fruitful discussions, on the train between Stockholm and Västerås, and we have authored several papers together.

Much of the work was initiated during a six month stay with Prof. Heinz Schmidt. Your ingenuity is amazing, and I have tried my best to keep up in your pace. Vielen Dank!

Dr. Mikael Åkerholm is my colleague and friend. We have known each other since we started the undergraduate studies together. Later we continued on to the Ph.D, and we have even worked together in industry. I'm looking forward to .... . Besides being a great colleague and co-author you are also a great friend!

I have spent many hours discussing life, education of children, development processes and research methodology with Dr. Rikard Land, with whom I have also authored several papers. You have taught me a lot, both academically and personally. Tack så mycket!

I have spent countless lunches discussing training and physiology with Prof. Hans Hansson and Prof. Christer Norström, I will sincerely miss that.

During my path towards my PhD, several people have made my work and life more enjoyable in many ways. Thank you Johan Kraft for being a good friend, and great to discuss ideas with. Thank you Dr. Daniel Sundmark for our training sessions and discussions, and for being a great friend. Thank you Anders "Petsson" Petterson for your encouragement, friendship and discussions.

I have shared project trips with several people during my time at the department. Thank you Dr. Thomas Nolte, Dr. Joakim Fröberg, Dr. Stig Larsson, Anders Möller, Dr. Radu Dobrin, Dr. Dag Nyström, Hüseyin Aysan, Moris Behnam, Markus Bohlin, Dr. Kaj Hänninen, Dr. Jan Carlsson, Dr. Mikael Åkerholm, Dr. Rikard Land, Prof. Hans Hansson, Prof. Christer Norström, Johan Kraft, Dr. Daniel Sundmark and Gunnar Widforss

I spent six months of my studies at the Centre for Distributed Systems and Software Engineering at Monash University, Melbourne, Australia. From my time in Melbourne I want to thank Dr. Christian Guttmann, Dr. Ian Peake, Prof. Heinz Schmidt and Shane Galdigau, for making my time so pleasant.

My personal trainer and friend Dr. Markus Nilsson ...

There are countless people who have given me so much during the last five years. Thank you Markus Lindgren keep on training. Thank you Dr. Massimo Tivoli. Thank you Anders Möller. Thank you Dr. Joakim Fröberg for the blues.
Thank you Dr. Radu Dobrin, Dr. Dag Nyström I really do like you too. Thank you Hüseyin Aysan for teaching me about photography.

Moris Behnam
Markus Bohlin
Dr. Jan Carlsson
Jonas Neander
Dr. Kaj Hänninen
Dr. Stig Larsson
Dr. Henrik Thane
Monica Wasell
Gunnar Widforss
Harriet Ekwall
Thesis reviewers: Thomas Nolte, Mikael Nolin, Kristian Sandström and Ivica Crnkovic, Stefan Bygde, Rikard Land, Radu Dobrin, Andreas Ermedahl, Jan Gustavsson, Hans Hansson, Björn Lisper, Frank Lüders, Dag Nyström, Paul Pettersson,

Supervisors, Dr. Thomas Nolte, Prof. Mikael Nolin, Dr. Kristian Sandström and Prof. Ivica Crnkovic

Co authors, Dr. Thomas Nolte, Dr. Andreas Ermedahl, Prof. Mikael Nolin, Prof. Ivica Crnkovic, Prof. Heinz Schmidt, Dr. Mikael Åkerholm, Dr. Jan Carlsson, Dr. Massimo Tivoli, Prof. Peter Alternbernd, Dr. Rikard Land, Dr. Kristian Sandström, Anders Möller, Prof. Hans Hansson, Prof. Ivica Crnkovic, Dr. Joakim Fröberg, Dr. Radu Dobrin, Irena Pavlova, Dr. Ian Peake.

Colleagues, Johan Kraft, Hüseyin Aysan, Markus Lindgren, Moris Behnam, Markus Bohlin, Jan Carlsson, Ewa Hansen, Jonas Neander, Andreas Hjärtström, Kaj Hänninen, Damir Isovic, Magnus Larsson, Stig Larsson, Jukka Mäki-Turja, Christer Norström, Sasikumar Punnekkat, Daniel Sundmark, Anders “Pettson” Petterson, Filip Sebek, Johan Stärner, Henrik Thane, Peter Wallin, Kurt Wallnau, Monica Wasell, Gunnar Widforss, Harriet Ekwall.

CC systems: Stefan Rönning, Jörgen Hannson, Johnnie Blom, Ken Lindfors, Anders Öberg, Andreas Olevik, Mats Kjellberg, Dr. Jochen Wendelbaum, Jörgen Martinsson, Göran Sohlman, Malin Olsson, Jonas Ehlin, Johan Persson, Mattias Lång, Ulf Sporrong, Carl Falk, David Wretling.

Former Colleagues: Dr. Joel Huselius, Dr. Markus Nilsson, Christian Andersson, Dr. Thomas Lenvall, Mohammed El Shobaki, Dr. Anders Wall.

During my .... I lived in Melbourne for half a year, and it has become one of my absolute favourite places.

My family: Mother Inger, Father Tommy, Sister Erika, Grandmother Karin,
Grandfather Evert. My beautiful “wife” Marie. Other family: Mikael, Angelica, Monica, Kjell.

In laws: Mimmis, Marja, Nils, Elias, Tina, Alexander, Sebastian.
Notes for the reader

This thesis deals with timing predictability and resource optimization for embedded real-time systems that are developed component-based. In Chapter 5 we introduce extensions to a development process component-based software engineering for facilitating predictability and resource optimization of embedded real-time systems. Chapter 6 outlines a framework for reuse of worst-case execution time analysis for software components. In Chapter 7 we describe a framework for optimization of resources for component-based embedded real-time systems. These Chapters present the research contributions of this thesis. Chapters 6 and 7 can be read independently of each other, and Chapter 5 describes the connection between these two frameworks.

To read about the results of the methods outlined in Chapters 6 and 7 the reader is referred to Chapter 8, which outlines the empirical studies and data collected from analysing the methods. Chapter 8 also describes a prototype tool that implements the ideas outlined in Chapter 6.

Chapter 2 gives an introduction to real-time systems, real-time analysis and worst-case execution time analysis, and Chapter 3 gives the basics of component-based software engineering and describes fundamental conceptions like component model, component, reuse and more.

For the reader interested in the research methodology, Chapter 4 provides detailed information about the research problems, questions and validations, and outlines the research method.

To get an introduction and a summary of the thesis; the problems we deal with, and the contributions, the reader is referred to Chapter 1 for an introduction and Chapter 9 for a summary of the thesis and the thesis contributions. Chapter 9 also outlines possible future research directions.

This thesis encloses two appendices; Appendix A provides an extended set of data from the validations described in Chapter 8. Appendix B provides a full list of all publications authored and co-authored by Johan Fredriksson.
Publications in this thesis

A full list of publications is presented in Appendix B. The following is the list of publications related to this thesis. The papers are categorized in publications on which this thesis is based, and papers that contribute to the thesis. The papers are presented in reverse chronological order.

Publications related to the thesis contributions

1: Deriving the Worst-Case Execution Time Input Values, Andreas Ermedahl, Johan Fredriksson, Peter Altenbernd, submitted for publication.


Publications related to the thesis


11: A Sample of Component Technologies for Embedded Systems, Mikael Åkerholm, Johan Fredriksson, Technical Report, Mälardalen University, November, 2004

12: Evaluation of Component Technologies with Respect to Industrial Requirements, Anders Möller, Mikael Åkerholm, Johan Fredriksson,
Mikael Nolin, in proceeding of the 30th IEEE Euromicro Conference, Component-Based Software Engineering Track, Rennes, France, August, 2004


## Contents

1 Introduction
   1.1 Embedded real-time systems ................................ 1
   1.2 Component-based software engineering ...................... 4
   1.3 Informal problem formulation ............................... 5
      1.3.1 Components and real-time .............................. 6
      1.3.2 Resource efficiency .................................. 6
      1.3.3 Developing embedded real-time systems ............... 7
   1.4 Overview of our solutions .................................. 8
      1.4.1 Reusable WCET analysis ............................... 8
      1.4.2 Derivation of WCET input combinations ............... 10
      1.4.3 Allocation to tasks .................................. 11
   1.5 Contributions .............................................. 13
   1.6 Thesis overview ........................................... 15

2 Embedded real-time systems .................................... 19
   2.1 Embedded systems - general concept ......................... 19
   2.2 Real-time systems - general concept ....................... 20
      2.2.1 Classification ......................................... 22
   2.3 Real-time model ............................................ 23
      2.3.1 Task model ............................................ 24
      2.3.2 Resource model ....................................... 25
      2.3.3 Scheduling policies .................................. 26
   2.4 Real-time analysis .......................................... 26
   2.5 Schedulability analysis .................................... 27
      2.5.1 Utilization based analysis ............................ 27
      2.5.2 Demand based analysis ............................... 27
      2.5.3 Response time analysis .............................. 28
## Contents

2.5.4 Transactions .............................................. 29  
2.5.5 Attribute assignment .................................... 29  
2.6 Worst-case execution time analysis ....................... 29  
2.6.1 Classification of WCET analysis ....................... 30  
3 Component-based development for ERTS .................... 33  
3.1 Motivation .................................................. 33  
3.2 Component reuse ............................................ 36  
3.3 Basic definitions in CBSE .................................. 36  
3.4 CBSE development process .................................. 37  
3.5 Component model ............................................ 37  
3.6 Component definition ...................................... 39  
3.6.1 Component interaction .................................. 41  
3.6.2 Component interface .................................... 42  
3.6.3 Component composition .................................. 43  
3.6.4 Component contracts .................................... 43  
3.7 Component technology ...................................... 43  
3.8 Component frameworks .................................... 44  
3.9 Summary .................................................... 44  
4 Research problem ............................................. 47  
4.1 Research introduction ...................................... 47  
4.2 Specific research goal ..................................... 48  
4.3 Research method ............................................ 48  
4.4 Industrial problems ........................................ 51  
4.4.1 General industrial problem ............................. 51  
4.4.2 Specific industrial problem ............................ 52  
4.5 Research setting ............................................ 53  
4.5.1 Limiting the problem using assumptions ............. 54  
4.5.2 Research problems ...................................... 57  
4.6 Requirements ............................................... 59  
4.6.1 Requirement definitions ................................. 60  
4.7 Partial solution proposals .................................. 62  
4.7.1 Resource aware development ........................... 63  
4.7.2 Input sensitive WCET analysis ....................... 63  
4.7.3 Allocation of components to real-time tasks ........ 64  
4.8 Validation of solutions .................................... 65  
4.8.1 Resource aware development ........................... 66  
4.8.2 Input sensitive WCET analysis ....................... 67
4.8.3 Transformation from components to real-time tasks

5 Resource aware development
5.1 Component-based software engineering
5.2 Reusable analysis
5.3 Allocating components to real-time tasks
5.4 System model
5.5 ACC Example
5.5.1 Usage profile assessment
5.5.2 Input sensitive WCET analysis
5.5.3 Allocating component to tasks
5.6 Summary

6 Input sensitive execution-time analysis
6.1 Input sensitive WCET analysis
6.1.1 Input value space partitioning
6.1.2 Partitioning primitives
6.1.3 WCET analysis tool assumptions
6.2 Reusable WCET analysis
6.2.1 Component WCET analysis
6.2.2 Binary search algorithm description
6.2.3 Approaching parametric WCET
6.2.4 Algorithm complexity
6.3 Finding WCET input combination
6.3.1 Algorithm description
6.3.2 Example
6.3.3 Algorithm complexity and back-tracking
6.4 Approaches for faster termination
6.4.1 Slicing the input space
6.4.2 Extreme value heuristic
6.4.3 User interaction
6.5 ACC example - input sensitive WCET analysis
6.6 Summary

7 Allocating components to real-time tasks
7.1 Introduction
7.1.1 Component to task allocation
7.2 Allocating components to real-time tasks
7.3 Allocation framework
## Contents

9.3.5 Worst best-case ............................................. 162  
9.3.6 Advanced real-time requirements ......................... 162  
9.3.7 Incremental WCET analysis ............................... 162  

A Complete list of tables and graphs .......................... 165  
B Complete list of publications ...................................... 171  
Bibliography .......................................................... 183
In the beginning the Universe was created. This has made a lot of people very angry and has been widely regarded as a bad move.

-Hitchiker’s guide to the galaxy

Chapter 1

Introduction

In this thesis we explore context aware, reusable Worst-Case Execution Time (WCET) predictions and optimization of resource utilization in software components for component-based embedded real-time systems. Such systems are typically found in embedded applications such as consumer electronics and vehicular systems.

We have developed methods for, (i) reusing WCET analysis for reusable software components, and, (ii) mapping components to tasks to minimize stack consumption and CPU-overhead while maintaining real-time constraints. Both methods have been implemented and validated.

In this chapter we give an introduction to our research, starting with an illustrative example from the real world before we give an overview of the specific research and contributions. We conclude this chapter with an overview of the rest of the thesis and a discussion.

1.1 Embedded real-time systems

Many modern products have impressive capabilities. Take as example a modern car with functions like Electronic Damper Control (EDC) and Adaptive Cruise Control (ACC). Only two decades ago such functions were impossible to achieve, only relying on mechanical solutions. New advanced functions are possible because mechanical systems are being replaced by electro mechanical systems controlled by software. Consider the Electronic Stability Control (ESC) which is an advanced function in modern cars [vZELP98]. ESC is a
technology that improves the vehicle’s handling by detecting and preventing skids. This function is possible because mechanically controlled brakes are replaced by computer controlled brakes where each wheel can be individually braked.

Figure 1.1: Conceptually electronic stability control

*Embedded systems* comprise electronics and software operating to adapt to, or control, its environment. Embedded systems are different from desktop computers in the sense that they do not commonly have a screen or keyboard for interaction, but rather have different inputs for analog and digital sensors, and, different types of communication buses. The embedded computers are often referred to as *Electronic Control Units* (ECUs). For example the ESC system in a car consists of a number of ECUs, and, several sensors and actuators. A typical example of sensors in the ESC system are a steering wheel sensor that determines the drivers intended path, a yaw sensor that reads the rotation of the car, wheel speed sensors that measure the speed of each individual wheel, and a lateral acceleration sensor that measures lateral (sideways) acceleration of the car. The ECU continuously reads the sensors to determine if the car is under steering ((B) in Figure 1.1) or over steering ((C) in Figure 1.1). The presumed vehicle path is calculated with the steering wheel sensor ((A) in Figure 1.1), and is compared to the actual path that is calculated with the lateral acceleration, wheel speed and yaw sensors. To prevent the car from over
steering or under steering the correct brake action is calculated and applied for each wheel individually together with reducing the engine power.

The ESC system relies on that observation and action are performed with a predictable timing pattern. The timing pattern usually comprise an exact periodicity \( T \) and a last finish time, i.e., a deadline \( D \) when all calculations and actions need to be finished. Typically an ESC system is triggered periodically observing the environment every 40 milliseconds, and the brake force should be applied within a few milliseconds. In order to prove that the system really fulfils these timing requirements, engineers use Real-Time Analysis (RTA), and one of the most important parts of that analysis is the WCET analysis. WCET analysis determines how long time the calculations and actions can possibly take in the worst case.

Systems that rely on time to function correctly are called Real-Time Systems (RTS). A definition that is commonly cited in literature is one by Stankovic [SR89]:

*Real-time systems are computer systems in which the correctness of the system depends not only on the logical correctness of the computations performed, but also on which point in time the results are provided.*

The power of software controlled systems has lead to modern cars having up to 90 on-board computers controlling the different functions in the car. Such computer systems that are embedded in apparently non-computerized electrical and electro-mechanical devices are known as embedded systems, and constitutes more than 99% of all computers in the world [Tur02, Lau06]. The IEEE has defined embedded systems as [IEE92]

*A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system.*

Development of software for embedded RTSs is considered a complex and difficult task, both due to the additional requirements imposed by such systems but also because of the inherent inobservability of embedded systems as they normally lack human machine interfaces (screen and keyboard). Software gives an increasing possibility for advanced functions and adaptive behaviour and has become the primary means for creating added value for customers. For instance software in cars help reduce gas consumption as well as increase performance, comfort and safety, and as a result systems become increasingly
software intensive. For example, the next generation of premium automobiles are estimated to carry around one gigabyte of binary code [ABGP05], which is comparable to a typical desktop workstation today. Reasons for this tremendous increase in code size include the demand for new functionality on the one hand, and the availability of powerful and cheap hardware on the other hand [PBKS07, ABGP05].

In addition customers expect new embedded systems to enter the market faster, at lower prices, and the competition for customers is tough. The decreasing time to market and increasing product differentiation leads to that software is required to be flexible enough for rapid reuse, extension and adaptation of system functions. As a result the trends in the embedded systems sector are:

- embedded RTSs become increasingly software-intensive [FdN08].
- costs shift from hardware to software [CAPD02].
- individual functions integrate increasing functionality [Cl02b].

This leads to requirements for new development paradigms that will enable an efficient and cost-effective development while ensuring low cost of prediction and high quality of the software.

### 1.2 Component-based software engineering

To cope with the decreasing time-to-market and the increasing software complexity, designers are looking for new ways of building systems. The notion of reuse has gained increasing interest as software complexity grows. However, ad-hoc reuse has proven to be difficult and not very successful [PD96, GSCK04]. Therefore, Component-Based Software Engineering (CBSE) has gained a lot of interest, and especially the possibility of integrating software from other vendors, i.e., third party composition. A definition that is regularly cited in publications is the one by Heinemann and Councill [HC01]

> A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

The term component model embraces the specification of components, how components are assembled (composition), and the component framework. In other words, the component model is a set of rules governing how the components may or may not be used. The composition of components is the process
of assembling components to form an application. Components are composed
to constitute systems by connecting their interfaces according to the rules de-
fining in the component model. The *component interface* is the entry to the
component functionality. A component composition is executed in the context
of a *component framework*. The component framework provides the necessary
run-time support that is not provided by the underlying run-time system, and
finally, a *component technology* is the concrete implementation of a compo-
nent model. To facilitate reuse, components are designed to be generic and
often with functionality for different deployments. At the same time systems
integrate more functions into single components, giving rise to increasingly
varying behaviour of these components. This in turn makes it harder to predict
important real-time properties of components.

### 1.3 Informal problem formulation

In this section we give an informal description of the problems that gives a
good overview of both the problem domain, and an overview of the problems
that we aim to solve. In Chapter 4 we provide a more formal formulation of
the industrial problems as well as the academic research setting and research
problems.

As software complexity increases, software reuse becomes interesting. Be-
cause software is “soft”, it is relatively easy to create tailored software that
exactly fulfils all system requirements. As the software complexity increases
the behaviour become increasingly complex and varying. When reusing soft-
ware components, it is unlikely to find a component that exactly fulfils derived
system requirements. The key to reuse is generality and context independence,
and for many specific component use cases, only parts of the component be-

behaviour is actually going to be used. Unfortunately generality and context
indipendence also leads to an increasing inability to make accurate predictions
of the component behaviour for each specific use-case. Hence, there is a need
for parametrization of the predictions in order to support reuse and at the same
time accurate predictions [PD96, FPDF98].

Expected benefits from using CBSE include more effective management
of complexity, shorter time-to-market and higher maintainability. Reuse is the
main characteristics for CBSE that would bring these benefits. Today issues
relevant to embedded *component-based systems* such as real-time and resource
efficiency are often addressed outside CBSE. There are many methods and
theories for, e.g., RTA, but few in relations to CBSE [MGL06, MYZC06].
1.3.1 Components and real-time

One of the most important activities for RTA is WCET analysis. There are many theories and tools for performing such analysis [SWE, Rap08, aIT, Bou, LME98, EY97, BCP02, FW99]. Common for WCET analysis is that it is a complex and time consuming activity not suitable for software with varying usage. This is inherent in that the analysis is context and usage unaware; and components are typically deployed in different contexts with different usage, and the usage can vary a lot between different contexts. Therefore, for components that are reused in different systems it is often not very meaningful to perform WCET analysis before the complete system has been designed, and each components’ usage has been determined.

As the complexity and diversity of component functionality increases it becomes harder to lower resource consumption and at the same time guarantee real-time constraints. This is because it becomes harder to make tight predictions and keep resource utilization with general components; at the same time reuse of general components have been the key to structured and efficient development. Paradoxically, components should be context-unaware to be reusable at the same time as they need to be context sensitive in order to support accurate WCET analysis. This seems to be a fundamental problem to overcome before the CBSE paradigm can be fully adopted in the embedded systems domain.

Summarizing the above:

- WCET is a pre-requisit for RTA.
- WCET analysis is complex and time consuming.
- Reuse of current WCET analysis results between different contexts leads to imprecise estimation of WCET.

1.3.2 Resource efficiency

In order to further support resource efficiency it is important to consider the allocation of components to real-time tasks. Components are often directly allocated to real-time tasks partly due to the ease of directly mapping timing constraints from components to tasks, and, partly due to that it is not trivial to find an allocation between components and tasks such that the stipulated timing requirements can be fulfilled. Allocating components to real-time tasks is a multi dimensional problem involving timing constraints and component architecture properties such as, e.g., component interaction and temporal or spatial
isolation between components. Each task and each task switch generates a re-
source overhead. Thus, the number of tasks is a trade-off between fulfilling
timing constraints and minimizing resource utilization. A higher number of
tasks lead to higher overhead in terms of memory and CPU usage.

When each component is allocated to a single real-time task the WCET
prediction error of each task is the same as the error of the component. When
several components are mapped to one task, the error scales with the number
of components, and the error can become quite large. The total system error
stays the same but greater errors of individual tasks have a greater impact on
properties like input jitter and output jitter, just to mention a few.

In other words:

- A high number of tasks increases the resource consumption, and de-
creases system performance.
- Allocation of components to real-time tasks must maintain the compo-
nent architecture.
- Allocation of components to real-time tasks must consider temporal con-
straints.

### 1.3.3 Developing embedded real-time systems

Even simple embedded systems today show more and more complex behaviours,
some triggered by usage-awareness or deployment-specific configuration pa-
rameters. Properties of the component such as time and reliability are variable
and usage-dependent, and the variance may be large. Together with global sys-
tem timing requirements that are required to be handled with scheduling and
increasing requirements on resource consumption for lowering hardware costs,
the embedded systems domain is facing a difficult problem.

In order to facilitate CBSE for Embedded Real-Time Systems (ERTS), is-
issues like real-time and resource consumption must be addressed as first class
Citizens in the component model. There are many theories and models on both
real-time and resource consumption, but very few in relation to CBSE. This
thesis is a step towards using CBSE for ERTS, with a particular focus on the
following aspects:

- Prediction of execution-time of components with respect to component
  usage.
- Classification of execution times with respect to usage.
• Allocations of components to real-time tasks with respect to real-time constraints.

• Optimization of system properties with respect to resource efficiency.

1.4 Overview of our solutions

We present three solutions that help tackle the outlined problems. The first two solutions are for supporting tight and reusable WCET analysis. The first solution is a methods for parametrizable and reusable prediction of the real-time property WCET for reusable software components. In the second solution we derive the input combination that triggers the execution of the program path that generates the WCET. The third solution is an allocation of components to real-time tasks for improved resource utilization and maintained real-time constraints. We present an overview of the solutions in the following sections.

1.4.1 Reusable WCET analysis

To support reuse of WCET predictions we need support for WCET analysis of different usage.

A component that is designed for reuse has to be general and free from context dependencies. By designing the component specifically for one particular context or usage it can be analysed and predicted with high accuracy, but not easily reused. In order for general reusable components to be predicted with higher accuracy we need new methods and frameworks. When the usage is not known at design time of a component, it is necessary to augment the component with information that can be used to accurately predict the WCET for a specific usage. The WCET can differ a lot between different uses of the same component. We want to define a contract as a function of an input-scenario to determine the WCET for that specific usage scenario. The reusable WCET analysis can be divided in three steps, namely:

1. **Component WCET analysis**: Analysing the WCET of the component with respect to many different general usage scenarios (inputs).

2. **Clustering WCETs**: Clustering inputs that lead to similar execution times.

3. **Component contracts**: Creating a contract that define the clustered inputs.
1.4 Overview of our solutions

We show how the precision and reusability of WCET can be increased for software components. We also discuss and give examples of how the proposed techniques can be used for (i) aiding run-time measurements for acquiring WCETs, and (ii) facilitating partial WCET analysis.

Summary of results

We have found that for industrial and academic components it is possible to achieve a reusable parametric WCET with high accuracy\(^1\). For most components every input combination is mapped to its corresponding exact\(^2\) execution time. Furthermore, we have found that the overall system WCET analysis become more accurate with our methods compared to that of traditional WCET analysis.

Related work

Recent case-studies show that it is important to consider mode- and context-dependent WCET estimates when analysing real sized industrial software systems\([SEG^\ast+06]\). There are suggested models of the overall component-based life cycle processes\([AZP03, CCL06]\) as well as more concrete methods for, e.g., component assessment\([BB05, HC01]\); our work illustrates how the division into context-unaware and context-sensitive analyses could be integrated into these models.

Staschulat et al.\([SESW05]\) make a similar partitioning of execution time behaviour of software modules based upon the context in which the module is derived. Our approach has some similarities with this work, but we use the partitioning for providing reusable and parametric analysis, whereas Staschulat et al. partitioning only for increasing the accuracy of WCET analyses.

Gheorghita et al. in\([GSBC05]\) use usage scenarios to determine tighter loop bounds. In\([MMH^\ast+05]\) Mohan et al. use run-time usage information for dynamic voltage scaling depending on the timing requirements. Wenzel et al.\([WRKP05]\) use both model checking and genetic algorithms to derive which input data makes a certain instrumented code part to be executed. Gross et al.\([SESW05]\) use evolutionary testing with measurement based WCET analysis to find a context dependent WCET. In\([DP04]\) a framework for probabilistic WCET with static analysis is presented. The probabilities are related to the probability of possible values of external and internal variables.

\(^1\)we give an exact definition of accuracy in Chapter 6

\(^2\)as exact as the underlying WCET analysis can provide
In [BCP03b, BCP03a] each basic block of a program is analysed with respect to execution times and probability distributions of the execution times are derived. This method is, in contrast to our method, based on measurements. In [LPB+05] a framework has been developed that considers the usage of a system; however, neither software components nor reuse is considered. Ji et al. in [JWLQ06] divide the source code in modes depending on input, and only the parts that are used in the a specific usage are analysed.

None of the above mentioned approaches have reusability or software components in mind. Also, our approach is more general and able to derive the input values that gives the program WCET for different usages.

In [HKR06, FBH05, MYZC06, CZM+03, Zsc04] methods for parametrizable contracts and their composition are proposed; however, they do not propose any specific analysis.

### 1.4.2 Derivation of WCET input combinations

A WCET analysis derives upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A problem with today’s WCET analyses is that there is no feedback on what input values that actually cause the WCET. However, this is an important information for the system’s designer for various reasons. It can, e.g., be used for identifying bottlenecks, and hence is very useful for further optimising the program. Further, the information gained is valuable for whole-system stress testing i.e., identifying the overall systems real worst-case behaviour, and for steering measurement-based timing analysis approaches, to select input value combinations to run for long execution times. The derivation of WCET input combination is based on the same technique as used for creating the reusable WCET contracts, and can be divided into the following steps:

1. **WCET analysis:** Analysing the WCET with respect to a large set of input combinations

2. **Reducing inputs:** Removing input combinations that do not lead to the WCET.

3. **Backtracking:** Backtracking to explore all possible WCET input combination candidates.
Summary of results

We have found for both industrial and academic components that our methods work, and for most of our benchmarks the WCET input variables can be quickly derived even though the value space is large.

Related work

To the author’s knowledge the problem of deriving which input values actually cause the WCET has been sparsely addressed.

Wenzel et al. [WRKP05] use both model checking and genetic algorithms to derive which input data makes a certain instrumented code part to be executed. The worst-case timing derived for different code parts are then combined to an overall program WCET estimate. In contrast, our approach is more general and able to derive the input values that gives the overall program WCET.

1.4.3 Allocation to tasks

In RTSs temporal constraints are of great importance. In RTSs the software is divided into tasks. A task is an entity that is associated with a Task Control Block (TCB) that stores information in memory about the state of the software. The task uses this information to control the execution of the software. Tasks are invoked periodically or at any time, i.e., by events, and usually have timing requirements. Components triggered by the same periodic event can often be coordinated and executed by the same task, preserving temporal constraints. Every time a task is executed the run-time system performs a context switch to activate the task, and each context switch consumes a specific amount of CPU-time. There can be memory profits in terms of fewer TCBs and profits in terms of CPU-overhead from context switches by allocating several components into one task. Moreover, many embedded RTSs use so called single shot tasks that share stack and in such systems the stack-size can be reduced manifold by co-allocating components.

An allocation can be performed in several different ways. In a small system all possible allocations can be evaluated and the best chosen. For larger systems it is not possible to explore all allocations due to the combinatorial explosion. Different algorithms can be used to find an allocation and scheduling of tasks that fulfils the timing requirements. For any algorithm to work there must be some way to evaluate an allocation.
We propose an allocation framework that is used to calculate schedulability, CPU-overhead and memory consumption. The framework is used together with an optimization algorithm to find feasible allocations that fulfill the given timing requirements at the same time as memory consumption and CPU-overhead are kept as low as possible. The framework has three main concerns:

1. **Allocation verification**: Verifying that the allocation is feasible with respect to a set of constraints, e.g., schedulability and isolation.

2. **System property calculation**: The properties stack usage and CPU-overhead are calculated for each allocation.

3. **Resource optimization**: An optimization technique is used for optimizing the allocations with respect to stack usage and CPU-overhead while maintaining real-time requirements.

**Summary of results**

We have evaluated the framework by using genetic algorithms to find allocations. We have found that for automatically generated systems with properties extracted from industrial systems, the properties stack size and CPU-overhead can be lowered. By allocating several components to one task the memory consumption and CPU-overhead are lowered by as much as $\sim 50\%$ and $\sim 30\%$ respectively, compared to allocating one component to one task.

**Related work**

The idea of assigning components to tasks for embedded systems while considering extra-functional properties and resource utilization is a relatively uncovered area. In [BMdW+04, BMdWC04] Bondarev et al. are looking at predicting and simulating real-time properties on component assemblies. However, there is no focus on increasing resource utilization through component to task allocation. There are also methods proposed for transforming structural models to run-time models [Dou99, Gom00, MG02], but extra-functional properties are usually ignored or considered as non-critical [KWS03]. In [SW00], an architecture for embedded systems is proposed, and it is identified that components have to be allocated to tasks, however there is no focus on the allocation of components to tasks. In [KWS03] the authors propose a model transformation where all components with the same priority are allocated to the same task; however no consideration is taken to lower resource usage. In [GLN01], the
authors discuss how to minimize memory consumption in real-time task sets, though it is not in the context of allocating components to tasks. Shin et al. [SLM02] are discussing the code size, and how it can be minimized, but does not consider scheduling and resource constraints.

A similar problem the allocation of real-time tasks to distributed processors, and different techniques are used to solve this problem; constraints programing is in [HCDJ08], simulated annealing in [BNTZ93, TBW92], branch and bound in [HS97, RRC03].

1.5 Contributions

The specific in-depth technical contributions of the thesis are (i) two methods for increasing accuracy and resource efficiency of WCET for embedded real-time components, and (ii) a method for allocating components to tasks for minimizing stack-usage and CPU-overhead, while maintaining real-time constraints.

The main contributions of the presented research are summarized as follows:

C1 A reusable WCET. The input space of reusable component is partitioned with respect to execution time, creating parametrizable component WCET contracts. A WCET contract is parametrizable and produces a WCET that is more accurate with respect to the specific usage. The result is that the WCET analysis can be reused with reusable components. The reusable WCET is evaluated with components from our industrial partners.

C2 Derivation of WCET input values. The input space of a component is divided with respect to component WCET, searching for an input combination that results in the execution of the worst-case path. The result can be used for guiding measurement-based WCET analysis. The method is evaluated with components from our industrial partners.

