Mälardalen University Doctoral Thesis
No.65

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

Johan Fredriksson

October 2008

MÄLARDALEN UNIVERSITY

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

ISSN 1651-4238
ISBN 978-91-86135-00-3
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

With increase of software complexity and demands for improved development efficiency, there is a need for new technologies and methods that can cope with these challenges. In certain business domains, such as distributed web-based systems and office applications, Component-Based Software Engineering (CBSE) has demonstrated advantages in achieving reusability of software components, shorter time to market and increased quality. Due to these advantages the approach is attractive also for other application domains, in particular for Embedded Real-Time Systems (ERTS). However, applying CBSE to ERTS is not trivial since ERTS have requirements related to timing and resources usage. One of the major challenges in providing CBSE for ERTS is to achieve performance efficiency and predictability while maintaining reusability. In this thesis we address this challenge, and three novel solutions are presented for improving predictability and utilization of resources in component-based ERTS.

The first solution 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 where a high degree of predictability is needed, classical techniques for WCET-estimation may result in unacceptable overestimations for reusable software components. Our solution 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 solution is a method for deriving the input combinations of a software component that produces the WCET. The information resulting from this method can be used, e.g., for guiding measurement-based WCET analysis.

The third solution is a framework for transformation of components to the real-time tasks aiming at providing efficient utilisation of resources. Efficient allocations can reduce memory usage and CPU-overhead considerably.

In addition this thesis demonstrates how the solutions can be incorporated in the CBSE development process. Furthermore, two tools have been implemented and used for the evaluation of the research results; the evaluations show that by using the methods outlined in the thesis resource efficiency and predictability can be substantially increased without negative impact on reuse.

Det är vackrast när det skymmer.
All den kärlek himlen rymmer
ligger samlad i ett dunkelt ljus
över jorden,
över markens hus.

Allt är ömhet, allt är smekt av händer.
Herren själv utplånar fjärran stränder.
Allt är nära, allt är långt ifrån.
Allt är givet
människan som lån.

Allt är mitt, och allt skall tagas från mig,
inom kort skall allting tagas från mig.
Träden, molnen, marken där jag går.
Jag skall vandra -
ensam, utan spår.

Per Lagerkvist - ¨Det är vackrast när det skymmer¨

-För att livet är för kort att slösas bort.

To Marie, Erika, Inger and Tommy

Without whom life would be a waste!

Preface and Acknowledgments

When I began my undergraduate studies at Mälardalen University I intended to study for three years and end with a bachelor's degree. However, after three years the studies had become so interesting that I decided to continued for a further one and a half years to get a master's degree. 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 who knows where I will end up in the future.

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", and the journey leading to this thesis has indeed been a long but rewarding journey in my life the past five years. I have had a lot of fun and met people from around the world whom I would never have met otherwise. I would not have made it this far, and especially wouldn't have had as much fun without all these people. It is impossible to quantify the Ph.D. studies in any proper way, instead I try to summarize different categories that have played different roles throughout these last five years.

Supervisors Without the encouragement, and guidance of my supervisors I would have been lost a long time ago.

Prof. Ivica Crnkovic is my main supervisor. Thank you for always believing in me, and for giving me this opportunity to be a Ph.D. student. I am impressed by your ability to get both the details and the big picture of my research. We have had a lot of fun together, especially after a few "pivo" in Croatia, 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,

and I am looking forward to working with you in the future. I am amazed by your ability to find the details in my research that no one else finds. I have you to thank for 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 privately. Your friendship and guidance have made me not only a better researcher, but also a better person and better photographer. I am eternally grateful for all your support. Tack!

Prof. Mikael Nolin is my assistant supervisor. Your guidance, both on a personal and professional level has helped me many times. You always believe in me, and back 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. Even though you are not officially one of my supervisor we have worked very much together, and you deserve thanks in this section as well.

Thesis reviewers Apart from my supervisors, the people who helped me in making this thesis better deserves an extra thank you. Thank you very much, Stefan Bygde, Marie Costallas, Dr. Radu Dobrin, Dr. Jan Gustafsson, Prof. Hans Hansson, Dr. Rikard Land, Prof. Björn Lisper, Dr. Frank Lüders, Dr. Dag Nyström, Prof. Paul Pettersson, Prof. Heinz Schmidt.

Co-authors I have authored and co-authored 34 different papers. I would never have made that without very competent and hard working co-authors. Tank you, Prof. Peter Alternbernd, Dr. Jan Carlson, Prof. Ivica Crnkovic, Dr. Radu Dobrin, Dr. Andreas Ermedahl, Dr. Joakim Fröberg, Prof. Hans Hansson, Dr. Rikard Land, Anders Möller, Dr. Thomas Nolte, Prof. Mikael Nolin, Irena Pavlova, Dr. Ian Peake, Dr. Kristian Sandström, Prof. Heinz Schmidt, Dr. Massimo Tivoli and Dr. Mikael Åkerholm.

Coffees Drinking coffee at the department has been a pleasant activity, and especially the breaks they have involved. Many ideas, both within and outside of the research were born during these breaks. Dr. Rikard Land estimated an intake of approximately 1500 cups of coffee in his thesis; hence I will not try to name all people with whom I have had coffees, and instead I thank the school of Innovation, Design and Engineering for providing the coffee.

Project trips During the Ph.D. studies, one of the most rewarding parts is traveling to conferences and workshops to present the research. I have travelled to 15 conferences and workshops, in 10 countries during the past five years.

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

Visits 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 especially want to thank Shane Gladigau, Dr. Christian Guttmann, Dr. Ian Peake and Prof. Heinz Schmidt, for making my time so pleasant. Much of the work was initiated together with Prof. Heinz Schmidt during this stay. Thank you Heinz, your ingenuity is amazing.

Colleagues Dr. Mikael Åkerholm is one of the colleagues who I have known the longest. We started the undergraduate studies together, and finished the Ph.D together, and we have also worked together at CC-Systems. Thank you Mikael, 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. Thank you. I have spent countless lunches discussing training and physiology with Prof. Hans Hansson and Prof. Christer Norström, I will sincerely miss having these conversations on a daily basis. During my path towards my Ph.D., 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 also for being a great friend. Thank you Anders "Pettson" Petterson for your encouragement, friendship and discussions. Also a big thank you to my personal trainer and friend Dr. Markus Nilsson.

There are countless people who have given much joy during the last five years. Thank you Hüseyin Aysan for teaching me about photography, Moris Behnam for the travel companionship, Markus Bohlin, Stefan Bygde for new mathematical insights, Dr. Jan Carlsson and Dr. Radu Dobrin for your discussions and friendship, Harriet Ekwall for always solving the unsolvable, Dr. Joakim Fröberg, Ewa Hansen, Dr. Kaj Hänninen, Andreas Hjärtström, Dr.

Damir Isovic and Dr. Magnus Larsson for your friendship, Dr. Stig Larsson for the lessons on life, Markus Lindgren for all the lunches, Dr. Jukka Mäki-Turja, Anders Möller, Dr. Dag Nyström (I do like you too, really!), Jonas Neander, Prof. Sasikumar Punnekkat, Filip Sebek, Johan Stärner, Dr. Henrik Thane, Dr. Massimo Tivoli, Monica Wasell for always helping out, Peter Wallin, Kurt Wallnau and Gunnar Widforss for winning a photo context for me. You have all contributed to make these past years enjoyable.

Industrial experience During the last two years I have spent a great deal of time at CC-systems developing industrial systems. I have learned a lot about the life in industry. I have had a wonderful time, and I especially wants to thank Johnnie Blom, Jonas Ehlin, Carl Falk, Jörgen Hansson, Mats Kjellberg, Ken Lindfors, Mattias Lång, Jörgen Martinsson, Andreas Olevik, Malin Olsson, Johan Persson, Stefan Rönning, Göran Sohlman, Ulf Sporrong, Dr. Jochen Wendebaum, David Wretling and Anders Öberg.

The important people I want to thank my family and friends; my Greek/Finnish side of the family, Marja, Mimmis, Tina, Alexander, Sebastian, Elias and Nils for believing in and supporting me in different ways. I also want to thank my part of the family who lives in Skåne; even though we don't meet as often as we should I am still grateful for your love and support.

In the end, there are some things that matter more than others. With those words I want to thank my mother Inger and my father Tommy for your love and for teaching me the value of knowledge; and I want to thank my little sister Erika for your never ending love and support.

Finally, the person to whom I am most grateful for her perseverance and love is my fiancée Marie. Without your support I would never have come this far, I am eternally thankful.

The liabilities The work leading up to this thesis was funded by the Swedish Foundation for Strategic Research (SSF), Vetenskapsrådet (VR) and a donation from ABB. Thank you all for providing this opportunity!

Johan Fredriksson
Solna, September, 2008

Notes for the reader

This thesis deals with timing predictability and resource optimization for Embedded Real-Time Systems (ERTS) that are developed using Component-Based Software Engineering (CBSE). In Chapter 5 we introduce extensions to a development process for CBSE facilitating predictability and resource optimization of ERTS. Chapter 6 outlines a framework for reuse of Worst-Case Execution Time (WCET) analysis for software components. In Chapter 7 we describe a framework for optimization of resources for component-based ERTS. These three 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 the two frameworks. Throughout Chapters 5, 6 and 7 we use an example of an Adaptive Cruise Controller application to illustrate our techniques.

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 analyzing the methods. Chapter 8 also describes a prototype tool that implements the ideas outlined in Chapter 6.

Chapter 2 give an introduction to real-time systems, real-time analysis and WCET analysis, and Chapter 3 gives the basics of CBSE and describes fundamental concepts 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. Furthermore, this thesis enclose 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 or co-authored by Johan Fredriksson.

Contents

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.

-Hitchhiker'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 vehicular systems and consumer electronics.

We have developed methods for, (i) reusing WCET analysis for reusable software components, and, (ii) allocating 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 real-world example 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.

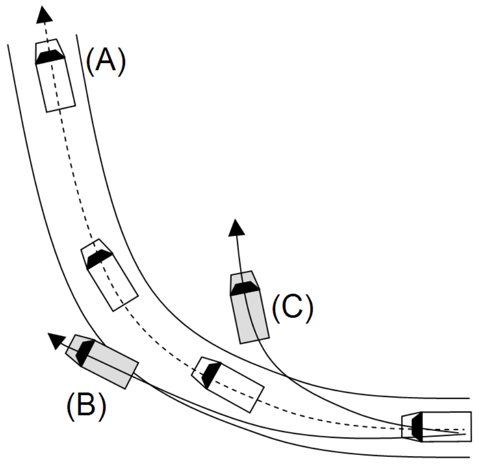


Figure 1.1: Conceptually electronic stability control.

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. In vehicular systems 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. For example, a typical sensors in the ESC system include 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 ECUs continuously read 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 while at the same time 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 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.*

The power of software controlled systems has lead to modern cars having up to 90 ECUs controlling the different functions in the car. Such computer systems that are embedded in apparently non-computerized electrical and electromechanical 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 real time systems (ERTS) 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 cars 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:

- ERTS 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 an 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 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, and finally, a *component technology* is the concrete implementation of a component model. To facilitate reuse, components are designed to be generic and often with functionality suitable 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. Because 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 software 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 behaviour is actually going to be used. Unfortunately generality and context-independence also leads to an increasing inability to make accurate predictions of the component behaviour for a given specific use-case. Hence, there is a need for parameterization 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 analyzing real-time properties, e.g., RTA, but few suitable when applying 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 these 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 while at the same time guaranteeing real-time constraints. This is because it becomes harder to make tight predictions and keep low 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 prerequisite for RTA.
- WCET analysis is complex and time consuming.
- Reuse of current WCET analysis results between a component's 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 in a one-to-one fashion 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 resource 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 larger errors of individual tasks have a greater impact on properties like input jitter and output jitter, just to mention a few.

Summarizing the above

- A high number of tasks increases the resource consumption, and decreases system performance.

- Allocation of components to real-time tasks must maintain the component architecture.

- Allocation of components to real-time tasks must consider temporal constraints.

1.3.3 Developing embedded real-time systems

Even simple embedded systems today show more and more complex behaviors, some triggered by usage-awareness or deployment-specific configuration parameters. Properties of the component such as time and reliability are variable and usage-dependent, and the variance may be large. Together with global system 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 ERTS, 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 directly applicable 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 method for parameterizable and reusable prediction of the WCET property 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 analyzed 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:** Analyzing 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.

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] WCET. 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 analyzing real sized industrial software systems [SEG$^+$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 Stachulat et al. use partitioning only for increasing the accuracy of WCET analyses.

Gheorghita et al. in [GSBC05] use usage scenarios to determine tighter loop bounds. In [MMH$^+$05] Mohan et al. use run-time usage information for

[1]We give an exact definition of accuracy in Chapter 6.
[2]As exact as the underlying WCET analysis can provide.

dynamic voltage scaling depending on the timing requirements. Wenzel et al. [WRKP05] use both model checking and genetic algorithms to derive which input data that 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.

In [BCP03b, BCP03a] each basic block of a program is analyzed 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. [JWLQ06] divide the source code in modes depending on input, and only the parts that are used in the a specific usage are analyzed.

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 parameterizable 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 designer for various reasons. It can, e.g., be used for identifying bottlenecks, and hence is very useful for further optimizing 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:** Analyzing 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 that actually cause the WCET has been sparsely addressed.