C3 A framework for allocating components to tasks aiming at minimizing resource consumption while maintaining real-time constraints. The framework calculates feasibility and fitness of an allocation. By exploring the state space of possible allocations, and comparing them to each other meta heuristic methods like genetic algorithms can be used. The framework is implemented with genetic algorithms, and evaluated with systems from our industrial partner.
<table>
<thead>
<tr>
<th>Contrib.</th>
<th>CBSE for ERTS</th>
<th>Resource efficiency and predictability</th>
<th>Development of ERTS</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>We create techniques for predicting WCET that can be reused through parametrization, yet with high accuracy.</td>
<td>Higher accuracy of predictions allows the developer to dimension hardware correctly.</td>
<td>ن</td>
<td>2 3 5</td>
</tr>
<tr>
<td>C2</td>
<td>Higher accuracy of predictions allows the developer to dimension hardware correctly.</td>
<td>ن</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>Helps to systematically transform components models to real-time models, something that otherwise often i performed ad-hoc.</td>
<td>Minimizing system overheads through efficient transformations from components to tasks, while at the same time maintaining both real-time requirements component architecture.</td>
<td>ن</td>
<td>6 7</td>
</tr>
<tr>
<td>C4</td>
<td>We integrate both WCET analysis and model transformation in the component based development process.</td>
<td>ن</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>Validation of the methods.</td>
<td>Validation of the methods.</td>
<td>ن</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: The relation between contributions and problem formulations

**C4** A *resource aware development process* that is an extension of the CBSE development process with the methods outlined in this thesis.

**C5** A *prototype tool* implementing the ideas from contributions C1 and C2. The prototype tool has a graphical user interface, graphically presenting WCET and BCET connected to inputs and component contracts. The tool supports several different heuristics for creating WCET contracts.

Table 1.5 presents how the contributions C1-C5 relate to the addressed problems as presented in Section 1.3

### 1.6 Thesis overview

The thesis structure is depicted in Figure 1.2. Note that the chapters 6 and 7 are shown as parallel in the thesis outline. This is because the solutions de-
scribed in Chapters 6 and 7 can be individually used for improving predictability and resource utilization for ERTS; however, they are synergistic when used together.

Chapter 1 introduces the reader to the particular problems this thesis seeks to solve. We discuss the motivation and objective of this thesis: to research and develop methods that facilitate component-based development in embedded real-time systems by increasing resource efficiency and analyzability.

Chapter 2 provides the reader a theoretical background to ERTS and gives a critical survey of the current state of research.

Chapter 3 provides the reader with a theoretical background to component-based development and give a critical survey of the current state of research.

Chapter 4 describes our research work and methods. We state and formalize the problem that we try to solve, and give an overview of how we solve the problem.

Chapter 5 gives an overview of the whole research, with both the allocation framework and the context-sensitive analysis framework. We briefly describe what the frameworks do and how they are positioned in the component-based development process.

Chapter 6 presents our reusable context-sensitive execution-time analysis framework. Here we describe the research and discuss and compare to similar or related work.

Chapter 7 presents our allocation framework. Here we describe the research in detail and discuss and compare similar or related work.

Chapter 8 describes examples and evaluations of each framework. In this chapter we discuss the results and their implications.

Chapter 9 summarizes the thesis by discussing the results, contributions and applicability of the research, and finally, discusses future work.

Appendix A presents an extended set of data and graphs.

Appendix B summarizes all publications, and reports written during the PhD studies.
Figure 1.2: An overview of the chapters in the thesis
Embedded real-time systems

In this chapter we give an introduction to *Embedded Real-Time Systems* (ERTS). We describe the terminology and definitions, used throughout this thesis.

2.1 Embedded systems - general concept

We do not need to search far to find an example of an ERTS in a modern everyday appliance. For example in a modern vehicle, the engine is controlled by an ERTS, measuring the airflow to the engine, pumping in just the right amount of fuel and igniting this in each cylinder at the exact right moment. The *Anti-Lock Brakes* (ABS) are controlled by an ERTS, continuously monitoring and controlling the brakes to ensure the maximum braking effect. In the unlikely case of a collision, an ERTS will detect the impact and deploy the airbag at exactly the right point in time [Cha02, Ros01]. What is common to all these systems is that they are parts of a bigger system and their actions have to be delivered at specified instants in time. If they fail to deliver their services at the right time, the consequences can lead to low performance, material damage or in the worst scenario, loss of human life.

There are several definitions of embedded systems. The definition given in Section 1.1 is quite vague since it only states that an embedded system is part of a larger system. Li and Yao [LY03] defines an embedded systems as follows:

*Embedded systems are computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function.*

-Time is an illusion. Lunchtime doubly so.-
-Douglas Adams
This definition includes the additional information that software is tightly coupled with hardware and that both are designed to perform a dedicated function.

Common to embedded systems is that they typically are characterized by a notion of embeddedness, i.e., it is not obvious that they are computers. An embedded system is a computer used to achieve a specific purpose and the computer is not the end-product itself. Such systems are typically found in, e.g., mobile phones, medical equipment, robotics, and, vehicular and automotive systems.

2.2 Real-time systems - general concept

ERTS usually control its environment by:

1. Observing the environment by reading sensors.
2. Making a decision by executing a control algorithm.
3. Affect the environment by writing a signal to actuators.

Real-time constraints can be split into two different parts, (i) how frequently the environment must be observed to get a coherent view of the real environment, and (ii) how quickly after each observation the environment must be affected in order to control the environment according to the control preferences, as depicted in Figure 2.1.

Figure 2.1: Simple real-time model
The observation frequency is designed into the system by triggering the software in an often predefined periodic pattern. There is a notion of that real-time has to be fast, but this is a misconception [Sta98]. It is only about “correct” time. The time scale can be seconds for some applications and micro seconds for some other application. Worst case, not average case matters. Not speed but predictability is the goal. The objective of fast computing is to minimize the average response time. The objective of real-time computing is to meet the individual timing requirement of each program.

A commonly given example of a hard Real-Time System (RTS) where predictability is very important, is an airbag in a car. An airbag system consists of sensors and an air bag control unit (ACU) which monitors a number of related sensors within the vehicle, including accelerometers, impact sensors, wheel speed sensors, gyroscopes, brake pressure sensors, and seat occupancy sensors. The different signals from the various sensors are fed into the Airbag control unit, and this determines the angle of impact, the severity, or force of the crash, along with other variables. Depending on the result of these calculations, the ACU may also deploy various additional restraint devices, such as seat belt pre-tensioners, and/or airbags including frontal bags for driver and front passenger, along with seat-mounted side bags, and "curtain" airbags which cover the side glass [MA95].

The greater the collision impact the earlier the airbag should be deployed because the airbag has to be deployed before an occupant moves forward 125 mm relative to the car. Normally it takes 30 milliseconds for an airbag to be deployed after it gets a trigger signal from the airbag sensor. Thus, an airbag sensor is to be designed in such a way that it can send a trigger pulse to the airbag deployment circuit 30 ms before the time when the occupant's head moves forward 125 mm with respect to the car. For a crash at 50 km/h the airbag should be triggered in the interval between 10 ms and 20 ms after the crash. Thus, timing is decisive to achieve maximum protection. The airbag must be opened in the right timing interval. If it opens too late, occupants could be injured. If it opens too early, they are not protected adequately, since the airbag then no longer has its ideal form on impact [WU93, Cha02].

A system is said to be real-time if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed. A definition that is commonly cited in literature is [Dou99]

Real-time systems encompass all devices with [temporal] performance constraints that, when violated, constitute a system failure of some kind.
Another more stringent definition is given by Stankovic [SR89]:

*Real-time systems are computer systems in which the correctness of the system depends not only on the logical correctness of the computations performed, but also on which point in time the results are provided.*

Both definitions agree on that time is a first class citizen and that the correctness of the system depends on both function and timing.

### 2.2.1 Classification

There are different types of RTS that are classified according to the criticality of a failure. Mission critical RTS, where a failure is considered to be a fault are denoted *hard*, while RSTs where occasional failures can be accepted are denoted soft.

**Hard real-time systems**

Hard RSTs are systems where the consequences of a failure to meet all constraint is a fatal fault. For hard real-time applications, the system must be able to handle all possible scenarios, including peak load situations. Thus, the worst-case scenario must be analysed and accounted for.

A RTS is said to be hard if completion after its deadline can cause catastrophic consequences [But97].

**Soft real-time systems**

Soft RTS are systems where some requirements may be violated to some defined extent. Late completion is undesirable but generally not fatal. Occasional missed deadlines or aborted execution is usually considered tolerable. The constraints are often specified in probabilistic terms.

A RTS is said to be soft if missing a deadline decreases performance of the systems, but does not jeopardize its correct behaviour [But97].

According to these definitions most RTS are soft, except some safety critical systems, such as airbags in cars. Systems where the consequences of failures are serious from the business point of view, are sometimes treated as hard RTS.
Another common classification of RTS is to distinguish how a task is activated, i.e., between time-triggered (TT) and event-triggered (ET) systems. Typically control functionality is by its nature time-triggered, i.e., the activation of its functionality is controlled by the progress of time.

**Time-triggered real-time systems**

Time-triggered systems are systems that are controlled by the progress of time. The system is often characterized by an enforced periodic activation pattern. [Kop91]

**Event-triggered real-time systems**

Event-triggered systems are systems that are controlled by the arrival of an event. A significant event is a change of state in an element of interest for the given purpose, e.g., a wheel speed sensor in an ABS system. These systems are characterized by aperiodic execution, with an unknown, and sometimes unbounded period time [Kop91].

### 2.3 Real-time model

Real-time software is often divided into so called tasks that execute a piece of software. Each task have some properties and requirements. The requirements are typically a periodicity for periodic tasks, a latest completion time (often referred to as deadline) and a priority, defining its execution order in a system with multiple tasks. Each task also has some properties, e.g., **Worst Case Execution Time** (WCET), and **Best-Case Execution Time** (BCET) and **Worst Case Blocking Time** (WCBT) in the case when shared resources are used.

The execution semantics are decided by the scheduling policy and the operating system. To be able to schedule the tasks they must conform to the specified rules of the scheduler and properties must be set, e.g., period, priority, deadline etc. To ensure that the software fulfils the stipulated requirements, real-time analysis has to be performed.

Time is important, what’s the problem, one might think? - The problem is manifold. First of all, the program languages used in most commercial software are C, C++, Java, to name a few. Most current de-facto standard languages do not have the notion of time built in to the language. Hence, ensuring timeliness requires complex validation procedures with analysis and simulations.
In the real-time domain there exist many theories, methods and tools. These methods use a number of real-time properties, such as WCET, execution period, deadlines, etc., and terms such as tasks and scheduling, in reasoning about timing and other related requirements.

A real-time model consists of:

- Task model
- Resource model
- Scheduling policy

### 2.3.1 Task model

The task model describes applications supported by system, and consists of:

- Temporal parameters
- Precedence constraints and dependencies
- Functional parameters

Tasks can be preemptable or non-preemptable. A preemptable task model is described in [But97]. A time-triggered, periodic task $\tau_i$ is described by (and depicted in Figure 2.2):

- A period, $T_i$, specifies the period of a periodic task, which is the time between an activation time $a_i$ and a finish time $f_i$. 

![Figure 2.2: Task model](image)
• Computation time, $C_i$, specifies the longest time it takes to execute the code of the task if it could run on the CPU uninterruptedly. To ensure that the software does not violate the longest allowed delay, the WCET must be known. The accuracy of the response time analysis is highly dependent on the accuracy of the WCET.

• Deadline, $d_i$, specifies a constraint on the completion time of the task. The task must finish no later than $d_i$ time units after it has been activated.

• Priority value, $v_i$ is a user defined integer value that represents the relative importance between tasks in the system.

An event-triggered, aperiodic, job $J_i$ is described by:

• An activation time $a_i$, which is the time when the event arrive at the system. Computation time, $C_i$, specifies the longest time it takes to execute the code of the task if it could run on the CPU uninterruptedly. To ensure that the software does not violate the longest allowed delay, the WCET must be known. The accuracy of the response time analysis is highly dependent on the accuracy of the WCET.

• Deadline, $d_i$, specifies a constraint on the completion time of the task. The task must finish no later than $d_i$ time units after it has been activated.

• Priority value, $v_i$ is a user defined integer value that represents the relative importance between tasks in the system.

The major difference between a periodic task $\tau_i$ and an aperiodic job $J_i$ is that $\tau_i$ is strictly periodic, triggered with a strict period time, whereas $J_i$ can arrive at the system at any time. In many systems periodic tasks have higher priority than aperiodic jobs.

2.3.2 Resource model

The resource model describes system resources available to applications. There are different types of resources, (i) Active resources, e.g., processor that executes tasks and communication networks, and, (ii) passive resources that are shared between tasks and may lead to blocking between tasks, e.g., shared inputs and outputs.

Usually each task need to allocate at least one active resource to execute, and the execution may depend on zero or more passive resources.
2.3.3 Scheduling policies

The scheduling policy defines how applications use resources at all times. A scheduling algorithm for ERTS aims at satisfying the timing requirements of the entire system functionality, i.e., meet all tasks deadline constraints, while minimizing the use of resources. There exist a wide variety of scheduling algorithms in the real-time literature. These can be classified in many ways, e.g., priority-based, value-based, rate-based, server algorithms [But97]. One common and coarse grained classification is based on when the actual scheduling decision, i.e., the decision of what task to execute at each point in time, is made. Scheduling that is performed before run-time is denoted off-line scheduling, and scheduling during run-time is denoted on-line scheduling.

Off-line scheduling

Off-line schedules are created and usually scheduled according to a time table. During run-time the dispatcher simply follows the table that was created before run-time. Off-line schedules can resolve complex constraints and requires no overhead during run-time. However, there is no flexibility for different load with respect to, e.g., aperiodic tasks (events).

On-line scheduling

On-line schedulers make decisions during run-time as opposed to off-line schedulers. This gives a penalty during run-time in terms of calculation overhead for deciding which task to be scheduled at any given time. On the other hand, on-line schedulers can implement more advanced features such as resource reclaiming in the case that the actual execution time of a task is lower than the predicted worst-case [FMAk03, BBB04]. The reclaimed resources can be used for executing aperiodic tasks or lower processor speed for power saving.

In this thesis we consider only priority-based real-time systems.

2.4 Real-time analysis

Real-time analysis is the method that is used analytically for determining if the system will behave according to the timing requirements that have been stipulated for the system.
2.5 Schedulability analysis

A task set is said to be schedulable if a schedule can be found which guarantees that all tasks will meet their timing constraints under all circumstances. Schedulability analysis aims to before run-time determine whether a task set is schedulable or not. For most real-time scheduling algorithms some kind of schedulability analysis test is available [But97]. In static scheduling, the schedulability analysis is combined with the construction of the schedule, a so called proof by construction approach. That is, if a schedule which fulfils all timing requirements and constraints can be constructed, the system is, by definition, schedulable.

There exists three different types of approaches for pre run-time schedulability analysis, utilization based, demand based and response-time based.

2.5.1 Utilization based analysis

In [LL73], Liu and Layland presents a utilization based test for determining the feasibility of a task set. Utilization based analysis is a fast but coarse grained analysis, that will guarantee that a task set is schedulable, i.e., the scheduling analysis is sufficient. However, in some cases when utilization based analysis reports that the task set is not schedulable, it may in fact be schedulable, i.e., the analysis is not necessary. The analysis is only valid for task sets where the deadline equals the period time ($D_i = T_i$).

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$$

The utilization $U$ is equivalent to the sum of the ratio between the execution-times and the periods of all tasks in the system.

$$U \leq 0.69 \text{ as } N \to \infty$$

Note that, as the number of tasks in the system approaches infinity, the system can be guaranteed to be scheduled if the utilization is less than or equal to 69%.

2.5.2 Demand based analysis

Processor demand, introduced by Baruah et al. in [BRH90] is a measure that indicates how much computation that is requested be the system’s task set, with
respect to the stipulated timing constraints. The processor demand \( h(t_1, t_2) \), over the time interval \( t \in [t_1, t_2] \), is given by

\[
h(t_1, t_2) = \sum_{t_1 \leq r_k, d_k \leq t_2} (C_k)
\]

where \( r_k \) is the release time of task \( k \), and \( d_k \) is the deadline for task \( k \). The processor demand is the execution time of all tasks that have their release time and deadline within the interval \( [t_1, t_2] \).

### 2.5.3 Response time analysis

Real-time research on schedulability in fixed priority scheduled systems has resulted in a wide variety of research results. Several different schedulability-analysis techniques for fixed priority systems exist [MT05, But97]. The most powerful approach, that provides the highest obtainable utilization and is able handle the most expressive task models, is to use Response-Time Analysis (RTA).

Joseph and Pandya presented the first basic RTA for the simple Liu and Layland task model [MJ86].

In addition, the following assumptions must hold in order for the analysis to be valid:

- Tasks must be independent, i.e., there can be no synchronization between tasks.
- Tasks must not suspend themselves.
- Deadlines must be less or equal to corresponding periods, i.e., \( D_i \leq T_i \).
- Tasks must have unique priorities.

The following formula determines the worst case response time, \( R_i \), of task \( \tau_i \):

\[
R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lfloor \frac{R_j^n}{T_j} \right\rfloor C_j
\]

Here task \( i \)'s worst-case response time, \( R_i \), is calculated first and then checked (trivially) with its deadline.

Starting with \( R_i^0 = C_i \) and iterating until \( R_i^{n+1} = R_i^n \) is guaranteed to yield the smallest possible solution and thus the response time for \( \tau_i \) [SH98].
In order to guarantee convergence either one must ensure a total task utilization is not greater than 100% or one can stop iterating when \( R_{i}^{n+1} > D_i \), i.e., a deadline violation has occurred.

### 2.5.4 Transactions

Transactions are collections of related tasks, which collectively perform some function or have some shared timing attributes. A transaction \( tr \) usually has a timing requirement, i.e., an end-to-end deadlines. A transaction usually has a period which denotes a lower bound on the time between re-arrivals of the transaction. Unfortunately exact analysis is computationally infeasible to evaluate for task-sets with transactions\(^{[Tin94]}\); hence some other approach has to be used. To use exact analysis the schedule has to be simulated over the hyper-period (LCM of all periods). A common approach is schedule simulation \(^{[Aud91]}\). In, e.g., \(^{[RT02, MT05]}\) fast methods for calculating response-times for task sets with transactions with advanced timing-properties such as offsets and jitter are presented.

### 2.5.5 Attribute assignment

For a real-time schedule to be feasible, task attributes have to be set accordingly. Several publications exist on the matter but many of them, e.g., \(^{[GHS94, Yer96]}\), are not very straightforward and difficult to use. That is because they are difficult to justify and they assume that all attributes are changeable. A more straightforward approach is the one by Bate and Burns \(^{[BB99]}\). Timing requirements such as \( \text{Period, Deadline, Jitter and Separation} \) are considered. Furthermore, transactions are sequences of tasks executing in a fixed order. The timing requirements for transactions are \( \text{Period, End-to-End Deadlines and Jitter} \). Bate and Burns use an iterative approach by considering subsequent instances of tasks within one transaction and derive the attributes from the iterative process. Their approach is somewhat similar to schedule simulation.

### 2.6 Worst-case execution time analysis

One very important part of the real-time analysis is the WCET analysis, that determines the longest time a piece of software will execute.

Reliable WCET estimates are a fundament for most of the research performed within the real-time research community. They are essential in real-
time systems development in the substantial step of creating schedules and to perform schedulability analysis, to determine if performance goals are met for tasks, and to check that interrupts have sufficiently short reaction times [Gan06].

The WCET is defined as the longest possible execution time of a program that could ever occur, on a specific hardware platform. There are different types of WCET analysis. Common for all types is that they should produce the an estimation of the longest possible time for executing a program on a specific hardware platform. Because of high complexity, WCET analysis tool are often not able to estimate the exact WCET. In those cases the WCET estimate should preferably be a safe, yet tight, over estimation.

### 2.6.1 Classification of WCET analysis

The different types of analyses are divided into static WCET analysis that performs a static analysis of the source code, producing an estimate of the execution time that is sure to be not less than the actual WCET - it is said to produce a safe over-estimation. Measurement based WCET analysis that measures the execution time during program execution. Measurement based analysis will report the longest observed execution time. Often, however, the absolute worst-case has not been observed (Figure 2.3). Hybrid WCET analysis that uses both static and dynamic analysis to get a tight WCET. A few approaches to parametric WCET exist where the WCET is expressed as a relation between the WCET and input parameters (possibly also hardware parameters). However, many of
them suffer greatly from exponentially increasing complexity with respect to the program size.

Static WCET analysis

A static WCET analysis derives WCET estimates without actually running the program. Instead, it takes into account all input value combinations, together with the characteristics of the software and hardware, to derive a safe WCET estimate. The analysis is commonly subdivided into three phases [Erm03, WEE’08]:

- **flow-analysis**: where bounds on the number of times different instructions can be executed are derived,
- **low-level analysis**: where bounds on the time different instructions may take to execute are derived, and
- **calculation**: where a WCET estimate is derived based on the information derived in the first two phases.

Due to the inherent complexity of modern software and hardware, it is not always possible to statically deduce the exact behaviour of a program. In these cases conservative approximations are made, e.g., a loop bound flow-analysis can report a larger loop bound than what is actually possible, or a low-level cache analysis can classify a memory access as a cache miss even though it always may result in a cache hit.

Some static WCET analyses are input-sensitive, meaning that they are able to take constraints on possible input variable values into account when calculating the WCET estimate. In general, such analyses should be able to derive more precise WCET estimates than non-input-sensitive ones.

Measurement based WCET analysis

A measurement-based WCET analysis executes the program on the hardware for some input value combinations, using some type of time measurement facility, such as oscilloscopes, logical analysers, or hardware trace mechanisms to derive the timing of the program or parts of the program [BCP02, BCP03b, BCP03a, WRP05, WKE02]. Since it is impossible for most programs to test all input value combinations, often only a subset of the possible executions are tested.

---

1Given that a cache miss always produces longer execution times than that of a cache hit (which may not always be true for all architectures)
run, hoping that the selected subset will include the WCET input value combination. If not, this may lead to dangerous underestimations of the WCET. The selection of test cases to reach the best path coverage is therefore crucial when using hybrid methods. An advantage of the hybrid approach may be that selection of test cases and control of coverage are well-known techniques in software engineering.

Hybrid WCET analysis

*Hybrid* WCET analysis methods combine measurement-based and static WCET analyses. Usually measurements are used to extract timing for smaller program parts, and static analysis to deduce the final WCET estimate from the program part timings. Examples of hybrid tools are RapiTime [Rap08] which is built on a probabilistic WCET approach [BCP02] and SymTA/P [Sym08]. There is a possibility that the hybrid methods underestimate the WCET, since the WCET estimate is based on measurements, and measurements may exclude the worst case path. At the same time, hybrid methods may also overestimate the WCET, since measurements from mutually exclusive parts of the program may be combined in the final WCET. For instance, the tool RapiTime is able to either analyse source code, adding instrumentation points on the source code level, or, otherwise use binary readers and instrument the generated code.

Parametric WCET analysis

Parametric (or symbolic) WCET analysis derives a formula for the execution time, expressed in parameters of the program, rather than just a single number. The parameters can be either external, or internal like a symbolic upper bound to a loop count. A parametric WCET formula contains much more information than just a single WCET estimate, and it can be used for applications like online scheduling of tasks where parameters are unknown until runtime, or to find which parts of a code that has the strongest influence on the WCET.

There are a few approaches where the WCET is expressed as a formula with respect to loops. For example Vivancos and Coffman [VHMW01, CHMW07], and, Bernat and Colin [BB00, CB02] present techniques that mainly parametrize loop bounds. However, in many cases it is necessary to be able to express excluding paths and infeasible paths in order to get a tight and parametric WCET.

Parametric WCET has been proposed by many researchers within the WCET community but there are still very few parametric WCET methods developed. In [Lis03] Björn Lisper outlines a technique for fully automatic parametric
WCET analysis, which is based on known mathematical methods. In a MSc thesis [Alt06, Hüm06] Altmeyer and Hümbert outlines a method inspired by Lispers work. Their work has been developed and tested with the aiT tool [aiT]. Current work on the methods outlined by Lispers is also presented in [BL08, AHLW08].
Chapter 3

Component-based development for ERTS

In this chapter we give an introduction to Component-Based Software Engineering (CBSE) terminology and definitions. We also discuss the industrial motivations for using CBSE.

3.1 Motivation

CBSE in general is the emerging discipline of the development of software components and development of systems incorporating software components [CL02a]. It is a promising approach for efficient software development, enabling well defined software architectures as well as reuse. Component technologies have been developed addressing different demands and domains, the most common technologies are perhaps Enterprise Java Beans and Java Beans from SUN, COM and .Net from Microsoft, and technologies implementing the CORBA standard as is from OMG. These technologies are used for desktop and distributed enterprise applications all over the world. However, these technologies are in general not used for all classes of systems. They are not used for (i) resource constrained systems; they are simply too demanding both in computing power and memory usage. They are not used for (ii) safety critical systems; it is hard to verify the functionality due to complexity and black box property of components. They cannot be used for (iii) real-time systems since they rely on unpredictable dynamic bindings and other complex run-time

\[\text{It is a mistake to think you can solve any major problems just with potatoes.}\]

-Douglas Adams
mechanisms. Embedded Real-Time Systems (ERTS) can often be classified as combinations of (i), (ii) and (iii).

Adoption of component technologies for the development of ERTS is significantly slower than for, e.g., desktop and business systems. Major reasons are that ERTS must satisfy requirements of timeliness, quality-of-service and predictability. Also these types of systems may have severely constrained resources (memory, processing power, communication). The widely adopted component technologies do not in general address timeliness, quality-of-service or similar extra-functional properties that are important for embedded real-time systems.

Component technologies have been developed for particular classes of ERTS. Often, these have been developed within some organizations, and their adoption outside that organization is limited. To avoid heavy-weight run-time platforms, these component-technologies mostly do not support run-time deployment of components and lack many services. Composition of components into a (sub)system is often performed in the design environment, prior to compilation, enabling static prediction of system properties and global optimizations. Examples of such technologies include the Koala component model for consumer electronics [vOvdLKM00], PECOS for industrial field devices [GCW+02, NAD+02, GCS+02], and PBO for robotics [SVK97]. Such component technologies are often tightly coupled to a specific operating system or a specific domain and only few of them consider non-functional properties. For these reasons they have not been general enough to be adopted for use in other domains [MÅFN03].

The life cycle of ERTS produced by the electronic and software industry is continuously shortening due to the acceleration of technologies and cutting time-to-market. ERTS are integrated into the products in many technology areas. The decreasing time to market leads to that software is required to be flexible enough for rapid reuse, extension and adaptation of system functions. In today’s highly competitive market, electronics Original Equipment Manufacturers (OEM) are faced with new software technologies that are introduced rapidly and just as rapidly become obsolete. Companies look for guidance when developing products and services that enable faster access to the components they use in their designs - they look to accelerate their time-to-market.

In fact time-to-market is so important that companies release products and software before they are finished developing the products. One specific example is the Popcorn hour media streamer, Figure 3.1 [Pop08], which is delivered with a “last minute note”, declaring that some functionality is unimplemented at delivery due to the requirements on short time-to-market.
Last minute notes:

A. Due to short time-to-market, some features may not be available yet and will be enabled in future firmware updates. Known missing features:

1. BT suspend key
2. File Operation key
   - For File Copy, Move, Rename
3. NFS server

B. If for some reason you are unable to update the firmware or NMT apps online, please visit www.popcornhour.com to download the firmware/NMT apps image and do the updates from USB thumbdrive.

C. Due to additional functions enabled via future firmware update, new Setup pages may differ from the one portrayed in the Quick Start Guide.

Date: November 1, 2007

Figure 3.1: Last minute notes
3.2 Component reuse

One of the more important component properties is unquestionable reuse. It is commonly accepted that reuse, if used properly, increases productivity and lowers development costs [Gri93, Jor97]. When software become complex, software reuse becomes more interesting due to the fact that software is "soft". By this we mean that it is easy to create tailored software that exactly fulfills all system requirements; however, as software become more complex the costs and efforts for re-creating the software also increase, and the benefits of reuse become more apparent.

To facilitate reuse, an important distinction between traditional software development and CBSE is that individual components are not specified and laid out according to existing other components that are supposed to integrate their services. Every single component is specified according to a more or less general requirements profile, so it can be reused and integrated in a number of different contexts. Generality is a key feature of components to facilitate reuse in many different contexts. Some of the benefits of reuse are:

- **Lower defect density** quantitative studies have shown that reused software components have significantly lower defect-density than non-reused software components [LGA+07, Moh04, BBB+00, BBCD+00].

- **More stable code** quantitative analyses has shown that the amount of modified code between releases is less in reused software components, than in non-reused [BBB+00, BBCD+00].

- **Increased reliability** The quality of the reusable components improves and it becomes more stable over several releases [LGA+07, BBB+00, BBCD+00].

- **Reduced time-to-market** Even though reuse requires a greater initial effort the benefits in time of reusing is often greater [Gri93, BBB+00, BBCD+00].

- **Reduced development costs** shorter development time, and therefore lower development cost is possible due to reuse of company assets, e.g., specialists knowledge. Several companies have reported reduced time and cost by reusing software [Jor97].

3.3 Basic definitions in CBSE

Outside the CBSE community, there is often confusion about the basic terms. The foundation of component-based systems is naturally the **component**. A
software component is a software entity that conforms to a component model and can be composed without modification [CL02a]. The term component model embraces the specification of components, how components are assembled, and the component framework. With other words, the component model is a set of rules governing how the components may or may not be used. The composition of components is the process of assembling components to form an application. Components are composed by constitute systems through connecting their interfaces according to the rules defined in the component model. The component interface is the entry to the component functionality. A component composition is executed in the context of a component framework. The component framework provides the necessary run-time support that is not provided by the underlying run-time system, e.g., scheduling. A component technology is the concrete implementation of a component model with the supporting tools, guidelines and imposed design constraints that a practitioner of CBSE deals with.

3.4 CBSE development process

An important distinction between traditional development and CBSE is that the CBSE process is divided in two parts: a system development process and a component development process [CCL06]. The interface between these two processes may be fairly complex. First during system development, existing components may be examined already during the requirements phase (and influence the entire scope and direction of the system). Later components are tested to assess functionality and quality characteristics; they are used in prototyping during design, and finally integrated and deployed with the system. And conversely, requirements on the system may also affect the evolution of its constituent components more or less directly (depending on the business relationship). Figure 3.2 shows a general model for CBSE processes.

3.5 Component model

The only way that a component can be distinguished from other forms of packaged software is through its compliance with a component model. However, no agreement on what should be included in a component model exists, but a component model should specify the standards and conventions imposed on developers of components. Common is that component models deals with different abstractions, component types, interaction schemes between components and
Figure 3.2: CBSE development process
clarifies how different resources are bound to components. Important parts of a component model are consequently:

- Component definitions
- Component interaction
- Component interfaces
- Component composition
- Component contracts

### 3.6 Component definition

The key concept of CBSE is that of software components that can be assembled into larger components or final products. One of the most influential definitions of software components is that of Szyperski [Szy98]

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Szyperski states that a component should be a unit of composition, meaning that the only visible part should be the interfaces. Furthermore, the interfaces should be contractually specified with respect to the interfaces and contextual dependencies - meaning that the component must be well documented. Szyperski also asserts that source code modules do not qualify as software components since they make it possible for the composer to rely on implementation details, thus violating the principle of black-box composition. The definition states that it should be possible to market software components as independent products and that buyers should be able to use them as parts in their own products. Naturally, independent deployment also has technical implications, namely that it must be possible to deploy (or upgrade) a single component without any modification, recompilation, or similar of the rest of the systems of which the component is a part.
In [HC01] Heineman and Councill present the following definition of software components:

*A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*

Heineman and Council states that a software components must conform to a component model, but does not say anything about requirements on the component model. The two definitions principally agree, since the requirement that components can be modified without modification can only be satisfied if interfaces and context dependencies are well defined and that compliance with a standard naturally supports composition by third parties.

In [Lüd06] Lüders discusses an alternative component definition from the UML2.0 standard and its relation to the other definitions:

*A component is a modular unit with well-defined required and provided interfaces that is replaceable within its environment. The concept can be used to model both logical and physical components.*

Lüders [Lüd06] states that the definition is somewhat broader than the previous two, as “replaceable within its environment” is a weaker requirement than “subject to independent deployment and composition by third parties”. This terminology is also used by Crnkovic and Larsson [CL02a], who define a software component as consisting of at least the following elements:

- A set of interfaces provided to, or required from the environment. These interfaces are particularly for interaction with other components, rather than with a component infrastructure or traditional software entities.
- An executable code, which can be coupled to the code of other components via interfaces.

From these previous definitions we conclude that components need to be:

**Standardized.** Component standardisation means that a component that is used in a CBSE development process has to conform to some standardised component model. This model may define, e.g., component interfaces, component meta-data, documentation, composition and deployment.
3.6 Component definition

**Independent.** A component should be independent, i.e., it should be possible to compose and deploy a component without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a “requires” interface specification.

**Composable.** For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself such as its methods and attributes. For ERTS a component must also provide quality of service information with respect to, e.g., execution time and resource consumption.

**Deployable.** To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed. For ERTS this may be a too stringent requirement.

**Documented.** Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics, and quality of service, of all component interfaces have to be specified.

### 3.6.1 Component interaction

Component interaction is the rules for how components can communicate, and how components can be assembled. Components must follow a common interaction model defined by the component model. The interaction models supported by the component model influences the architecture of the systems that are built with the component technology. A few common interaction models are:

**Pipes and filters.** The components in this style are called filters and each have a set of inputs and a set of outputs. The outputs of a filter can be attached to inputs of other filters via simple connectors called pipes. Typically, the filters transform streams of input data to streams of output data in an incremental fashion. An important constraint is that filters should be independent in the sense that they do not share state and each filter is unaware of the identities of the other filters it is connected to.
Black board. The basic model of a blackboard system is composed of three main entities: the blackboard, a set of knowledge sources and a control mechanism [CL92]. The blackboard is a globally accessible database, which is shared by knowledge sources. It contains the data and intermediate solutions. The blackboard is structured as a hierarchy of abstraction levels, which determine where data is input and where solutions are collected. Partial solutions are associated with each level and may be linked to information on other levels.

Client server. Client/server computing systems are comprised of two logical parts: a server that provides services and a client that requests services of the server. Together, the two form a complete computing system with a distinct division of responsibility. More technically, client/server computing relates two or more threads of execution using a consumer/producer relationship.

3.6.2 Component interface

CBSE relies heavily on interfaces. They must handle all properties that lead to inter-component dependences, since the component implementation is hidden to a developer (black-box property). Indications show that although interfaces are familiar and has existed for several years, CBSE may require more of an interface than earlier applications. In [WBE], Weck et al. states that an interface is “a collection of service access points, each of them including a semantic specification.”

A component realizes an interface by providing services or entry points for data and control, this interface is called provided interface. A component requires services from or pass data to other components, thus it is not reasonable for a component to only provide services. Required services need to be specified in a required interface. A required interface specifies services that are required by, or data and control passed to, other components.

Provided interface. Defines the services or data and control entry points that are provided by the component.

Required interface. Defines the services that must be made available by, or data and control passed to, other components.
3.6.3 Component composition

Composition is to bring together components so that they give the desired behaviour. The possibilities for composition should be defined by the component model [Lar04]. Typically the possible interaction patterns are component to component, component to framework and framework to framework. Under composition, resource binding are also treated, in terms of early or late. It is during composition the system is formed and it is at this moment most predictions of run-time properties can be done by supporting tools.

3.6.4 Component contracts

A contract is a specification of obligations of a component. There are several types of contracts for software components. They have in common that they specify some expected behaviour or property of the component. A commonly cited classification of contracts is one by Beugnard et al. [BJPW99], where four levels of contracts are defined, namely:

**Syntactic.** Conformance of functionality.

**Behavioural.** Behavioural contracts describes the behaviour of the component with respect to the interfaces. Behavioural contracts can be realized with textual descriptions, formal methods or component pre and post conditions. Behavioural contracts can also specify certain context dependencies.

**Synchronization.** Synchronization between components.

**Quality of service.** In contrast to the behavioural contract the quality of service contract describes the performance of the component and specifies non-functional properties such as timing aspects, memory consumption required by component and system analyses.

3.7 Component technology

A component technology is a concrete implementation of a component model and consists of tools and models for supporting assembling of, and interoperation between components. A component technology should provide necessary run-time support for the components and mature component technologies often offer different development tools simplifying the engineering process [CL02a].
3.8 Component frameworks

A component framework is based on a software architecture, a set of components and their interaction mechanisms. It provides the run-time mechanisms required by the component model, and that are not provided by the underlying run-time system. Thus, a component framework can be imagined as a small operating system that offers the services that components require [CL02a].

3.9 Summary

During the last decade advances have been made in component-based development for desktop and internet applications. A few de-facto standards have completely transformed the way such software is developed. These standards are mainly Microsoft’s .NET, SUN’s Enterprise Java Beans and OMG’s Corba Component Model (CCM). Component models for embedded systems are usually designed with very domain specific requirements in mind [MÅFN05]. For instance the well known component model by Philips, Koala [vO02], considers low resource usage, but does not consider, e.g., real-time properties that are important in many embedded domains. There is a large set of different component technologies that approach different problems in different ways such as ABB’s PECOS [GCW+02], Rubus [LLL03, HMTN+08, HMTS+08], which originates around research on Basement [HLB+97], SaveComp [ÅCF+07] and many more. None of these however have yet been successful outside of their intended domain. Thus, for ERTS it seems difficult to define de-facto standards due to highly diverging requirements on different industrial segments [Crn04].

One of the more important component properties is unquestionable reuse. It is commonly accepted that reuse, if used properly, increases productivity and lowers development costs. However, support for reuse requires generality of components which often leads to low accuracy of component properties which is enough for desktop systems. But for ERTS low accuracy of component properties leads to low resource efficiency, and low resource efficiency leads to higher manufacturing costs in terms of hardware resources. On the other hand, lack of support for reuse increase development costs and increases time-to-market.

Reusable components should, by definition, be used in different applications [Crn02], i.e., they should be context unaware. All possible deployments are not known and the extra-functional behaviour of components in a new deployment is often very hard to predict. This is not a problem for desktop appli-
cations where resources are abundant and the requirements on, e.g., timing and safety are relatively low. Few component models support general component properties at the same time as they are highly resource and run-time aware, and vice versa. Component models that are specifically designed for a particular group of systems are often not adaptable and general enough to be used in other systems or other domains. In order to achieve reuse, most component technologies of today intentionally do not consider the system context, e.g., inputs, hardware and run-time system. As a result performance prediction is often inaccurate.
Chapter 4

Research problem

In this chapter we describe the industrial and research problems considered, and we discuss our solutions and the research methodology we have used.

The research problem stems from (i) the desire to lower development costs and shorten time-to-market by using Component-Based Software Engineering (CBSE), and (ii) the requirements on resource efficiency and predictability in Embedded Real-Time Systems (ERTS). The general and overarching question that we ask is

*How can efficient reuse of software components be realized, while fulfilling the requirements of resource efficiency and predictability in embedded real-time systems?*

We do not aim to fully answer this question, but we will explore the underlying problem domain and provide knowledge that contributes to the partial answering of the question.

4.1 Research introduction

One of the most important aspects of ERTS is that they must exhibit a predictable timing behaviour; furthermore, these systems are often embedded in larger systems where resources are scarce. In order to meet the challenges of predictability and resource efficiency in the increasing complexity of embedded software, developers need appropriate development and analysis tools. One of the development strategies that industry is interested in is CBSE, mainly

Once you do know what the question actually is, you’ll know what the answer means...

-Hitchiker’s guide to the galaxy
for its focus on structured reuse. However, because of lack of models and tools for analysing predictability and resource consumption in relation to components, CBSE has not yet been as successful in the embedded domain as in the desktop and web system domains.

Several component technologies for ERTS have been developed. Examples of component technologies for particular classes of ERTS are PECOS [GCW+02], PBO [SVK97], RubusCM [LLL03], SaveCCT [ACF+07]. There has been less focus on developing real-time tools and analyses for particular classes of component-technologies, which is the focus of our research.

4.2 Specific research goal

We believe that CBSE will be one of the main future development strategies for software. The increasing complexity of software is an incitament for reuse, because the effort to re-create the software will become higher and higher. For most systems in the industrial segment of embedded systems it is not enough to only consider correct functionality. Even if the functionality is reused it still requires a lot of work to re-analyze the software in order to be able to generate correct extra-functional behaviour on the software.

The specific goals of this thesis are to

- provide means for reusable WCET analysis for reusable software components
  - specify WCET with respect to input with higher precision than compared to traditional WCET for reusable software components.
- provide methods for efficient allocation of components to real-time tasks.
  - reduce system CPU and memory overhead while maintaining real-time requirements.

4.3 Research method

In this section we start by clarifying our view of research, research methods, and what kind of research we have performed.

Research is the systematic collection, analysis and interpretation of data to answer a question or solve a problem [Moh06]. Research relies on a research
method that guides the research. Research is often classified into two different categories, viz., basic research and applied research. Applied research is undertaken with the intention of applying results or previous research to an identified problem. Our research is focused on real problems; we solve a real problem and try to generalize the solution. This type of research is a mix between applied and basic research and is sometimes referred to as frontier research [Har05].

The starting point of our research has been selected from the study of industrial problems. Thus, early on in the research we performed qualitative studies [MÅFN04, ÅFSC04], trying to understand the problems in the embedded systems industry. The studies are slanted towards the heavy vehicular domain. As with most real problems they are complex, difficult to grasp. Thus, we have broken down the real problem in smaller, understandable sub problems. To support the validity (construct validity) of our solutions, we deduce a set of requirements from the research problems to make sure that we really approach the problem we intend to solve.

![Diagram: Transformation of industrial problem to academic problems in a reduced academic setting.](image)

Figure 4.1: Transformation of industrial problem to academic problems in a reduced academic setting.

Good research requires not only a clearly stated problem and results, but also convincing evidence that the results are sound. In order to fulfil these criteria and approach the problem methodically, our research method is divided it into the following five steps inspired by Shaw [Sha01]:

1. We identify the industrial problem and derive five problem statements **S1-S5.** (Section 4.4)

2. We transfer the industrial problem to an academic setting where a set of
50 Chapter 4. Research problem

assumptions that restrict the academic problem are identified. Further, we formulate three specific academic research problems \( P_1, P_2 \) and \( P_3 \), based on the assumptions and the problem statements \( S_1-S_5 \), as depicted in Figure 4.1 (Section 4.5).

3. We devise general solutions to the different problems in the form of requirements. (Section 4.6).

4. We describe specific partial solutions (Section 4.7).

5. We validate the specific solutions with respect to the corresponding industrial problem (Section 4.8).

The structure of this chapter follows a logical reasoning that leads from an industrial problem \( \rightarrow \) problem statements and assumptions \( \rightarrow \) research problems \( \rightarrow \) requirements \( \rightarrow \) solution proposals \( \rightarrow \) validation. This logical chain is depicted in Figure 4.2.

In Section 4.4 we describe the industrial problem. In Section 4.5 we derive three research problems. We refine the research problems in Section 4.6 to form a set of requirements. We continue by proposing partial solutions to these requirements in Section 4.7, and finally in Section 4.8 we describe the validations of the solutions.

In the following sections we describe how the industrial problems are transferred to a set of research problems.
4.4 Industrial problems

In this section we describe the studied industrial problem in general together with the specific part that we consider in this thesis.

Some of the embedded industry domains are sensitive to resource consumption. In segments like consumer electronics, where the product cost and time-to-market are two of the main competitive factors there is not room for an increase of hardware costs due to increased resource consumption.

Many companies in the embedded systems domain view CBSE as a promising approach to more efficient software development. However, several key issues have not yet been solved for CBSE, specifically relating to the demanding requirements industry is facing on resource consumption, timeliness and reliability [BCC'03].

New legislation for machinery (2006/42/EEC) [SAE06] has forced several domains that before were exempted from the machine directive to follow the new directives. The machine directive is ultimately about product safety, and the safety aspects must be integrated in all aspects, from construction to use. For machinery that are controlled by software correctness of the software with different type of software analysis, e.g., WCET analysis need to be proven.

4.4.1 General industrial problem

The industrial problem we discuss exists in the realm of ERTS. Within this realm industry is facing problems with increasingly complex software [HKK04, But06, PBKS07, HMTN06]. Issues like pressure to decrease time-to-market and increasing complexity rapidly increases the cost for designing, developing and testing the software; and more often the development involves domain experts whose time is very expensive. This has lead to an increasing desire to reuse software [Bro06, Gil05, Acc06].

Due to these issues, many companies are looking for new development strategies that can handle these problems [But06]. CBSE is an approach that promises remedy for increasing complexity, long development times and costs. Structured reuse is the main activity that provides the promised benefits [HKK04]. CBSE has been proven to be useful in several domains, such as desktop, internet and business systems. Compared to these domains, the ERTS segment struggles with more stringent requirements on, e.g., timing and resource consumption. In order to bring CBSE and ERTS closer, CBSE needs to support such requirements [HKK04].
Problem statement S1: The efficiency of reuse has been proven mainly outside of the embedded and real-time systems domain.

One of the classes of requirements that are imposed on embedded real-time systems are real-time requirements. ERTS have historically mostly been used in mission critical systems, such as automotive, factory automation or aerospace. However, the use of real-time systems has become widespread also in machinery and consumer electronics to satisfy increasingly demanding customers [Bro06, PBKS07].

Real-time requirements origins from the fact that embedded software needs to interact with a physical environment. This has created a need for being able to analytically reason about the software behaviour. For safety critical systems in particular it is important to analytically show that the system will behave correctly. Examples of well established uses of such analytical properties is the increasing use of WCET analysis, which is a critical activity for proving the correctness of real-time systems [WEE08].

Problem statement S2: Without real-time analysis it is difficult to prove that the embedded real-time systems will react correctly and predictably to its physical environment.

4.4.2 Specific industrial problem

In this section we further limit the general industrial problem to form a more narrow industrial problem that we can use as a basis for our research.

When developing embedded RTS it is of main concern to make accurate predictions of component properties such as timing and memory consumption. At the same time it is desirable to gain from the development benefits offered by CBSE. The key activity that will bring the CBSE benefits is reuse. In order to support reuse in the CBSE development process, components need to be made general. Reuse traditionally only considers functional parts, leaving timing prediction to the system developer. Hence, to facilitate reuse, components are made unaware of their context, i.e., their usage and environment, which makes it difficult to perform accurate timing predictions [HKK04, PD96, But06, BCC03].

Problem statement S3: The efficiency of software component reuse is lowered if software components have too much context-specific dependencies.
By reusing traditional WCET analysis results of a component, reuse is increased, but the analysis must be performed with respect to all possible contexts, resulting in inaccurate estimations for most contexts. On the other hand, by analysing a specific system with the knowledge of the system context, the analysis results are more accurate estimations, but the key property reusability is decreased since the predictions are not reused. Also timing prediction is a difficult and time-consuming activity, resulting in a higher development effort compared to if the predictions are reused [KP03, HK07].

**Problem statement S4:** Without context-specific knowledge it is difficult to accurately predict the behaviour of software components.

An obstacle to combine predictability with correctly dimensioned hardware is the inaccuracy of the system analysis. Real-time analysis is based on worst-case assumptions, and the composition of overestimated worst-cases make the system impractically oversized and under utilized [Dur06]. Few component technologies methodically consider the allocation of components to real-time tasks, leading to worse than possible resource utilization. A faulty allocation from components to real-time tasks can even lead to violated real-time requirements [HMTN06, FSÅ05].

**Problem statement S5:** Poor allocation of components to real-time tasks may lead to inefficient resource utilization or violated real-time requirements.

From the problem statements S1-S5 we see that there are several trade-offs between the generality required by components for efficient reuse in the CBSE process, and the specific usage required by embedded RTS for achieving accurate predictions required by real-time analysis; and between resource consumption and real-time constraints when allocating components to real-time tasks.

### 4.5 Research setting

We use Shaw’s classification of software engineering research paradigms in terms of research settings and products/approaches [Sha01] to characterize the work in this dissertation. The research settings of this work, according to Shaw’s definitions, are characterization and methods/means. Thus, the corresponding questions are:
Characterization: what are the important characteristics for increasing resource efficiency and predictability for reusable software components in embedded real-time systems?

Methods/means: how can we accomplish increased resource efficiency and predictability for reusable software components in embedded real-time systems?

We create a research setting that is a simplification of the industrial problem by stating a set of assumptions.

By making assumptions the industrial problem is reduced to a simpler problem that can be tackled easier. In subsequent research, these assumptions can be relaxed to make the problems and solutions more generally applicable.

4.5.1 Limiting the problem using assumptions

We have identified a set of assumptions that we make explicit. The number of assumptions can be made large but we have chosen a subset of assumptions that we believe are important. There is no other rationale for choosing exactly these assumptions than that we believe that they are important for reducing the complexity of the problem.

Many small embedded systems do not have complex interaction models, but rather require models that are simple and analysable. Therefore several component technologies for ERTS. For instance SaveCCT [ÅCF+07], PBO [SVK97] and Pecos [GCW+02]) have chosen to limit the interaction model to the pipes and filter model. Hence, we limit the problem to only consider component technologies that use the pipes-and-filter interaction model.

Assumption: Only pipe-and-filter interaction is used between components.

In this research we do not aim to develop yet another WCET analysis tool, and therefore we use existing tools. In this research we have used the SWEET [GESL06] WCET analysis tool, which is one tool that can produce different WCETs depending on restrictions on the input.

Assumption: Input-sensitive WCET analysis is available.

Another assumption is that a component usage is always known for a component in a specific system, and the analysis is performed with respect to this specific system usage, and the analysis should be accurate with respect to that
Partial solution proposal 2
Combine WCET analysis with structured search over the input space to create input parameterizable component WCET contracts.

Partial solution proposal 3
Transformation from components to tasks such that resource efficiency is maximized while temporal constraints are met.

Industrial and academic problem formulations

Problem P1
Lack of development support for reusable WCET analysis complicates reuse of software components.

Problem P2
Reuse requires general and context unaware components while accurate WCET analysis requires context awareness and component specialization.

Problem P3
Inefficient transformations from components to real-time models leads to low resource efficiency.

Requirement R1
Automation

Requirement R2
CBSE Process

Requirement R3
Reusable WCET analysis

Requirement R4
Accurate analysis

Requirement R5
Resource efficiency

Requirement R6
Temporal correctness

Validation
Do the reusable analysis and component to task transformation we propose achieve sufficient resource utilization and predictability for reusable components in embedded real-time systems?
usage. However, we do not make any assumptions on the accuracy of the usage. The resulting WCET is only as accurate as the accuracy of both the usage and the usage dependent WCET.

**Assumption:** Components usage is known for each specific context.

In this research we do not aim to develop yet another memory analyser, and we use existing tools.

**Assumption:** Stack/memory analysis is available.

We do not provide methods or tools for analysing or binding context switch times, or other run-time properties; and we also assume that such methods, tools and analyses exist, and that these times are known.

**Assumption:** Known and predictable context switch time for the run-time system.

We assume that all components can be analysed with respect to memory, cpu-overhead and execution time. We do not determine if a component is analysable, or how suitable the component is.

**Assumption:** All components can be analysed.

We assume that all components are reusable from a strictly functional point of view. We do not consider how the reuse of a component is affected depending on implementation specific details.

**Assumption:** All components are reusable.

Most analyses are developed and known for single processor systems. For multi processor systems or distributed systems the analyses may be very different. Thus to limit the problem, we assume single processor, non-distributed systems.

**Assumption:** Single processor, non-distributed, systems.

We do not consider is different variants of hardware. Different alignment of software in memory, could potentially lead to different cache behaviour and thus different timing behaviour. There may also exist other issues that influence timing behaviour, e.g., different pipeline effects. However, to limit the complexity of the problem, we assume that a WCET prediction is valid, given that the component resides on the same type of hardware.
This of course limits the reuse to systems with the same hardware. However, we believe that this assumption is justified in many cases as components often are distributed as binaries, compiled for a specific hardware. Thus, variants of the same component are required for reuse on different hardware.

**Assumption:** Invariant hardware in the component execution environment.

Furthermore, when reusing components on the same type we assume that prediction results can be reused given that it is reused on the same hardware.

**Assumption:** WCET predictions are always valid for components reused on the same hardware.

From the problem statements (S1-S5) and the assumptions we form three research problems. From these research problems we identify a set of requirements that define important characteristics for increasing resource efficiency and predictability. The requirements also increase the confidence that we consider the correct problem. To fulfill these requirements we propose three partial solutions. The dependence between these parts is depicted in Figure 4.3.

### 4.5.2 Research problems

The nature of the studied industrial problem lies in the component-based development of ERTS, where general and context-unaware software components meets requirements on accurate timing predictions and low resource consumption.

The research problems detail some of many possible views of the industrial problem. We do not claim that our views are more correct than any other or that we cover all aspects of accurate timing predictions or low resource consumption. However, the research problems identifies parts of the industrial problem.

The problem statements S1-S5 stem from trade-offs between the generality required for efficient reuse, and the particularity of accurate component properties and efficient transformation to real-time system. The potential benefits of reuse are especially high in the embedded domain where product differentiation is ever increasing and competitiveness is driven by time-to-market and costs; thus there are strong reasons to find a solution to the trade-off. We continue by deriving a set of research problems from the industrial problem statements S1-S5, as outlined in Figure 4.1.
There are many incitaments to reuse software components in a structured way to lower, among many things, time-to-market and development costs. Reuse has been proven efficient for some domains within software engineering, however, the domain of ERTS has not yet been one of these domains. It is widely believed that one of the main things that obstructs reuse in this domain is the pervasive use of usage and context dependent properties, such as, e.g., WCET [Lüd06].

In order to successfully reuse components in ERTS it is necessary to consider the development of components and to combine the context freeness required by reuse and the context awareness required by the analysis.

**Problem P1** Lack of development support for reusable WCET analysis complicates reuse of software components.

**Statements:** S1, S3 and S4.

**Motivation:**

- WCET analysis is difficult and time consuming to use [HK07].
- Reuse is the main activity in CBSE to lower development time and cost [PD96].

To support reuse, context-freeness is vital. If a component has strong dependencies to one or a set of contexts, its reuse is limited to only the systems that conform to that specific context. Predicting the behaviour of a component without knowing its intended use may lead to very inaccurate predictions (or the inability to give any predictions at all).

To provide evidence of predictable behaviour for ERTS, one of the most important real-time properties is the WCET. While reusable components should be context free, WCET is a context sensitive property, meaning that it is sensitive to both the hardware it is executed upon and the usage, i.e., how it is used in that particular setting. If WCET is predicted without respect to context the predictions become much to inaccurate, pessimistic or even impossible to state. Inaccurate predictions leads to hardware being under-utilized, or even worse to faulty and unreliable systems.
Problem P2  Reuse requires general and context unaware component while accurate WCET analysis requires context awareness and component specialization.

Statements: S2 and S3 and S4.

Motivation:
-准确 WCET 分析需要上下文信息 [GESL06, KP05]
- 准确分析对于正确尺寸的硬件和正确的系统行为 [Dur06]。

Although predictability is one of the most important aspects of ERTS, it is also important with high resource utilization efficiency in order not to overdimension hardware. To achieve high resource efficiency it is important to consider how components are deployed. Even resource efficient components that are deployed without considering resource utilization may lead to resource inefficient systems, i.e., it is of little importance to have accurate predictions if the underlying system does not take advantage of them. Thus, the allocation of components to real-time tasks must be considered.

Components are reused at design and development of a system. However, transformation from component models to real-time models are not reused. A system needs to be transformed from a component model to a real-time model for each new context. Improper transformation from component models to real-time models may both reduce resource efficiency, and violate system properties.

Problem P3  Inefficient transformation from component models to real-time models may reduce resource efficiency, and violate real-time constraints.

Statements: S2 and S5.

Motivation:
- It is often desired to keep resource consumption low in embedded real-time systems [Crn04].
- Real-time constraints must be satisfied in a system with real-time constraints, in order to guarantee correct behaviour [But97].
- Components must be synthesized to real-time tasks [MG02, KWS03].
4.6 Requirements

From the formulated problems we break down the problems and identify a set of key requirements. The rationale for defining requirements from the research problems is to be able to validate the solutions with respect to the requirements in order to increase the confidence that our solutions actually solve (at least partially) the problems.

We define a set of requirements based on the research problems to make sure that we tackle the problem we intend, in a way that is adequate to industry. By fulfilling the requirements we increase the confidence that we tackle the correct problem. This can also be seen as a step in increasing construct validity of the thesis.

We do not claim that the requirements we have defined are exhaustive in the sense that they cover all issues in increasing prediction accuracy and resource utilization in ERTS. Also, each requirements could potentially be refined. However, the given requirements reflects issues that the embedded systems research community, real-time research community, component-based development communities and industry consider to be important.

It is possible to tackle the problems P1-P3 in many different ways. We want to direct our solutions to be of interest to the industry. Thus we also consider the component properties that we found to be important when defining the requirements.

The following component properties were found to be important to industry in earlier studies that we have presented in, e.g., Möller et al. [MÅFN05], Åkerholm et al. [ÅFSC04] and is further supported by Hänninen et al. [HMTN06].

**Predictable:** To what extent a component’s behaviour can be analysed.

**Resource efficiency:** How much resources a component requires in order to successfully fulfil its operation.

**Reusable:** How easily a component can be reused.

**Simplicity:** How much effort is required to use the component

**Usable:** How easy the component is to use in a certain context.

We synthesize the component properties and the research problems to form specific requirements that we consider in our research.
4.6 Requirements

4.6.1 Requirement definitions

WCET analysis is both time consuming and difficult. Having tools that automatically derive a reusable WCET increases the development efficiency by making WCET analysis simpler and more accessible.

\[ Requirement \text{ R1} \] **Automation:** The WCET analysis shall be automated as much as possible, requiring a minimum of human interaction.

**Derived from:** P1, Simplicity, Usable

The CBSE development process is different from traditional development in the sense that it is divided into component development and system development. Thus the WCET analysis should be divided in two different parts, a component part, developed to be reusable by the component developer, and a system part, to be used by the system designer. This facilitates the adoption of the technique in the CBSE development process.

\[ Requirement \text{ R2} \] **CBSE Process:** The WCET analysis shall be performed in the component development part of the CBSE development process. The WCET shall be known in the system development part of the CBSE development process.

**Derived from problem:** P1, Reusable, Usable

One problem is that, to gain maximum benefit from reuse not only the functional parts of components need to be reused, but also non-functional parts, e.g., WCET analyses. Thus, we need to find a way to reuse a component without re-analyzing the WCET.

\[ Requirement \text{ R3} \] **Reusable WCET analysis:** It shall be possible to reuse a software component without re-doing WCET analysis.

**Derived from:** P2, Portable, Reusable

WCET analysis can of course always be reused but to make sure that the WCET is always a safe over estimation, all possible usages must be considered. This will potentially lead to very inaccurate predictions.

\[ Requirement \text{ R4} \] **Accurate analysis:** The reusable WCET analysis results shall reach a pre-defined accuracy, and it shall be possible to reach the same accuracy as with current state of the art WCET analyses.

**Derived from:** P2, Predictable
We want to transform the component based system to a real-time system conforming to a specific real-time model. We must be able to separate an efficient transformation from an inefficient. A common approach for transforming components to tasks is simply to view one component as one task.

\(\text{Requirement R5}\) **Resource efficiency:** Component shall be transformed to real-time tasks such that the resource efficiency is higher than for a system where one component is allocated to one real-time task.

**Derived from:** P3, Resource efficiency

There exists many possible allocations from components to real-time tasks. At the same time as we want the system as resource efficient as possible, a transformation from components to real-time tasks may not violate temporal requirements.

\(\text{Requirement R6}\) **Temporal correctness:** Components must be transformed to tasks in such a way that the temporal correctness of the system is maintained.

**Derived from:** P3, Predictable

The requirements are not exhaustive in the sense that they cover the problems, as depicted in Figure 4.4. The problems are too big and complex and we only provide partial solutions to the problems in this thesis.

![Figure 4.4: Coverage of requirements with respect to the research problems in this thesis.](image)
4.7 Partial solution proposals

In [ÅFSC04, MÅFN04] we have surveyed several component technologies for ERTS, and investigated different methods that are commonly used with ERTS. Often predictable behaviour and resource efficiency are conflicting properties because many methods for lowering resource consumption are simply not suitable for ERTS. Dynamic resource management and adaptive behaviour are examples of techniques widely used outside the embedded and real-time domain for enforcing efficient resource usage.

4.7.1 Resource aware development

We have found that the most common techniques in ERTS for resource efficiency are compile-time techniques. Run-time techniques introduce uncertainties in the execution, making it difficult to predict correct behaviour.

Component functionality is reused between products, analysis on the other hand is typically performed for each system rather than for each component. Many analysis tools are expensive, time consuming and difficult to use, and there are potential big time gains to be made if component analysis can be reused in the same way as the component.

Partial solution proposal 1: Using the CBSE development process for combining the benefits of structured reuse of both functionality and WCET.

Fulfils requirements: R2

How does this solution fulfil requirements R2? We propose that the WCET analysis is divided in two parts to fit the CBSE development process. The first part is a reusable component WCET analysis, where inputs are partitioned with respect to WCET to form input-parameterizable component WCET contracts. The second part is the parametrization and composition to find the usage dependent WCET.

How does this solution contribute to the overarching question? We propose to integrate our techniques in the CBSE development process. This facilitates the use of the techniques in relation to CBSE because the CBSE development process differs from traditional development models.

4.7.2 Input sensitive WCET analysis

Reuse gives lower accuracy with respect to analyzability and increased resource consumption because of the generality needed for efficient reuse. How-
ever, how much lower prediction accuracy can we accept to lower time-to-market and what is sufficient resource utilization?

Reaching high resource utilization and predictability implies high effort. Thus, there is a clear trade-off between effort and better properties. Effort must be connected to some quantitative property like, e.g., cost, to be able to reason about. Sufficient accuracy and resource utilization is of course then entirely dependent on the application domain. In safety critical applications it is probably easier to motivate a higher cost for reaching high predictability than in, e.g., cheap consumer electronics. Because of this it is desirable to have parametrizable methods. Then the methods may be applicable to a larger set of domains.

Partial solution proposal 2: Parametrizing WCET analysis results with respect to context information for increasing reusability of the WCET analysis.

Fulfils requirements: R1, R3, R4

How does this solution fulfil requirements R1? Automatic slicing can be used for finding variables and variable value bounds that affect the program flow, and thus the WCET. Human intervention is not required, but can be used for speeding up the analysis by manually specifying variables and their bounds.

How does this solution fulfil requirements R3? By creating parametrizable component WCET contracts, that are parametrizable with usage, to get a usage dependent WCET, then, given the assumptions stated in Section 4.5.1 the component WCET can be reused together with the software component.

How does this solution fulfil requirements R4? By gradually exploring the program state space by using automatically derived annotations it is possible to achieve an equally accurate WCET compared to a manually annotated traditional WCET analysis.

How does this solution contribute to the overarching question? By parametrizing WCET with respect to usage reuse of software components is facilitated.

4.7.3 Allocation of components to real-time tasks

Making accurate predictions and supporting reuse is sufficient for the construction and analysis of a system; however, eventually the system needs to be synthesised to a run-time system. In ERTS tasks are the run-time entities that
control the execution of the components. One common approach to map components to tasks is *one-to-one allocation* where one component is allocated to one task. This allocation is very convenient from an analysis perspective because real-time analysis can be performed early in the development. However, the allocation is not very efficient due to high overhead.

Partial solution proposal 3: Allocate components to real-time tasks in such a way that resource efficiency is maximized while temporal constraints are met.

**Fulfils requirements:** R5, R6

*How does this solution fulfil requirements R5?* The allocation framework guarantees that any allocation is never worse than the one-to-one, by always using the one-to-one allocation as a starting point in the search for better allocations.

*How does this solution fulfil requirements R6?* The allocation framework has a set of rules for evaluating a allocation if it is feasible. Those rules lie also encode the temporal correctness. Thus a transformation that is resource efficient but does not fulfil the stipulated timing requirements is not considered to be a feasible allocation.