Wenzel et al. [WRKP05] use both model checking and genetic algorithms to derive which input data that 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, and in such systems the software is often divided into a number of *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 task. A *scheduler* uses this information to control the execution of the task. The scheduler invokes tasks periodically or at any time, i.e., aperiodically, and usually have timing requirements. Components triggered with the same periodicity 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 ERTSs 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 calculate 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 system 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 $\approx 50\%$ and $\approx 30\%$ 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 has 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 is the allocation of real-time tasks to distributed processors, and different techniques are used to solve this problem, e.g., constraints programming is [HCDJ08], simulated annealing [BNTZ93, TBW92], branch and bound [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 a component's WCET for component-based ERTS, and (ii) a method for allocating components to tasks while minimizing stack-usage and CPU-overhead, and at the same time maintaining real-time constraints. The main contributions of the presented research are summarized as follows:

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

C2 *Methods for deriving WCET input values*. The input space of a component is divided into partitions 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 derivation of WCET input values 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 augmented with the methods outlined in this thesis. The WCET analysis is divided and positioned in both the component and system part. The component to task allocation is positioned after the reusable WCET analysis for providing a tight WCET to the allocation framework.

C5[3] *A prototype tool* implementing the ideas from contributions C1 and C2. The prototype tool graphically presents 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 outline

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 described in Chapters 6 and 7 can be individually used for improving predictability and resource utilization for ERTS; however, they are synergistic.

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 CBSE for ERTS by increasing resource efficiency and analyzability.

Chapter 2 provides the reader with a theoretical background on 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 survey of the current state of research.

[3]C5 is not a scientific contribution.

Contrib.	**CBSE for ERTS**	**Resource efficiency and predictability**	**Development of ERTS**	**Paper**
C1	We create techniques for predicting WCET that can be reused through parametrization, yet with high accuracy.	Higher accuracy of predictions allows the developer to dimension hardware correctly.		2 3 5
C2		Higher accuracy of predictions allows the developer to dimension hardware correctly.		1
C3	Helps to systematically allocate components to real-time tasks, something that otherwise often is performed ad-hoc.	Minimizing system overheads through efficient allocations from components to tasks, while at the same time maintaining both real-time requirements and component architecture.		6 7
C4			We integrate both WCET analysis and model transformation in the component-based development process.	4
C5	Validation of the methods.	Validation of the methods.		

Table 1.1: The relation between contributions, problem formulations and publications (a list of publications is presented in Appendix B).

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 evaluation data.

Appendix B summarizes all publications, and reports written during the Ph.D. studies.

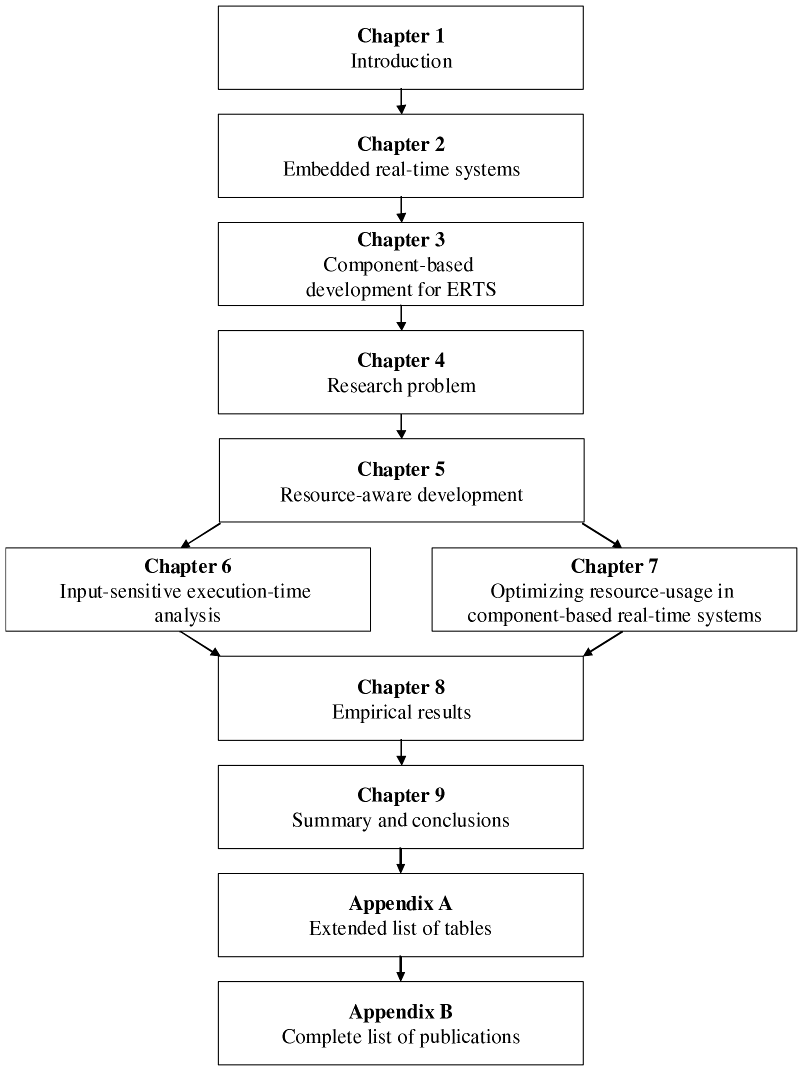


Figure 1.2: An overview of the chapters in the thesis.

Time is an illusion. Lunchtime doubly so.

-Hitchhiker's guide to the galaxy

Chapter 2

Embedded real-time systems

In this chapter we give an introduction to Embedded Real-Time Systems (ERTS). We describe 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 Breaks (ABS) are controlled by an ERTS, continuously monitoring and controlling the brakes to ensure the maximum braking effect. In the unlikely event 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 an embedded system. 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] define 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.*

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 controlled system used to achieve a specific purpose and the computer is not the end-product itself. Such systems are typically found in, e.g., medical equipment, robotics, and, vehicular and automotive systems.

2.2 Real-time systems - general concepts

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 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 temporal requirements, as depicted in Figure 2.1.

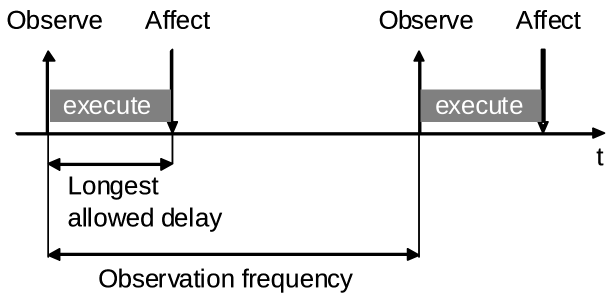


Figure 2.1: Simple real-time model.

The observation frequency is enforced in 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, i.e., 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 systems consists of sensors and an Airbag 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 ACU which 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 pretensioners, 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 as the airbag has to be deployed before an occupant moves forward 125 mm relative to the car. Normally it takes 30 ms 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, i.e., 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 a real-time system 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 systems where a failure is considered to be a fault, are denoted *hard* RTS, while systems where occasional failures can be accepted are denoted *soft* RTS.

Hard real-time systems

Hard RST are systems where the consequences of a failure to meet all constraints 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 analyzed 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 executions are usually considered tolerable. The constraints in such systems 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. However, 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-triggred 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 has 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 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 periodic task τ_i is described by (and depicted in Figure 2.2):

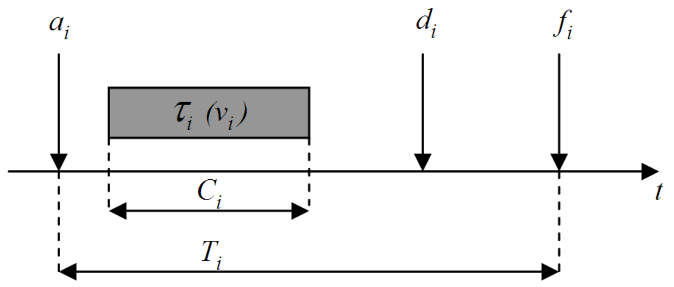


Figure 2.2: Task model.

- 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.

- 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 task τ_j is described by:

- An activation time a_j which is the time when the event arrive at the system. Computation time, C_j, 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_j, specifies a constraint on the completion time of the task. The task must finish no later than d_j time units after it has been activated.
- Priority value, v_j is a user defined integer value that represents the relative importance between tasks in the system.

The major difference between a periodic task τ_i and an aperiodic task τ_j is that τ_i is strictly periodic, triggered with a strict period time, whereas the latter can arrive at the system at any time. In many systems periodic tasks have higher priority than aperiodic tasks.

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 in- and outputs.

Usually each task need to allocate at least one active resource to execute, and the progress of 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, i.e., meet all tasks deadline constraints while at the same time minimizing the use of resources. There exist a wide range 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 require 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 [FÅDS03, BBB04]. The reclaimed resources can be used for executing aperiodic tasks or to 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 several different types of approaches for pre run-time schedulability analysis, two of the most commonly used are utilization-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 sufficient but 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.

$$U \leq 0.69 \; as \; N \rightarrow \infty$$

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

2.5.2 Response time analysis

Research on schedulability for 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 their 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 τ_i:

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

Here the worst-case response time R_i of task τ_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 τ_i [SH98].

In order to guarantee convergence one must either ensure a total task utilization not greater than 100% or one can stop iterating when $R_i^{n+1} > D_i$, i.e., a deadline violation has occurred.

2.5.3 Transactions

Transactions are collections of related tasks, which collectively perform some function or have some shared timing attributes. A transaction usually has a timing requirement, i.e., an end-to-end deadline. 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* (least common multiple of all periods), and a common approach for this 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.

2.5.4 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 straight forward and difficult to use. That is because they are difficult to justify and they assume that all attributes are changeable. A more straight forward approach is the one by Bate and Burns [BB99]. Timing requirements such as *Period, Deadline, Jitter* and *Separation* are considered. Furthermore, transactions are sequences of tasks executing in a fixed order. The timing requirements for transactions are *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 tools are often not able to estimate the exact WCET. In those cases the WCET estimate should preferably be a safe, yet tight, overestimation.

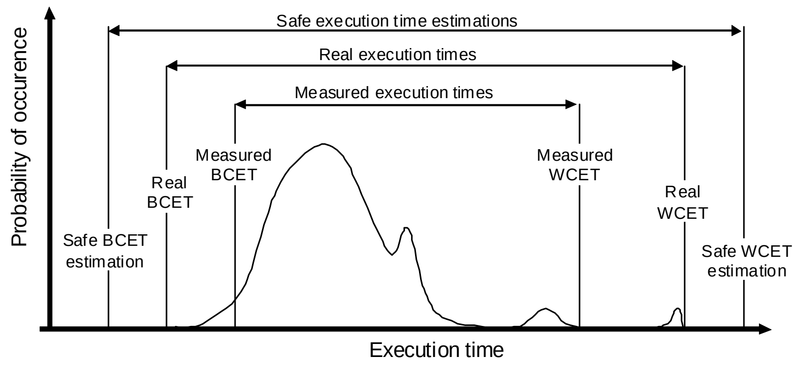


Figure 2.3: Execution time analysis.

2.6.1 Classification of WCET analysis

The different types of analyses include *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, and will report the longest observed execution time. Often, however, the absolute worst-case has not been observed (Figure 2.3). *Hybrid WCET analysis* 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[1].

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 analyzers, or hardware trace mechanisms to derive the timing of the program or parts of the program [BCP02, BCP03b, BCP03a, WRKP05, WKE02]. Since it is impossible for most programs to test all input value combinations, often only a subset of the possible executions are 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 measurement-based methods. An advantage of the measurement-based 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

[1]Given that a cache miss always produces longer execution-times than that of a cache hit (which may not always be true for all architectures).

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 analyze 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 on-line 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 parameterize 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. Lisper [Lis03] 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 Lisper's work. Their work has been developed and tested with the aiT tool [aiT]. Current work on the methods outlined by Lisper is also presented in [BL08, AHLW08].

2.7 Summary

This chapter has presented basic concepts and terminology used throughout this thesis for reasoning about embedded real-time systems.

It is a mistake to think you can solve any major problems just with potatoes.

-Hitchhiker's guide to the galaxy

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 from OMG. These technologies are used for desktop and distributed enterprise applications all over the world. However, these technologies are typically not used for all classes of systems. They are not used for (i) resource constrained systems; they are simply to 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.

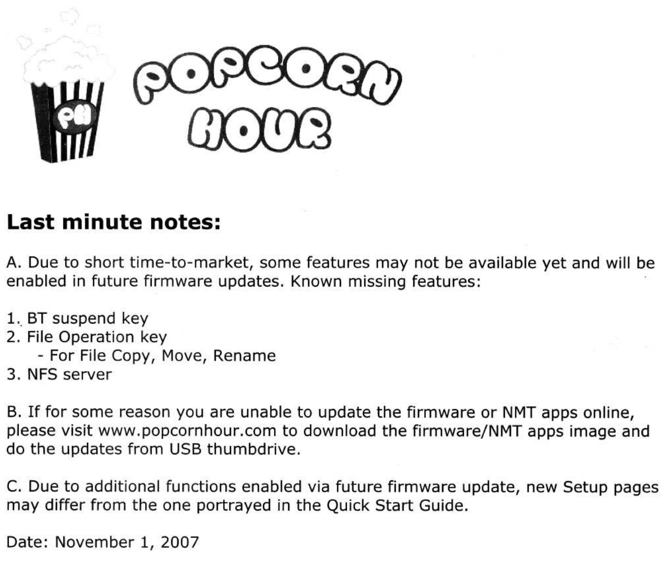
Looking at Embedded Real-Time Systems (ERTS) they often contain elements of (i), (ii) and (iii).

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 ERTS.