*How does this solution contribute to the overarching question?* By providing a framework for structured allocation of components to real-time tasks, resource utilization is increased and real-time requirements are maintained from design to synthesis. Allocation of components to real-time tasks is one step towards increased use of CBSE in ERTS.

### 4.8 Validation of solutions

One of the things that distinguishes good software engineering research is the presence of proper validation. Validation is required before the claims of effectiveness and/or usefulness can be determined. We revisit the research method proposed in Section 4.3 and describe how our results have been validated. We strive to validate the solution with respect to the industrial problem by validating the results with respect to the requirements that were derived from the industrial problem. In the summary of Chapter 9 we also revisit the industrial problem and discuss to what degree it has been solved.

The question that the validation should answer is why our research is valid? The simple answer to the question is that the research is valid because we have
used a proper research method. To further answer the question we need to known what we mean by validity. Validity is usually divided into four categories, i.e., Construct validity, Reliability, External validity and Internal validity.

Construct validity means, put simply, did we implement the program we intended to implement and did we measure the outcome we wanted to measure?

Reliability means, can we repeat the research and get the same results?

External validity means, how well the results can be generalized outside the study.

Internal validity means, put simply, did the input to the program cause the outcome to happen?

According to Shaw [Sha02] there exist 5 types of evaluations in software engineering, Persuasion, Implementation, Evaluation, Analysis and Experience. We use several of these methods, and we go through each partial solution and describe the type of validation used.

4.8.1 Resource aware development

Validation of partial solution 1: We develop extensions for the CBSE development process and reasons about the usage of the process together with the proposed methods. We are extensively useing existing literature, discussions, examples and evaluations for validating the solution.

Type of validation: Persuasion

Construct validity: To ensure construct validity, several sources of evidence is used, both use of existing literature, and experiments.

External validity: The results are only generalizable with respect to our explicit assumptions. When the assumptions are not valid we can not guarantee external validity. The systems that are evaluated are both academic benchmarks and industrial code from two compamies in different business segments within the ERTS domain.

Internal validity: We disregard the internal validity; we do not see any threats because we have control over all variables, and we know their relationships.
4.8 Validation of solutions

Reliability: The experiments leading to the results are well controlled. The same evaluation has been performed multiple times to generate multiple data points.


Described in thesis: Chapter 5.

4.8.2 Input sensitive WCET analysis

Validation of partial solution 2: We propose and implement methods for reusable WCET analysis for permitting reuse of accurate WCET predictions of components in ERTS. We perform empirical evaluations through empirical models from both industrial code and academic benchmarks.

Type of validation: System implementation, empirical evaluation

Construct validity: To ensure construct validity we have well defined frameworks, clearly defining what is measured.

External validity: The results are only generalizable with respect to our explicit assumptions. When other variables are considered we cannot guarantee external validity. The systems that are evaluated are both academic benchmarks and industrial code from two companies in different business segments within the ERTS domain.

Internal validity: We disregard the internal validity; we do not see any threats because we have control over all variables, and we know their relationships.

Reliability: The experiments leading to the results are well controlled. The same evaluation has been performed multiple times to generate multiple data points.


Described in thesis: Chapter 6 and Chapter 8.

4.8.3 Transformation from components to real-time tasks

Validation of partial solution 3: We propose and implement a method for transforming components to tasks in such a way that temporal constraints are preserved and resource usage is increased compared to a reference transformation. We perform empirical evaluations through simulations.
Type of validation: System implementation, empirical evaluation

Construct validity: To ensure construct validity we have well defined frameworks, clearly defining what is measured.

External validity: The results are only generalizable with respect to our explicit assumptions. When other variables are considered we cannot guarantee external validity. The allocation strategy has not been extensively tested for different types of systems, thus the results are not generalizable in other systems than ERTS.

Internal validity: We disregard the internal validity; we do not see any threats because we have control over all variables, and we know their relationships.

Reliability: The experiments leading to the results are well controlled. The same evaluation has been performed multiple times to generate multiple data points.

Original research: Paper A, Paper D and Paper G

Described in thesis: Chapter 7 and Chapter 8.
Chapter 5

Resource aware development

As discussed in the previous chapters, an important research issue in the field of Component-Based Software Engineering (CBSE) for Embedded Real-Time Systems (ERTS) is how to handle resources and predictability. This chapter positions our proposed methods for increasing resource utilization and predictability, in the component-based development process. We first introduce the CBSE process, and then briefly describes the methods and their relationships to the process.

Traditional software development (e.g., the waterfall model [Roy70]) considers the system view, and develops a system with normally well defined system context and usage. In contrast, the CBSE process is divided in two parts: a system development process and a component development process and interactions between the two processes [CCL06]. In the CBSE development process the component development is focused on developing general components to be used in many different systems, while the system development is focused on reusing existing components to build a system. Thus, there is a significant difference between the traditional and CBSE software development processes.

To enable CBSE for ERTS it is important to address reusable predictions and low resource consumption. Since the CBSE development process is significantly different from traditional development, it is important to address these issues in relation to the CBSE development process.
5.1 Component-based software engineering

To facilitate reuse, an important distinction between traditional software development and CBSE is that individual components are not specified and laid out according to existing other components that are supposed to integrate their services. Every single component is specified according to a more or less general requirements profile, so it can be reused and integrated in a number of different contexts. Generality is a key feature of components because they should be reused in many different contexts. However, the interface between the component development and system development processes may be fairly complex. Existing components are surveyed in the system development process during the requirements phase, influencing the entire scope and direction of the system because the system is easier built with existing components [LBCC08]. If no suitable existing components can be found, new components are developed in the component development process. During later phases in the CBSE process the components are tested to assess functionality and quality characteristics; they are used in prototyping during design, and finally integrated and deployed with the system.

Figure 5.1 shows a general model for the CBSE processes where components and systems are developed independently. Most of the interaction between the component and system development processes are performed during the requirements and design phases, and, the verification phase.

5.2 Reusable analysis

Worst-Case Execution Time (WCET) analysis has been used in traditional software development for ERTS for a long time, and is a vital part of real-time analysis. However, to introduce reusable WCET analysis for CBSE, it must be assessed how the reusable analysis can be incorporated in the CBSE development process. WCET analysis is normally performed in the verification activity of traditional software development processes. The analysis of component-based systems is different since components are developed to be general and reusable, and the systems are developed from existing components.

Predicting the amount of resources required by ERTS is of prime importance for verifying that the system will fulfil its real-time and resource constraints. Particularly important in ERTS is to predict the WCETs of tasks, so that it can be proven that task temporal constraints (typically deadlines) will be met.
The division between development of components and development of systems in the CBSE process implies that one efficient way of reusing WCET analysis is to position the analysis in the component development process, so that the results can be reused with the component in the system development process. By introducing an analysis process where part of the analysis is performed by the component developer the overall development process becomes arguably more efficient compared to traditional analysis.

Traditionally the complete analysis is required to be performed by the system developer because current WCET analysis techniques requires the complete system to be available in order to make accurate predictions. However, WCET analysis is both difficult and time consuming and often requires manual tuning and annotation of the program [HK07]. If the WCET analysis can be reused it also implies that its effort is reused. Moreover, if an equally accurate, and reusable, prediction can be produced the benefit is potentially very high.

On a high level we divide the reusable WCET analysis into (i) the construction and analysis of the component where the component developer performs a reusable WCET analysis, and (ii) the development of systems and usage where the system developer uses the reusable WCET analysis on a specific system

Figure 5.1: CBSE development process (from [CCL06])
to find the specific WCET of each component for the system, as depicted in Figure 5.2.

The component developer produces a parametrizable, reusable WCET analysis for individual components; the system developer parametrize the components’ reusable WCET analysis is with information about the system specific usage in order to acquire the system specific WCET.

Figure 5.2: Resource centric development process view.

Figure 5.3 shows how a general model for CBSE processes is extended with the reusable WCET analysis as part of the Verification activity, and the analysis results packaged with the component in the Release activity. These results are reused during the system WCET analysis, which is performed in the Verify phase of the component assessment process.

This development process allows for the effort of WCET analysis to be moved from the system development to the component development. In this way, not only the component itself but also the WCET analysis is reused several times. The system developer escapes the effort of learning and using advanced WCET analysis tools, and the overall process becomes more efficient. We
Figure 5.3: CBSE development process extended with reusable WCET analysis
deepen these discussions in Chapter 6, and the reusable WCET analysis is evaluated in Chapter 8.

5.3 Allocating components to real-time tasks

The problem of allocating components to tasks does not exist in traditional systems, since traditional systems are normally designed considering tasks (i.e., processes or threads of execution). However, when the system is developed from reusable components, it is not obvious how these components shall be, or can be, allocated to real-time tasks. And especially how components shall be allocated to real-time tasks in order to minimize resource consumptions while maintaining real-time constraints, e.g., deadline.

The allocation of components to real-time tasks depends on real-time properties. The allocation must be performed in such a way that real-time analysis guarantees that stipulated real-time requirements are fulfilled. The allocation is therefore positioned in the development process after the reusable WCET analysis as shown in Figure 5.4.

Because of the real-time requirements imposed on ERTS, the allocation of components to real-time tasks is a problem in current CBSE development practices [KWS03]. Therefore it is vital that the allocation considers temporal attributes, such as WCET, deadline and period time. In a system with many components, the overhead due to context switches is quite high. ERTS consist of periodic and aperiodic events, often with associated end-to-end timing requirements. Periodic, time-triggered components can often be coordinated and executed by the same task, while preserving temporal constraints. Co-allocating several components to one real-time task may lead to better performance in terms of, e.g., memory and CPU usage. Hence, it is easy to understand that there can be positive effects as a result of grouping several components into one task.

There are several ways to allocate components to tasks, one common allocation strategy, which may have lead to the confusion between what are components and what are tasks, is the one-to-one allocation where one component constitutes one task\(^1\). This mapping is used in several component technologies including, e.g., Rubus [Lun08, LLL03], PBO [SVK97] and Auto-comp [SFÅ04]. Components can be co-allocated, where many components form a task. Nevertheless, a component can be distributed over several tasks.

---

\(^1\)Components and tasks are in fact two orthogonal concepts; components are design entities and tasks are run-time entities, the concepts are still sometimes confused
5.3 Allocating components to real-time tasks

Figure 5.4: CBSE development process extended with reusable WCET analysis and component to task allocation.
The one-to-one mapping is often chosen due to its matching properties with real-time analysis, since a component assembly can be checked at design-time with standard real-time analysis. However, to ensure that temporal requirements are met, while at the same time resource usage is minimized, special methods for mapping component to tasks need to be developed. One such method is described in detail in Chapter 7, and evaluated in Chapter 8.

5.4 System model

We describe a general system model that is used throughout the rest of this thesis to reason about our methods for reusable WCET analysis and component to task allocation.

Component characteristics

The component interaction model is a pipe-and-filter model with transactions, which is commonly used within the embedded systems domain. Each component has a trigger; a time trigger or an event trigger or a trigger from a preceding component. A component transaction describes an order of components to be executed and defines an end-to-end timing requirement. In Figure 5.5, the notation of a component assembly with six components and four transactions is described. The graphical notation is similar to the one used in UML.

![Graphical notation of the component model.](image)

Many component models for embedded systems have the notion of transactions built in; however, if a component model lacks the notion of transactions, it is possible to model end-to-end timing requirements and execution order at
a higher abstraction level. In general a system is described as a set of components, and transactions (flow) among components. The component model is described with:

**Component** $c_i$ is a tuple $(P_i, R_i, S_i, Q_i, m_i, f_i, prog_i)$, where $P_i$ is the provided interface, which is a set $\{p_{i,0}, p_{i,1}, \ldots, p_{i,n-1}\}$ of input variables and $R_i$ is the required interface, i.e., a set $\{r_{i,0}, r_{i,1}, \ldots, r_{i,n-1}\}$ of output variables. Both input and output variables can pass data or control. $S_i$ represents a trigger; a time trigger, an event trigger or a trigger from a preceding component. $Q_i$ represents the periodicity of a trigger. $f_i$ is a contract as a function with respect to a usage that returns the estimated WCET of the component with respect to the specified usage $U_i$ such that $f_i : U_i, pt_i \rightarrow WCET_i$, where $U_i$ is a usage profile and $pt_i$ is a probability threshold used for removing WCETs with low probability. $m_i$ is the amount of stack memory required by the component. $prog_i$ is the software behaviour of the component in some form, e.g., source code, graphical model or binary format.

**Usage profile** $U_i$ is a set of inputs and probabilities for those inputs. $U_i$ represents predicted inputs for the component in a specific context and usage. The usage profile is described in detail in Chapter 6.2.3.

**Component Transaction** $\Gamma_i$ is an ordered relation between components $c_a, c_b, c_c, \ldots, c_n$, and an end-to-end deadline $d_{ci}$. The deadline is relative to the event that triggered the component transaction, and the first component within a transaction defines the transaction trigger. $\Gamma^T_i$ defines that the transaction is time-triggered and $\Gamma^E_i$ denotes that the transaction is event triggered. A component transaction can stretch over one or several components, and a component can participate in several component transactions. The component $c_a$ should execute before the component $c_b$, and the component $c_b$ should execute before $c_c$ to produce the expected results etc. The correct execution behaviour for a component transaction is formalized with the regular expression denoted in 5.1.

$$c_a \Sigma^* c_b \Sigma^* c_c \Sigma^* \ldots \Sigma^* c_n$$ (5.1)

Where $\Sigma^*$ denotes all components defined by the transaction in any order. This means that a transaction allows arbitrary execution ordering as long as the pattern $c_a$ before $c_b$ before $c_c$ exists, i.e., $c_a, c_c, c_b, c_d, c_e$ is a valid transaction since at some point, $c_b$ executes after $c_a$, and $c_c$ executes after $c_b$. However, in order to use most current response time...
analyses it is required that a transaction only consists of components with harmonic period times.

In a component assembly, event triggers are treated different from periodic triggers as the former are not strictly periodic. There is only a lower boundary restricting how often it can occur, but there is no upper bound restricting how much time may elapse between two invocations. Thus, if an event trigger could exist inside or last in a transaction, it would be impossible to calculate the response time for the transaction, and hence a deadline could never be guaranteed.

**Task characteristics**

Our task model specifies real-time properties similar to many traditional real-time task models, e.g., [MJ86], with the addition of the organization of entities in the component model into tasks. During the allocation from components to real-time tasks, properties like schedulability and response-time constraints must be analysed in order to ensure the correctness of the final system. Components only interact through explicit interfaces; hence tasks do not synchronize outside the component model. The task model is for evaluating schedulability and other properties of a system.

Task \( \tau_j \) is a tuple \( \langle Z_j, T_j, C_j, stack_j \rangle \) where \( Z_j \) is an ordered set of components. Components within the same task are executed in sequence following the order of \( Z \) and with the same priority as the task. \( T_j \) is the period of the task. The parameter \( C_j \) is the estimated WCET. \( stack_j \) is the stack usage of the component. \( C_j, stack_j \) and \( T_j \) are deduced from the components in \( Z_j \). \( C_j \) is the sum of WCETs for all components in \( Z_j \). Hence, for a task \( \tau_j \), the parameters \( C_j \) is calculated with Equation 5.2. \( stack_j \) is the maximum of all components specified stack usage and is calculated with Equation 5.3. The task inherits the trigger(s) of the first component \( c_i \) in the set \( Z_j \) to facilitate scheduling analysis.

\[
C_j = \sum_{\forall i (c_i \in Z_j)} (f_i (U_i, pt_i)) \quad (5.2)
\]

\[
stack_j = \max_{\forall i (c_i \in Z_j)} (m_i) \quad (5.3)
\]

The task model specifies the organization of entities in the component model into tasks and transactions over tasks. During the allocation of components to real-time tasks, extra-functional properties like response-time constraints must be respected in order to ensure the correctness of the final system.
System characteristics
The system consists of system parameters and a schedulable task set.

System $K$ is described with the tuple $< A, \beta, \rho >$ where $A$ is a task set to be scheduled by the system scheduler. The constant $\beta$ is the size of a task control block, i.e., the size of the data structure needed by the scheduler, containing information for managing the task. The task control block is considered constant and the same for all tasks. The constant $\rho$ is the time associated with a task switch, i.e., the time for storing and restoring the state of the CPU such that several tasks can share the CPU. The system kernel is the only explicit shared resource between tasks; hence we do not consider blocking effects due to, e.g., synchronization.

5.5 ACC Example
To illustrate our models and the techniques described in Chapters 6 and 7 we use an Adaptive Cruise Control (ACC) application as an illustrative example. The ACC is developed with the SaveCCM component model [ÅCF+07] as depicted in Figure 5.6 and is constructed with four components, one switch and one assembly. For a detailed description of SaveCCM and the ACC we refer to [ÅCF+07].

The ACC is a conceptual design with features like automatic scanning of road signs, brake assist if objects approach too fast, and adaptive speed considering the distance to cars in front.

In this example we assume that a car OEM develops the ACC from a set of pre-fabricated components (and an assembly). We assume that the ACC is developed for two car models, a high-end luxurious car model and a low-end car model. The car manufacturer (i.e., system developer in Figure 5.2) must analyse the ACC with respect to timing, in order to assert that it will behave according to the system requirements. Once this is done, the ACC is allocated to real-time tasks in such a way that the temporal requirements are met. Furthermore, the allocation should minimize the resource consumption.

Looking at Figure 5.6, the components in the ACC are detailed in Table 5.1 together with the properties trigger $S_i$, period time $Q_i$, where the period time is determined by the trigger. If a component $c_j$ is triggered by a preceding component $c_i$, then the period time $Q_j$ is inherited from the component $c_i$, thus $Q_j = Q_i$. The memory consumption $m_i$ is the memory required by a
component. Moreover, the ACC is designed with three time triggered transactions \(\langle \Gamma^0_T, \Gamma^1_T, \Gamma^2_T \rangle\) and associated end-to-end deadlines \(\langle d_{c0}, d_{c1}, d_{c2} \rangle\). The transactions are described in Table 5.2.

The component \textit{Speed Limit} is periodically triggered by a 50Hz clock. The components \textit{Object Recognition}, \textit{Brake Assist} and \textit{ACC Controller} are triggered in sequence with the same period as \textit{Speed Limit}. The \textit{Logger HMI Output} is triggered by a 10Hz clock. Furthermore, we consider the assembly \textit{ACC Controller} the same as a component, which is allowed by the \textit{SaveCCM} component model.

![Adaptive Cruise Control Diagram](image)

**Figure 5.6: Adaptive Cruise Control**

<table>
<thead>
<tr>
<th>Component</th>
<th>Name</th>
<th>((S_i, Q_i, m_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed limit</td>
<td>(c_{speed})</td>
<td>((50Hz, 20, 1024))</td>
</tr>
<tr>
<td>Object recognition</td>
<td>(c_{obj})</td>
<td>((c_{speed}, 20, 512))</td>
</tr>
<tr>
<td>Brake assist</td>
<td>(c_{break})</td>
<td>((c_{obj}, 20, 512))</td>
</tr>
<tr>
<td>Logger</td>
<td>(c_{log})</td>
<td>((10Hz, 100, 2048))</td>
</tr>
<tr>
<td>ACC controller (assembly)</td>
<td>(c_{acc})</td>
<td>((c_{obj}, 20, 2048))</td>
</tr>
</tbody>
</table>

**Table 5.1:** The five components \textit{Speed limit}, \textit{Object recognition}, \textit{Brake assist}, \textit{Logger} and \textit{ACC controller} as depicted in Figure 5.6.
Following the CBSE workflow as depicted in Figure 5.2, we discuss the following actions in relation to the ACC example.

- Usage profile assessment in Section 5.5.1.
- Input sensitive component WCET analysis in Section 5.5.2.
- Allocation of components to real-time tasks in Section 5.5.3.

<table>
<thead>
<tr>
<th>$\Gamma_i$</th>
<th>$N_i$</th>
<th>$dc_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$c_{\text{speed}} \rightarrow c_{\text{obj}} \rightarrow c_{\text{break}}$</td>
<td>1800</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$c_{\text{speed}} \rightarrow c_{\text{obj}} \rightarrow c_{\text{acc}}$</td>
<td>2500</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$c_{\text{log}}$</td>
<td>5000</td>
</tr>
</tbody>
</table>

Table 5.2: The three transactions in the ACC

### 5.5.1 Usage profile assessment

In our example, the car OEM uses the ACC for two different car models, one high-end car and one low-end car. In the high-end car, more features are enabled than for the low-end car.

In Tables 5.3 and 5.4, we outline the usage profiles $U_1$ and $U_2$, i.e., the possible values for the different inputs, with respect to the two car models. The inputs considered are Road Signs Enabled (RSE), ACC Max speed (AMS), Road Sign Speed (RSS), Distance (D), Current Speed (CS), ACC Enabled (AE) and Brake Pedal Used (BPU), as depicted in Figure 5.6.

Table 5.3 describes the usage profile for the high-end car where road signs scanning can be activated or deactivated (RSE). The ACC maximum speed is 250 km/h (AMS), while the operation speed intervals for the road signs scanning system is between 0 and 130 km/h (RSS). The distance for the adaptive speed is between 0 m and 2 km. The current speed of the ACC can be between 0 and 250 km/h (CS). The adaptive behaviour of the ACC can be enabled or disabled, switching between an adaptive and a normal cruise controller (AE), and the brake pedal can be pressed or fully released (BPU).

Table 5.4 describes the usage profile for the low-end car where both the adaptive part of the ACC and the road sign scanning are always disabled, resulting in a normal cruise controller. As a consequence of this, the distance (D), ACC enabled (AE) and road sign speed (RSS) are always 0.

A usage profile is a description of the usage of the system, and a usage-dependent WCET is valid only in the given usage profile. It is therefore important to describe the usage profile as accurately as possible.
5.5.2 Input sensitive WCET analysis

The developer of the individual components performs an input sensitive WCET analysis that is parametrized with a usage profile. The system developer and the domain expert make an assessment of the system usage, creating usage profiles for the system. During the system analysis the contract \( f_1 : U \rightarrow WCET \) is parametrized with the usage profile for respective car model, and the WCET is given.

<table>
<thead>
<tr>
<th>Comp.</th>
<th>ui</th>
<th>( U_1 )</th>
<th>( U_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpeedLimit</td>
<td>384</td>
<td>384</td>
<td>105</td>
</tr>
<tr>
<td>ObjectRecog.</td>
<td>301</td>
<td>301</td>
<td>220</td>
</tr>
<tr>
<td>BrakeAssist</td>
<td>214</td>
<td>191</td>
<td>88</td>
</tr>
<tr>
<td>Logger.</td>
<td>627</td>
<td>433</td>
<td>433</td>
</tr>
<tr>
<td>ACC Controller</td>
<td>1341</td>
<td>1341</td>
<td>869</td>
</tr>
</tbody>
</table>

Table 5.5: WCETs according to the methods usage independent (ui) and usage dependent (ud) and usage dependent with usage profiles ud(\( U_1 \)) and ud(\( U_2 \))

To illustrate the difference in WCET for different inputs, we present both the usage independent and usage dependent WCETs in Table 5.5. We present the usage independent WCETs (ui), and the usage dependent WCETs considering usage profiles \( U_1 \) and \( U_2 \) described in Section 5.5.1. Notice that the usage \( U_1 \) produce the same WCETs as compared to the usage independent (ui) WCETs except for the components \( c_{\text{log}} \) (Logger) and \( c_{\text{brake}} \) (Brake Assist). For \( U_2 \) the difference with respect to the usage independent (ui) WCETs are greater.

The context sensitive WCET analysis produces a contract \( f_1 : U \rightarrow WCET \) for each component \( c_i \) that is parametrized with a usage. The contract for, e.g., \( c_{\text{speed}} \) (Speed Limit) consists of three predicates as outlined in Table 5.6.

<table>
<thead>
<tr>
<th>Comp.</th>
<th>ui</th>
<th>( U_1 )</th>
<th>( U_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpeedLimit</td>
<td>384</td>
<td>384</td>
<td>105</td>
</tr>
<tr>
<td>ObjectRecog.</td>
<td>301</td>
<td>301</td>
<td>220</td>
</tr>
<tr>
<td>BrakeAssist</td>
<td>214</td>
<td>191</td>
<td>88</td>
</tr>
<tr>
<td>Logger.</td>
<td>627</td>
<td>433</td>
<td>433</td>
</tr>
<tr>
<td>ACC Controller</td>
<td>1341</td>
<td>1341</td>
<td>869</td>
</tr>
</tbody>
</table>

Table 5.6: Contracts for the components according to the methods usage independent (ui) and usage dependent (ud) and usage dependent with usage profiles ud(\( U_1 \)) and ud(\( U_2 \))

Not considering the usage potentially results in very inaccurate WCETs In Section 6.5 we outline the detailed analysis for the component \( c_{\text{speed}} \) (Speed Limit), and describe the contract.
Table 5.6: \( c_{speed} \) resulting contract \( f_{speed} \)

### 5.5.3 Allocating component to tasks

When the system is developed and analysed, and it has been asserted that all timing requirements are met, the system is allocated from components to real-time tasks, to be scheduled by a real-time scheduler. The components are allocated to real-time tasks, and then task properties for each task are derived from the allocated components.

![Figure 5.7: Allocation of components to real-time tasks with respect to the different WCETs predicted considering \( U_1 \) and \( U_2 \).](image)

There exists a large number of possible allocations of components to tasks. By utilizing our framework two allocations are produced for usage profiles \( U_1 \) and \( U_2 \), as depicted in Figure 5.7. The framework results in a set of tasks, to which the components are allocated. We denote the resulting task set \( A \). We consider the strategy of allocating each component to a single task (1-to-1 allocation) to be the standard allocation. Thus, we compare our allocations to the 1-to-1 allocation and calculate the improvement of context-switch overhead \( p_A \) and stack usage \( s_A \) for all tasks.

The context switch overhead \( p_A \) and stack usage \( s_A \) are presented in Table 5.7 for the task set \( A \) with respect to the different allocations. The improvement is presented in parenthesis, and is calculated with respect to the 1-to-1 allocation as the worst possible allocation, and the Optimal as the best allocation.
Chapter 5. Resource aware development

The optimal allocation is where all components are allocated to one task. We will show in Section 7.6 that this allocation is feasible only with input-sensitive WCET prediction considering usage profile $U_2$. For WCET predictions with respect to usage profile $U_2$ and for a usage independent WCET prediction this allocation is not possible with respect to schedulability.

Thus, for the low-end car the car OEM can use less powerful and less expensive hardware while still guaranteeing that stipulated timing constraints are fulfilled.

<table>
<thead>
<tr>
<th>Allocation</th>
<th>$P_A$ (%)</th>
<th>$\pi_A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1$ allocation</td>
<td>4.6% (0%)</td>
<td>7644 (0%)</td>
</tr>
<tr>
<td>$U_2$ allocation</td>
<td>2.4% (63%)</td>
<td>6020 (31%)</td>
</tr>
<tr>
<td>Optimal allocation</td>
<td>1.1% (100%)</td>
<td>2348 (100%)</td>
</tr>
</tbody>
</table>

Table 5.7: CPU overhead and stack usage with respect to different allocations.

The goal for the allocations depicted in Figure 5.7 is to minimize context switch overhead, task control block size and stack size, while maintaining the stipulated timing constraints. We discuss the numbers in Table 5.7 and the allocations in detail in Section 7.6.

5.6 Summary

CBSE in general is the emerging discipline of the development of software components and development of systems incorporating software components [CL02a]. It is a promising approach for efficient software development, mainly through the activity reuse. In general, with some exceptions, component technologies are not used for (i) resource constrained systems; they are simply too demanding both in computing power and memory usage. They are not used for (ii) safety critical systems; it is hard to verify the functionality due to complexity and black box property of components. They cannot be used for (iii) real-time systems since they rely on unpredictable dynamic bindings and other complex run-time mechanisms. Embedded systems can often be classified as combinations of (i), (ii) and (iii).

Traditional WCET analysis has been used in traditional software development for embedded real-time systems for a long time. Examples of such development processes are the waterfall model [Roy70] and the V-model [Pre01]. Traditional software development only considers the system view, and develops one system, where the system context and usage normally are well known.
In contrast, the CBSE development process is divided into two different processes: component development process, and system development processes and interactions between the two processes. In the CBSE development process, component development is focused on developing general components to be used in many different systems, while the system development is focused on reusing existing components to build a system. Thus, there is a significant difference between the traditional and CBSE software development processes.

The problem of allocating components to tasks does not exist in traditional systems, since traditional systems are designed with respect to tasks (i.e., processes or threads of execution). However, when the system is developed from reusable components, it is not obvious how these components should be, or can be, mapped to real-time tasks. This is not a problem that limits the reusability of the components (as with the WCET analysis) but the problem stems from that the system is developed from existing components that may have constraints on how they can be allocated to tasks.

It is important to address how the techniques we propose in this thesis can be positioned in the CBSE development processes. The proposed solutions must fit the CBSE development process.

In this chapter we have outlined the usage of the methods described in Chapters 6 and 7. We have positioned the reusable WCET analysis, and the component to task allocation in the CBSE development process in order to facilitate the usage of these techniques in relation to CBSE. Furthermore we have outlined relevant definitions needed for reasoning about the problems in Chapters 6 and 7.
Chapter 6

Input sensitive execution-time analysis

In this chapter we outline two novel methods, based on a combination of static WCET analysis and systematic search over the value space of input variables, for deriving the WCET behaviour of a software component. We use the input sensitiveness of computer software components to express a relationship between their inputs and execution times. In particular we present:

- a novel approach to input parametric WCET analysis for reusable software components.
- a novel method for deriving the values of the input variables that gives the WCET.
- various approaches to speed up the search, allowing the WCET input values in most cases to be quicker derived.

The methods are evaluated in Chapter 8, and the results show that the proposed methods can express relationships between inputs and execution times with equally high precision compared to traditional WCET analysis. The methods have been evaluated with both industrial and academic software.
6.1 Input sensitive WCET analysis

To facilitate reuse, an important distinction between traditional software development and CBSE is that individual components are not specified and laid out according to existing other components that are supposed to integrate their services. Every single component is specified according to a more or less general requirements profile, so it can be reused and integrated in a number of different contexts. Generality is a key feature of components to facilitate reuse in many different contexts [BBB⁺00, BBCD⁺00].

In this work we use the relationship between component input and component execution times to form a WCET contract. To facilitate reuse of WCET different component execution times are classified with respect to component input. A component WCET contract is defined as a set of input classes with corresponding WCET. The contract is not created with respect to any specific component usage or, but describes the timing behaviour of the component only with respect to its inputs.

WCET is a context-dependent property that varies depending on component inputs, internal component state and hardware state. In this work we only consider the parts that can be observed from outside of the component. The component inputs are observable, but the component’s internal state and the CPU hardware state, are not observable outside of the component.

We derive WCETs for a component with respect to input combinations. Internal program states and hardware states will have to be considered by the WCET analysis tool; which is outside the scope of this thesis.

6.1.1 Input value space partitioning

Given a component $c_i$ that has a provided interface $P_i$ with $n$ inputs, such that $P_i = \{p_{i,0}, p_{i,1}, ..., p_{i,n-1}\}$. Each provided input $p_{i,j}$ is associated with a variable $v_{i,j}$ and a value domain $v_{i,j} \in V_{i,j}$. The set of variables $v_{i,0}, v_{i,1}, ..., v_{i,n-1}$ represent an input value space as a set of input value combinations.

**Definition 6.1.** An input variable $v_{i,j} \in V_{i,j}$ represents the possible values for a component provided input $p_{i,j} \in P_i$.

Input value combinations for a provided input interface $P_i = \{p_{i,0}, p_{i,1}\}$ are described with tuples $\langle v_{i,0}, v_{i,1} \rangle$. Consider the two input variables $v_{i,0}$ and $v_{i,1}$, with the value domains $V_{i,0} = \{v: 0 \leq v < 1\}$ and $V_{i,0} = \{v: 0 \leq v < 1\}$, then the possible input variable combinations are described with the input value space $D_i = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$.
Definition 6.2. An input value space $D_i$ is a set of input value combinations, as formalized in 6.1

$$D_i = \{(v_{i,0}, v_{i,1}, \ldots, v_{i,n-1}) : v_{i,j} \in V_{i,j}\}$$ (6.1)

An input combination tuple $d$ represents one combination of single values on the provided inputs. Consider a provided interface $P_i = \{p_{i,0}, p_{i,1}\}$ and the input combination $d = \langle 0, 1 \rangle$ represents that the input $p_{i,0}$ is given the value 0 and $p_{i,1}$ is given the value 1.

Definition 6.3. A tuple $d = \langle v_{i,0}, v_{i,1}, \ldots, v_{i,n-1} \rangle$ describes an input combination for the provided interface $P_i$.

$D_i$ represents all possible value combinations tuples $d$. In the input sensitive WCET analysis, the input value space $D_i$ is partitioned in subsets. For partitioning we use a predicate $\phi$ considering a tuple $d$. For example the predicate $\phi$ may put restrictions on the input variables $v_{i,0}, v_{i,1}$ and $v_{i,2}$ such that $\phi = [0 \leq v_{i,0} < 1, 0 \leq v_{i,1} < 1, 0 \leq v_{i,2} < 2]$.

In this case $d$ is restricted to the single value 0 for $v_{i,0}$ and $v_{i,1}$ and to the values 0 and 1 for $v_{i,2}$. A predicate $\phi$ may also express dependencies between input variables, e.g., $\phi = [1 \leq v_{i,j} < 3, v_{i,k} < v_{i,j}]$.

Definition 6.4. $\phi$ is a predicate on an input combination tuple $d$, such that the tuple $d$ must fulfill $\phi$, which is denoted $\phi(d)$.

$D_{i|\phi}$ represents the tuples in $D_i$ that fulfill the predicate given in $\phi$. Thus, $D_{i|\phi}$ is a condition subset of $D_i$ such that all tuples must fulfills the predicate $\phi$, and all tuples $d$ that violates $\phi$ are removed from $D_{i|\phi}$.

Definition 6.5. An input value space partition $D_{i|\phi}$ is a condition subset of all value combination tuples with respect to a predicate $\phi$, as formalized in 6.2

$$D_{i|\phi} = \{d \in D_i : \phi(d)\}$$ (6.2)

If $\phi$ defines dependencies between input variables, e.g., $\phi = [v_{i,0} < v_{i,1}]$ some method must be used for deriving the possible values combination of the input value space partition. An examples of one such method is, e.g., constraint programming with finite domain constraints [RvBW06].

Figure 6.1 outlines a simple algorithm for building an input value space partition by resolving dependencies between input variables. The algorithm
90 Chapter 6. Input sensitive execution-time analysis

iterates through all input value combinations and removes those that violate the predicate \( \phi \).

For example, consider two 8-bit integer input variables \( v_{i,j} \) and \( v_{i,k} \) and the predicate \( \phi = [1 \leq v_{i,j} < 3, \ v_{i,k} < v_{i,j}] \). The input value space partition \( D_{i|\phi} \) consists of a set of value combination tuples \( \langle v_{i,j}, v_{i,k} \rangle \) and from the predicate \( \phi \) we see that \( v_{i,j} \) can assume the values 1 or 2. At the same time \( v_{i,k} \) must be lower than \( v_{i,j} \). Thus the input value space partition will consist of the three tuples \( D_{i|\phi} = \{ \langle 1, 0 \rangle, \langle 2, 0 \rangle, \langle 2, 1 \rangle \} \).

The number of possible concrete input tuples for the initial input value space is \( |D_i| = 2^{8} \times 2^{8} = 2^{16} \), and the resulting number of possible concrete input tuples in the input value space partition \( |D_{i|\phi}| = 3 \).

```
1. BEGIN build_value_space_partition
2.  D_i = v_{i,0} \times v_{i,1} \times \cdots \times v_{i,n-1};
3.  D_{i|\phi} = D_i;
4.  DO
5.    FOREACH tuple d \in D_{i|\phi} DO
6.      IF d violates \( \phi \) THEN
7.        remove d from D_{i|\phi};
8.    FI
9.  END FOREACH
10. UNTIL all dependencies in \( \phi \) are solved;
11. RETURN D_{i|\phi};
12. END build_value_space_partition
```

Figure 6.1: Building an input value space partition

Notice that an input value space partition \( D_{i|\phi} \), associated with a predicate \( \phi \), is always a subset of \( D_i \) such that \( |D_{i|\phi}| \leq |D_i| \).

**Definition 6.6.** \( |D_i| \) is the concrete number of input combination tuples in the set \( D_i \).

The number of possible non-empty input value space partitions \( |P(D_i)| \) is then \( 2^{|D_i|} - 1 \). The powerset \( P(D_i) \) contains all subsets \( D_{i|\phi} \subseteq D_i \).

Each input value space partition \( D_{i|\phi} \) can be associated with two execution times, WCET (Worst-Case Execution Time) and BCET (Best-Case Execution Time).

**Definition 6.7.** \( WCET_{i|\phi} = est_{wcet}(\text{prog}_i, D_{i|\phi}) \) is an estimation of the WCET of an input value space partition \( D_{i|\phi} \) with respect to a software component behaviour \( \text{prog}_i \).

**Definition 6.8.** \( BCET_{i|\phi} = est_{bcet}(\text{prog}_i, D_{i|\phi}) \) is an estimation of the BCET of an input value space partition \( D_{i|\phi} \) with respect to a software component behaviour \( \text{prog}_i \).
Due to large number of possible program states precision is lost in a WCET-analysis tool, generating increasingly large overestimations.

A smaller input domain generates fewer program states; thus, limiting the input domain $D_{i|\phi}$ increases the precision of the WCET estimate. Because of this we experience that the WCET estimates become more accurate for the whole system due to less over approximation when analysing smaller parts of the behaviour by limiting the inputs:

$$\text{est}_{\text{wcet}}(\text{prog}, D_{i}) > \max_{\forall D_{i|\phi}}\left(\text{est}_{\text{wcet}}(\text{prog}, D_{i|\phi})\right)$$

### 6.1.2 Partitioning primitives

We define a set of primitives used in pseudo-code examples throughout the rest of this chapter.

**Primitive 6.1.** $\text{new}_D(D_{i}, \phi)$ $\rightarrow D_{i|\phi}$ creates a new input value space partition $D_{i|\phi}$ with respect to an initial input value space partition $D_{i}$ and a predicate $\phi$.

**Primitive 6.2.** $\text{new}_Vc(\phi, v_{i,j}, \text{predicate})$ $\rightarrow \phi'$ assigns the predicate considering the input variable $v_{i,j}$, in predicate $\phi$. This implies that a predicate may be more or less restricted with respect to the chosen input variable, and the predicates $\phi$ and $\phi'$ restrict the input value space partition $D_{i}$.

**Primitive 6.3.** $\text{select}_\text{var}(D_{i|\phi}, \text{strategy}_{j})$ $\rightarrow v_{i,j}$ selects the $j^{th}$ input variable $v_{i,j}$ from the input value space partition $D_{i|\phi}$ to be divided. An input variable $v_{i,j}$ is chosen for division depending on $\text{select}_\text{var}$ strategy, and such that $v_{i,j}$ can be further divided, i.e., $v_{i,j}$ is not restricted to a single value by $\phi$. Examples of strategies for variable selection are:

- **Last used** same input variable is chosen as last division if possible.
- **Next** next input variable is chosen if possible.

If no input variable fulfills these criteria, then $\text{select}_\text{var}(D_{i|\phi})$ will return nil.

**Primitive 6.4.** $\text{min}_\text{var}(v_{i,j}, \phi)$ $\rightarrow l$ returns the lowest value $l$ of $v_{i,j}$ with respect to the predicate $\phi$.

**Primitive 6.5.** $\text{max}_\text{var}(v_{i,j}, \phi)$ $\rightarrow u$ returns the highest value $u$ of $v_{i,j}$ with respect to the predicate $\phi$. 
Notice that we use “WCET” and “WCET estimate” interchangeably. If nothing else is mentioned, we always mean “WCET estimate”. The same applies for BCET.

In examples we may use the notation \( v_{i,j} \leftarrow \{0..1\} \) instead of \( 0 \leq v_{i,j} < 2 \) for the sake of readability.

### 6.1.3 WCET analysis tool assumptions

In order to restrict the problem we make a set of assumptions with respect to the WCET analysis tool used in our proposed methods. We explicitly state the assumptions below.

Let \( \text{est\_wcet}(\text{prog}_i, D_i|\phi) \) be the WCET estimate calculated by a WCET analysis tool for a software component behaviour \( \text{prog}_i \) and an input value space partition \( D_i|\phi \). Further, let \( \text{real\_wcet}(\text{prog}_i, D_i|\phi) \) be the real WCET for the component behaviour \( \text{prog}_i \) and the input value space partition \( D_i|\phi \).

For the algorithm to work correctly the following assumptions should hold:

**Assumption 6.1.** The WCET calculation should never underestimate the WCET, i.e., \( \text{real\_wcet}(\text{prog}_i, D_i|\phi) \leq \text{est\_wcet}(\text{prog}_i, D_i|\phi) \) should be true for any \( D_i|\phi \).

**Assumption 6.2.** A WCET calculation run with a single-valued input value space partition should produce a WCET estimate equal to the time for running the program with these inputs, i.e., for \( |D_i|\phi| = 1 \) then \( \text{est\_wcet}(\text{prog}_i, D_i|\phi) = \text{real\_wcet}(\text{prog}_i, D_i|\phi) \) should hold.

**Assumption 6.3.** For any two predicates \( \phi \) and \( \phi' \), and the input value space partitions \( D_i|\phi \) and \( D_i|\phi'| \), such that \( D_i|\phi' \subseteq D_i|\phi \) then \( \text{est\_wcet}(\text{prog}_i, D_i|\phi') \leq \text{est\_wcet}(\text{prog}_i, D_i|\phi) \) should hold.

**Assumption 6.4.** For any two predicates \( \phi \) and \( \phi' \), and the input value space partitions \( D_i|\phi \) and \( D_i|\phi'| \), such that \( D_i|\phi' \subseteq D_i|\phi \) then \( \text{est\_bcet}(\text{prog}_i, D_i|\phi') \geq \text{est\_bcet}(\text{prog}_i, D_i|\phi) \) should hold.

**Assumption 6.5.** For any input variable \( v_{i,j} \) there should be a natural ordering such that \( 0 \leq v_{i,j} < n \).

**Assumption 6.6.** Each input variable \( v_{i,j} \) can only be assigned a finite set of discrete values.

We claim that assumptions 6.1 and 6.3 - 6.6 are sound and valid for most type of today’s input-sensitive WCET analysis tools. However, assumption 6.2
might not always be true in reality due to imprecision introduced by a WCET analysis tool.

In order for assumption 6.2 to hold, the initial hardware state and internal program states must be known. Both these aspects are internal to the component and are not visible outside of the component. Since a component can be delivered as a pre-compiled entity, where the only visible parts are its input variable, the WCET tool will have to deal with the internal states. This leads to that the estimated WCET most likely will be overestimated. This is because the WCET analysis must consider the worst possible combination of hardware and internal states. Similarly, the BCET will be underestimated. However, technical details of WCET analysis tools are outside of the scope of this thesis.

6.2 Reusable WCET analysis

For components that are reused in different systems it is often not very meaningful to perform WCET analysis. This is because traditional WCET analysis considers only one specific usage of a system. Each component can be analysed with respect to a specific system and usage; however that prediction is only valid for that specific usage, and the usage can vary a lot. To support reuse of WCET predictions we need support for WCET analysis of different usage.

Our viewpoint is to make the analysis both reusable, and tight. To achieve this we propose a component contract that can be parametrized with usage-information to get the usage dependent WCET.

Reusable software components

The key to reuse is generality and context freeness, and when using a software component in a specific system often only parts of the component behaviour is used. Therefore generality and context-freeness leads to an increasing inability to make accurate predictions of the component behaviour for each specific use-case.

By designing the component specifically for one particular usage it can be analysed and predicted with high accuracy, but not always reused. In order to support reuse and at the same time support accurate predictions we need new parametrizable methods and frameworks [PD96].

We augment components with information that can be used to accurately predict the WCET by parametrization of the prediction with respect to different
Chapter 6. Input sensitive execution-time analysis

use cases. The WCET is the longest time it takes to execute an execution path of a program. If the WCET execution path of that program can not be executed due to limitations on the inputs, then the usage dependent WCET is reasonably lower.

Components for embedded systems especially need to be reusable both with respect to functional and reusable extra-functional specifications, since the main idea in component-based software engineering is to quickly assemble systems out of pre-fabricated reusable components.

If a piece of software is analysed with a usage independent WCET the predictions may be overly pessimistic resulting in the system being under utilized. Predictions with respect to the actual behaviour allows for a considerably tighter WCET than would be predictable from the usage independent WCET analysis. In a large software system the usage independent WCET may be orders of magnitude inaccurate compared to the usage dependent WCET.

The reusable software component concept

Let's assume two components $c_a$ and $c_b$, where each component has a set of inputs $P_a = \{p_{a,0}, p_{a,1}\}$ and $P_b = \{p_{b,0}, p_{b,1}\}$. The input value spaces $D_a$ and $D_b$ represents the possible input value combinations. Analysing these components considering the input value spaces $D_a$ and $D_b$ give the worst-case execution times $est_{wcet}(prog_a, D_a) = 1200ms$ and $est_{wcet}(prog_b, D_b) = 2200ms$.

Let's assume that the components $c_a$ and $c_b$ are part of a system where their input values are limited due to a specific usage. These limitations can be described with predicates $\phi$ and $\phi'$, and the possible input combinations are the input value space partitions $D_{a|\phi}$ and $D_{b|\phi'}$ respectively. Thus, the resulting usage dependent WCETs are $est_{wcet}(prog_a, D_{a|\phi}) = 500ms$ and $est_{wcet}(prog_b, D_{b|\phi'}) = 200ms$ as depicted in Figure 6.2.

The composite WCET of components $c_a$ and $c_b$ are $900ms$ for the usage $D_{a|\phi}$ and $D_{b|\phi'}$, as compared to the usage independent ($D_a$ and $D_b$) WCET $3400ms$. The difference is quite large, and in a system with many components the difference between the context dependent and context-free WCETs can potentially be quite large, which leads to costly over dimensioning of the system resources.

We define a contract as a function of a set of an input value space partition to determine the WCET for that specific usage scenario. The reusable WCET analysis can be divided in three steps, namely:
6.2 Reusable WCET analysis

Component WCET analysis  Analysing the WCET of the component with respect to input.

Finding input value space partitions  Finding input value space partitions in which all input combinations lead to similar execution times.

Parametric component contracts  Creating parametric contracts that express a relationship between input value space partitions and WCETs.

6.2.1 Component WCET analysis

Components are reused in different products and different usages. A different usage can substantially change the behaviour of a component. To predict the execution time with high accuracy of a complex component, components must today be reanalysed for every new usage - a very costly activity. Furthermore, it is not certain that the source code is available for components as they may be delivered as binaries by subcontractors. In this case analysis become even more costly [Kor99].

We can not know the usage of a component before the component has been deployed, therefore it is not meaningful to create input value space partitions with respect to possible usages before deployment. However, we can acquire knowledge of the WCET and BCET with respect to different input value space partitions. Dividing input value space partitions to minimize the difference
between WCET and BCET increases accuracy for the different input variable combinations.

Our method overcomes the problem that components must be reanalysed for every new usage by analysing the execution times and their probability as a function of the input of the component. We assume that execution time varies with different inputs and their associated modes.

When WCET analysis is performed with restrictions on the input parameters, it is desired to not analyse all single value input combinations, but rather a set of input value space partitions \( D_{i|\phi} \subseteq D_i \), such that:

\[
\bigcup_{\forall \phi} D_{i|\phi} = D_i
\]

As with all static WCET analyses all execution time estimates are safe over-estimations.

**Finding input value space partitions**

Theoretically, by ignoring hardware effects, each single input combination has only one fixed execution-time (not considering hardware effects). Thus, the difference between \( WCET_{i|\phi} \) and \( BCET_{i|\phi} \) of an input value space partition \( D_{i|\phi} \) shows the greatest difference between two execution times within the input value space partition. This is an indicator of how similar the execution times are in the input value space partition. A challenge is to select input value space partitions \( D_{i|\phi} \) in such a way that each different execution time is mapped to only one input value space partition.

In other words, an input value space partition \( D_{i|\phi} \) should be chosen such that every input combination tuple \( d \in D_{i|\phi} \) should produce similar execution times, i.e., \( est_{wcet}(prog, D_{i|\phi}) \) is close to \( est_{bcet}(prog, D_{i|\phi}) \) and the input value space partitions should be as large as possible, i.e., encompass as many input combinations as possible.

**Definition 6.9.** The accuracy of an input value space partition \( D_{i|\phi} \) is the difference between the highest and lowest execution time within that input value space partition, compared to the context-free WCET and BCET, as formalized in Equation 6.3.

\[
1 - \frac{WCET_{i|\phi} - BCET_{i|\phi}}{WCET_i - BCET_i}
\] (6.3)
6.2 Reusable WCET analysis

The accuracy is an indication of the difference of the execution times for each input combination in the input value space partition $D_{i\mid \phi}$. Less difference between $WCET_{i\mid \phi}$ and $BCET_{i\mid \phi}$ leads to higher accuracy. Consider the Figure 6.3, all “real execution times” are represented by an estimated WCET and an estimated BCET. If there is a large difference between the estimated WCET and BCET, then the accuracy for the input value space partition $D_{i\mid \phi}$ is low; and $D_{i\mid \phi}$ should be chosen differently.

Note that the “real execution times” in an input value space are unknown, and an estimation of the WCET with respect to an input value space partition $\text{est}_w CET(prog_i, D_{i\mid \phi})$ results in one single estimate. Also the estimation of BCET, $\text{est}_b CET(prog_i, D_{i\mid \phi})$ results in a single estimate. Thus, the only times known for an input value space partition $D_{i\mid \phi}$ are $WCET_{i\mid \phi}$ and $BCET_{i\mid \phi}$.

In the ideal case, for an input value space partition $|D_{i\mid \phi}| = 1$ the estimated $WCET_{i\mid \phi}$ and $BCET_{i\mid \phi}$ should be the same. However, due to possible differences in, e.g., the initial hardware state, the WCET will most likely be overestimated and the BCET underestimated, i.e., resulting in $WCET_{i\mid \phi} > BCET_{i\mid \phi}$ even when $|D_{i\mid \phi}| = 1$.

![Figure 6.3: Input value space partition accuracy over the input space partition $D_{i\mid \phi}$ and execution time.](image)

The sum of the difference between $WCET_{i\mid \phi}$ and $BCET_{i\mid \phi}$ of all input value space partitions should be minimized to get the highest accuracy for a component $c_i$. In the extreme to achieve the greatest accuracy, each input value space partition contains one single element; a good solution is therefore a trade-off between acceptable accuracy and the number of input value space partitions. If the difference between $WCET_{i\mid \phi}$ and $BCET_{i\mid \phi}$ of each in-
put value space partition is larger than the required accuracy the input value space partition should be chosen differently. The acceptable difference between $WCET_{i|\phi}$ and $BCET_{i|\phi}$ of the input value space partition depends on the required accuracy of the reusable analysis.

Each WCET analysis consumes some time, and it is desirable to achieve highest possible accuracy with lowest possible effort. Thus, trying to keep the number of runs with a WCET analysis tool low, is important to lower the effort of the outlined approach.

The accuracy of a whole component WCET prediction is the sum of the accuracy of each input value space partition, where each input value space partition is weighted with respect to its size.

**Definition 6.10.** Component WCET prediction accuracy is the sum of all weighted input value space partition accuracy, as formalized in Equation 6.4.

$$\sum_{\forall r} \left( \left( 1 - \frac{WCET_{i|\phi} - BCET_{i|\phi}}{WCET_i - BCET_i} \right) \cdot |D_i| \right)$$ (6.4)

The search for input value space partitions can be terminated when any of the following criteria are fulfilled:

1. WCETs for all single value input combinations are estimated.
2. A pre-defined accuracy is achieved.
3. A pre-defined effort is reached.

To find accurate input value space partitions with least effort and in bounded time we propose a binary tree search approach, dividing the input space into new input value space partitions until the required accuracy has been found, as outlined in Figure 6.4. The only data initially known is the longest and shortest execution time for the input value space partition $D_i$ (the $WCET_i$ and $BCET_i$). There are several other possible approaches to solve similar search problems, such as simulated annealing [KGV83] and evolutionary algorithms [Hol92].

### 6.2.2 Binary search algorithm description

We have described a general framework for deriving WCETs with respect to component inputs. In this section we give an example of a concrete search
Figure 6.4: Binary search for WCET over the input domain represented by input variables $v_{i,0}$ and $v_{i,1}$. Each gray block indicates that the desired accuracy has been achieved, and the figure shows how the search tree expands, and divides the inputs.
heuristics for deriving input sensitive WCET. In this example we do not express dependencies between input variables.

The arguments to the algorithm, described in Figure 6.5, are the program \( \text{prog}_i \), the input variables’ input value space partition \( \mathbf{D}_i \), and the initial predicate \( \phi \). We use the primitives introduced in Section 6.1.2 to describe the algorithm.

First an initial input variable is chosen for division. The input variable is chosen with the primitive \texttt{select\_var} and one of the strategies \texttt{next} or \texttt{last\_used} as described in Section 6.1.2.

The highest and lowest values of the input variable \( v \) are acquired in order to divide the predicate \( \phi \) into new predicates \( \phi' \) and \( \phi'' \). The predicates are used for dividing the input value space partition. The new input value space partitions \( \mathbf{D}_{i|\phi'} \) and \( \mathbf{D}_{i|\phi''} \) are added to the \textit{wcet\_list}.

The \textit{wcet\_list} can be considered to contain the “leaves” in a tree, where input value space partitions are added with WCET and BCET as the keys. The \textit{wcet\_list} is sorted with, e.g., one of the following sorting strategies:

**Sort strategy Worst accuracy** the predicate with worst accuracy is returned and removed from the \textit{wcet\_list}.

**Sort strategy Best accuracy** the input value space partition with greatest accuracy is returned and removed from the \textit{wcet\_list}.

**Sort strategy Highest WCET** the input value space partition with highest WCET is returned and removed from the \textit{wcet\_list}.

**Sort strategy Last used** the input value space partition last added is returned and removed from the \textit{wcet\_list}.

As the \textit{wcet\_list} is populated, several input value space partitions are eligible for further division. The first input value space partition in the \textit{wcet\_list} is chosen for further division. Thus, the next input value space partition is chosen depending on the sorting strategy of the \textit{wcet\_list}.

A new input value space partition \( \mathbf{D}_{i|\phi} \) is chosen with \texttt{wcet\_less.pop(sort\_strategy)} such that the first element is returned and removed from the \textit{wcet\_list} such that the following criteria are fulfilled:

1. there exists at least one input variable \( v_{i,j} \) that can be further divided with respect to \( \phi \).

2. the accuracy of \( \mathbf{D}_{i|\phi} \) is lower than the desired accuracy.
If there does not exist an input value space partition that fulfills these criteria, then \texttt{wcet\_list.pop(sort\_strategy)} will return \texttt{nil}.

A new input variable is chosen for dividing the predicate, and the algorithm iteratively refines the input value space partitions. Finally, the algorithm terminates when:

1. desired accuracy for all input value space partitions is achieved, OR
2. all input value space partitions are fully divided, OR
3. highest effort is reached (e.g., longest allowed search time has passed).

![Figure 6.5: Finding input value space partitions with binary search algorithm](image)

### 6.2.3 Approaching parametric WCET

Parametric WCET is an approach to execution time analysis where the WCET is expressed as a mathematical expression parametrized with inputs and possibly other context parameters. Our approach categorizes WCETs with respect to input parameters. The granularity depends on the effort and time put into finding value space partitions and the creation of WCET contracts.
Current problems with fully automatic parametric WCET relates to its very high complexity in terms of state space. Several simpler approaches only parametrize input variables that affect loops, which leads to potentially lower accuracy compared to our approach due to that they disregard many other program effects (e.g., mutually excluding program parts). Our approach is exact in the sense that it considers the whole program flow and relates the execution time to their corresponding input values. The complexity of our approach is related to the search depth, allowing different accuracy (and complexity). Thus, by running the algorithm a short time, a less accurate yet still reusable parametrizable WCET may be achieved; while running the algorithm longer achieves higher accuracy.

A parametrizable analysis must be parametrized with some information, i.e., the parameters. In our case we define a contract to be parametrized with usage data. Usage data is a probability mass function with occurrences of values for all inputs defined in the contract. We denote these probability mass functions usage profiles.

Usage profile

In the “real” physical world, distinct modes exist and are often engineered into systems, for example, as modes of operation. We hypothesize that modes are significant discriminators of WCET and can be utilized for more accurate WCET modelling of systems constructed out of software components.

Except the natural limitations given by, e.g., a variable’s type, the variable’s possible input values can be further constrained by the system developer. For example, a variable speed declared as an 8-bit unsigned integer can hold integer values between 0 and 255. Assuming that speed holds the value of a vehicle speed sensor, and the vehicle can not go faster than 200 km/h, then speed can be further constrained to speed $\leftarrow \{0..200\}$. The number of possible values that the variable can assume is called the value domain size. Thus, we can use the concept of input value space partitions with predicates to define a set of input combinations as a usage. We denote a predicate $\phi^U$ to define a usage. For the component $c_i$ with the input space partition $D_i$, a usage is denoted $D_{i|\phi^U}$. Thus, the input value space partition $D_{i|\phi^U}$ describes the possible values of a certain usage.

**Definition 6.11.** A usage profile $U = \langle \phi^U, Pr, pt \rangle$ is a predicate $\phi^U$ connected with a probability mass function $Pr$ and $0 \leq pt < 1$ is a given priority threshold to ignore low probability inputs (and consequently their WCETs).
The usage scenario is associated with a probability mass function $Pr : D_{i|\phi^U} \rightarrow [0, 1]$ for the occurrence of the values in $D_{i|\phi^U}$ such that the sum of the probabilities of all input combinations is 1, as outlined by Equation 6.5. Consider the example in (Figure 6.6), where the probability mass function of a usage profile $D_{i|\phi^U}$ is depicted as the light gray area. The dark gray area is the probability of a subset $D_{i|\phi} \subseteq D_{i|\phi^U}$.

$$Pr(D_{i|\phi}) = \int_{D_{i|\phi}} Pr(D_{i|\phi^U}) \, dD_{i|\phi^U}$$  (6.5)

$$\int_{D_i} Pr(D_{i|\phi^U}) \, dD_{i|\phi^U} = 1$$  (6.6)

The sum of the probabilities of all input combinations in a value partition $D_{i|\phi}$ is the probability of that input value space partition. The sum of all input combinations over the usage ($D_{i|\phi^U}$) is always 1.

Figure 6.6: Probability mass function of a usage profile $D_{i|\phi^U}$. The dark gray area is the probability of a subset $D_{i|\phi} \subseteq D_{i|\phi^U}$.
Component WCET contracts

The search algorithm may result in a large number of input value space partitions if high accuracy is required. In the worst case, the number of input value space partitions is equal to the number of input combinations. It is desired to create as few input value space partitions as possible and yet acquire as high accuracy as possible. Too many input value space partitions will result in an unmanageable amount of information. Consider two 32-bit integer inputs rendering \(2^{32}\) possible input combinations, and equally many corresponding WCETs.

To lower the number of input value space partitions, if possible two input value space partitions \(D_i|φ'\) and \(D_i|φ\) are merged if their associated WCETs are the same, i.e., \(D_i|φ' \cup D_i|φ = WCET_i|φ'\). Actually two input value space partitions can be merged if their corresponding WCETs are close enough to maintain the desired accuracy, such that \(D_i|φ' = D_i|φ' \cup D_i|φ\) and \(\max(WCET_i|φ', WCET_i|φ) - \min(BCET_i|φ', BCET_i|φ) \leq accuracy\).

A component contract is used for acquiring a WCET with respect to a usage. The contract is parametrized with the usage profile. The parametrization match the input combinations of the usage \(D_i|φ U\) with the combinations of all input value space partitions \(D_i|φ\). All input value space partitions \(D_i|φ \cap D_i|φ U \neq \emptyset\) are referred to as active input value space partitions, i.e., all input value space partitions \(D_i|φ\) that are active and their respective WCETs are eligible for the usage dependent component WCET. The usage dependent WCET \(WCET_U\) is defined in Equation 6.7.

\[
WCET_U = \max_{\forall r (D_i|φ \cap D_i|φ U \neq \emptyset)} (WCET_i|φ) \tag{6.7}
\]

A component contract is a function of a usage \(U\) that results in a usage dependent WCET (Equation 6.8).

\[
f_U: D_i|φ U \rightarrow \max_{\forall r (D_i|φ \cap D_i|φ U \neq \emptyset)} (WCET_i|φ) \tag{6.8}
\]

The usage dependent WCET together with the probability \(Pr\) of a WCET allows for a contract user to disregard usage dependent WCETs with low probabilities. Thus, the component contract can be defined as a function of an input value space partition and a priority threshold \(pt\) disregarding inputs with low probability, as defined in Equation 6.9. Notice that the priority threshold may be set to 0 for critical systems, thus considering all WCETs.
6.3 Finding WCET input combination

\[ f_{U(i)} : D_{\phi \neq U}, pt \rightarrow \max \{ WCET_{\phi \neq U} \} \quad (6.9) \]

**Contract composition**

Each input value space partition can be associated with a set of possible outputs. Abstract interpretation can be used to make a safe over-approximation of limitations on outputs given limitations on inputs by analysing possible values of the output variables. Each component produces output given the input such that the required interface \( R_i \) of component \( c_i \) is a function of the input \( f_i : P_i \rightarrow R_i \). By adding this information to the predicates the approach is composable since one component will automatically give a component usage scenario to the next connected component. SWEET [GESL06] is one tool that can produce such restrictions on the output with respect to the input.

### 6.2.4 Algorithm complexity

Our algorithm will have a worst-case behaviour of:

\[ O(|D_i| + 2 \times \log |D_i|) = O(|D_i|) \]

This is because in each step of the algorithm the size of the currently selected input value space partition is divided by two, resulting in a worst-case behaviour of \( 2 \times \log |D_i| \). Furthermore, each input value combination must be evaluated resulting in a worst-case behaviour of \(|D_i|\).

### 6.3 Finding WCET input combination

The second novel method described in this chapter is an approach to find the input combination that generates the worst-case path for a given input partition. The knowledge of this input combination can be used for steering measurement-based timing analysis approaches to select input value combinations that produce long execution times.

Knowing the input value combinations that result in the worst-case behaviour enriches the user’s knowledge about the software component of interest. It can be used for identifying bottlenecks, and hence is very useful for
further optimising the software component. Moreover, to cope with the complexity of the software and hardware of interest, WCET analysis tools often make over-approximations in their inherent subanalyses, which may result in non-tight WCET estimates [FH08]. Thus, knowing the WCET input values allows the user to get an estimate on the imprecision introduced by the WCET analysis.

Similar to the search algorithm described in the previous sections, we subdivide the input value space partitions. However, instead of subdividing until a desired accuracy has been achieved for every branch, here we subdivide the branch or branches that exhibits the worst-case execution time.

The algorithm is presented in Figure 6.7. It works by iteratively calculating WCET estimates for different partitions of the program’s input value space. In each iteration the part of the input value space with the largest calculated WCET estimate, which has not yet been subdivided, is selected and subdivided into two smaller partitions for which WCET estimates are calculated. The process continues until the selected partition corresponds to only one concrete input value combination. The partition then holds the WCET input value combination and is returned.

6.3.1 Algorithm description

The arguments to the algorithm described in Figure 6.7 are the behaviour of the component under analysis $prog_i$, an input value space partition $D_i$ and the initial predicate $\phi$. We use the primitives described in Section 6.1.2 to describe the algorithm.

An input variable is chosen for dividing the predicate in two parts. In order to as quickly as possible search for individual input combinations the last_used strategy is used in the search for the WCET input combination.

The predicate $\phi$ is set to divide $D_i$ in half; and the WCET for the resulting input value space partition is estimated and added to the $wcet_list$.

Input value space partitions inserted in the $wcet_list$ are sorted according to the highest WCET strategy. When popping an item from the queue the input value space partition with the largest WCET estimate will be returned and removed from the queue.