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 they 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 being shortened 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 that I bought in spring 2008, Figure 3.1 [Pop08], which was 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

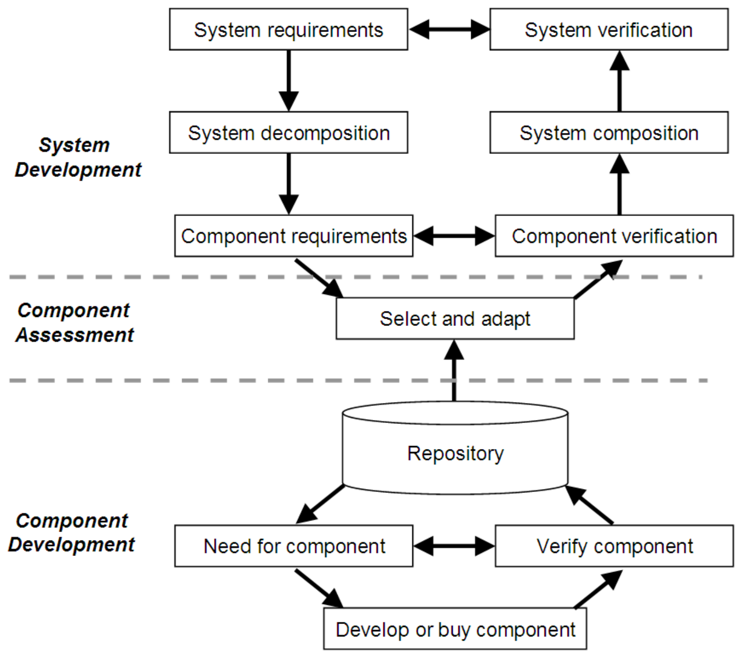


Figure 3.2: CBSE development process.

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 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

3.5.1 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 Szypersky [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. Szypersky 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 do not say anything about requirements on the component model. The two definitions principally agree, since the requirement that components can be composed 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 standardization means that a component that is used in a CBSE development process has to conform to some standardized component model. This model may define, e.g., component interfaces, component meta-data, documentation, composition and deployment.

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.

3.5.2 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.

3.5.3 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 have existed for several years, CBSE may require more of an interface than earlier applications. In [WBS97], 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.

3.5.4 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.

3.5.5 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.

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

Synchronization. Synchronization between components.

Quality of service. In contrast to the behavioral 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.

3.6 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].

3.7 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 offer the services that components require [CL02a].

3.8 Summary

During the last decade advances have been made in component-based development for desktop and business 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 other 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 original 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 applications 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.

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

-Hitchhiker's guide to the galaxy

Chapter 4

Research problem

In this chapter we describe the industrial and research problems considered, and we discuss our solutions and the research methodology that 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 methods and analysis tools. One of the development strategies that industry is interested in is CBSE, mainly for its focus on structured reuse. However, because of lack of models and tools for analyzing 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 [ÅCF+07]. In general 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 in embedded systems. The increasing complexity of software is an incitement 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 that we intend to solve.

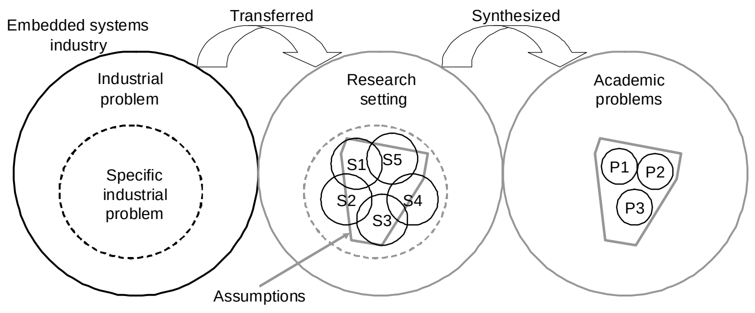


Figure 4.1: Transformation of industrial problem to academic problems in a reduced academic setting.

Good research requires not only a clearly stated problem and results, but also convincing evidence that the results are sound. In order to fulfill these criteria and approach the problem methodically, our research method is divided 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

assumptions that restricts the academic problem are identified. Further, we formulate three specific academic research problems **P1**, **P2** and **P3**, based on the assumptions and the problem statements **S1**-**S5**, as depicted in Figure 4.1 (Section 4.5).

3. We devise general solutions to the different problems in the for 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 → problem statements and assumptions → research problems → requirements → solution proposals → validation. This logical chain is depicted in Figure 4.2.

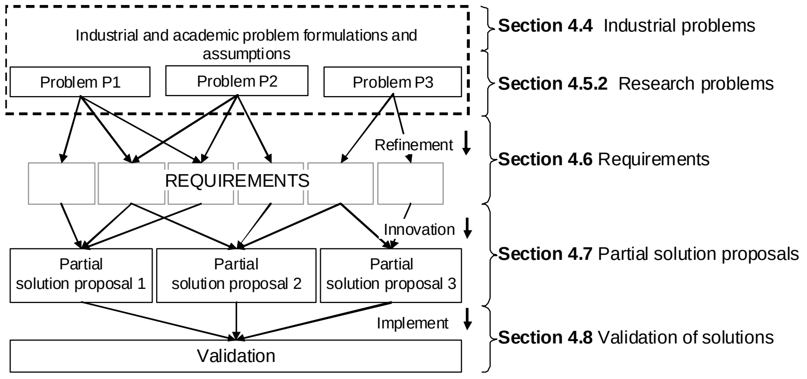

Industrial and academic problem formulations and assumptions
Problem P1
Problem P2
Problem P3
Refinement
REQUIREMENTS
Innovation
Partial solution proposal 1
Partial solution proposal 2
Partial solution proposal 3
Implement
Validation
Section 4.4 Industrial problems
Section 4.5.2 Research problems
Section 4.6 Requirements
Section 4.7 Partial solution proposals
Section 4.8 Validation of solutions

Figure 4.2: Logical structure of chapter 4.

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.

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], and 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 *temporal 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 wide spread also in machinery and consumer electronics to satisfy increasingly demanding customers [Bro06, PBKS07].

Temporal 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's temporal 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 temporal correctness of real-time systems [WEE+08].

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 ERTS 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, BCC+03].

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, the level of 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 analyzing 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 can not easily be reused. Also timing prediction is a difficult and time-consuming activity, resulting in a higher development effort compared to if the predictions would be 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 multiple 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 an ERTS 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 analyzable. 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 WCET analysis tool [GESL06], 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 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.

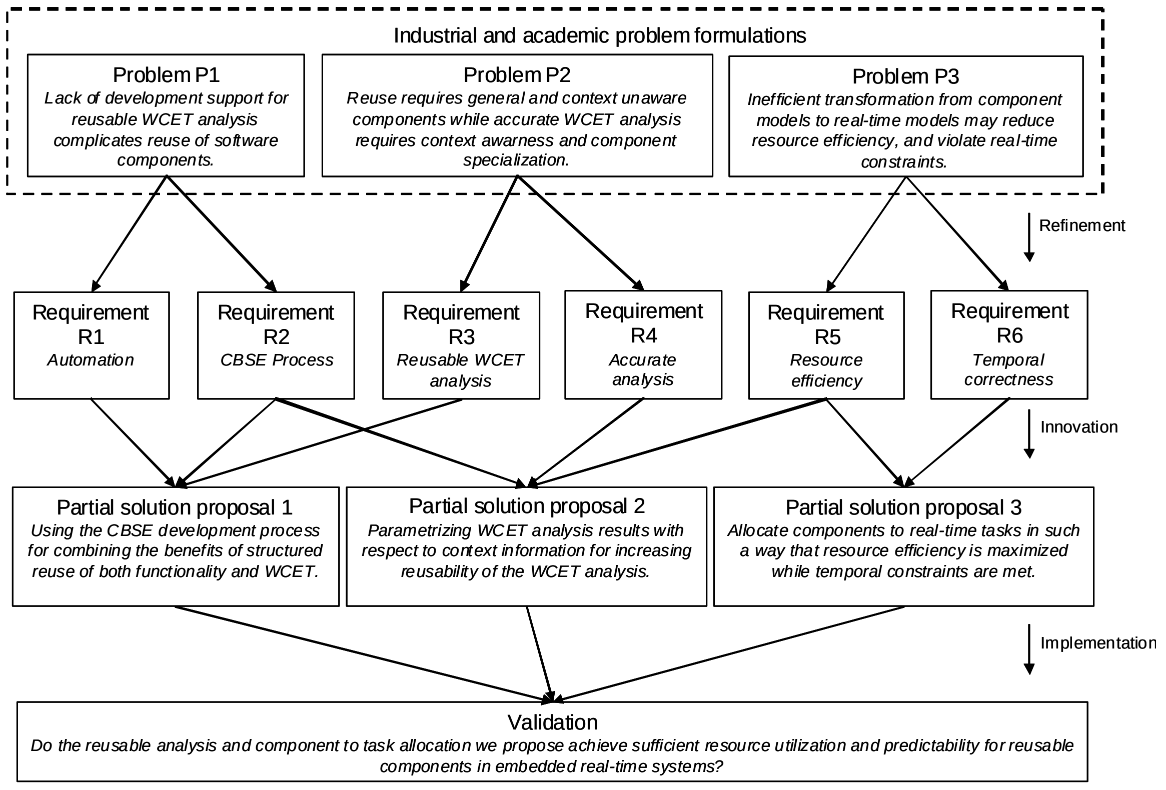

Industrial and academic problem formulations
Problem P1
Lack of development support for reusable WCET analysis complicates reuse of software components.
Problem P2
Reuse requires general and context unaware components while accurate WCET analysis requires context awarness and component specialization.
Problem P3
Inefficient transformation from component models to real-time models may reduce resource efficiency, and violate real-time constraints.
Refinement
Requirement R1
Automation
Requirement R2
CBSE Process
Requirement R3
Reusable WCET analysis
Requirement R4
Accurate analysis
Requirement R5
Resource efficiency
Requirement R6
Temporal correctness
Innovation
Partial solution proposal 1
Using the CBSE development process for combining the benefits of structured reuse of both functionality and WCET.
Partial solution proposal 2
Parametrizing WCET analysis results with respect to context information for increasing reusability of the WCET analysis.
Partial solution proposal 3
Allocate components to real-time tasks in such a way that resource efficiency is maximized while temporal constraints are met.
Implementation
Validation
Do the reusable analysis and component to task allocation we propose achieve sufficient resource utilization and predictability for reusable components in embedded real-time systems?

Figure 4.3: Research flow with dependencies between solutions and problems.

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

Assumption: Stack/memory analysis is available.

We do not provide methods or tools for analyzing 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, and, task control block size for the run-time system.

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

Assumption: All components can be analyzed.

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 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 incitements 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:

- Accurate WCET analysis requires context information [GESL06, KP05].
- Accurate analysis is required for correctly dimensioned hardware, and correct system behaviour [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 analyzed.

Resource efficiency: How much resources a component requires in order to successfully fulfill 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.

4.6.1 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 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 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 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 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.

⟨*Requirement R5*⟩ **Resource efficiency:** Components 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.

⟨*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 completely 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.

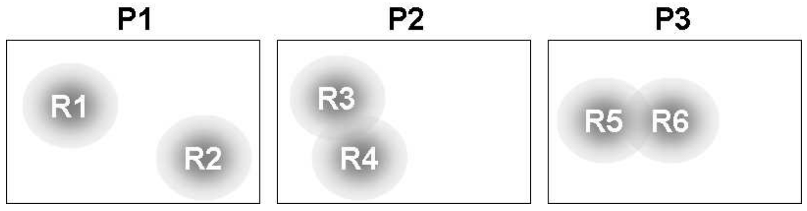


Figure 4.4: Conceptual coverage of requirements with respect to the research problems in this thesis.

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 parameterization 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. However, 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 parameterizable methods. Then the methods may be applicable to a larger set of domains.

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

Fulfils requirements: R1, R3, R4

How does this solution fulfill 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 fulfill requirements R3? By creating parameterizable component WCET contracts, that are parameterizable 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 fulfill 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 parameterizing 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 synthesized 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 fulfill 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 fulfill requirements R6? The allocation framework has a set of rules for evaluating an allocation if it is feasible, those rules also encode the temporal correctness. Thus a transformation that is resource efficient but does not fulfill 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 know 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 using 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 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.

Original research: Paper 2 and Paper 4.

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 can not 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.

Original research: Paper 1, Paper 2, Paper 3 and Paper 5

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 can not 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 2, Paper 6 and Paper 7.

Described in thesis: Chapter 7 and Chapter 8.

4.9 Summary

This chapter has outlined the research method used when addressing the research problems in the thesis. This chapter is also the concluding chapter in the introductory part, and the following chapters will address the technical contributions.

Even he, to whom most things that most people would think were pretty smart were pretty dumb, thought it was pretty smart.

-Salmon of Doubt

Chapter 5

Resource-aware development

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 describe 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 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, and, to address these issues in relation to the CBSE development process.

5.1 Component-based software engineering process

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 [ABGP05]. 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.

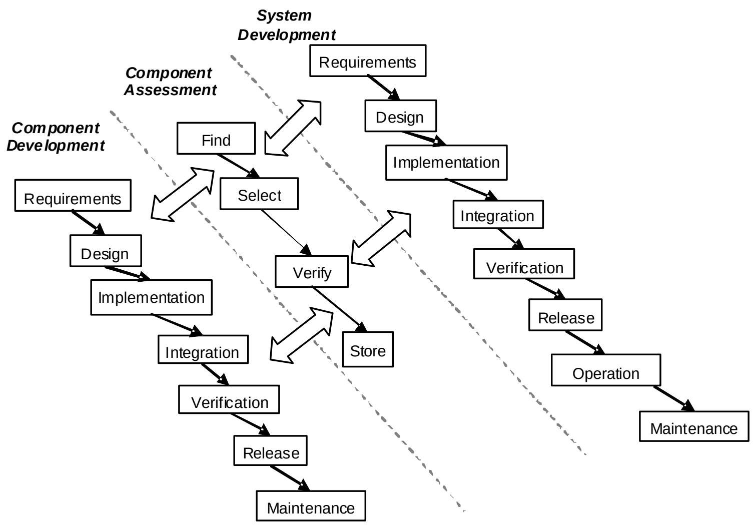


Figure 5.1: CBSE development process (from [CCL06]).

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 fulfill its real-time and resource constraints. Particularly important in ERTS is to predict the WCET 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]. A lot of effort can be saved if the WCET analysis can be 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 components where the component developer performs reusable WCET analysis, and (ii) the development of systems and usage where the system developer uses the reusable WCET analysis on a specific system to find the specific WCET of each component for the system, as depicted in Figure 5.2.