To guarantee that all partitions simultaneously stored in the $wcet_list$ are disjunct, i.e., the same input value combinations should only occur in one par-
6.3 Finding WCET input combination

The input variables should always be selected in some predefined order. Moreover, the currently selected input variable’s range is divided down into a single value before selecting the next input variable.

The algorithm iteratively refines the input value space partitions until one input value space partition with the size $|D_i| = 1$ has a larger WCET than all input value space partitions with the size $|D_i| > 1$.

```plaintext
BEGIN find WCET_input_value(prog, D_1), φ
  D_1 ← new_D(D_i, φ);
  WHILE v ≠ nil DO
    l ← min_var(v, φ);
    u ← max_var(v, φ);
    vc_φ ← new_var(φ, v, "l ≤ v < (l + (u − l)/2)′′");
    D_φ ← new_D(D_1, vc_φ);
    WCET_φ ← est_wcet(prog, D_φ);
    wcet_list.insert(WCET_φ, D_φ);
    D_φ ← wcet_list.pop("largest_wcet");
  END WHILE
END find WCET_input_value
```

Figure 6.7: Finding WCET input combination with binary search algorithm

6.3.2 Example

Figure 6.8 illustrates how the algorithm works. The component $c_i$ has a provided interface $P_i$ with three inputs $\{p_{i,0}, p_{i,1}, p_{i,2}\}$ and has been given the initial input value space of $\langle v_{i,0} ← 0..15, v_{i,1} ← 0, v_{i,2} ← 0..1 \rangle$ which corresponds to 16 * 1 * 2 = 32 concrete input value combinations. Input variable $v_{i,0}$ is first selected to do range division upon. This produces two new partitions for which WCET calculations are made. The $\langle v_{i,0} ← 0..7, v_{i,1} ← 0, v_{i,2} ← 0..1 \rangle$ partition gives the largest WCET estimate and the analysis therefore continues with this partition during the next iteration. This time $v_{i,0}$’s range is subdivided into 0.3 and 4..7, both producing partitions for which WCET estimates are calculated.

The division of $v_{i,0}$ continues until the value of $v_{i,0}$ which produces the largest WCET estimate when $v_{i,1} ← 0$ and $v_{i,2} ← 0..1$ has been found. Since $v_{i,1}$ only can hold a single value, the next input variable selected is $v_{i,2}$. The
Figure 6.8: Example of basic algorithm execution
division of $v_{i,2}$’s range produces two partitions, for which $\langle v_{i,0} \leftarrow 5, v_{i,1} \leftarrow 0, v_{i,2} \leftarrow 1 \rangle$ gives the largest WCET. There are no other partitions in the priority queue with a larger WCET estimate, so the iteration stops and the partition is returned.

6.3.3 Algorithm complexity and back-tracking

Our algorithm will have a best-case behaviour of $O(2^t \log |D_i|) = O(\log |D_i|)$ where $D_i$ is the input value space. This is because in each step of the algorithm the size of the currently selected input value space partition is divided by two. In many cases this will also be the algorithm’s worst-case behaviour, since the worst-case input values are likely to be found in one of the two partitions originating from the currently selected one.

Unfortunately, due to over-approximations made in the WCET analysis, this is not always true, i.e., sometimes both partitions originating from the currently selected partition get a WCET estimate smaller than, in the priority queue already stored, WCET estimate. The analysis then has to back-track and continue with this partition. For example, assume that the $\langle V_{i,0} \leftarrow 5, V_{i,1} \leftarrow 0, V_{i,2} \leftarrow 1 \rangle$ partition in Figure 6.8 gave a WCET estimate of 65 instead of 70. The analysis should then continue with $\langle v_{i,0} \leftarrow 6..7, v_{i,1} \leftarrow 0, v_{i,2} \leftarrow 0..1 \rangle$ instead of terminating. In the worst-case this type of back-tracking gives that a WCET calculation must be made for each concrete input value combination plus the WCET calculation for the binary search. Thus, the algorithm has a worst-case behaviour of $O(|D_i| + 2^t \log |D_i|) = O(|D_i|)$. To reduce the amount of back-tracking, it is important that the WCET calculation is tight, i.e., $est_{wcet}(D_i|\phi)$ is close to $max_{\phi \in D_i} real_{wcet}(\phi)$ even though $|D_i|\phi$ is large. However, if the complexity for deriving a tight WCET estimate is significantly higher than deriving a less tight WCET estimate, some back-tracking might still be worth doing.

6.4 Approaches for faster termination

A potential problem with both the algorithms outlined in Sections 6.2 and 6.3 is that many WCET calculations might be needed, especially when $|D_i|$ is large. This section outlines some approaches for faster algorithm termination.
6.4.1 Slicing the input space

A component may have inputs that affect the program flow (and thus the execution time), and inputs that do not affect the program flow (execution time). To determine which input variables that affect the execution time, and which do not, we make the observation that an input variable’s different values may cause the execution time variations in two different ways:

**Conditional branch instructions:** The value of an input variable affects the outcome of conditionals expressions in the software which in turn determines how many times an instruction can be executed. This includes all conditional instructions in the software such as loop exit conditions, switch or if-statements. The input variables’ different values may decide how many times different instructions can execute and in what order they can be executed.

**Input-sensitive instructions:** The value of an input variable makes an instruction execute with different values, and some of these values result in a different execution time for the instruction compared to other values. This might happen, for example, if the values to the instruction affect where in memory a certain load executes (given that different memory addresses have different access time). Another example is arithmetic instructions with variable execution time due to argument values.

Practically to find which inputs that affect *Conditional branch instructions* or *Input-sensitive instructions* we use slicing. Slicing is a technique for simplifying programs by focusing on selected aspects of semantics. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest [Wei81]. We perform slicing with respect to the above stated instructions, and in our methods we identify all input variables which are part of the resulting slice. Only those variables may cause the program execution time to vary due to their input value assignments.

6.4.2 Extreme value heuristic

A general observation is that it is more likely that either the smallest or the largest value in an input variable’s value domain will be the value that gives the WCET. This is especially true if the outcome of loop conditions are dependent on this input variable. As an illustrative example consider the min and max input variables in Figure 6.9. The largest number of loop iterations will occur when min is as small as possible and max is as large as possible.
6.4 Approaches for faster termination

1. // Inputs that may be given input values
2. int a[100];
3. int min=0, max=0, index=0, val=0, sum=0;
4. // The code to analyse
5. int sum_selected_irray_elements(void) {
6. int tmp = 0;
7. int i = min;
8. sum = 0;
9. while(i <= max) {
10. tmp = a[index];
11. sum = sum + tmp;
12. a[index] = val;
13. index++;
14. i++;
15. }
16. return sum;
17. }

Figure 6.9: Illustrating example of extreme values in a loop.

Our extreme-value search heuristic builds on this observation. It modifies the algorithm outlined in Figure 6.7 as follows: whenever a variable \( v \) with a range \( l..u \) is selected for the first time it will be divided into three new ranges; \( l..l, l+1..u-1, \) and \( u..u \), each producing an input value space partition for which WCET estimates are calculated. If the medium range (\( l+1..u-1 \)) is selected for further division, normal binary range division is performed.

Figure 6.10 illustrates how the heuristic works. The program has two input variables \( v_{i,0} \) and \( v_{i,1} \), with given input values of \( 0..15 \) and \( 0..255 \) respectively. \( v_{i,0} \) is first selected to do input range division upon. This produces three new partitions for which WCET estimates are calculated. The \( \langle v_{i,0} ← 0, v_{i,1} ← 0..255 \rangle \) partition gets the largest WCET estimate and the analysis continues with a division of \( v_{i,0} \). This produces three new partitions for which WCET estimates are calculated. The \( \langle v_{i,0} ← 0, v_{i,1} ← 255 \rangle \) partition gives a WCET estimate which is larger than all other partitions in the priority queue. Thus, the WCET was given when \( v_{i,0} \) and \( v_{i,1} \) were assigned its minimum and maximum input values respectively.

6.4.3 User interaction

Another option for improving the overall analysis time, is to let the user provide an input value space partition in which he/she believes hot spots are found. For example, in Figure 6.8 the user might believe that one worst-case is when \( v_{i,2} = 1 \) and \( 0 ≤ v_{i,0} ≤ 6 \). Then the analysis can be started with an initial set of partitions according to the user's assumptions, e.g., \( \langle v_{i,0} ← 0..6, v_{i,1} ← \)
Chapter 6. Input sensitive execution-time analysis

Figure 6.10: Example of extreme value heuristic
We revisit the example outlined in Section 5.5, and discuss the component \( c_{\text{speed}} \) (Speed Limit) in detail, and explore the resulting input value space partitions. \( c_{\text{speed}} \) has 3 variables \( \text{ACC Max Speed} \) (AMS), \( \text{Road Signs Enabled} \) (RSE) and \( \text{Road Sign Speed} \) (RSS). The input value space partition \( D_{\text{speed}} \) has a size \( |D_{\text{speed}}| = 301 \cdot 2 \cdot 301 = 181202 \) possible input combinations.

The algorithm iteratively divides the input domain in input value space partitions until the desired accuracy is achieved. In this example we attempt to achieve 100% accuracy (according to our definition of accuracy, Definition 6.10). We explore two heuristics of the select \_ var in this example (see Primitive 6.3), \text{Last used} with \text{eXtreme value heuristics} (LX) and \text{Last used} (L). The strategy \text{Last used} is described in Section 6.1.2 and \text{eXtreme value heuristics} is described in Section 6.4.2.

The LX strategy finds input value space partition with 100% accuracy much faster than the L strategy. It only requires 20 WCET tool runs for the entire input domain, as compared to the L strategy that requires more than 1000 runs. Thus, we see that it is important to choose appropriate heuristics.

Our algorithm has divided the inputs in their respective input value space partitions, as shown in Table 6.1. It is straightforward to extract 3 predicates from these input value space partitions for the contract \( f_{\text{speed}} \), that are used for the parametrization of speed. We see that the input AMS is the same for all input value space partitions, thus does not affect the WCET. The resulting predicates that form the contract are shown in Table 6.2.

<table>
<thead>
<tr>
<th>( D )</th>
<th>WCET</th>
<th>RSE</th>
<th>AMS</th>
<th>RSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D_{\text{speed}} )</td>
<td>384</td>
<td>105</td>
<td>0.300</td>
<td>0</td>
</tr>
<tr>
<td>( D_{\text{speed},0} )</td>
<td>384</td>
<td>384</td>
<td>0.300</td>
<td>1</td>
</tr>
<tr>
<td>( D_{\text{speed},1} )</td>
<td>188</td>
<td>188</td>
<td>0.300</td>
<td>0</td>
</tr>
<tr>
<td>( D_{\text{speed},2} )</td>
<td>105</td>
<td>105</td>
<td>0.300</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: \( c_{\text{speed}} \) resulting input value space partitions
A software component that is reused in different settings is used with different usage profiles, i.e., with different inputs. Unfortunately, a change in the usage of a component can also invalidate past experience about the component’s quality of performance. Indeed it is safe to assume the worst possible usage scenario for estimating the component’s performance, however, this results in pessimistic and inaccurate system property predictions. Especially for embedded real-time systems, where not only the correct predictions are important, but also resource consumption, it is necessary to have more accurate methods.

One possible way of acquiring accurate predictions is to perform predictions for every new usage profile. However, this undermines the CBSE main action reuse. Hence, it is desired to gain accurate predictions of component properties in a reusable way.

In this chapter we have introduced two novel methods based on static WCET analysis and structured search over the input domain of a reusable software component. The first method provides a parametrizable and reusable WCET for reusable software components. We have introduced a concept of WCET analysis accuracy based on the difference between the WCET and BCET for a given input domain. With the accuracy as a search criteria we guide the structured search to produce as accurate predictions as possible in as short time as possible.

In our experience it is a matter of several man weeks to setup, learn and effectively use many of the commercial and research WCET analysis tools. The work effort is effectively moved from the system developer to the component developer, and for every reuse there are potentially big time gains. For many WCET tools it is also required to do a lot of “hands on tuning” and adaptations of the code and annotations of the inputs in order to run the WCET tool.

Due to less over approximations when analysing smaller parts of the behaviour by limiting the inputs we have also observed that the WCET become more accurate for the whole system.

Another problem that arises from reusing components is that they can be aligned differently in memory between systems. In a system with caches this
may cause cash-lines to be invalidated frequently as a result from interference between components. A possibility for limiting this problem is to predefine possible alignments in memory for each component. In this way, interference may be analyzable, and predictions can be made tighter.

The second novel method derives an input combination that triggers the execution of the WCET path, that produces the WCET. In this method we use techniques similar to the first outlined method in the sense that we use static WCET analysis and structured search over the component input domain. But rather than searching the entire input domain, we iteratively remove parts of the domain that does not generate execution of the WCET program path.

The knowledge of the WCET input combination is useful for, e.g., producing efficient test cases for measurement based WCET.

Finally, we show with complexity analysis that these methods can be used effectively.
Allocating components to real-time tasks

Following the extension of the Component-Based Software Engineering (CBSE) process presented in Chapter 5, this chapter moves beyond the WCET analysis and presents a method for optimizing the resource usage in component-based real-time systems. The method presented is based on real-time analysis, resource consumption calculations, and genetic algorithms for deriving allocations from components to tasks that are optimized for low resource consumption, while maintaining stipulated real-time requirements.

Allocating components to tasks, and scheduling of tasks are both complex problems due to the exponentially growing search space imposed by the amount of possible combinations. Simulated annealing [KGV83] and genetic algorithms [Hol73, FB70, Hol92] are examples of algorithms that are frequently used for optimization of such problems. However, to be able to use such algorithms, a framework to calculate properties, such as memory consumption and CPU-overhead, is needed.

This chapter describes a general framework for reasoning about trade-offs concerning allocation of components to tasks while preserving extra-functional requirements. Temporal constraints are verified and the allocations are optimized for low memory consumption and CPU-overhead. The framework is evaluated using industrially relevant component assemblies, and the results show that CPU-overhead and memory consumption can be reduced by as much as $\sim 50\%$ and $\sim 30\%$ respectively.
7.1 Introduction

In many component-based Embedded Real-Time Systems (ERTS) there is no explicit strategy for deriving real-time tasks from components, i.e., translating a system described with software components to a system of run-time entities, e.g., tasks. This has lead to that many systems use a one-to-one allocation from components to tasks creating one real-time task from each and every component. If available memory is limited by physical footprint, cost, or power consumption constraints, or if low overhead is needed, the allocation from components to tasks need to be performed with a more sophisticated approach.

Let’s consider two components $c_a$ and $c_b$, depicted in Figure 7.1, they can be allocated to tasks in three different ways:

1. $c_a$ in one task $\tau_1$, and $c_b$ in one task $\tau_2$.

2. $c_a$ and $c_b$ in the same task $\tau_1$.

3. $c_b$ and $c_a$ in the same task $\tau_1$ (reverse order compared to allocation 2).

Figure 7.1: Possible allocations of 2 components to tasks.
7.1 Introduction

The allocations (2) or (3) will, e.g., result in lower memory consumption. If the components have different periodicity a co-allocation may lead to high processor utilization, which may be acceptable if the main concern is to minimize memory usage. However depending on their real-time constraints the allocations may be feasible or infeasible; thus, the allocation must be evaluated with respect to real-time constraints, and some allocations may not be feasible, i.e., deemed unschedulable. Another issue is the number of task switches that results from a specific allocation, as well as processor utilization that may vary a lot depending on the timing constraints of the components. All these things may be traded against each other: memory, CPU overhead and schedulability.

In a system with only a small number of components it is trivial to find the best allocation from components to tasks, but already in a system with 10 components the number of possible allocations exceeds several millions, e.g., a system with 4 components results in 73 different allocations, and a system with 10 components can be allocated in 58941091 different ways (sets of segments)\(^1\). This is because 10 components can be allocated to any number between 1 and 10 tasks, with any combination of components in each task, and the order of components in each task is significant. In such systems it may be difficult to find a good allocation manually. Many industrial ERTS consist of as many as 50 different components. In order to remedy this challenge we have defined a theoretical framework for reasoning about the allocation from components to tasks. The reasoning framework is well suited to be applied with optimization techniques such as, e.g., genetic algorithms.

7.1.1 Component to task allocation

Definition 7.1. \(\Gamma_T^i\) denotes a component transaction that is time-triggered. Components and component transactions are defined in Section 5.4.

Definition 7.2. \(\Gamma_E^i\) denotes a component transaction that is event-triggered. Components and component transactions are defined in Section 5.4.

An isolation set \(I\) defines a relation between components that should not be allocated. It is described as a set of component pairs \(\{ (c_a, c_b), (c_c, c_d) \}\) that define what components may not be allocated to the same task. There may be memory protection requirements or other legitimate engineering reasons to

\(^1\)Sets of segments can be calculated with the recursive formulae \(a(n) = (2n - 1) \cdot a(n - 1) - (n - 1) \cdot (n - 2) \cdot a(n - 2) [Slo08]\), where \(a(n)\) give the number of possible allocations, and \(n\) is the number of components
avoid allocating certain combinations of components; for example, if a com-
ponent has a highly uncertain WCET. The predicate $Iso(a, b)$ defines that com-
ponents $c_a$ and $c_b$ has an isolation requirement, and should not be co-allocated.

Definition 7.3. $Iso(c_a, c_b)$ defines an isolation requirements between two com-
ponents $c_a$ and $c_b$. The isolation requirement implies that the components $c_a$
and $c_b$ may not be allocated to the same task.

Components may have precedence relations in terms of triggering. If a com-
ponent is triggered by a preceding component, they are said to have a
precedence relation, component $c_a$ precedes component $c_b$ means that com-
ponent $c_b$ is triggered by component $c_a$. $prcd(c_a, c_b, \Gamma_d)$ defines that component $c_b$ is preceded by component component $c_a$, i.e., component $c_b$ is triggered by component $c_a$ and both component are part of transaction $\Gamma_d$.

Definition 7.4. $prcd(c_a, c_b, \Gamma_d)$ defines that component $c_a$ triggers component $c_b$ and both components $c_a, c_b \in \Gamma_d$ are part of transaction $\Gamma_d$.

7.2 Allocating components to real-time tasks

Temporal constraints are of great importance when dealing with ERTS and
tasks control the execution of software. Hence, components need to be allo-
cated to tasks in such a way that temporal requirements are met, while at the
same time resource usage is minimized. Given an allocation the framework
determines if the allocation is feasible and the memory consumption and task
switch overhead are calculated. To impose timing constraints, the framework
defines end-to-end timing requirements and denote them transactions. Transac-
tions are defined by a sequence of components and a deadline, before which the
sequence of tasks must have finished their execution (relative to the triggering
of the transaction). Thus, this work has three main concerns:

1. Verification of allocations from components to tasks.

2. Calculating system properties for an allocation

3. Minimizing resource utilization

7.3 Allocation framework

The allocation framework is a set of models for calculating properties of al-
locations of components to tasks. The properties calculated with the frame-
work are used for optimization algorithms to find feasible allocations that fulfil given real-time requirements and minimizes memory consumption and CPU-overhead.

In order to not over constrain the allocation from components to tasks it is important to have tight WCET estimates, which has been a problem for reusable software components with rich behaviours. Also, if several components are allocated to one task, the error of each component will scale linearly to form a large task output jitter, possibly making it harder to schedule the task set.

For a task set $A$ that has been created from components with a one-to-one allocation, it is trivial to calculate the system memory consumption and CPU-overhead since each task will inherit the same properties as the basic component. When several components are allocated to one task we need to calculate the appropriateness of the allocation as well as the tasks properties. For a set of components, $c_1,...,c_n$, allocated to the task set $A$, the following resource properties are considered.

- **CPU-overhead $p_A$**
- **Memory consumption $s_A$**

Each component $c_i$ has a pre-defined maximum stack size. Since all components allocated to one task will use the same stack, the stack of the task is equal to the maximum size of the stack of all components allocated to the task. The CPU overhead $p_A$ and the memory consumption $s_A$ for a task set $A$ are formalized in Equations 7.1 and 7.2:

$$p_A = \sum_{\forall n (\tau_n \in A)} \frac{\rho}{T_n} \quad (7.1)$$

$$s_A = \sum_{\forall n (\tau_n \in A)} (stack_n + \beta) \quad (7.2)$$

where $\rho$ is the context switch time and $\beta$ is the task control block of the system (as described in Section 5.4). $p_A$ represents the sum of the task switch overhead divided by the period for all tasks in the task set, and $s_A$ represents the total amount of memory used for stacks and task control blocks for all tasks in the task set.
7.3.1 Constraints on allocations

There is a set of constraints that must be considered when allocating components to tasks, these are:

- Component isolation
- Intersecting transactions
- Periodic and aperiodic events and their period times
- Schedulability

Each constraint is discussed in detail below, and we use the notation of components and tasks as described Section 5.4.

Isolation

It is not realistic to expect that components can be allocated in an arbitrary way. There may be explicit dependencies that prohibits that certain components are allocated together, the isolation set $I$ defines which components may not be allocated together. There may be specific engineering reasons to why some components should be separated. For instance, it may be desired to minimize the output jitter for some tasks, e.g., components with highly uncertain WCET could be isolated. There may also be safety integrity reasons to separate certain components. Hence it must be assured that two components that are defined to be isolated do not reside in the same task. This is validated with Equation 7.3

$$\neg \exists_n (c_i \in Z_n \land c_j \in Z_n \land Iso(i, j))$$  \hspace{1cm} (7.3)

where there must not exist any task $\tau_n$ that allocates two components $c_j$ and $c_k$ such that these components have an isolation requirement. $Iso(i, j)$ is a predicate returning if there is a isolation requirement between component $c_i$ and $c_j$, and $Z_n$ is the set of allocated components in task $\tau_n$, as described in Section 5.4.

Intersecting transactions

If two or more component transactions intersect one task, there are different strategies of how to allocate the component to tasks. A task inherits the trigger(s) of the first allocated component, i.e., the component $c_i \in Z_n$ where $c_i$
is the first element in $Z_n$, therefore it is important to consider which components are allocated to a task with respect to multiple transactions. We do not want a task to be event triggered, while at the same time having timing constraints from a time-triggered transaction. The feasibility condition taking care of intersection transactions is described in Equations 7.4 and 7.5.

\[
\neg \exists_n \left( \left( c_i \in \Gamma^E_j \land c_m \in \Gamma^E_k \land c_i \in Z_n \land c_m \in Z_n \right) \land 
\left( \text{prcd}(c_m, c_i, \Gamma^E_k) \lor \text{prcd}(c_m, c_i, \Gamma^E_j) \right) \right), j \neq k \tag{7.4}
\]

if a component $c_i$ is allocated to a task $\tau_n$, and the component is part of an event triggered transaction $\Gamma^E_j$, and a component $c_m$ is also allocated to the task $\tau_n$ and is part of another event triggered transaction $\Gamma^E_k$, then no preceding components in either transaction shall be allocated to the same task. This is because a component should be triggered by both transactions. The triggering of a component is inherited from one component. If that component is triggered by only one of the transactions then it is not possible to guarantee the timing requirements. $Z_n$ is the set of allocated components in task $\tau_n$, as described in Section 5.4.

\[
\neg \exists_n \left( \left( c_i \in \Gamma^T_j \land c_m \in \Gamma^E_k \land c_i \in Z_n \land c_m \in Z_n \right) \land 
\left( c_m \in \Gamma^E_k \land \text{prcd}(c_m, c_i, \Gamma^E_k) \right) \right), j \neq k \tag{7.5}
\]

if a component $c_i$ is allocated to a task $\tau_n$ and the component is part of an event triggered transaction $\Gamma^E_j$, and a component $c_m$ allocated to the task $\tau_n$ and the component $c_m$ is part of a time triggered transaction $\Gamma^T_k$, then no preceding components that are part of the event triggered transaction $\Gamma^E_j$ shall be allocated to the task $\tau_n$. This is because, in our model, the task inherits the triggering of only one component, and if that trigger is event driven, then it is impossible to guarantee the timing constraints of the time-triggered transaction.

**Triggers**

Some allocations from components to tasks can be performed without impacting the schedulability negatively. A component that triggers a subsequent component can be allocated into a task if it has no other explicit dependencies, see (1) in Figure 7.2. Components with the same period times can be allocated to
the same task if they do not have any other explicit dependencies, see (2) in Figure 7.2. Since a task may only have one trigger, time triggered components with the same or harmonic period can be triggered by the same trigger and thus allocated to the same task. However, event triggered components may only be allocated to the same task if they are in fact triggered by the same event, see (3) in Figure 7.2.

Figure 7.2: Component to task allocation considering triggers.

**Schedulability**

Schedulability analysis is highly dependent on the scheduling policy chosen. Depending on the system design, different analysis approaches have to be considered. The task and task transaction meta-models are constructed to fit different scheduling analyses. In the evaluations in Chapter 8 we have used fixed priority exact analysis. However, the model can easily be extended with jitter and blocking for real-time analysis techniques that allow those properties. The framework assigns each task a unique priority pre run-time, and it uses exact analysis for schedulability analysis, together with the Bate and Burns [BB99] approach for verifying that the transaction deadlines are met.

### 7.4 Using the framework

An allocation can be performed in several different ways. In a small system all possible allocations can be evaluated and the best chosen. For a larger system, however, this is not possible due to the combinatorial explosion of possible allocations. Different algorithms can be used to find a feasible allocation of
components to tasks. For any algorithm to work there must be some way to evaluate an allocation. The proposed allocation framework can be used to calculate schedulability, CPU-overhead and total memory load. Each allocation is compared to the worst and best possible allocations. The worst-case allocation is a one-to-one allocation where every component is allocated to one task, and the best-case allocation is all components allocated to one single task. However, allocating all components to one task very seldom produces a schedulable task set because of too varying timing constraints between different components and transactions.

Simulated annealing [KGV83], genetic algorithms [Hol73, FB70] and bin packing [Baa88] are well known algorithms often used for optimization problems. We briefly discuss how these algorithms can be used with the described framework, to perform component to task allocations.

**Bin Packing (BP) [Baa88]** is a method well suited for our framework. In [JO95] a bin packing technique that handles arbitrary conflicts (BPAC) is presented. The BPAC model constrains certain elements from being packed into the same bin, which directly can be used in our model as the isolation set \( I \), and the bin-packing feasibility function is the schedulability.

**Genetic algorithms (GA) [FF95]** can solve, roughly, any problem as long as there is some way of comparing two solutions, judging which is better according to a fitness criteria. The proposed framework gives the possibility to use the properties memory consumption, CPU-overhead and schedulability as grades for an allocation. In, e.g., [MBD98] and [MBMB98], genetic algorithms are used for scheduling complex task sets and scheduling task sets in distributed systems.

**Simulated annealing (SA) [KGV83]** is a global optimization technique that is regularly used for solving NP-Hard problems. An energy function consists of a schedulability test, the memory consumption and CPU-overhead. In, e.g., [TBW92][CA95] simulated annealing is used to place tasks on nodes in distributed systems.

### 7.5 Genetic algorithm setup

In order to evaluate the performance of the allocation framework, it has been implemented. We have chosen to perform a set of allocations and compare
the results to a corresponding one-to-one allocation where each component is allocated to a task. The allocations are evaluated with respect to feasibility, memory consumption and CPU overhead.

The implementation is based on GA, and as Figure 7.3 shows, each gene represents a component and contains a reference to the task in which it is assigned. Each chromosome represents the entire system with all components assigned to tasks. Each allocation produced by the GA is evaluated by the framework, and is given a fitness value depending on the feasibility, memory consumption and CPU overhead of the allocation.

![Chromosome diagram](image)

Figure 7.3: The genetic algorithm view of the component to task allocation; a system with ten components, allocated to four tasks.

### 7.5.1 Fitness function

The fitness function is based on the feasibility of the allocation together with the memory consumption and CPU overhead. The feasibility part of the fitness function is mandatory, i.e., the fitness value for a low memory and CPU overhead can never exceed the value for a feasible allocation. The feasibility function consists of the following parts, with their respective possible values used in the fitness function:

- \( I \) which represents component isolation \([1, 100]\).
- \( IT \) representing intersecting transactions \([1, 100]\).
- \( Tr \) representing triggers \([1, 100]\).
- \( Sc \) represent schedulability \([1, 100]\).

Consider that each of these feasibility tests is assigned a value greater than 1 if they are true, and a value of 0 if they are false. The parameter \( n \) represents
the total number of components. Then, the fitness function can be described as with Equation 7.6.

\[
\text{Fitness} = \left( (I + IT + Tr + Sc)F + \left( \frac{n}{s_A} + \sum_{\forall\{t_i \in A\}} \frac{\rho \cdot n}{T_i} O \right) \right) \cdot (I \cdot IT \cdot Tr \cdot Sc + 1)
\]

where the fitness is the sum of all feasibility values times a factor \(F\), added with the inverted memory usage and performance overhead, times a factor \(O\), and \(F \gg O\). The total fitness is multiplied with 1 if any feasibility test fail, and the products of all feasibility values plus 1 if all feasibility tests succeed.

### 7.6 ACC example - transforming components to tasks

We revisit the example outlined in Section 5.5, and discuss the allocation of components to tasks in different ways. For the 5 components outlined in Section 5.5 there exists 501 different possible allocations. However, only a few of them are feasible. We show 5 different feasible allocations in Figure 7.4. In this example we choose to consider allocation (E) in the figure, i.e., the allocation of all components to one task.

A search heuristics searches for feasible allocation with high performance (i.e., low CPU overhead and low memory consumption), and it uses the framework to evaluate an allocation. The framework calculates feasibility and performance. The feasibility is calculated with respect to:

- Isolation
- Triggers
- Intersecting transactions
- Schedulability

The performance of an allocation is based on:

- CPU overhead
- memory consumption
An allocation is compared to the optimal allocation, where the optimal allocation is all components allocated to one task (independent of feasibility).

Let’s study the allocation of all components to one task (allocation (E) in Figure 7.4). This allocation is optimal with respect to CPU overhead and memory consumption, since there is no other allocation for the given components that can yield a lower CPU overhead or memory consumption. We validate the feasibility of the allocation of the components given in Section 5.5 with the different WCETs given by the usage dependent WCET analysis, and the usage profiles $U_1$ and $U_2$, as presented in Section 5.5.1.

We reintroduce the components and properties in Table 7.1. We discuss the evaluation of the allocation with respect to the feasibility requirements and performance properties for the different WCETs.

<table>
<thead>
<tr>
<th>Component</th>
<th>Name</th>
<th>$(S_i, Q_i, m_i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed limit</td>
<td>$c_{speed}$</td>
<td>($50Hz, 20, 1024$)</td>
</tr>
<tr>
<td>Object recog.</td>
<td>$c_{obj}$</td>
<td>$(c_{speed}, 20, 512)$</td>
</tr>
<tr>
<td>Brake assist</td>
<td>$c_{brake}$</td>
<td>$(c_{obj}, 20, 512)$</td>
</tr>
<tr>
<td>Logger</td>
<td>$c_{log}$</td>
<td>$(10Hz, 100, 2048)$</td>
</tr>
<tr>
<td>ACC controller</td>
<td>$c_{acc}$</td>
<td>$(c_{obj}, 20, 2048)$</td>
</tr>
</tbody>
</table>

Table 7.1: The five components Speed limit, Object recognition, Brake assist, Logger, and ACC controller

### 7.6.1 Isolation, triggers and intersecting transactions

In the example there are no isolation requirements between any components; thus, the allocation is feasible with respect to this requirement.

There are some constraints on allocating different types of triggers to the same task. In this allocation there exists two different triggers, one 50Hz clock and on 10Hz clock. The period times of these clocks are harmonic. Thus, the allocation is feasible with respect to triggers. However, by allocating all components to one task, it is required to use the lowest period time in order to fulfil all components requirements.

There are also no intersecting transactions in this allocation. However, intersecting transactions are exclusively a problem when there exist event-triggered transactions.
7.6 ACC example - transforming components to tasks

7.6.2 Schedulability

By adding the components’ WCETs we get the task’s WCET. We present the task’s WCET for the usage independent \((ui)\) WCET as well as for the usage profiles \(U_1\) and \(U_2\), together with the utilization in Table 7.2. The utilization is calculated with:

\[
U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i}
\]

We see right away that the specified allocation is only feasible for usage profile \(U_2\).