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

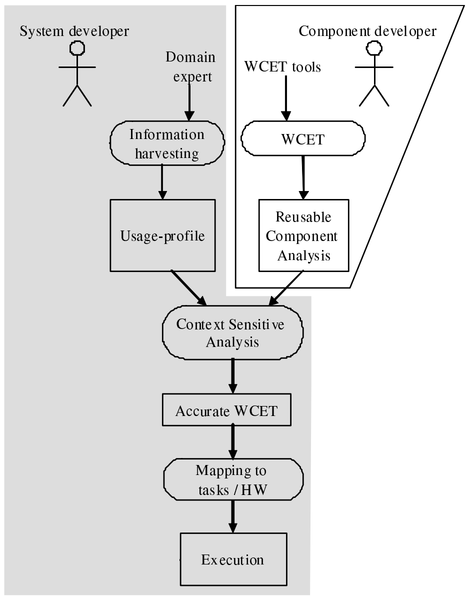


Figure 5.2: Resource centric development process view.

Figure 5.3 shows how a general model for a CBSE process 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.

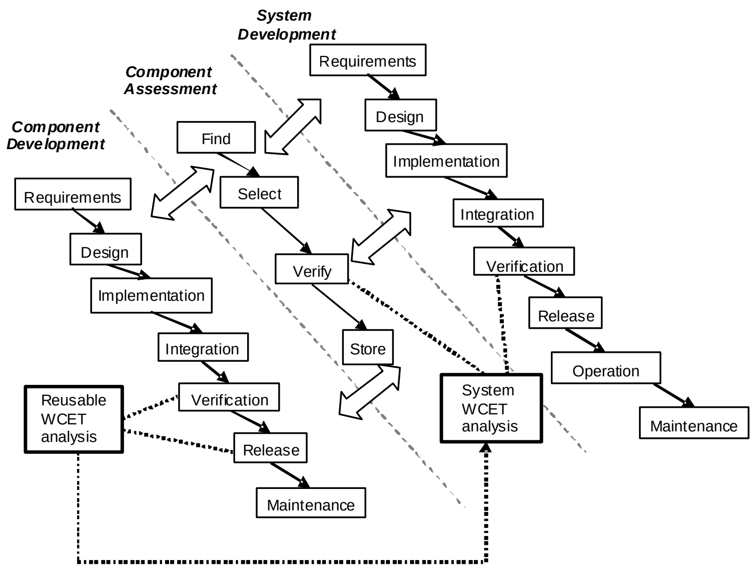


Figure 5.3: CBSE development process extended with reusable WCET analysis.

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 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 ERTS, since traditional ERTS 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 allocated to real-time tasks in order to minimize resource consumptions while maintaining real-time constraints, e.g., respecting deadlines.

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.

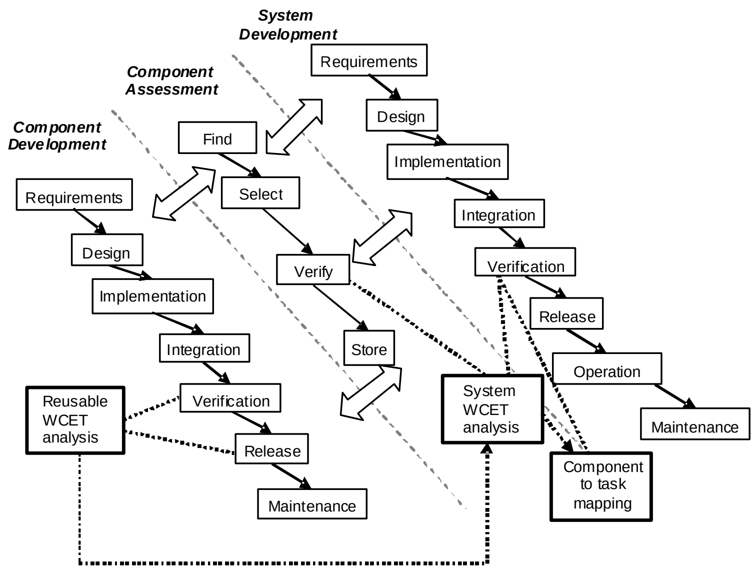


Figure 5.4: CBSE development process extended with reusable WCET analysis and component to task allocation.

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 Autocomp [SFÅ04]. Components can be co-allocated, where many components form a task. Nevertheless, a component can be distributed over several tasks. 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 models

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.

Descriptive models

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

[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.

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.

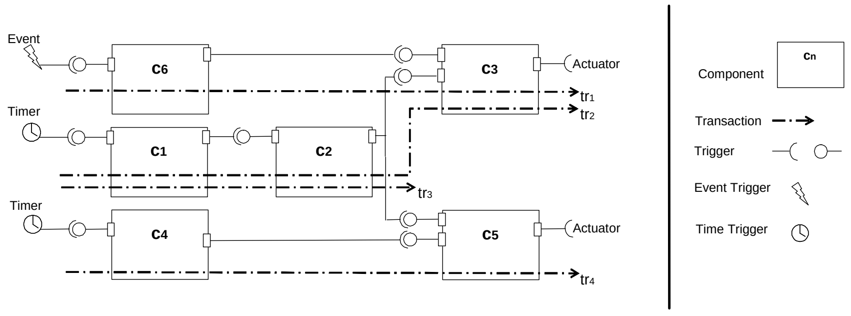


Figure 5.5: Graphical notation of the component model.

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 $\langle \mathbf{P}_i, \mathbf{R}_i, S_i, Q_i, m_i, f_i, prog_i \rangle$, where $\mathbf{P}_i$ is the provided interface, which is a set $\{p_{i,0}, p_{i,1}, ..., p_{i,n-1}\}$ of input variables and $\mathbf{R}_i$ is the required interface, i.e., a set $\{r_{i,0}, r_{i,1}, ..., r_{i,n-1}\}$ 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. m_i is the amount of stack memory required by the component. 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 $\mathbf{U}_i$ such that $f_i : \mathbf{U}_i, pt_i \rightarrow WCET_i$, where $\mathbf{U}_i$ is a *usage profile* and pt_i is a probability threshold used for removing WCET with low probability. $prog_i$ is the software behaviour of the component in some form, e.g., source code, graphical model or binary format.

Usage profile $\mathbf{U}_i$ is a set of inputs and probabilities for those inputs. $\mathbf{U}_i$ 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 Γ_i is an ordered relation between components $c_a, c_b, c_c, ..., c_n$, and an end-to-end deadline dc_i. The deadline is relative to the event that triggered the component transaction, and the first component within a transaction defines the transaction trigger. Γ_i^T defines that the transaction is time-triggered and Γ_i^E 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.

$$c_a\Sigma^* c_b\Sigma^* c_c\Sigma^* ...\Sigma^* c_n \tag{5.1}$$

Where Σ^* 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_a, c_c 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 model

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 analyzed 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 τ_j is a tuple $\langle \mathbf{Z}_j, T_j, C_j, stack_j \rangle$ where $\mathbf{Z}_j$ is an ordered set of components. Components within the same task are executed in sequence following the order of $\mathbf{Z}$ 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. T_j, C_j and $stack_j$ are deduced from the components in $\mathbf{Z}_j$. C_j is the sum of WCET for all components in $\mathbf{Z}_j$. Hence, for a task τ_j, the parameter 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 $\mathbf{Z}_j$ to facilitate scheduling analysis.

$$C_j = \sum_{\forall_i (c_i \in \mathbf{Z}_j)} \left(f_i \left(\mathbf{U}_i, pt_i \right) \right) \tag{5.2}$$

$$stack_j = \max_{\forall_i (c_i \in \mathbf{Z}_j)} (m_i) \tag{5.3}$$

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 model

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 β 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 ρ 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 $\langle dc_0, dc_1, dc_2 \rangle$. The transactions are described in Table 5.2.

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

Component	**Name**	$\langle \mathbf{S_i}, \mathbf{Q_i}, \mathbf{m_i} \rangle$
Speed Limit	c_{speed}	$\langle$ 50Hz, 20, 1024$\rangle$
Object Recognition	c_{obj}	$\langle c_{speed}, 20, 512\rangle$
Brake Assist	c_{break}	$\langle c_{obj}, 20, 512\rangle$
Logger HMI Output	c_{log}	$\langle$ 10Hz, 100, 2048$\rangle$
ACC Controller (assembly)	c_{acc}	$\langle c_{obj}, 20, 2048\rangle$

Table 5.1: The five components *Speed Limit*, *Object Recognition*, *Brake Assist*, *Logger HMI Output* and *ACC Controller* as depicted in Figure 5.6.

Following the CBSE work flow as depicted in Figure 5.2 we discuss the

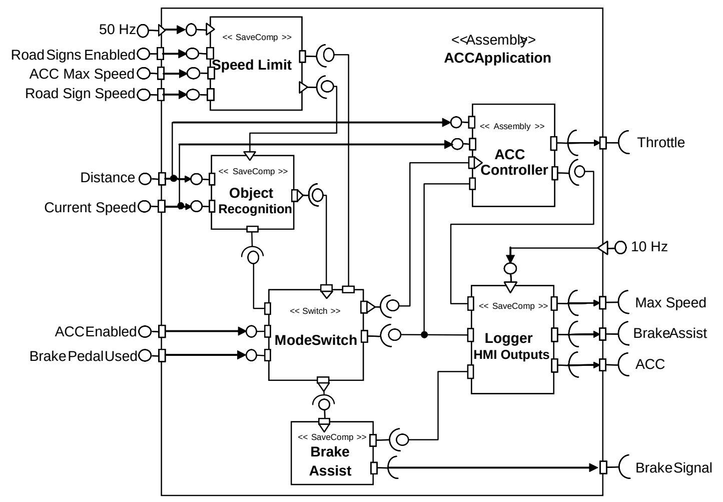


Figure 5.6: Adaptive Cruise Control.

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.

Γ_i	N_i	dc_i
Γ_0	$c_{speed} \rightarrow c_{obj} \rightarrow c_{break}$	800
Γ_1	$c_{speed} \rightarrow c_{obj} \rightarrow c_{acc}$	2000
Γ_2	c_{log}	10000

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 $\mathbf{U}_1$ and $\mathbf{U}_2$, 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 *BrakePedal 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.

RSE	**AMS**	**RSS**	**D**	**CS**	**AE**	**BPU**
0,1	250	0..130	0..2k	0..250	0	0,1

Table 5.3: High-end car usage profile $\mathbf{U}_1$.

RSE	**AMS**	**RSS**	**D**	**CS**	**AE**	**BPU**
0	130	0	0	0..250	0	0,1

Table 5.4: Low-end car usage profile $\mathbf{U}_2$.

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 $f_i : U \rightarrow WCET$ is parameterized with the usage profile for respective car model, and the WCET is given.

Comp.	**ui**	$\mathbf{U}_1$	$\mathbf{U}_2$
SpeedLimit	304	284	105
ObjectRecog.	201	201	120
BrakeAssist	174	91	88
Logger.	400	303	303
ACC Controller	1241	1241	769

Table 5.5: WCETs according to the methods usage independent (ui) and usage dependent (ud) and usage dependent with usage profiles ud($\mathbf{U}_1$) and ud($\mathbf{U}_2$).

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

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

Predicate for component Speed limit
$RSE = 1 \rightarrow WCET = 304$
$RSE = 0 \wedge RSS > 0 \rightarrow WCET = 284$
$RSE = 0 \wedge RSS = 0 \rightarrow WCET = 105$

Table 5.6: c_{speed} resulting contract f_{speed}.

Not considering the usage potentially results in very inaccurate WCET. In Section 6.5 we outline the detailed analysis for the component c_{speed} (Speed Limit), and describe the contract.

5.5.3 Allocating component to tasks

When the system is developed and analyzed, 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.

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,

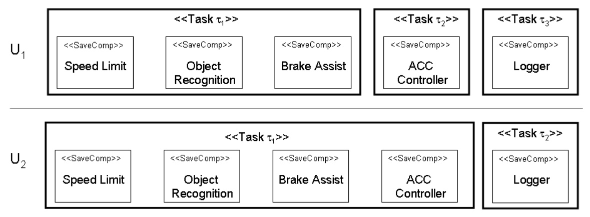


Figure 5.7: Allocation of components to real-time tasks with respect to the different WCET predicted considering U_1 and U_2.

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 7.4 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.

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.

Allocation	$\mathbf{p_A}$	$\mathbf{s_A}$
1-1 allocation	4.6% (0%)	7644 (0%)
U_1 allocation	2.4% (63%)	6020 (31%)
U_2 allocation	1.1% (100%)	2384 (100%)
Optimal allocation	1.1% (100%)	2348 (100%)

Table 5.7: CPU overhead and stack usage with respect to different allocations (improvement in parenthesis).

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 7.4 and the allocations in detail in Section 7.6.

5.6 Summary

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 the development of a system requires the system context and usage to be well known. In contrast, the CBSE development process is divided in 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.

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.

Forty two

-Answer to Life, the Universe, and Everything.

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 component 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 component inputs that gives the WCET.
- *various approaches* to speed up the search over the value space, allowing the input parametric WCET, and the WCET input values, to be quicker derived, in some cases.

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 estimates, 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 estimates. The contract is not created with respect to any specific component usage, but describes the timing behaviour of the component solely with respect to its inputs.

The WCET estimate 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.

For our results this implies that the estimated WCET will always be safely overestimated.

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 $v_{i,j} \in \mathcal{V}_{i,j}$. 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 $v_{i,j}$ represents the value of the provided input $p_{i,j}$, and $\mathcal{V}_{i,j}$ represents the possible values for $v_{i,j} \in \mathcal{V}_{i,j}$.*

Input value combinations for a provided input interface $P_i = \{p_{i,0},\ p_{i,1}\}$ are described as tuples $\langle v_{i,0}, v_{i,1}, \ldots, v_{i,n-1}\rangle$. Consider the two input variables $v_{i,0}$ and $v_{i,1}$, with the value domains $\mathcal{V}_{i,0} = \{v : 0 \leq v \leq 1\}$ and $\mathcal{V}_{i,0} = \{v :$

$0 \leq v \leq 1\}$, then the possible input variable combinations are described by the input value space $\mathbf{D}_i = \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$.

Definition 6.2. *An input value space $\mathbf{D}_i$ is a set of input value combinations, as formalized in 6.1.*

$$\mathbf{D}_i = \{\langle v_{i,0}, v_{i,1}, ..., v_{i,n-1}\rangle : v_{i,j} \in \mathcal{V}_{i,j}\} \tag{6.1}$$

An input combination tuple $\mathbf{d}$ 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 $\mathbf{d} = \langle 0, 1\rangle$ 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 $\mathbf{d} = \langle v_{i,0}, v_{i,1}, ..., v_{i,n-1}\rangle$ describes an input combination for the provided interface P_i.*

$\mathbf{D}_i$ represents all possible value combinations tuples $\mathbf{d}$ for a component c_i. In the input-sensitive WCET analysis, the input value space $\mathbf{D}_i$ is partitioned in subsets. A predicate ϕ considering the input value space $\mathbf{D}_i$ is used for partitioning. For example, the predicate ϕ 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, each tuple $\mathbf{d} \in \mathbf{D}_i$ 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 ϕ 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. *ϕ is a predicate on an input value space $\mathbf{D}_i$, and $\phi(\mathbf{d})$ notates that the tuple $\mathbf{d}$ fulfills ϕ.*

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

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

$$\mathbf{D}_{i|\phi} = \{\mathbf{d} \in \mathbf{D}_i : \phi(\mathbf{d})\} \tag{6.2}$$

If ϕ 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 iterates through all input value combinations $\mathbf{d} \in \mathbf{D}_i$ and removes those that violate the predicate ϕ to derive $\mathbf{D}_{i|\phi}$.

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

The number of possible concrete input tuples for the initial input value space is $|\mathbf{D}_i| = 2^8 \cdot 2^8 = 2^{16}$, and the resulting number of possible concrete input tuples in the input value space partition $|\mathbf{D}_{i|\phi}| = 3$.

```
BEGIN build_value_space_partition
  $\mathbf{D}_i = v_{i,0} \times v_{i,1} \times \cdots \times v_{i,n-1}$;
  $\mathbf{D}_{i|\phi} = \mathbf{D}_i$ ;
  DO
    FOREACH tuple $\mathbf{d} \in \mathbf{D}_{i|\phi}$ DO
      IF $\mathbf{d}$ violates $\phi$ THEN
        remove $\mathbf{d}$ from $\mathbf{D}_{i|\phi}$ ;
      FI
    END FOREACH
  UNTIL all dependencies in $\phi$ are solved ;
  RETURN $\mathbf{D}_{i|\phi}$ ;
END build_value_space_partition
```

Figure 6.1: Building an input value space partition.

Note that an input value space partition $\mathbf{D}_{i|\phi}$, associated with a predicate ϕ, is always a subset of $\mathbf{D}_i$ such that $|\mathbf{D}_{i|\phi}| \leq |\mathbf{D}_i|$.

Definition 6.6. $|\mathbf{D}_i|$ *is the concrete number of input combination tuples in the set* $\mathbf{D}_i$.

The number of possible non-empty input value space partitions $|\mathcal{P}(\mathbf{D}_\mathtt{i})|$ is then $2^{|\mathbf{D}_i|} - 1$. The powerset $\mathcal{P}(\mathbf{D}_\mathtt{i})$ contains all subsets $\mathbf{D}_{i|\phi} \subseteq \mathbf{D}_i$. An input value space partition $|\mathbf{D}_i| = 1$ is denoted *single valued* input value space partition.

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

Definition 6.7. $WCET_{i|\phi} = est\_wcet(c_i, \mathbf{D}_{i|\phi})$ *is an estimate of the WCET* *for an input value space partition $\mathbf{D}_{i|\phi}$ with respect to a software component c_i.*

Definition 6.8. $BCET_{i|\phi} = est\_bcet(c_i, \mathbf{D}_{i|\phi})$ *is an estimate of the BCET for an input value space partition $\mathbf{D}_{i|\phi}$ with respect to a software component c_i.*

Due to large number of possible program states precision may be lost in a WCET-analysis tool, generating large overestimations for WCET and underestimations for BCET. A smaller input domain generates fewer program states; thus, limiting the input domain $\mathbf{D}_{i|\phi}$ normally increases the precision of the WCET estimate. Thus the WCET estimates normally become more accurate when analyzing smaller parts of the behaviour by limiting the inputs:

$$est\_wcet\,(c_i, \mathbf{D}_i) \geq \max_{\forall_\phi \mathbf{D}_{i|\phi}} \big(est\_wcet\big(c_i, \mathbf{D}_{i|\phi}\big)\big)$$

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. $new\_D\,(\mathbf{D}_i, \phi) \rightarrow \mathbf{D}_{i|\phi}$ *creates a new input value space partition $\mathbf{D}_{i|\phi}$ with respect to an initial input value space partition $\mathbf{D}_i$ and a predicate ϕ.*

Primitive 6.2. $new\_vc\,(\phi, v_{i,j}, predicate) \rightarrow \phi'$ *created a new predicate ϕ' with respect to the predicate ϕ and the additional predicate on $v_{i,j}$, i.e., ϕ' is more restrictive than ϕ.*

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

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

If no input variable fulfills these criteria, then $select\_var(\mathbf{D}_{i|\phi}, strategy_i)$ will return nil.

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

Primitive 6.5. *$max\_value\,(v_{i,j}, \phi) \rightarrow u$ returns the highest value u of $v_{i,j}$ with respect to the predicate ϕ.*

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 $v_{i,j} \leftarrow \{l..u\}$ instead of $l \leq v_{i,j} \leq u$ 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 $real\_wcet(c_i, \mathbf{D}_{i|\phi})$ be the real WCET for the component c_i and the input value space partition $\mathbf{D}_{i|\phi}$. For the input-sensitive WCET analysis to work correctly the following assumptions should hold:

Assumption 6.1. *The WCET calculation should never underestimate the WCET, i.e., $real\_wcet(c_i, \mathbf{D}_{i|\phi}) \leq est\_wcet(c_i, \mathbf{D}_{i|\phi})$ should be true for any $\mathbf{D}_{i|\phi}$.*

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., when $|\mathbf{D}_{i|\phi}| = 1$, then $est\_wcet(c_i, \mathbf{D}_{i|\phi}) = real\_wcet(c_i, \mathbf{D}_{i|\phi})$.*

Assumption 6.3. *For any two predicates ϕ and ϕ', and the input value space partitions $\mathbf{D}_{i|\phi}$ and $\mathbf{D}_{i|\phi'}$, such that $\mathbf{D}_{i|\phi'} \subseteq \mathbf{D}_{i|\phi}$, then $est\_wcet(c_i, \mathbf{D}_{i|\phi'}) \leq est\_wcet(c_i, \mathbf{D}_{i|\phi})$ should hold.*

Assumption 6.4. *For any two predicates ϕ and ϕ', and the input value space partitions $\mathbf{D}_{i|\phi}$ and $\mathbf{D}_{i|\phi'}$, such that $\mathbf{D}_{i|\phi'} \subseteq \mathbf{D}_{i|\phi}$, then $est\_bcet(c_i, \mathbf{D}_{i|\phi'}) \geq est\_bcet(c_i, \mathbf{D}_{i|\phi})$ should hold.*

Assumption 6.5. *For any input variable's value domain $\sqsubseteq_{i,j}$ 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, 6.3 and 6.4 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 analyzed with respect to a specific system and usage; however that prediction is only valid for that specific usage, and the usage for a software component can vary a lot. To predict the execution time with high accuracy of a complex component, components must be reanalyzed 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].

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

Reusable software components

The key to reuse is generality and context freeness. Often only parts of the component behaviour is used in a specific system and context. 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 a component specifically for one particular usage the component can be analyzed and predicted with high accuracy, but not always reused.

In order to support reuse and at the same time support accurate predictions, new parameterizable methods and frameworks are needed [PD96].

Components are augmented with information that can be used to accurately predict the WCET by parameterization of the prediction with respect to different 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 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 analyzed 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 potentially be orders of magnitude inaccurate compared to the usage dependent WCET.

The reusable software component concept

Lets 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 $\mathbf{D}_a$ and $\mathbf{D}_b$ represents the possible input value combinations. Analyzing these components considering the input value spaces $\mathbf{D}_a$ and $\mathbf{D}_b$ give the worst-case execution times $est\_wcet(c_a, \mathbf{D}_a) = 1200ms$ and $est\_wcet(c_b, \mathbf{D}_b) = 2200ms$.

Lets 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 ϕ and ϕ', and the possible input combinations are the input value space partitions $\mathbf{D}_{a|\phi}$ and $\mathbf{D}_{b|\phi'}$ respectively. The resulting usage dependent WCETs are $est\_wcet(c_a, \mathbf{D}_{a|\phi}) = 500ms$ and $est\_wcet(c_b, \mathbf{D}_{b|\phi'}) = 200ms$ as depicted in Figure 6.2.

The composite WCET of components c_a and c_b are $900ms$ for the usage $\mathbf{D}_{a|\phi}$ and $\mathbf{D}_{b|\phi'}$, as compared to the usage independent ($\mathbf{D}_a$ and $\mathbf{D}_b$) 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.

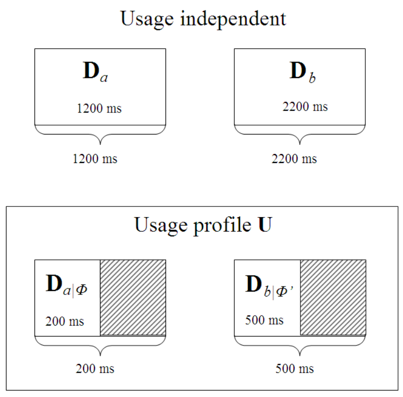


Figure 6.2: Context free vs. context dependent component behaviour.

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:

Component WCET analysis Analyzing 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

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.

Finding input value space partitions

The difference between $WCET_{i|\phi}$ and $BCET_{i|\phi}$ of an input value space partition $\mathbf{D}_{i|\phi}$ is an upper approximation of the difference between the WCET and BCET for a component c_i, which indicates the largest difference between two execution times within the input value space partition. This is an indicator of how close the execution times are for the different input value combinations within the same input value space partition $\mathbf{D}_{i|\phi}$. A challenge is to select input value space partitions $\mathbf{D}_{i|\phi}$ 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 $\mathbf{D}_{i|\phi}$ should be chosen such that every input combination tuple $\mathbf{d} \in \mathbf{D}_{i|\phi}$ should produce similar execution times, i.e., $est\_wcet(c_i, \mathbf{D}_{i|\phi})$ is close to $est\_bcet(c_i, \mathbf{D}_{i|\phi})$ and the input value space partitions should be as large as possible, i.e., encompass as many input combinations as possible.

When WCET analysis is performed with restrictions on the input parameters, it is desirable to not analyze *all* single value input combinations, but rather a set of input value space partitions $\mathbf{D}_{i|\phi} \subseteq \mathbf{D}_i$, such that:

$$\bigcup_{\forall \phi} \mathbf{D}_{i|\phi} = \mathbf{D}_i$$

Definition 6.9. *The accuracy of an input value space partition $\mathbf{D}_{i|\phi}$ 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.*

$$1 - \frac{WCET_{i|\phi} - BCET_{i|\phi}}{WCET_i - BCET_i} \tag{6.3}$$

The accuracy is an indication of the difference of the execution times for each input combination in the input value space partition $\mathbf{D}_{i|\phi}$. Less difference between $WCET_{i|\phi}$ and $BCET_{i|\phi}$ 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 $\mathbf{D}_{i|\phi}$ is low; and $\mathbf{D}_{i|\phi}$ 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 $est\_wcet(c_i, \mathbf{D}_{i|\phi})$ results in *one* single estimate. Also the estimation of

BCET, $est\_bcet(c_i, \mathbf{D}_{i|\phi})$ results in a single estimate. Thus, the only times known for an input value space partition $\mathbf{D}_{i|\phi}$ are $WCET_{i|\phi}$ and $BCETa|\phi$.

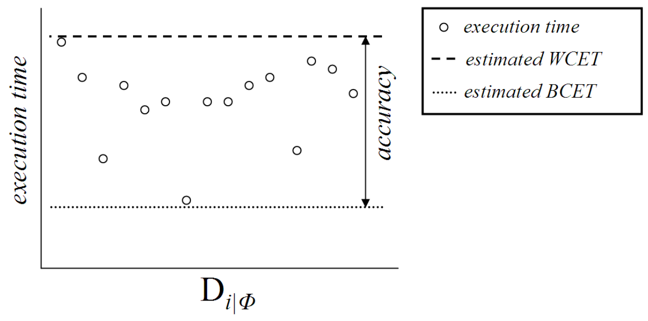


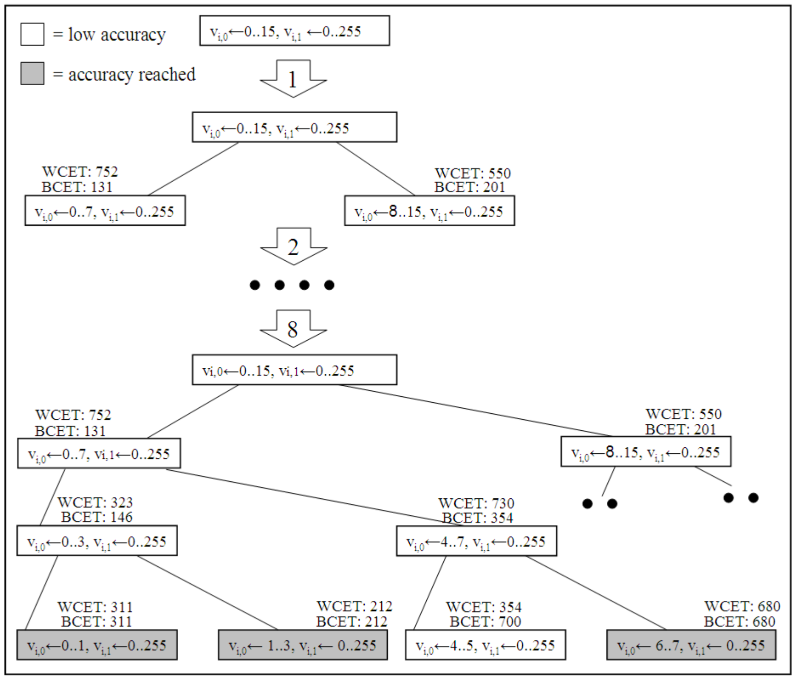
Figure 6.3: Input value space partition accuracy over the input space partition $\mathbf{D}_{i|\phi}$ and execution time.