We also have to consider the transactions \(\Gamma^T_1\), \(\Gamma^T_2\) and \(\Gamma^T_3\). The transactions are presented in Table 7.3. We see that one single execution if the task will fulfil the precedence relation of all transactions. \(c_{\text{break}}\) and \(c_{\text{acc}}\) are executed after \(c_{\text{obj}}\), which is executed after \(c_{\text{speed}}\). Furthermore all deadline \(d_{c_1}, d_{c_2}\) and \(d_{c_3}\) are less than the task WCET.

<table>
<thead>
<tr>
<th>Period</th>
<th>WCET</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>2927</td>
<td>2650</td>
</tr>
</tbody>
</table>

Table 7.2: Period time, WCET and Utilization for all components allocated to one task

<table>
<thead>
<tr>
<th>(\Gamma^T_0)</th>
<th>(N_t)</th>
<th>(c_{\text{speed}} \rightarrow c_{\text{obj}} \rightarrow c_{\text{break}})</th>
<th>(d_{c_1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma^T_1)</td>
<td>(c_{\text{speed}} \rightarrow c_{\text{obj}} \rightarrow c_{\text{acc}})</td>
<td>1800</td>
<td></td>
</tr>
<tr>
<td>(\Gamma^T_2)</td>
<td>(c_{\text{log}})</td>
<td>5000</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3: The three transactions in the ACC

7.6.3 Other allocations

We show 5 different allocations in Figure 7.4. Some of them exhibit bad performance and/or are not feasible. We mention that allocation (A) in the figure is feasible for all usage profiles, but this allocation exhibits the worst performance.
The best feasible allocation for the usage independent (ui) WCET prediction is allocation (C) in the figure. For usage profile $U_1$ the best feasible allocation is allocation (D) in the figure. And for the usage profile $U_2$, as we have discussed in this example, the best allocation is all components allocated to one task (allocation (E) in Figure 7.4).

![Figure 7.4: 5 feasible component to task allocations of the ACC example.](image)

### 7.7 Summary

Resource efficiency is important in ERTS, both regarding performance and memory. Schedulability, considering resource efficiency, has gained much focus, however the allocation between components to tasks has gained very little focus. Hence, in this Chapter we have described an allocation framework for allocating components to tasks, to facilitate existing scheduling and optimization algorithms such as genetic algorithms, bin packing and simulated annealing. The framework is designed to be used during compile-time to minimize resource usage and maximize timeliness. The framework can also be used it-
eratively in case of design changes; however with some obvious drawbacks on the results. The framework can easily be extended to support other optimization criteria besides task switch overhead and memory consumption.

Results from simulations show that the framework gives substantial improvements both in terms of memory consumption and task switch overhead. The described framework also has a high ratio in finding feasible allocations. Moreover, in comparison to allocations performed with a one-to-one allocation our framework performs very well, with \( \sim 30\% \) reduced memory size and \( \sim 50\% \) reduced task switch overhead. The simulations show that the proposed framework performs allocations on systems of a size that covers many embedded systems, and in a reasonable time for an off-line tool. We have also shown how CPU load and deadline laxity affects the allocation.
Chapter 8

Empirical results

In this chapter we present the evaluations and results of the technical contributions presented in Chapters 6 and 7.

8.1 Input sensitive execution-time analysis

In this section we begin with presenting our prototype tool for creating WCET contracts and finding WCET input values. We continue with presenting evaluations of the methods outlined in Sections 6.2 and 6.3. The evaluations are performed with industrial and academic software components. In particular we evaluate the effort needed to gain improvement in the methods.

8.2 Prototype tool

Many of the outlined ideas presented in Chapter 6 have been implemented in our prototype tool. Our prototype tool is fully automatic in the sense that it performs program slicing for deriving the input variables whose different values may cause the program execution time to vary, allowing the input data search space to be reduced for any type of measurement-based or static analysis method to derive the WCET input values.

It is possible to provide an pre-defined set of inputs, overriding the slicing. The user may have to provide limitations for the inputs if the tool can not determine them automatically. The tool tries to analyze the source code to determine max and min values, thus if the source code is not available the
inputs limits must be provided by either the user or the WCET tool (in our case SWEET).

The input domain is divided according to one of the heuristics presented in Chapter 6, continually analysing the WCET and (if required) BCET for the current value space partition. The prototype tool graphically presents the WCETs and BCETs and show the derived WCET partitions. Statistics other than execution time are also logged such as, e.g., flow analysis time and low-level analysis time.

Our tool is based on the WCET analysis tool SWEET (SWEdish Execution time Tool) [SWE06]. Unlike most WCET analysis tools, SWEET is integrated with a compiler and performs the flow analysis on its intermediate code representation (NIC) [WPO08]. SWEET includes an input-sensitive flow analysis called Abstract Execution (AE) [GESL06], which is a form of symbolic execution. The AE has several options for trading precision and analysis time. SWEET also do a type of slicing to restrict the AE to only analyse those statements and variables that may affect the program flow [SEGL06].

Constraints on input values are given in SWEET’s annotation language. Numeric variables are constrained by intervals. Pointer constraints are sets of abstract addresses, each representing a range of NIC addresses. Annotations
can constrain the variable values in specific program points. Normally, when constraining inputs, this is the program entry point. The value space partitions are directly translated to annotations for SWEET in the prototype tool.

The low-level analysis of SWEET [Eng02] supports the NECV850E and ARM9 processors. Three calculation methods are supported: a path-based (PB) calculation, a global Implicit Path Enumeration Technique (IPET) [LM95] method, and a hybrid clustered method [Erm03, ESE05].

The clustered calculation method is a combination of PB and IPET. With respect to complexity and expressiveness PB and IPET have different strengths and weaknesses. IPET expresses the search of the WCET as an Integer Linear Programming problem where the program execution time is to be maximized under some constraints on the execution counts of the basic blocks. With this technique, an execution path is defined by the set of the executed blocks with their respective execution counts but the order in which they are executed is not expressed. PB expresses the search of the WCET as a graph, searching for the overall path with the longest execution time [HAM+99], and can be solved with, e.g., Dijkstra’s algorithm [HSRW98].

8.2.1 Benchmarks

We have used programs from the Mälardalen WCET Benchmark suite [SWE06], together with some source codes provided by our industrial partner [BEG+08], to evaluate our methods. We have only included benchmarks which may be run with many different input values. Table 8.1 gives some details of the benchmark used. #LOC gives lines of C code, #Vars give the number of input variables and |D| represents the size of the input domain.

8.2.2 Reusable WCET

All benchmarks’ inputs have been divided until the point where (i) all inputs are single conditions, or, (ii) the accuracy is 100%. For some benchmarks this has not been possible due to limited time or limitations in the WCET analysis tool.

We have run all the benchmarks with four different strategies for dividing the input value space partition. These strategies are, firstly, Last used, where the next variable to be divided in the value space partition is the same variable as was divided last. This strategy is similar to a depth first tree search algorithm, since we use binary search to find the value space partitions. The
Table 8.1: Benchmark programs used

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LOC</th>
<th>Vars</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>Cyclic redundancy check computation on 40-bytes of data.</td>
<td>258</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>edn</td>
<td>Finite Impulse Response (FIR) filter calculations.</td>
<td>285</td>
<td>3</td>
<td>2.10</td>
</tr>
<tr>
<td>inssort</td>
<td>Insertion sort on a reversed array of size 10.</td>
<td>92</td>
<td>10</td>
<td>10.220</td>
</tr>
<tr>
<td>icomplex</td>
<td>Nested loop program.</td>
<td>64</td>
<td>2</td>
<td>2.16</td>
</tr>
<tr>
<td>lcdnum</td>
<td>Read ten values, output half to LCD.</td>
<td>64</td>
<td>2</td>
<td>2.16</td>
</tr>
<tr>
<td>ns</td>
<td>Search in a multi-dimensional array.</td>
<td>535</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>nsichneu</td>
<td>Simulates an extended Petri net. Generated code with more than 250 if-statement.</td>
<td>4253</td>
<td>6</td>
<td>2.16</td>
</tr>
<tr>
<td>esab_mod</td>
<td>Industrial code developed by CC-Systems and Esab to control welding machine</td>
<td>3064</td>
<td>17</td>
<td>2.17</td>
</tr>
<tr>
<td>task1</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>55</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>task3</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>58</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>task4</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>77</td>
<td>18</td>
<td>2.14</td>
</tr>
<tr>
<td>task5</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>86</td>
<td>8</td>
<td>2.16</td>
</tr>
<tr>
<td>task7</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>123</td>
<td>26</td>
<td>2.17</td>
</tr>
</tbody>
</table>

Second strategy is Next, where the next variable to be divided is the next variable in their annotated order, giving a search similar to breadth first. Each of these strategies are also analyzed with the basic and extreme heuristics in determining how the value space partitions is divided. Following basic heuristics, the value space partition is divided in two new value space partitions, with respect to the chosen variable. In the extreme heuristics the value space partition is divided equally to the basic except for the first time, when it is divided in three new value space partitions, with respect to the chosen variable; those three partitions are the min-value, the max-value and the rest.

For each benchmark we show the number of analyses (#Runs) required to reach a specific accuracy. The accuracy is in comparison to a single run, annotated with the initial value space partition (that has not been divided). The flow facts used in SWEET are presented below, and generated flow facts for each benchmark are presented in Table 8.2.

- **ip**: infeasible paths, i.e., paths that can not be taken together.
- **ep**: excluding pairs, i.e., nodes that can not be visited together.
- **ina**: infeasible nodes calculations for all iteration, i.e., nodes that can not be visited.
8.2 Prototype tool

**ine** infeasible paths calculation in each iteration, i.e., nodes that can not be visited.

**mmh** minimum and maximum header counts, i.e., the number of times the header\(^1\) node can be visited.

**mmnl** minimum and maximum nested loop count, i.e., loop count for nested loops.

**mmnc** minimum and maximum node count, i.e., number of times a node is visited.

<table>
<thead>
<tr>
<th>Program</th>
<th>Flow facts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ip</td>
</tr>
<tr>
<td>crc</td>
<td>X</td>
</tr>
<tr>
<td>edn</td>
<td>X</td>
</tr>
<tr>
<td>insort</td>
<td>X</td>
</tr>
<tr>
<td>lcomplex</td>
<td>X</td>
</tr>
<tr>
<td>lcomplex</td>
<td>X</td>
</tr>
<tr>
<td>ns</td>
<td>X</td>
</tr>
<tr>
<td>nsichneu</td>
<td>X</td>
</tr>
<tr>
<td>ssab_mod</td>
<td>X</td>
</tr>
<tr>
<td>task1</td>
<td>X</td>
</tr>
<tr>
<td>task2</td>
<td>X</td>
</tr>
<tr>
<td>task4</td>
<td>X</td>
</tr>
<tr>
<td>task5</td>
<td>X</td>
</tr>
<tr>
<td>task7</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 8.2: Generated flow fact for each benchmark

When we have run the algorithm on these testbenches, we have considered the value space partitions with the slightest difference in either WCET or BCET to be different value space partitions. In practice it is likely to have a certain range or accuracy where value space partitions with similar execution times are grouped to the same value space partition.

The hardware and calculation methods in Table 8.3 are:

**ARM9** Hardware model of the ARM9 processor with calculation with respect to pipeline and cache effects.

**NECV850E** Hardware model of the NEC V850E microcontroller, with respect to pipeline and cache effects.

---

\(^1\)A header node is the first node in a scope, determining if any nodes in that scope will be executed.
Chapter 8. Empirical results

**IPET** Implicit Path Enumeration Technique for WCET calculations [LM95]

**PB** Path Based WCET calculations [HSRW98]

**CB** Cluster Based WCET calculations [Erm03]

<table>
<thead>
<tr>
<th>Program</th>
<th>Hardware</th>
<th>Calculation method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARM9</td>
<td>NEC-V850E</td>
</tr>
<tr>
<td>crc</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>emon</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>lessort</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>jcomplex</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>icdnum</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ns</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>nschned</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>esab_mod</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>task1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>task3</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>task4</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>task5</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>task7</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3: Hardware model and calculation method

For several of the benchmarks we see a trend that we achieve a fairly high accuracy (above 50%) with a quite low effort (low number of runs). However, to achieve high accuracy a considerable effort is required. This is goo in that there is no threshold required to pass in order to get any improvement, and the more effort is put into the analysis the more accurate results. To get a level of accuracy that may be acceptable for many systems the required level of effort is not very high. We denote such a trend, depicted in Figure 8.2, as exponential behaviour.

**8.2.3 crc and insertsort benchmarks**

The crc and insertsort benchmarks give no improvement from the input division. The input domain for crc is very small (only 81 input combinations). Even when each single value input combination is analyzed there is no difference in WCET and BCET. This shows that the benchmark is not input sensitive with respect to execution time. The insertsort input space is quite large, but even after 12000 WCET runs no improvement was achieved. After studying the source code of these benchmarks, it is clear that the effect of different inputs are very small. These benchmarks are therefore not highly representative to the proposed methods.
8.2 Prototype tool

8.2.4 jcomplex and lcdnum benchmarks

For the benchmarks jcomplex and lcdnum we see a clear trend towards the exponential-like behaviour in that a low number of runs give a fairly high accuracy. In Figure 8.3 and Figure 8.5 we show the accuracy of the first 100 runs, where an accuracy of 50% is achieved already after about 30 runs for the Next (N) strategy. After 100 runs an accuracy of more than 70% has been achieved. However, to reach 100% accuracy another order of magnitude is required, i.e., around 1000 runs, as shown in Figure 8.4 and Figure 8.6.

8.2.5 nsichneu benchmark

Similarly to the jcomplex benchmark we see the same exponential trend for the nsichneu benchmark. The Last used (L) strategy outperforms the N strategy (Figure 8.8) it still exhibits the same trend (Figure 8.7). Due to too much inaccuracy introduced by the WCET analysis tool even for single valued space partitions the accuracy does not reach 100% for the nsichneu benchmark.
Figure 8.3: jcomplex first 100 runs

Figure 8.4: jcomplex

Figure 8.5: lednum first 100 runs

Figure 8.6: lednum

Figure 8.7: nsichneu first 50 runs

Figure 8.8: nsichneu
8.2.6 edn and ns benchmarks

![Figure 8.9: edn](image1)
![Figure 8.10: ns](image2)

The edn benchmark shown in Figure 8.9 exhibits a different behaviour than previous benchmarks in that it has a threshold for the L strategy. The ns benchmark shown in Figure 8.10, exhibits the exponential trend. The ns benchmark only has one input variable, thus resulting in the same behaviour independent of using the Next or Last used strategies. We do see a small penalty in using the extreme value heuristics.

8.2.7 task1 benchmark

![Figure 8.11: task1 first 40 runs](image3)
![Figure 8.12: task1](image4)

task1 exhibits different behaviours depending on strategy. For the L strategy it exhibits the exponential trend, but for the N strategy the results show a
threshold-like behaviour that is not desirable. Note that the accuracy is still relatively high for a low number of WCET runs.

### 8.2.8 task3 and task5 benchmarks

(task3 and task5 shown in Figures 8.13 and 8.14 both exhibits the exponential trend for at least one strategy. The different strategies performs very differently. task3 exhibit different behaviours depending on strategy, e.g., LX exhibits an exponential like behaviour while, e.g., the NX and L strategies have a more linear behaviour.

(task5 has a clearer exponential-like behaviour, and the LX strategy shows the best result.

### 8.2.9 task7 benchmark

(task7 has not reached 100% accuracy due to that we have not run until each input value space partition is single valued, and our definition of accuracy has not been held true. Note in Figures 8.15 and 8.16 that the graph stops at 40%. There is a big difference between the NX and LX behaviours, where NX performs well. The results would most likely have been better if assumption 6.2 was held true. In this benchmark, the WCET and BCET are not the same for single value space partitions.)
8.2 Prototype tool

Figure 8.15: task7 first 700 runs

Figure 8.16: task7

Figure 8.17: task4 first 50 runs

Figure 8.18: task4
8.2.10 **test4 benchmark**

task4 has not reached 100% accuracy due to that we have not run until all input value space partitions are single valued, and our definition of accuracy has not been held true. The results however show that a low effort give a large effect, and the behaviour is similar to the other benchmarks. If assumption 6.2 was to be held true, the results would arguably have been better. This is a problem that would need to be approached in future work in order to make the methods more practically useful.

8.2.11 **esab_mod benchmark**

**Comment 1:** More data is coming continually

![Figure 8.19: esab_mod first 50 runs](image1)

![Figure 8.20: esab_mod](image2)

esab_mod has not reached 100% accuracy due to that we have not run until each input value space partition is single valued, and our definition of accuracy has not been held true. The esab_mod benchmark exhibits similar behaviours as most of the other benchmarks. Also for the esab_mod benchmark the hardware effects prove to make the results worse than they potentially could be.

8.2.12 **Discussion**

We notice a big difference in many of the tests between the different approaches. It is important to choose the right method for the right program. We propose that this is automated by monitoring the difference in accuracy between each run and change the strategy if the accuracy is not improving.
L is last used and N is next strategies. LX is last used with extreme value heuristics, and NX is next with extreme value heuristics. We do not draw any final conclusions from the findings in Table 8.4, but the results are slightly slanted towards that the N strategy performs slightly better in; thus, the N strategy may be the one to start with. It is still probably a good idea to change the strategies during the analysis according to some heuristics. This should be investigated in future work.

**Discussion**

To some of the industrial benchmarks we do not have access to the code (all benchmarks called task), and therefore all benchmarks have been treated as black-box components. For these benchmarks we only have information about the inputs. It is also worth mentioning that the benchmarks called “task” are designed according to the Rubus [HMTN+08] component model.

We finally note that our results may not be fully representative for all type of input-dependent embedded system programs. Even though the included code may be quite complex, several of the benchmarks are quite small. Moreover, some of the used benchmarks are not true input-sensitive programs, instead the input-sensitivity seems to have been added to originally single-input value programs.
### 8.2.13 Finding WCETs

<table>
<thead>
<tr>
<th>Program</th>
<th>#Vars</th>
<th>$[D]$</th>
<th>$#WC$</th>
<th>Basic MinT</th>
<th>Basic MaxT</th>
<th>Basic #BT</th>
<th>TotalT</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>6922</td>
<td>7421</td>
<td>0</td>
<td>49934</td>
</tr>
<tr>
<td>edn</td>
<td>3</td>
<td>$2^{15}$</td>
<td>12</td>
<td>5063</td>
<td>5531</td>
<td>0</td>
<td>63512</td>
</tr>
<tr>
<td>inssort</td>
<td>10</td>
<td>$10^{12}$</td>
<td>619</td>
<td>344</td>
<td>765</td>
<td>0</td>
<td>326894</td>
</tr>
<tr>
<td>incomplex</td>
<td>2</td>
<td>512</td>
<td>455</td>
<td>32</td>
<td>828</td>
<td>155</td>
<td>175541</td>
</tr>
<tr>
<td>icdnum</td>
<td>2</td>
<td>1024</td>
<td>7</td>
<td>31</td>
<td>859</td>
<td>0</td>
<td>3436</td>
</tr>
<tr>
<td>ns</td>
<td>1</td>
<td>512</td>
<td>6</td>
<td>1093</td>
<td>38172</td>
<td>0</td>
<td>155205</td>
</tr>
<tr>
<td>nsichneu</td>
<td>6</td>
<td>$2^{17}$</td>
<td>25</td>
<td>17733</td>
<td>95203</td>
<td>0</td>
<td>1716109</td>
</tr>
<tr>
<td>esab_mod</td>
<td>19</td>
<td>$2^{17}$</td>
<td>12</td>
<td>16359</td>
<td>19364</td>
<td>0</td>
<td>188599</td>
</tr>
<tr>
<td>task1</td>
<td>4</td>
<td>64</td>
<td>11</td>
<td>16359</td>
<td>19364</td>
<td>0</td>
<td>188599</td>
</tr>
<tr>
<td>task2</td>
<td>7</td>
<td>18</td>
<td>9</td>
<td>14368</td>
<td>18968</td>
<td>0</td>
<td>173226</td>
</tr>
<tr>
<td>task5</td>
<td>8</td>
<td>$2^{10}$</td>
<td>53</td>
<td>47</td>
<td>157</td>
<td>7</td>
<td>5434</td>
</tr>
<tr>
<td>task7</td>
<td>26</td>
<td>$2^{17}$</td>
<td>167</td>
<td>375</td>
<td>719</td>
<td>2</td>
<td>72165</td>
</tr>
</tbody>
</table>

Table 8.5: Finding WCET analysis results for benchmarks without program slicing

<table>
<thead>
<tr>
<th>Program</th>
<th>#Vars</th>
<th>$[D]$</th>
<th>$#WC$</th>
<th>Extreme MinT</th>
<th>Extreme MaxT</th>
<th>Extreme #BT</th>
<th>TotalT</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>6813</td>
<td>7362</td>
<td>0</td>
<td>71872</td>
</tr>
<tr>
<td>edn</td>
<td>3</td>
<td>$2^{15}$</td>
<td>2</td>
<td>4987</td>
<td>4984</td>
<td>0</td>
<td>33636</td>
</tr>
<tr>
<td>inssort</td>
<td>10</td>
<td>$10^{12}$</td>
<td>619</td>
<td>485</td>
<td>780</td>
<td>0</td>
<td>326894</td>
</tr>
<tr>
<td>incomplex</td>
<td>2</td>
<td>512</td>
<td>455</td>
<td>47</td>
<td>922</td>
<td>141</td>
<td>97078</td>
</tr>
<tr>
<td>icdnum</td>
<td>2</td>
<td>1024</td>
<td>4</td>
<td>63</td>
<td>734</td>
<td>0</td>
<td>2357</td>
</tr>
<tr>
<td>ns</td>
<td>1</td>
<td>512</td>
<td>1</td>
<td>35256</td>
<td>35256</td>
<td>0</td>
<td>35256</td>
</tr>
<tr>
<td>nsichneu</td>
<td>6</td>
<td>$2^{17}$</td>
<td>20</td>
<td>13968</td>
<td>95203</td>
<td>0</td>
<td>130985</td>
</tr>
<tr>
<td>esab_mod</td>
<td>19</td>
<td>$2^{17}$</td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>task1</td>
<td>4</td>
<td>64</td>
<td>8</td>
<td>14203</td>
<td>20234</td>
<td>0</td>
<td>121687</td>
</tr>
<tr>
<td>task2</td>
<td>7</td>
<td>18</td>
<td>10</td>
<td>14125</td>
<td>21890</td>
<td>0</td>
<td>153654</td>
</tr>
<tr>
<td>task5</td>
<td>8</td>
<td>$2^{10}$</td>
<td>43</td>
<td>62</td>
<td>187</td>
<td>8</td>
<td>4514</td>
</tr>
<tr>
<td>task7</td>
<td>26</td>
<td>$2^{17}$</td>
<td>174</td>
<td>328</td>
<td>625</td>
<td>3</td>
<td>72912</td>
</tr>
</tbody>
</table>

Table 8.6: Finding WCET analysis results for benchmarks with extreme value heuristics and without program slicing

Tables 8.5 and 8.6 give the analysis results when using no slicing, i.e., all input variables are assumed to affect the program execution time. #Vars gives number of input variables. $[D]$ gives the size of the input value space. Basic gives the results for the basic input derivation method, while Extreme gives the corresponding ones for our extreme value search method. $\#WC$ gives the number of WCET calculations performed. MinT and MaxT gives the minimum and maximum analysis time (in seconds) used for any of the WCET calculations. $\#BT$ gives the number of back trackings performed in the input-
Table 8.7: Analysis results for benchmarks affected by program slicing

| Program  | #Vars | $|D|_1$ | #WC | MinT | MaxT | #BT | TotT |
|----------|-------|------|------|------|------|------|------|------|
| nsichneu | 5     | $2^{10}$ | 24 | 17453 | 92780 | 0 | 1638990 |
| ssab_mod | 15    | $2^{10}$ | 216 | 67501 | 610125 | 17 | 3956334 |
| task7    | 17    | $2^{10}$ | 167 | 328 | 656 | 2 | 70164 |

Table 8.8: Analysis results for benchmarks affected by with extreme value heuristics and program slicing

| Program  | #Vars | $|D|_1$ | #WC | MinT | MaxT | #BT | TotT |
|----------|-------|------|------|------|------|------|------|
| nsichneu | 5     | $2^{10}$ | 17 | 17087 | 46702 | 0 | 644249 |
| ssab_mod | 15    | $2^{10}$ | 90 | 76687 | 616016 | 10 | 1185105 |
| task7    | 17    | $2^{10}$ | 174 | 328 | 625 | 3 | 72893 |

Figure 8.21: Number of runs and analysis time with respect to input space
value search analysis. \textit{TotT} gives the total analysis time (in seconds) required to derive the WCET input values.

A general conclusion is that we indeed can derive the input combination that forces the execution of the WCET program path. Moreover, as can be expected, the number of WCET calculations are for most programs highly related to the size of the program’s input value space. For most programs the WCET input value combination can be derived without any back-tracking at all. However, for one program, \textit{jcomplex}, over-estimations in the static WCET calculations lead to extensive backtracking, resulting in a fairly large number of WCET runs for relatively small input domains.

We also note that for many programs there is a variability in the time for doing different WCET calculations. In general, when the input value size decreases, the time for performing the WCET calculation also decreases. Thus, the first analysis generally consumes most time, while subsequent analyses are faster. This indicates that the time penalty for performing multiple runs is less than linear.

We also see that the two different heuristics (\textit{Basic} and \textit{Extreme}) are highly dependent on the structure of the analyzed program. For some programs the benefits are very large for using the extreme heuristics, while other programs instead get a smaller penalty.

Tables 8.7 and 8.8 give the analysis results when using slicing. The slicing was made under the assumption that we were using a hardware architecture with always constant time for different memory accesses from the same instruction, and no instructions with variable execution time due to argument values. Thus, the program slicing could therefore be performed on conditionals only. \textit{#Vars} gives number of input variables left after the slicing and \(|D|\) gives the size of the corresponding input value space.

Only the benchmarks \textit{(nsichneu, esab_mod and task7)}, that had their value space reduced by the slicing are included in the Table 8.7. We see that the reduction in analysis time, as well as the size of the input value space, when using slicing, are significant for \textit{nsichneu}. However, for \textit{task7} the effects are more moderate. This was because most input data removed for \textit{task7} were relatively small in value size, or static pointers holding only a single abstract pointer value.

We finally note that our results may not be fully representative for all type of input-dependent embedded system programs. Even though the included code may be quite complex, several of the benchmarks are quite small. Moreover, some of the used benchmarks are not true input-sensitive programs, instead the input-sensitivity seems to have been added to originally single-input
8.3 Allocating components to tasks

8.3.1 Simulation set up

This section describes the simulation method and set up for evaluating the allocation of components to tasks. For each simulation the genetic algorithm allocates components to tasks and evaluates the allocation, and incrementally finds new allocations.

The system data is produced by creating a random schedulable task set, on which all components are randomly allocated, i.e., we create an allocation backwards from which component and transactions properties are derived. The component properties are deduced from the task they are allocated. Transactions are deduced the same way from the task set. In this way there is always at least one solution for each system. However, it is not sure that all systems are solvable with a one-to-one allocation. The components and component transactions are used as input to the framework. Hereafter, systems that are referred to as generated systems are generated to form input to the framework. Systems that come out of the framework are referred to as allocated systems. The simulation parameters are set up as follows:

- The number of components of a system is randomly selected from a number of predefined sets. The number of components in the systems are ranging in twenty steps from 40 to 400, with a median on 120 components.

- The period times for the components are randomly selected from a predefined set of different periods between 10 and 100 ms.

- The Worst Case Execution Time (WCET) is specified as a percentage of the period time and chosen from a predefined set. The WCETs together with the periods in the system constitutes the system load.

- The transaction size is the size of the generated transactions in percentage of the number of components in the system. The transaction size is randomly chosen from a predefined set. The longer the transactions, the more constraints, regarding schedulability, on how components may be allocated.
• The transaction deadline laxity is the percentage of the lowest possible transaction deadline for the generated system. The transaction deadline laxity is evenly distributed among all generated systems and is always greater or equal to one, to guarantee that the generated system is possible to map. The higher the laxity, the less constrained transaction deadlines.

One component can be involved in more than one transaction, resulting in more constraints in terms of timing. The probability that a component is participating in two transactions is set to 50% for all systems.

To get as realistic systems to simulate as possible, the values used to generate systems are gathered from some of our industrial partners. The industrial partners chosen are active within the vehicular embedded system segment. A complete table with all values and distributions, of the system generation values, can be found in Chapter 8. The task switch time used for the system is 22 µs, and the TCB size is 300 bytes. The task switch time and tcb size are representative of commercial RTOS TCB sizes and context switch times for common CPUs.

The simulations are performed for four different utilization levels, 30%, 50%, 70% and 90%. For each level of utilization 1000 different systems are generated with the parameters presented above. The evaluation results are found in Chapter 8.

In Table 8.3.1 we show the specific data used for generating systems to the simulations. The GA was setup with an initial population of 300 individuals, and every simulation was run for 500 generations. The simulations were run on a 1.8 GHz Pentium 4m processor with 768 MB of RAM. The mean time for each simulation is 133 seconds.

### 8.3.2 Results

A series of simulations have been carried out to evaluate the performance of the proposed framework. To evaluate the schedulability of the systems, FPS scheduling analysis is used. The priorities are randomly assigned by the genetic algorithm, and no two tasks have the same priority. We compare the allocation approach described in this paper to one-to-one allocations. Table 8.3.2 summarizes the results from the simulations. The columns entitled "stack" and "CPU" displays the average memory size (stack + tcb) and CPU overhead respectively, for all systems with a specific load and transaction deadline laxity. The column entitled "success" in the 1-1 allocation section displays the rate of systems that are solvable with the 1-1 allocation. The column entitled "success" in the GA
### 8.3 Allocating components to tasks

<table>
<thead>
<tr>
<th>Number components</th>
<th>WCET in % of period</th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1,25</td>
<td>4</td>
</tr>
<tr>
<td>50</td>
<td>6,25</td>
<td>8</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>Stack size</td>
</tr>
<tr>
<td>70</td>
<td>6,25</td>
<td>10</td>
</tr>
<tr>
<td>80</td>
<td>2,75</td>
<td>512</td>
</tr>
<tr>
<td>100</td>
<td>7,5</td>
<td>1024</td>
</tr>
<tr>
<td>120</td>
<td>13</td>
<td>2048</td>
</tr>
<tr>
<td>140</td>
<td>7,5</td>
<td>4096</td>
</tr>
<tr>
<td>150</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>2,5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>transactions size</th>
<th>% of num. comp.</th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>180</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>200</td>
<td>5,25</td>
<td>13</td>
</tr>
<tr>
<td>210</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>240</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>250</td>
<td>1,25</td>
<td>25</td>
</tr>
<tr>
<td>280</td>
<td>5</td>
<td>Laxity % of critical</td>
</tr>
<tr>
<td>300</td>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>320</td>
<td>1</td>
<td>130</td>
</tr>
<tr>
<td>350</td>
<td>1,25</td>
<td>150</td>
</tr>
<tr>
<td>400</td>
<td>0,25</td>
<td>Utilization %</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Isolation %</th>
<th>Dist. %</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>56</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>70</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
<td>90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GA parameters</th>
<th>GA property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>Population</td>
<td>300</td>
</tr>
<tr>
<td>25000</td>
<td>Generations</td>
<td>500</td>
</tr>
<tr>
<td>50000</td>
<td>Elite rate</td>
<td>5%</td>
</tr>
<tr>
<td>100000</td>
<td>Cull rate</td>
<td>40%</td>
</tr>
<tr>
<td></td>
<td>Mutation rate</td>
<td>1%</td>
</tr>
</tbody>
</table>

Table 8.9: Data used for generating systems, and GA parameter
allocation section displays the rate at which our framework finds allocations, since all systems have at least one solution. The stack and CPU values are only collected from systems where a feasible allocation was found.