The sum of the difference between $WCET_{i|\phi}$ and $BCET_{i|\phi}$ 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 $WCET_{i|\phi}$ and $BCET_{i|\phi}$ of each input 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 desired 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.*



= low accuracy
= accuracy reached
$v_{i,0}$←0..15, $v_{i,1}$ ←0..255
1
$v_{i,0}$←0..15, $v_{i,1}$←0..255
WCET: 752
BCET: 131
$v_{i,0}$←0..7, $v_{i,1}$←0..255
WCET: 550
BCET: 201
$v_{i,0}$←8..15, $v_{i,1}$←0..255
2
8
vi,0←0..15, vi,1←0..255
WCET: 752
BCET: 131
$v_{i,0}$←0..7, vi,1←0..255
WCET: 550
BCET: 201
$v_{i,0}$←8..15, $v_{i,1}$←0..255
WCET: 323
BCET: 146
$v_{i,0}$←0..3, $v_{i,1}$←0..255
WCET: 730
BCET: 354
$v_{i,0}$←4..7, $v_{i,1}$←0..255
WCET: 311
BCET: 311
$v_{i,0}$←0..1, $v_{i,1}$←0..255
WCET: 212
BCET: 212
$v_{i,0}$← 1..3, $v_{i,1}$← 0..255
WCET: 354
BCET: 700
$v_{i,0}$←4..5, $v_{i,1}$←0..255
WCET: 680
BCET: 680
$v_{i,0}$← 6..7, $v_{i,1}$← 0..255

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.

$$\sum_{\forall\phi}\left(\left(1-\frac{WCET_{i|\phi}-BCET_{i|\phi}}{WCET_i-BCET_i}\right)\cdot\frac{|\mathbf{D}_{i|\phi}|}{|\mathbf{D}_i|}\right) \tag{6.4}$$

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

1. All input value space partitions are single valued, i.e., $|\mathbf{D}_{i|\phi}| = 1$.
2. A pre-defined accuracy is achieved.
3. A pre-defined effort is reached.

6.2.2 Binary search algorithm description

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 desired 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 $\mathbf{D}_i$ (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].

In previous sections 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 heuristics for deriving input-sensitive WCET. The arguments to the algorithm, described in Figure 6.5, are the component c_i, the input variables' input value space partition $\mathbf{D}_i$, and the initial predicate ϕ. 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 $select\_var$ and one of the strategies $next$ or $last\_used$. The highest and lowest values of the input variable v are acquired through the $max\_value(v,\phi)$ and $min\_value(v,\phi)$ primitives. The highest and lowest values are for creating two new predicates ϕ' and ϕ''. The predicates are used for dividing the input value space partition into two disjunct input value space partitions $\mathbf{D}_{i|\phi'}$ and $\mathbf{D}_{i|\phi''}$. 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 partition, the input variables are selected in a predefined order determined by $select\_var(\mathbf{D}_{i|\phi}, strategy_i)$.

The new input value space partitions are added to the $wcet\_list$. The $wcet\_list$ contain the "leaves" in the binary tree and the $wcet\_list$ is sorted with, e.g., one of the following *sorting strategies*:

Worst accuracy the input value space partition with worst accuracy, i.e., largest difference $WCET-BCET$, is returned and removed from the $wcet\_list$.

Best accuracy the input value space partition with greatest accuracy, i.e., smallest difference $WCET - BCET$, is returned and removed from the $wcet\_list$.

Highest WCET the input value space partition with highest WCET is returned and removed from the $wcet\_list$.

Last used the input value space partition latest added is returned and removed from the $wcet\_list$.

As the $wcet\_list$ is populated, several input value space partitions are eligible for further division. The next input value space partition is chosen depending on the sorting strategy of the $wcet\_list$. An input value space partition $\mathbf{D}_{i|\phi}$ is chosen with $wcet\_less.pop(sort\_strategy)$ such that the first element is returned and removed from the $wcet\_list$ 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 ϕ.

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 $wcet\_list.pop(sort\_strategy)$ will return nil. An input variable is chosen and used for creating new predicates. 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).

```
$BEGIN\ find\_value\_space\_partitions(c_i, \mathbf{D}_i, \phi, strategy_i, max\_effort)$
  $\mathbf{D}_{i|\phi} \leftarrow new\_D(\mathbf{D}_i, \phi)$;
  $effort = 0$;
  $v \leftarrow select\_var(\mathbf{D}_{i|\phi}, strategy_i)$;
  WHILE $v \neq nil \wedge effort < max\_effort$ DO
     $l \leftarrow min\_value(v, \phi)$;
     $u \leftarrow max\_value(v, \phi)$;
     $\phi' \leftarrow new\_vc(\phi, v,'' l \leq v < (l + (u - l)/2)'')$;
     $\phi'' \leftarrow new\_vc(\phi, v,'' (l + ((u - l)/2) \leq v < u'')$;
     $\mathbf{D}_{i|\phi'} \leftarrow new\_D(\mathbf{D}_i, \phi')$;
     $\mathbf{D}_{i|\phi''} \leftarrow new\_D(\mathbf{D}_i, \phi'')$;
     $WCET_{i|\phi'} \leftarrow est\_wcet(c_i, \mathbf{D}_{i|\phi'})$;
     $WCET_{i|\phi''} \leftarrow est\_wcet(c_i, \mathbf{D}_{i|\phi''})$;
     $BCET_{i|\phi'} \leftarrow est\_bcet(c_i, \mathbf{D}_{i|\phi'})$;
     $BCET_{i|\phi''} \leftarrow est\_bcet(c_i, \mathbf{D}_{i|\phi''})$;
     $wcet\_list.insert(WCET_{i|\phi'}, BCET_{i|\phi'}, \mathbf{D}_{i|\phi'})$;
     $wcet\_list.insert(WCET_{i|\phi''}, BCET_{i|\phi''}, \mathbf{D}_{i|\phi''})$;
     $\mathbf{D}_{i|\phi} \leftarrow wcet\_list.pop(sort\_strategy)$;
     /*$\phi$ is the predicate of $\mathbf{D}_{i|\phi}$*/
     $v \leftarrow select\_var(\mathbf{D}_{i|\phi}, strategy_i)$;
     $increase\ effort$;
  END WHILE
  RETURN $wcet\_list$;
END find_value_space_partitions
```

Figure 6.5: Finding input value space partitions with binary search algorithm.

6.2.3 Approaching parametric WCET

Parametric WCET is an approach to execution time analysis where the WCET is expressed as a mathematical expression parameterized with inputs and possibly other context parameters. Current problems with fully automatic parametric WCET relates to its very high complexity in terms of state space. Several simpler approaches only parameterize 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 categorizes WCETs with respect to input values. The granularity depends on the effort and time put into finding value space partitions and the creation of WCET contracts. The 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 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 parameterizable WCET may be achieved; while running the algorithm longer achieves higher accuracy.

All parameterizable analyses must be parameterized with some information, i.e., the parameters. In our case we define a contract to be parameterized with a usage profile. A usage profile consists of an input value space partition defining the value combinations of the specific usage, a probability mass function with occurrences of value combinations for all inputs defined in the contract and a priority threshold for ignoring low probability WCETs.

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 modeling 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 ϕ^U to define a usage. For the component c_i with the input space partition $\mathbf{D}_i$ a usage is denoted $\mathbf{D}_{i|\phi^U}$. Thus, the input value space partition $\mathbf{D}_{i|\phi^U}$ describes the possible values of a certain usage.

Definition 6.11. *A usage profile* $\mathbf{U} = \langle \phi^U, \mathcal{P}, pt \rangle$ *is a predicate* ϕ^U *connected with a probability mass function* $\mathcal{P}$ *and* $0 \leq pt < 1$ *is a given priority threshold to ignore low probability WCETs.*

The probability mass function for the occurrence of the input combination tuples $\mathbf{d} \in \mathbf{D}_{i|\phi^U}$, is outlined in Equation 6.5, where $Pr(\mathbf{d})$ is the probability of the occurrence of the input value combination $\mathbf{d}$. The sum of the probabilities of all input combinations in the usage profile equals to 1, as outlined by Equation 6.6. Consider the example in (Figure 6.6), where the probability mass function of a usage profile $\mathbf{D}_{i|\phi^U}$ is depicted as the light gray area. The dark gray area is the probability of a subset $\mathbf{D}_{i|\phi} \subseteq \mathbf{D}_{i|\phi^U}$.

The probability for an input value space partition $\mathbf{D}_{i|\phi}$ is given by $\Pi(\mathbf{D}_{i|\phi})$, and is the probability of the occurrence of all input combination tuples in the intersection between the input value space partition and the usage profile, as outlined in Equation 6.7.

$$\mathcal{P}(d) = \begin{cases} Pr(\mathbf{d}) & \text{if } \mathbf{d} \in \mathbf{D}_{i|\phi^U}, \\ 0 & \text{if } \mathbf{d} \notin \mathbf{D}_{i|\phi^U}. \end{cases} \tag{6.5}$$

$$1 = \sum_{d \in D_{i|\phi^U}} \mathcal{P}(\mathbf{d}) \tag{6.6}$$

$$\Pi(\mathbf{D}_{i|\phi}) = \sum_{d \in (D_{i|\phi} \cap D_{i|\phi^U})} \mathcal{P}(\mathbf{d}) \tag{6.7}$$

The sum of the probabilities of all input combinations in a value partition $\mathbf{D}_{i|\phi}$ is the probability of that input value space partition.

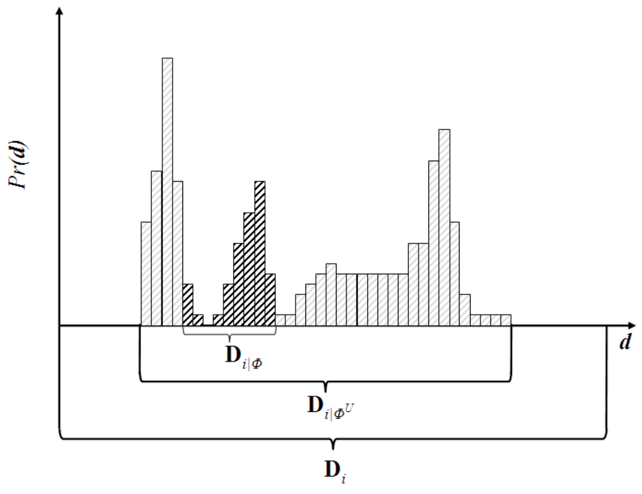


Figure 6.6: Probability mass function of a usage profile $\mathbf{D}_{i|\phi^U}$. The dark gray area is the probability of a subset $\mathbf{D}_{i|\phi} \subset \mathbf{D}_{i|\phi^U}$.

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^{64} possible input combinations, and equally many corresponding WCETs.

To lower the number of input value space partitions, two input value space partitions $\mathbf{D}_{i|\phi'}$ and $\mathbf{D}_{i|\phi}$ are merged if their associated WCETs are the same, i.e., $\mathbf{D}_{i|\phi'} \cup \mathbf{D}_{i|\phi} | WCET_{i|\phi'} = WCET_{i|\phi}$. Actually two input value space partitions can be merged if their corresponding WCETs are close enough to maintain the desired accuracy, such that $\mathbf{D}_{i|\phi\phi'} = \mathbf{D}_{i|\phi'} \cup \mathbf{D}_{i|\phi}$ and $\max(WCET_{i|\phi'}, WCET_{i|\phi}) - \min(BCET_{i|\phi'}, BCET_{i|\phi}) \leq accuracy$.

A component contract is used for acquiring a WCET with respect to a usage. The contract is parameterized with the *usage profile*. The parameterization match the input combinations of the usage $\mathbf{D}_{i|\phi^U}$ with the input combinations of all input value space partitions $\mathbf{D}_{i|\phi}$. All input value space partitions $\mathbf{D}_{i|\phi} \cap \mathbf{D}_{i|\phi^U} \neq \emptyset$ are referred to as *active input value space partitions*, i.e., all input value space partitions $\mathbf{D}_{i|\phi}$ that are *active* and their respective WCETs are eligible for the usage dependent component WCET. The usage dependent $WCET_U$ is defined in Equation 6.8.

$$WCET_U = \max_{\forall_r \left(\mathbf{D}_{i|\phi} \cap \mathbf{D}_{i|\phi^U} \neq \emptyset\right)} \left(WCET_{i|\phi}\right) \tag{6.8}$$

A component contract is a function of a usage $\mathbf{U}$ that results in a usage dependent WCET (Equation 6.9).

$$f_U : \mathbf{D}_{i|\phi^U} \rightarrow \max_{\forall_r \left(\mathbf{D}_{i|\phi} \cap \mathbf{D}_{i|\phi^U} \neq \emptyset\right)} \left(WCET_{i|\phi}\right) \tag{6.9}$$

The usage dependent WCET together with the probability $\Pi(\mathbf{D}_{i|\phi})$ of a WCET allows for a contract user to disregard WCETs with low probabilities in a specific usage. 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 6.10. Note that the priority threshold may be set to 0 for critical systems, thus considering all WCETs.

$$f_U : \mathbf{D}_{i|\phi^U}, pt \rightarrow \max_{\forall_r \left(\left(\mathbf{D}_{i|\phi} \cap \mathbf{D}_{i|\phi^U} \neq \emptyset\right) \wedge \left(\Pi(\mathbf{D}_{i|\phi}) \geq pt\right)\right)} \left(WCET_{i|\phi}\right) \tag{6.10}$$

Contract composition

Each input value space partition can be associated with a set of possible outputs. 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.