Table 8.10: Memory, CPU overhead and success ratio for 1-1 and GA allocations

<table>
<thead>
<tr>
<th>Load</th>
<th>Laxity</th>
<th>1-1 allocation</th>
<th>GA allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Stack</td>
<td>CPU</td>
</tr>
<tr>
<td>30%</td>
<td>All</td>
<td>28882</td>
<td>4.1%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>65949</td>
<td>3.5%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>33077</td>
<td>4.4%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>26755</td>
<td>4.1%</td>
</tr>
<tr>
<td>50%</td>
<td>All</td>
<td>37227</td>
<td>4.8%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>35391</td>
<td>4.3%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>38251</td>
<td>4.8%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>37043</td>
<td>4.9%</td>
</tr>
<tr>
<td>70%</td>
<td>All</td>
<td>44455</td>
<td>5.1%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>44226</td>
<td>5.0%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>44267</td>
<td>5.1%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>44619</td>
<td>5.2%</td>
</tr>
<tr>
<td>90%</td>
<td>All</td>
<td>46943</td>
<td>5.6%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>49607</td>
<td>5.5%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>53535</td>
<td>5.7%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>53535</td>
<td>5.7%</td>
</tr>
</tbody>
</table>

The first graph of the simulations (Figure 8.22) shows the success ratio, i.e., the percentage of systems that were possible to map with the one-to-one allocation, and the GA allocation respectively. The success ratio is relative to the effort of the GA, and is expected to increase with a higher number of generations for each system. Something that might seem confusing is that the success ratio is lower for low utilization than for high utilizations, even though, intuitively, it should be the opposite. The explanation to this phenomenon is that the timing constraints become tighter as fewer tasks participate in each transaction (lower utilization often leads to fewer tasks). With fewer tasks the task phasing, due to different periods, will be lower, and the deadline can be set tighter. This has happened because the systems are automatically generated.

The second graph (Figure 8.23) shows that the deadlines are relaxed with higher utilization, since the allocations with relaxed deadlines perform well, and the systems with a more constrained deadline show a clear improvement with higher utilization.

The third graph (Figure 8.24) shows for both approaches the average stack size for the systems at different utilization. The comparison is only amongst allocations that have been successfully mapped by both strategies. The memory size consists of the tcb and the stack size, and the tcb size is 300.
8.3 Allocating components to tasks

bytes. As described earlier, each task allocates a stack that is equal to the size of the largest stack among its allocated components.

The fourth graph (Figure 8.25) shows the average task switch time in microseconds for the entire system. The task switch overhead is only dependent on how many tasks there are in the system. The average improvement of GA allocation in comparison to the 1-1 allocation is, for the success ratio, 10%. The memory size is reduced by 32%, and the task switch overhead is reduced by 48%. Hence we can see a substantial improvement in using more sophisticated methods to allocate components to tasks. A better strategy for setting priorities would probably lead to an improvement in the success ratio. Further we observe that lower utilization admits larger improvements than higher laxity of the deadlines; and since lower utilization in the simulations often gives tighter deadlines, we can conclude that the allocation does not negatively impact schedulability. However, regarding the improvements, the more components the more constraints are put on each transaction, and thereby on the components, making it harder to perform good allocations.

Results from simulations show that the allocation framework gives substantial improvements both in terms of memory consumption and task switch overhead. The described framework also has a high ratio in finding feasible allocations. Moreover, in comparison to allocations performed with a one-to-one allocation our framework performs very well, with 32% reduced memory size and 48% reduced task switch overhead. The simulations show that the proposed framework performs allocations on systems of a size that covers many embedded systems, and in a reasonable time for an off-line tool. We have also
shown how CPU load and deadline laxity affects the allocation.
Chapter 9

Summary and Conclusions

In this chapter we summarize and conclude the thesis and the contributions presented in this thesis, and we discuss possible future research directions.

9.1 Summary

This thesis is a link between Component-Based Software Engineering (CBSE) and Embedded Real-Time Systems (ERTS), indicating some of the difficulties in, and, providing several solutions for, developing predictable and resource constrained systems by reusing pre-fabricated software components.

In chapter 1 we give an introduction to our research, starting with an illustrative example from the real world before we give an overview of the specific research and contributions.

Chapter 2 gives an introduction to embedded and real-time systems and its terminology and definitions. We describe the state-of-research in terms of real-time scheduling and WCET analysis.

In chapter 3 we give an introduction to component-based software engineering terminology and definitions. We also discuss the industrial motivations for using component-based software engineering.

In chapter 4 we describe the industrial and research problems and discuss the research methodology used in order to partially solve the research problem.

Chapter 5 positions our methods (that are described in chapters 6 and 7) in the component-based development process, and proposes an extension of a CBSE development process to support the proposed methods.
156 Chapter 9. Summary and Conclusions

In chapter 6 we outline two novel methods, based on a combination of static WCET analysis and systematic search over the value space of input variables, for deriving WCET behaviour of the program.

Chapter 7 presents a method for optimizing the resource usage in component-based real-time systems based on real-time analysis, calculating resource consumption and genetic algorithms for deriving mappings between components and tasks that are optimized for low resource consumption, while maintaining stipulated real-time requirements. Throughout Chapters 5, 6 and 7 we use an example of an Adaptive Cruise Controller application to illustrate our techniques.

Finally in chapter 8 we present the evaluations of the technical contributions presented in Chapters 6 and 7.

9.1.1 Contributions

In this thesis we have introduced several novel methods for improving utilization and prediction accuracy for reusable software components. We have extended the component-based software engineering (CBSE) development process with our methods, and we have also developed tools and evaluated the methods with both industrial code and academic benchmarks.

The specific in-depth technical contributions of the thesis are (i) two methods for increasing accuracy and resource efficiency of WCET for embedded real-time components, and (ii) a method for allocating components to tasks for minimizing stack-usage and CPU-overhead, while maintaining real-time constraints.

<table>
<thead>
<tr>
<th>Contribution</th>
<th>Described in Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>C2</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>C3</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>C4</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>C5</td>
<td>Chapter 8</td>
</tr>
</tbody>
</table>

Table 9.1: The relation between contribution and chapters in the thesis.

The main contributions of the presented research are summarized as follows:

C1 A reusable WCET. The input space of reusable component is partitioned with respect to execution time, creating parametrizable component WCET contracts. A WCET contract is parametrizable and produces a WCET
that is more accurate with respect to the specific usage. The result is that the WCET analysis can be reused with reusable components. The reusable WCET is evaluated with components from our industrial partners.

**C2 Derivation of WCET input values.** The input space of a component is divided with respect to component WCET, searching for an input combination that results in the execution of the worst-case path. The result can be used for guiding measurement-based WCET analysis. The method is evaluated with components from our industrial partners.

**C3 A framework for allocating components to tasks** aiming at minimizing resource consumption while maintaining real-time constraints. The framework calculates feasibility and fitness of an allocation. By exploring the state space of possible allocations, and comparing them to each other meta heuristic methods like genetic algorithms can be used. The framework is implemented with genetic algorithms, and evaluated with systems from our industrial partner.

**C4 A resource aware development process** that is an extension of the CBSE development process with the methods outlined in this thesis.

**C5 A prototype tool** implementing the ideas from contributions C1 and C2. The prototype tool has a graphical user interface, graphically presenting WCET and BCET connected to inputs and component contracts. The tool supports several different heuristics for creating WCET contracts.

Table 9.1.1 presents how the contributions C1-C5 relate to the chapters in the thesis in Section 1.3

### 9.2 Discussion

In Chapter 4 several industrial and academic problems were formulated, and solution proposed. We revisit the questions identified from Shaw’s classification of software engineering research. These questions are:

**Characterization:** what are the important characteristics for increasing resource efficiency and predictability for reusable software components in embedded real-time systems?
We have found that important characteristics for increasing resource utilization and predictability for reusable software components are:

- WCET analysis is both time consuming and difficult. Having tools that automatically derive a reusable WCET increases the development efficiency by making WCET analysis simpler and more accessible. Thus the WCET analysis should be automated as much as possible, requiring a minimum of human interaction.

- The CBSE development process is different from traditional development in the sense that it is divided into component development and system development. Thus the WCET analysis should be divided in two different parts, a component part, developed to be reusable by the component developer, and a system part, to be used by the system designer. This facilitates the adoption of the technique in the CBSE development process. The WCET analysis should be performed in the component development part of the CBSE development process. The WCET should be known in the system development part of the CBSE development process.

- WCET analysis can of course always be reused but to make sure that the WCET is always a safe over estimation, all possible usages must be considered. This will potentially lead to very inaccurate predictions. Thus, the reusable WCET analysis results should reach a pre-defined accuracy, and it should be possible to reach the same accuracy as with current state of the art WCET analyses.

- A component based system should be transformed to a real-time system conforming to a specific real-time model. It must be possibleable to separate an efficient allocation from an inefficient. A common approach for allocating components to real-time tasks is to view one component as one task. However, component should be allocated to real-time tasks such that the resource efficiency is higher than for a system where one component is allocated to one real-time task.

- There exists many possible allocations between components to real-time tasks. At the same time as we want the system as resource efficient as possible, a transformation from components to real-time tasks may not violate temporal requirements. Thus, components should be transformed to tasks in such a way that the temporal correctness of the system is maintained.
The assumptions restrict the usefulness of the approach in practice, and in future work the assumptions need to be relaxed. It is worth noting though that much research is based on assumptions that are in practical in general, especially much real-time scheduling work. However, as the research progresses, more assumptions may be relaxed.

**Methods/means:** how can we accomplish increased resource efficiency and predictability for reusable software components in embedded real-time systems?

- The reusable WCET analysis method has shown to provide accurate WCET estimates that can be used under the assumptions given in this thesis. The method come with a penalty of increased effort in terms of calculation time for achieving the reusable WCET.

  The methods are evaluated in Chapter 8, and the results show that the proposed methods can express relationships between inputs and execution times with equally high precision compared to traditional WCET analysis. The methods have been evaluated with both industrial and academic software.

  Due to less over approximations when analysing smaller parts of the behaviour by limiting the inputs we have also observed that the WCET become more accurate for the whole system.

  Moreover, as can be expected, the number of WCET calculations are for most programs highly related to the size of the program’s input value space. The method scales well up to around an accuracy\(^1\) \( \sim 60\% \), after that the method requires more effort. The overall complexity is still never higher than \( O(n) \), where \( n \) is the input value size.

- The second novel method derives an input combination that triggers the execution of the WCET path, that produces the WCET. In this method we use techniques similar to the first outlined method in the sense that we use static WCET analysis and structured search over the component input domain. But rather than searching the entire input domain, we iteratively remove parts of the domain that does not generate execution of the WCET program path.

  The evaluations show that it is possible to derive the input combination that generates the execution of the WCET program path. ...

---

\(^1\) Accuracy is defined in Chapter 6
Moreover, as can be expected, the number of WCET calculations are for most programs highly related to the size of the program’s input value space.

- The third method allocates components to real-time tasks in such a way that resource utilization is maximized. The evaluations show that allocation of components to tasks potentially give high benefits in terms of increased resource efficiency. We have also shown that tighter WCET estimations produces a higher number of feasible allocation, and hence a greater chance to find a better allocation compared to one-to-one allocations.

Results from simulations show that the allocation framework gives substantial improvements both in terms of memory consumption and task switch overhead. The described framework also has a high ratio in finding feasible allocations. Moreover, in comparison to allocations performed with a one-to-one allocation our framework performs very well, with 32% reduced memory size and 48% reduced task switch overhead. The simulations show that the proposed framework performs allocations on systems of a size that covers many embedded systems, and in a reasonable time for an off-line tool. We have also shown how CPU load and deadline laxity affects the allocation.

have we then solved the outlined problems by providing the outlined methods? No, we certainly have not solved the problems; however, we take one step in the right direction, and underline the difficulty in reusing software components in ERTS. We believe that the research show that it is possible to acheive efficient reuse of software components and at the same time have efficient use of resources and high accuracy of real-time predictions.

To make the research more general some of the assumptions given in this work should be relaxed; especially those related to varying hardware.

### 9.3 Future research directions

From the assumptions and conclusions from our presented research, we have identified possible future research directions. In this section we present a selection.
9.3 Future research directions

9.3.1 Varying hardware

In future work it may be possible to also parametrize the contracts with respect to hardware to facilitate reuse over hardware boundaries. This gives another dimension to the problem and increases the search domain. Different aspects of hardware can be considered. For example different alignment in memory, different hardware architectures and much more. It is not obvious how this information should be treated; one possibility is to view the hardware parameters as a larger input domain. However, a more sophisticated approach is probably preferable in order to limit the search effort.

9.3.2 Architectures

Many embedded systems are built using the pipe-and-filter or blackboard architectures, supported in this thesis. However, the proposed methods would be more general if other architectures were supported. Some architectural styles are more difficult to analyze; take as example a client server architecture, where components may have arbitrarily complex interaction patterns.

9.3.3 Distribute the search algorithms

The reusable WCET analysis is inherent suitable for parallelizing. This is future work that has been discussed, and would dramatically improve the performance of the methods.

The analysis times for some of the test-benches are several hours. The times are still, in the opinion of the author, reasonable given the complexity of the problem. The approach however, is easily parallelized using SMP or clusters of computers. The technique can be used for, e.g., speculative WCET analysis of value space partitions.

9.3.4 Dynamic search strategies

The methods for reusable WCET and deriving the WCET input combination, a binary search where the proposed search strategies are evaluated during the search. A simple heuristics is to cycle the different variable selection strategies depending on the improvement in accuracy between two consecutive WCET runs.
9.3.5 Worst best-case

We see the use for best-case prediction with best-case behaviour but worst-case hardware effects for a program and a given input set. This would make our predictions better.

9.3.6 Advanced real-time requriements

By adding jitter constraints, blocking or other more advanced contraints, several new trade-offs arise. Furthermore, a different scheduling policy and priority assignments could be applied. Due to the nature of GA it is easy to add new optimizations as the ones suggested above.

9.3.7 Incremental WCET analysis

In all software development processes we compile components or modules, but we don’t compile all modules again and again, but we compile only the modified ones and link them with the others. This should also be true for WCET analysis, however, today there are no efficient way of reusing WCET analysis results. If we analyze a piece of code, we should be able to store this information in a way that can be used later on and make the analysis easier. So that it is not required to do the whole complex analysis again and again, but rather do the complex analysis tasks only once, and reuse our results cheaply, so we don’t have to go through all the analysis and all the components again and again.

A big problem today is that when large systems that are analysed with WCET analysers are exposed to minor changes, the whole system is still re-analysed. For large and complex system like, e.g., avionics, the WCET analyses may take several days. Therefore there is a strong desire to be able to do partial WCET analysis.

For example, in many WCET analysis tools the low-level analysis result, which gives upper bounds on the execution time for individual instructions and basic blocks, are likely to not be affected by the different input value partitions as the analysis is run. Similarly, by performing a forward slice upon the input variables that had their input changed, it should be possible to determine what code parts the may have had their flow analysis results affected by the change. Only those code parts may need to get their flow information recalculated. Thus, it should be possible to reuse flow- and low-level analysis results from
already performed WCET calculations, thereby speeding up the WCET input values determination analysis.

Figure 9.1: Three different annotation scenarios w.r.t basic blocks.

With the described method we propose that more information can be stored in the parametrizable contract. For instance, a promising approach is to store a list of all basic blocks that have been affected by the annotated analysis. If the WCET analysis is annotated in such a way that some parts of the source code is not affected, the basic blocks representing that code would not be in that list. Most state of the practice WCET tools can provide this information, and can also give a mapping between basic blocks and source code. This implies that if a component is changed, only those clusters that “contain” the affected basic blocks needs to be re-analyzed as depicted in figure 9.1. Effectively this gives us the possibility of incremental WCET analysis.

When re-compiling parts of the code it needs to be re-linked. This may in turn result in that the different pieces (basic blocks) may be aligned different in the memory leading to, e.g., different cache behaviours. We propose that this can easily be solved with using a static link-map, where the linker is forced to align the different basic-blocks in a specified way. In many safety critical systems it is desired to have control over the linker anyway.

It can also be used iteratively in case of design changes; however with some obvious drawbacks on the results. The framework can easily be extended to support other optimizations, besides task switch overhead and memory con-
sumption.
Appendix A

Complete list of tables and graphs

For each benchmark we show the number of analyses (#Runs) required to reach a specific accuracy, and how many different value space partitions that resulted in (#D).

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next Used</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Runs</td>
<td>#D</td>
</tr>
<tr>
<td>crc</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table A.1: crc benchmark
### Table A.2: insertsort benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>% Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#D</td>
<td>#D</td>
</tr>
<tr>
<td>insertsort</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### Table A.3: jcomplex benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>% Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#D</td>
<td>#D</td>
</tr>
<tr>
<td>jcomplex</td>
<td>10%</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>40</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>48</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>62</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>80</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>112</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>160</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>264</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>622</td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>968</td>
<td>160</td>
</tr>
</tbody>
</table>

### Table A.4: ns benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>% Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#D</td>
<td>#D</td>
</tr>
<tr>
<td>ns</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>16</td>
<td>19</td>
</tr>
</tbody>
</table>

Table A.2: insertsort benchmark

Table A.3: jcomplex benchmark

Table A.4: ns benchmark
### Table A.5: edn benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs #D</td>
<td>Extreme #Runs #D</td>
</tr>
<tr>
<td>edn</td>
<td>10%</td>
<td>18 3</td>
<td>5 3</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>29 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>24 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>26 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>26 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>35 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>34 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>34 2</td>
<td>9 2</td>
</tr>
</tbody>
</table>

### Table A.6: nsichneu benchmark - 138 runs

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs #D</td>
<td>Extreme #Runs #D</td>
</tr>
<tr>
<td>nsichneu</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>26 3</td>
<td>9 3</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>48 4</td>
<td>7 4</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>44 11</td>
<td>20 14</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>46 13</td>
<td>29 20</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>34 15</td>
<td>35 24</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>90 23</td>
<td>38 26</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>122 25</td>
<td>189 129</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>192 304</td>
<td>50 328</td>
</tr>
</tbody>
</table>

### Table A.7: task1 benchmark - 72 runs

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs #D</td>
<td>Extreme #Runs #D</td>
</tr>
<tr>
<td>task1</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>29 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>32 3</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>32 4</td>
<td>7 4</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>32 4</td>
<td>9 4</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>48 13</td>
<td>14 13</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>72 5</td>
<td>23 5</td>
</tr>
</tbody>
</table>

Table A.5: edn benchmark

Table A.6: nsichneu benchmark - 138 runs

Table A.7: task1 benchmark - 72 runs
### Table A.8: task3 benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Runs</td>
<td>#Runs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

### Table A.9: task4 benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Runs</td>
<td>#Runs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

### Table A.10: task5 benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Runs</td>
<td>#Runs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>10%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td>Prog</td>
<td>%Acc</td>
<td>Strategy Last Used</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>--------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
</tr>
<tr>
<td>lcdnum</td>
<td></td>
<td>#D</td>
<td>#D</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>n/a</td>
<td>2</td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td>n/a</td>
<td>2</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>n/a</td>
<td>2</td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td>n/a</td>
<td>3</td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td>14</td>
<td>38</td>
</tr>
<tr>
<td>70%</td>
<td></td>
<td>209</td>
<td>13</td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>90%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table A.11: LCDNUM benchmark - ## runs

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th></th>
<th>Strategy Next</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
</tr>
<tr>
<td>task7</td>
<td></td>
<td>#D</td>
<td>#D</td>
<td>#D</td>
<td>#D</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>n/a</td>
<td>3</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>n/a</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td>n/a</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>n/a</td>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td>n/a</td>
<td>3</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td>14</td>
<td>38</td>
<td>58</td>
<td>109</td>
</tr>
<tr>
<td>70%</td>
<td></td>
<td>209</td>
<td>13</td>
<td>694</td>
<td>149</td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>90%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table A.12: TASK7 benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th></th>
<th>Strategy Next</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
<td>Basic #Runs</td>
<td>Extreme #Runs</td>
</tr>
<tr>
<td>esab_mod</td>
<td></td>
<td>#D</td>
<td>#D</td>
<td>#D</td>
<td>#D</td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>70%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>90%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table A.13: ESAB_MOD benchmark
Appendix B

Complete list of publications

In this appendix we present the complete list of publications. The publications are divided into three categories (i) papers that are founding for the thesis contributions, (ii) papers that are related to the thesis, but not not directly (however, maybe indirectly) to the thesis contribution, and, (iii) papers that are not related to the thesis. The papers that are not related to the thesis have been written due to cooperation with researchers, and often the contributions of these papers are somewhat outside my main research track.

With the papers that are founding for the thesis contribution, I also state my contribution, and how the paper is used in this thesis. Also refer to Table B for an overview of how the papers relate to the chapters in this thesis.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Directly contributing publications</th>
<th>Related publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 3</td>
<td>30,12,25,26</td>
<td></td>
</tr>
<tr>
<td>Chapter 4</td>
<td>9,11,13,18,22</td>
<td></td>
</tr>
<tr>
<td>Chapter 5</td>
<td>2,4</td>
<td>8,12</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>1,2,3,5</td>
<td>10,19</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>2,6,7</td>
<td>12,20</td>
</tr>
<tr>
<td>Chapter 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.1: The relation between publications and the Chapters of the thesis.
Publications related to the thesis contributions

1: Deriving the Worst-Case Execution Time Input Values, Andreas Ermedahl, Johan Fredriksson, Peter Altenbernd, submitted for publication.

Abstract: A Worst-Case Execution Time (WCET) analysis derives upper bounds for the execution times of programs. Such bounds are crucial when designing and verifying real-time systems. A major problem with today’s analyses is that there is no feedback on what input values that actually cause the WCET. However, this is an important information for the system’s designer for various reasons.

In this article we present several novel approaches to overcome this problem. In particular, we show how to use program slicing to derive the input variables whose different values may cause the program execution time to vary. Furthermore, we present a method, based on a combination of input-sensitive static WCET analysis and systematic search over the value space of the input variables, to derive the input value combination that causes the WCET. Since the basic method may be time-consuming when the input-value space is large, we also present different approaches for faster termination.

Usage in thesis: This paper is founding for the contribution for the WCET input combination derivation.

My contribution: The work has been equally distributed among the authors of the paper. Johan Fredriksson has also been responsible for the evaluation of the methods.


Abstract: We present a contract-based technique to achieve reuse of known worst-case execution times (WCET) in conjunction with reuse of software components. For resource constrained systems, or systems where high degree of predictability is needed, classical techniques for WCET-estimation will result in unacceptable overestimation of the execution-time of reusable software components with rich behavior. Our technique allows different WCETs to be associated with subsets of the component behavior. The appropriate WCET for any usage context of the compo-
nent is selected by means of component contracts over the input domain.
In a case-study we illustrate our technique and demonstrate its potential
in achieving tight WCET estimates for reusable components with rich behavior.

Usage in thesis: This paper is founding for several of the contributions
in this thesis. This paper describes the reusable input sensitive WCET
analysis and its relation to the component to task allocation framework.

My contribution: Johan Fredriksson has been the main author and con-
tributor of the paper and the work has been performed in close coop-
eration with Thomas Nolte and Mikael Nolin. The paper was initiated by
Johan Fredriksson.

3: Clustering Worst-Case Execution Times for Software Components,
Johan Fredriksson, Thomas Nolte, Andreas Ermedahl and Mikael Nolin,
in proceedings of the 7th International Workshop on Worst-Case Exe-
cution Time Analysis (WCET’07), in conjunction with the 19th IEEE
Euromicro Conference on Real-Time Systems (ECRTS’07), Pisa, Italy,

Abstract: For component-based systems, classical techniques for Worst-
Case Execution Time (WCET) estimation produce unacceptable overes-
timations of a components WCET. This is because software components
more general behavior, required in order to facilitate reuse. Existing
tools and methods in the context of Component-Based Software Engi-
neering (CBSE) do not yet adequately consider reusable analyses.

We present a method that allows different WCETs to be associated with
subsets of a components behavior by clustering WCETs with respect
to behavior. The method is intended to be used for enabling reusable
WCET analysis for reusable software components. We illustrate our
technique and demonstrate its potential in achieving tight WCET-estimates
for components with rich behavior.

Usage in thesis: This paper is partly founding for the reusable input
sensitive WCET analysis contribution.

My contribution: Johan Fredriksson has been the main author and con-
tributor of the paper and the work has been performed in close coop-
eration with Thomas Nolte, Andreas Ermedahl and Mikael Nolin. The
paper was initiated by Johan Fredriksson.

**Abstract:** Component-Based Software Engineering (CBSE) promises an improved ability to reuse software which would potentially decrease the development time while also improving the quality of the system, since the components are (re-)used by many. However, CBSE has not been as successful in the embedded systems domain as in the desktop domain, partly because requirements on embedded systems are stricter (e.g. requirements on safety, real-time and minimizing hardware resources). Moreover these requirements differ between industrial domains. Paradoxically, components should be context-unaware to be reusable at the same time as they should be context sensitive in order to be predictable and resource efficient. This seems to be a fundamental problem to overcome before the CBSE paradigm will be successful also in the embedded systems domain. Another problem is that some of the stricter requirements for embedded systems require certain analyses to be made, which may be very complicated and time-consuming for the system developer.

This paper describes how one particular kind of analysis, of worst-case execution time, would fit into the CBSE development processes so that the component developer performs some analyses and presents the results in a form that is easily used for component and system verification during system development. This process model is not restricted to worst-case execution time analysis, but we believe other types of analyses could be performed in a similar way.

**Usage in thesis:** This paper describes and is founding for Chapter 5 and the positioning of the proposed methods in the CBSE development process.

**My contribution:** Johan Fredriksson has been the main author and contributor of the paper and the work has been performed in close cooperation with Thomas Nolte and Mikael Nolin. The paper was initiated by Johan Fredriksson.

5: **Predicting Execution-Time for Variable Behaviour Embedded Real-Time Components**, Johan Fredriksson, Thomas Nolte, Mikael Nolin and Heinz Schmidt in proceedings of IEEE Workshop on Model and Analysis Methods for Automotive Systems (WMAAS’06) in con-
Abstract: Embedded systems for vehicle control critically depend on efficient and reliable control software, together with practical methods for their production. Component-based software engineering for embedded systems is currently gaining ground since variability, reusability, and maintainability are supported. However, existing tools and methods do not guarantee efficient resource usage in these systems.

In this paper we present a method, which increases the accuracy of execution time predictions for embedded components without lowering reusability of the components. For assessing the correct timing behaviour, the method classifies run types by their time in addition to their probability.

Usage in thesis: This paper is partly founding for the reusable input sensitive WCET analysis contribution.

My contribution: Johan Fredriksson has been the main author and contributor of the paper and the work has been performed in close cooperation with Thomas Nolte and Mikael Nolin and Heinz Schmidt. The paper was initiated by Johan Fredriksson.

Johan Fredriksson, Kristian Sandstrom, Mikael Akerholm in proceedings of 8th in proceeding of the 8th LNCS International Symposium on Component-based Software Engineering (CBSE8), St.Louis, MO, USA, May, 2005.

Abstract: The embedded systems domain represents a class of systems that have high requirements on cost efficiency as well as run-time properties such as timeliness and dependability. The research on component-based systems has produced component technologies for guaranteeing real-time properties. However, the issue of saving resources by allocating several components to real-time tasks has gained little focus. Trade-offs when allocating components to tasks are, e.g., CPU-overhead, footprint and integrity. In this paper we present a general approach for allocating components to real-time tasks, while utilizing existing real-time analysis to ensure a feasible allocation. We demonstrate that CPU-overhead and memory consumption can be reduced by as much as 48% and 32% respectively for industrially representative systems.
Usage in thesis: This paper is partly founding for the component to task allocation framework.

My contribution: Johan Fredriksson has been the main author and contributor of the paper and the work has been performed in close co-operation with Kristian Sandström.


Abstract: Industry is constantly looking for new developments in software for use in increasingly complex computer applications. Today, the development of component-based systems is an attractive area for both Industry and Academia. The systems we focus on in this thesis are embedded computers, in particular those in automotive systems. A modern car incorporates several embedded computers that control different functions of the car, e.g., anti-spin and anti-lock breaks.

The main purpose of this thesis is to investigate how component technologies for use in embedded systems can reduce resource usage without compromising non-functional requirements, such as timeliness.

The component-technologies available have not yet been used extensively in the vehicular domain. To understand why this is the case we have conducted a survey and performed evaluations of the requirements of the vehicular industry with respect to software and software development. The purpose of the evaluation was to provide a foundation for defining models, methods and tools for component-based software engineering.

The main contribution of this work is the implementation and evaluation of a framework for resource-efficient mappings between component models and real-time systems. Few component technologies today consider the mapping between components and run-time tasks. We show how effective mappings can reduce memory usage and CPU-overhead. The implemented framework utilizes genetic algorithms to find feasible, resource efficient mappings.

We show how component models designed for resource constrained safety-critical embedded real-time systems can use powerful compile-time techniques to realize the component-based approach and ensure predictable behaviour.

Further, we propose a resource reclaiming strategy for component-based real-time systems, to decrease the impact of pessimistic execution time
predictions. In our approach, components run in different quality levels as unused processor time is accumulated.

Usage in thesis: This Licentiate Thesis\(^1\) is founding for the component to task allocation framework.

My contribution: Johan Fredriksson has been the main author and contributor of this Licentiate thesis.

Publications related to the thesis


11: Evaluation of Component Technologies with Respect to Industrial Requirements, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin, in proceeding of the 30\(^{th}\) Euromicro Conference, Component-Based Software Engineering Track, Rennes, France, August, 2004


13: An Industrial Evaluation of Component Technologies for Embedded-Systems, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael

\(^1\)The Swedish Licentiate Thesis is a “half time thesis” in the PhD studies
Nolin, MRT report ISSN 1404-3041 ISRN MDH-MRTC-155/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2004


22: Context Aware Optimizations for Embedded Real-Time Components, Johan Fredriksson, PhD Proposal


25: Component Based Software Engineering for Embedded Systems - A literature survey, Mikael Nolin, Johan Fredriksson, Jerker Hammarberg (external), Joel Huselius, John Håkansson (Department of Information Technology, Uppsala University), Annika Karlsson (external), Ola Larse (external), (external), Goran Mustapic, Anders Möller, Thomas Nolte, Jonas Norberg (external), Dag Nyström, Aleksandra Tesanovic (external), Mikael Åkerholm, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-102/2003-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June, 2003

26: Component-Based Development of Safety-Critical Vehicular Systems, Ivica Crnkovic, DeJiu Chen (KTH), Johan Fredriksson, Hans Hansson, Jörgen Hansson (external), Joel Huselius, Ola Larse (external), Joakim Fröberg, Mikael Nolin, Thomas Nolte, Christer Norström, Kristian Sandström, Aleksandra Tesanovic (external), Martin Törngren (external), Simin Nadjm-Tehrani (external), Mikael Åkerholm, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-190/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, September, 2005

27: Worst-Case Execution Time Clustering for Software Components, Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, Mikael Nolin, Technical Report, Department of Computer Science and Electronics, Mälardalen University, April, 2007


**Publications not related to the thesis**


35: **Handling Subsystems using the SaveComp Component Technology**, Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, Mikael Nolin, Thomas Nolte, John Håkansson, Paul Pettersson, Workshop on Models and Analysis for Automotive Systems (WMAAS’06) in conjunction with the 27th IEEE Real-Time Systems Symposium (RTSS’06), Rio de Janeiro, Brazil, Editor(s):Marco Di Natale and Luis Almeida, December, 2006

36: **On the Teaching of Distributed Software Development**, Ivica Crnkovic, Igor Cavrak (external), Johan Fredriksson, Rikard Land, Mario Zagar
(external), Mikael Åkerholm, 25th International Conference INFORMATI ON TECHNOLOGY INTERFACES, IEEE, Dubrovnik, Croatia, June, 2003

37: Interference Control for Integration of Vehicular Software Components, Mikael Åkerholm, Kristian Sandström, Johan Fredriksson, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-162/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May, 2004

38: Application of Built-In-Testing in Component-Based Embedded Systems, Mikael Åkerholm, Irena Pavlova (Faculty of Math. and Informatics, Sofia University), Johan Fredriksson, Technical Report, MRTC, May, 2006

Bibliography


ait execution time analyzer.
http://www.absint.com/ait/.


Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord,


[LGA+07] Jingyue Li, Anita Gupta, Jon Arvid, Borretzen Borretzen, and Reidar Conradi. The empirical studies on quality benefits


[SFÅ04] Kristian Sandström, Johan Fredriksson, and Mikael Åkerholm. Introducing a component technology for safety critical