Abstract interpretation can be used to make a safe over-approximation of limitations on outputs given limitations on inputs by analyzing possible values of the output variables, however in most current tools, all probabilities are lost. SWEET [GESL06] is one tool that can produce restrictions on the output with respect to the input, however, without probabilities.

6.2.4 Algorithm complexity

The binary search algorithm will have a worst-case behaviour of:

$$O(2 \cdot |\mathbf{D}_i| - 1) = O(2 \cdot |\mathbf{D}_i|) = O(|\mathbf{D}_i|)$$

This is because the algorithm will analyze all leaves in the tree resulting in $|\mathbf{D}_i|$ plus all the branches in the tree $|\mathbf{D}_i| - 1$, resulting in a complexity of $O(|\mathbf{D}_i| + |\mathbf{D}_i| - 1)$, resulting in a worst case behaviour of $O(|\mathbf{D}_i|)$.

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.

Knowing the input value combinations that result in the worst-case behaviours 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 optimizing 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. 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.

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.

Basically we search for an input value space partition $\mathbf{D}_{i|\phi}$ such that:

$$|\mathbf{D}_{i|\phi}| = 1 \wedge \left(WCET_{i|\phi} \geq WCET_{i|\phi'}\right) \wedge \phi \neq \phi'$$

6.3.1 Algorithm description

The arguments to the algorithm described in Figure 6.7 are the behaviour of the component under analysis c_i, an input value space partition $\mathbf{D}_i$ and the initial predicate ϕ. 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 ϕ is set to divide $\mathbf{D}_i$ 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 removing 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 partition, the input variables are selected in a predefined order determined by $select\_var(\mathbf{D}_{i|\phi}, strategy_i)$. 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 $|\mathbf{D}_{i|\phi}| = 1$ has a larger WCET than all input value space partitions with the size $|\mathbf{D}_{i|\phi}| > 1$.

```
BEGIN find_WCET_input_value($c_i, \mathbf{D}_i$), $\phi$
  $\mathbf{D}_{i|\phi} \leftarrow new\_D(\mathbf{D}_i, \phi)$;
  $v \leftarrow select\_var(\mathbf{D}_{i|\phi}, '' last\_used'')$;
  WHILE $v \neq nil$ DO
    $l \leftarrow min\_var(v, \phi)$;
    $u \leftarrow max\_var(v, \phi)$;
    $\mathbf{vc}_p \leftarrow new\_vc(\phi, v, '' l \leq v < (l + (u - l)/2)'')$;
    $\mathbf{vc}_q \leftarrow new\_vc(\phi, v, '' (l + ((u - l)/2) \leq v < u'')$;
    $\mathbf{D}_{i|\phi'} \leftarrow new\_D(\mathbf{D}_i, \mathbf{vc}_p)$;
    $\mathbf{D}_{i|\phi''} \leftarrow new\_D(\mathbf{D}_i, \mathbf{vc}_q)$;
    $WCET_{i|\phi'} \leftarrow est\_wcet(c_i, \mathbf{D}_{i|\phi'})$;
    $WCET_{i|\phi''} \leftarrow est\_wcet(c_i, \mathbf{D}_{i|\phi''})$;
    $wcet\_list.insert(WCET_{i|\phi'}, \mathbf{D}_{i|\phi'})$;
    $wcet\_list.insert(WCET_{i|\phi''}, \mathbf{D}_{i|\phi''})$;
    $\mathbf{D}_{i|\phi} \leftarrow wcet\_list.pop(``largestwcet'')$;
    /*$\phi$ is the predicate of $\mathbf{D}_{i|\phi}$*/
    $v \leftarrow select\_var(\mathbf{D}_{i|\phi}, '' last\_used'')$;
  END WHILE
  RETURN $\mathbf{D}_{i|\phi}$;
ENDfind_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 $\langle v_{i,0} \leftarrow 0..15, v_{i,1} \leftarrow 0, v_{i,2} \leftarrow 0..1\rangle$ which corresponds to $16 \cdot 1 \cdot 2 = 32$ 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 $\langle v_{i,0} \leftarrow 0..7, v_{i,1} \leftarrow 0, v_{i,2} \leftarrow 0..1\rangle$ partition gives the largest WCET estimate 72 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 $v_{i,1} \leftarrow 0$ and $v_{i,2} \leftarrow 0..1$ has been found. Since $v_{i,1}$ now only can hold a single value, the next input variable selected is $v_{i,2}$. The 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 70. There are no other partitions in the $wcet\_list$ with a larger WCET estimate, so the iteration stops and the partition is returned.

Note that the gray boxes in Figure 6.8 has a different meaning compared to Figure 6.4.

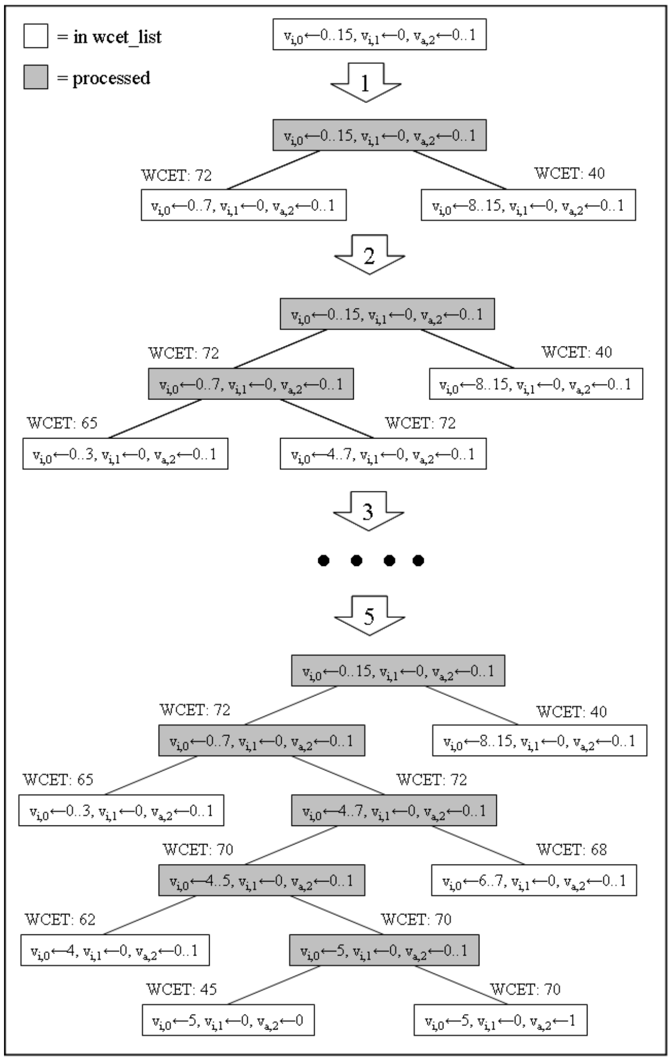

= in wcet_list
= processed
$v_{i,0} \leftarrow 0..15, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
1
$v_{i,0} \leftarrow 0..15, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 72
WCET: 40
$v_{i,0} \leftarrow 0..7, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
$v_{i,0} \leftarrow 8..15, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
2
$v_{i,0} \leftarrow 0..15, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 72
WCET: 40
$v_{i,0} \leftarrow 0..7, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
$v_{i,0} \leftarrow 8..15, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 65
WCET: 72
$v_{i,0} \leftarrow 0..3, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
$v_{i,0} \leftarrow 4..7, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
3
5
$v_{i,0} \leftarrow 0..15, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 72
WCET: 40
$v_{i,0} \leftarrow 0..7, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
$v_{i,0} \leftarrow 8..15, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 65
WCET: 72
$v_{i,0} \leftarrow 0..3, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
$v_{i,0} \leftarrow 4..7, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 70
WCET: 68
$v_{i,0} \leftarrow 4..5, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
$v_{i,0} \leftarrow 6..7, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 62
WCET: 70
$v_{i,0} \leftarrow 4, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
$v_{i,0} \leftarrow 5, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0..1$
WCET: 45
WCET: 70
$v_{i,0} \leftarrow 5, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 0$
$v_{i,0} \leftarrow 5, v_{i,1} \leftarrow 0, v_{a,2} \leftarrow 1$

Figure 6.8: Example of basic algorithm execution.

6.3.3 Algorithm complexity and back-tracking

Our algorithm will have a best-case behaviour of $O(2 \cdot log\,|\mathbf{D}_i|) = O(log\,|\mathbf{D}_i|)$ where $\mathbf{D}_i$ 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 $wcet\_list$ 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 $O(|\mathbf{D}_i| + 2 \cdot log\,|\mathbf{D}_i|) = O(|\mathbf{D}_i|)$.

To reduce the amount of back-tracking, it is important that the WCET calculation is tight, i.e., $est\_wcet(c_i, \mathbf{D}_{i|\phi})$ is close to $real\_wcet(c_i, \mathbf{D}_{i|\phi})$ even though $|\mathbf{D}_{i|\phi}|$ 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 $|\mathbf{D}_i|$ is large. This section outlines some approaches for faster algorithm termination.

6.4.1 Slicing the input space

A component may have inputs where different inputs may affect the execution time, and inputs that do not affect the 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 for many programs 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.

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.

```
// Inputs that may be given input values
int a[100];
int min=0, max=0, index=0, val=0, sum=0;
// The code to analyse
int sum_selected_irray_elements(void) {
   int tmp = 0;
   int i = min;
   sum = 0;
   while(i <= max) {
      tmp = a[index];
      sum = sum + tmp;
      a[index] = val;
      index++;
      i++;
   }
return sum;
}
```

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

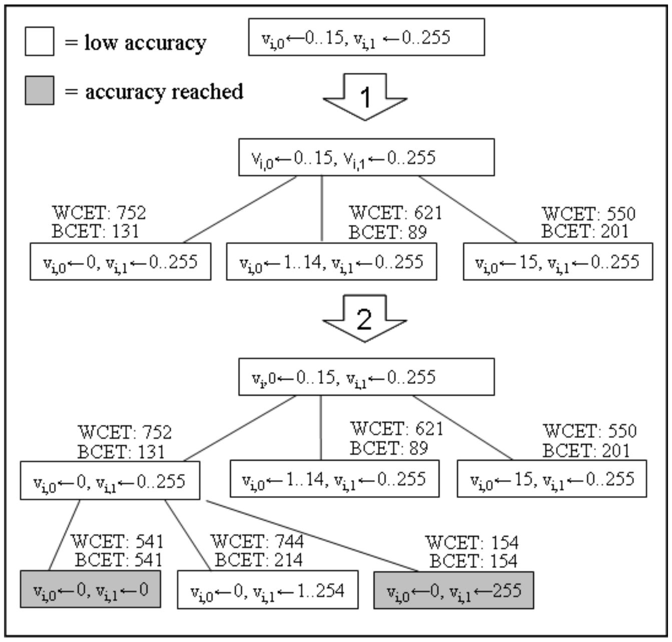

= low accuracy
= accuracy reached
v_{i,0}←0..15, v_{i,1}←0..255
1
v_{i,0}←0..15, v_{i,1}←0..255
WCET: 752 BCET: 131
WCET: 621 BCET: 89
WCET: 550 BCET: 201
v_{i,0}←0, v_{i,1}←0..255
v_{i,0}←1..14, v_{i,1}←0..255
v_{i,0}←15, v_{i,1}←0..255
2
v_{i,0}←0..15, v_{i,1}←0..255
WCET: 752 BCET: 131
WCET: 621 BCET: 89
WCET: 550 BCET: 201
v_{i,0}←0, v_{i,1}←0..255
v_{i,0}←1..14, v_{i,1}←0..255
v_{i,0}←15, v_{i,1}←0..255
WCET: 541 BCET: 541
WCET: 744 BCET: 214
WCET: 154 BCET: 154
v_{i,0}←0, v_{i,1}←0
v_{i,0}←0, v_{i,1}←1..254
v_{i,0}←0, v_{i,1}←255

Figure 6.10: Example of extreme value heuristic.

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 $\langle v_{i,0} \leftarrow 0, v_{i,1} \leftarrow 0..255\rangle$ 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 $\langle v_{i,0} \leftarrow 0, v_{i,1} \leftarrow 255\rangle$ 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 she/he believes hot spots are to be found. For example, in Figure 6.8 the user might believe that the worst-case is when $v_{i,2} = 1$ and $0 \leq v_{i,0} \leq 6$. Then the analysis can be started with an initial set of partitions according to the user's assumptions, e.g., $\langle v_{i,0} \leftarrow 0..6, v_{i,1} \leftarrow 0, v_{i,2} \leftarrow 1\rangle$, $\langle v_{i,0} \leftarrow 0..6, v_{i,1} \leftarrow 0, v_{i,2} \leftarrow 0\rangle$, and $\langle v_{i,0} \leftarrow 7..16, v_{i,1} \leftarrow 0, v_{i,2} \leftarrow 0..1\rangle$, where the first partition corresponds to the user provided assumption.

6.5 ACC example - input-sensitive WCET analysis

We revisit the example outlined in Section 5.5, and discuss the component c_{speed} (Speed Limit) in detail, and explore the resulting input value space partitions. c_{speed} has 3 variables ACC Max Speed (AMS), Road Signs Enabled (RSE) and Road Sign Speed (RSS). The input value space partition $\mathbf{D}_{speed}$ has a size $|\mathbf{D}_{speed}| = 301 \cdot 2 \cdot 301 = 181202$ 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**), Last used with eXtreme value heuristic (LX) and Last used (L). The strategy Last used is described in Section 6.1.2 and eXtreme value heuristic 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.

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_{speed}, that are used for the parameterization 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.

$\mathbf{D}$	WCET	BCET	AMS	RSE	RSS
$\mathbf{D}_{speed}$	304	105	0..300	0..1	0..300
$\mathbf{D}_{speed,0}$	304	304	0..300	1	0..300
$\mathbf{D}_{speed,1}$	284	284	0..300	0	1..300
$\mathbf{D}_{speed,2}$	105	105	0..300	0	0

Table 6.1: c_{speed} resulting input value space partitions.

Predicate
$RSE = 1 \rightarrow WCET = 304$
$RSE = 0 \wedge RSE > 0 \rightarrow WCET = 284$
$RSE = 0 \wedge RSE = 0 \rightarrow WCET = 105$

Table 6.2: c_{speed} resulting contract f_{speed}.

6.6 Summary

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 components 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 parameterizable 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 analyzing 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.

We partly rely on the presence of BCET analysis which is not present in most of today's WCET analysis tools. However, we note that we actually want to know the best-case execution path with the worst-case hardware effects. The reasons for this is that we want to have a single value for input value space partitions with $|\mathbf{D}_{i|\phi}| = 1$. By using a "real" BCET (best-case path and best-case hardware effects) a single valued input value space partition will get two values. There should be only one execution path with a single valued input value partition, but the hardware may behave differently in the best and worst cases.

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.

Space is big. Really big. You just won't believe how vastly hugely mind-bogglingly big it is. I mean you may think it's a long way down the road to the chemist, but that's just peanuts to space.

-Hitchhiker's guide to the galaxy

Chapter 7

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 $\approx 50\%$ and $\approx 30\%$ respectively, compared to allocating each component to one task.

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.

Lets 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 τ_1, and c_b in one task τ_2.
2. c_a and c_b in the same task τ_1.
3. c_b and c_a in the same task τ_1 (reverse order compared to allocation 2).

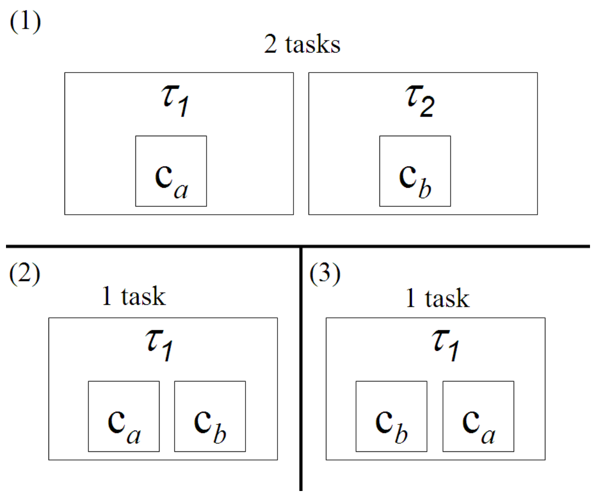


Figure 7.1: Possible allocations of 2 components to tasks.

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. Γ^T *denotes a component transaction that is time-triggered. Components and component transactions are defined in Section 5.4.*

Definition 7.2. Γ^E *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 co-allocated. It is described as a set of component pairs $\langle (c_i, c_j), (c_k, c_l) \rangle$ that define what components may not be allocated to the same task. There may be memory protection requirements or other legitimate engineering reasons to avoid allocating certain combinations of components; for example, if

[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.

a component has a highly uncertain WCET. The predicate $Iso(i, j)$ defines that components c_i and c_j has an isolation requirement, and should not be co-allocated.

Definition 7.3. $Iso(c_i, c_j)$ *defines an isolation requirement between two components* c_i *and* c_j*. The isolation requirement implies that the components* c_i *and* c_j *may not be allocated to the same task.*

Components may have precedence relations in terms of triggering. If a component is triggered by a preceding component, they are said to have a precedence relation, component c_i precedes component c_j means that component c_j is triggered by component c_i. $prcd(c_i, c_j, \Gamma_k)$ defines that component c_j is preceded by component c_i, i.e., component c_j is triggered by component c_i and both component are part of transaction Γ_k.

Definition 7.4. $prcd(c_i, c_j, \Gamma_k)$ *defines that component* c_i *triggers component* c_j *and both components* $c_i, c_j \in \Gamma_k$ *are part of transaction* Γ_k.

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 allocated 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. Transactions 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 allocations of components to tasks. The properties calculated with the framework are used for optimization algorithms to find feasible allocations that fulfill given real-time requirements and minimizes memory consumption as well as 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. 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_0,...,c_{n-1}, 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:

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

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

where ρ is the context switch time and β is the size of 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 periods 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

$$\neg\exists\tau_n(c_i \in \mathbf{Z}_n \wedge c_j \in \mathbf{Z}_n \wedge Iso(i,j)) \tag{7.3}$$

where there must not exist any task τ_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 $\mathbf{Z}_n$ is the set of allocated components in task τ_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 $c_i \in \mathbf{Z}_n$ where c_i is the first element in $\mathbf{Z}_n$. Therefore it is important to consider which components that 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 intersecting transactions is described in Equations 7.4 and 7.5.

$$\neg\exists\tau_n\Big(\big(c_i \in \Gamma_j^E \wedge c_m \in \Gamma_k^E \wedge c_i \in \mathbf{Z}_n \wedge c_m \in \mathbf{Z}_n\big) \wedge \big(prcd(c_m, c_i, \Gamma_k^E) \vee prcd(c_m, c_i, \Gamma_j^E)\big)\Big), j \neq k \tag{7.4}$$

If a component c_i is allocated to a task τ_n, and the component is part of an event triggered transaction Γ_j^E, and a component c_m is also allocated to the task τ_n and is part of another event triggered transaction Γ_k^E, 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. $\mathbf{Z}_n$ is the set of allocated components in task τ_n, as described in Section 5.4.

$$\neg\exists\tau_n\Big(\big(c_i \in \Gamma_j^T \wedge c_m \in \Gamma_k^E \wedge c_i \in \mathbf{Z}_n \wedge c_m \in \mathbf{Z}_n\big) \wedge \big(c_m \in \Gamma_k^E \wedge prcd(c_m, c_i, \Gamma_k^E)\big)\Big), j \neq k \tag{7.5}$$

If a component c_i is allocated to a task τ_n and the component is part of an event triggered transaction Γ_k^E, and a component c_m allocated to the task τ_n and the component c_m is part of a time triggered transaction Γ_k^T, then no preceding components that are part of the event triggered transaction Γ_j^E shall be allocated to the task τ_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.

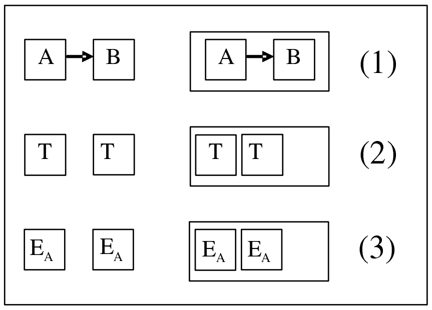


Figure 7.2: Component to task allocation considering triggers.

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.

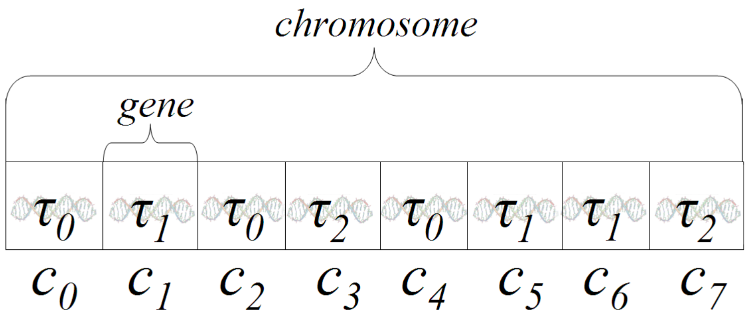

chromosome
gene
τ_0 τ_1 τ_0 τ_2 τ_0 τ_1 τ_1 τ_2
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7

Figure 7.3: The genetic algorithm view of the component to task allocation; a system with eight components $(c_0 - c_7)$, allocated to three tasks $(\tau_0 - \tau_2)$.

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.

$$Fitness = \Big((I+IT+Tr+Sc)F + \big(\frac{n}{s_A} + \sum_{\forall i(\tau_i \in A)} \frac{\rho \cdot n}{T_i}\big)O\Big) \cdot (I \cdot IT \cdot Tr \cdot Sc + 1) \tag{7.6}$$

where the fitness is the sum of all feasibility values multiplied with a factor F, added with the inverted memory usage and performance overhead, multiplied with a factor O, and $F >> O$. 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 - allocating 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.

A search heuristics searches for a *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 that all components are allocated to one task (independent of feasibility).

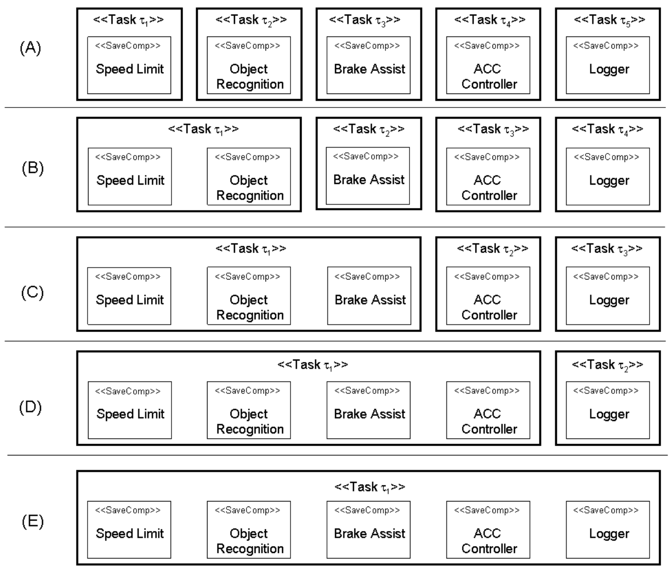


Figure 7.4: 5 feasible component to task allocations of the ACC example.

Allocation E is optimal with respect to CPU overhead and memory consumption since there exist no other allocation for the given components that can yield a lower CPU overhead or memory consumption. The feasibility of the allocation is validated for the components given in Section 5.5 and the WCETs given by the usage dependent WCET analysis for the usage profiles U_1 and U_2, as presented in Section 5.5.1. In Table 7.1 we reintroduce the components and properties and we discuss the evaluation of the allocation with respect to the feasibility requirements and performance properties for the different WCETs.

Component	**Name**	$\langle \mathbf{S_i}, \mathbf{Q_i}, \mathbf{m_i} \rangle$
Speed limit	c_{speed}	$\langle$ 50Hz, 20, 1024$\rangle$
Object recognition	c_{obj}	$\langle c_{speed}, 20, 512\rangle$
Brake assist	c_{break}	$\langle c_{obj}, 20, 512\rangle$
Logger HMI Output	c_{log}	$\langle$ 10Hz, 100, 2048$\rangle$
ACC Controller	c_{acc}	$\langle c_{obj}, 20, 2048\rangle$

Table 7.1: The five components *Speed Limit*, *Object Recognition*, *Brake Assist*, *Logger HMI Output* 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 exits 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 fulfill 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.

Γ_i	N_i	dc_i
Γ_0	$c_{speed} \rightarrow c_{obj} \rightarrow c_{break}$	800
Γ_1	$c_{speed} \rightarrow c_{obj} \rightarrow c_{acc}$	2000
Γ_2	c_{log}	10000

Table 7.2: The three transactions in the ACC.

7.6.2 Schedulability

In Table 7.3 we present the tasks' WCET for the usage independent (ui) WCET as well as for the usage profiles U_1 and U_2. The transactions Γ_0^T, Γ_1^T and Γ_2^T must also be considered. The transactions are presented in Table 7.2. c_{break} and c_{acc} are executed after c_{obj}, which is executed after c_{speed}.

We calculate the response time for allocation A, where task τ_4 (last in transaction Γ_2) has a response time of 10000 for (ui), which is equal to the stipulated deadline dc_2. The response time for task τ_3 (last in transaction Γ_1) is calculated to 1920 for (ui) and is lower than the stipulated deadline dc_1. Finally we calculate the response time for task τ_2 (last in transaction Γ_0), which has a response time of 679 for (ui), which is lower than the stipulated deadline dc_0. On the other hand, if we consider allocation D, where task τ_0 needs to fulfill the deadlines of both transactions Γ_0 and Γ_1, we see that only the WCET for U_2 fulfills this requirement. The other WCETs are greater than dc_0. The transactions Γ_i and deadlines dc_i are presented in Table 7.2.

In Table 7.4 the resource usage for each allocation is presented. The context switch ρ_A is 22 and the task control block size β is 300. The difference in CPU-overhead p_A and memory usage s_A between, e.g., allocations A and E is large, allowing for large benefits in finding *good* allocations.

Allocation	Task	Period	WCET		
			ui	$\mathbf{U_1}$	$\mathbf{U_2}$
A	τ_0	2000	304	284	105
	τ_1	2000	201	201	120
	τ_2	2000	174	91	88
	τ_3	2000	1241	1241	769
	τ_4	10000	400	303	303
B	τ_0	2000	505	485	225
	τ_1	2000	174	91	88
	τ_2	2000	1241	1241	769
	τ_3	10000	400	303	303
C	τ_0	2000	679	576	313
	τ_1	2000	1241	1241	769
	τ_2	10000	400	303	303
D	τ_0	2000	1920	1817	1082
	τ_1	10000	400	303	303
E	τ_0	2000	2320	2120	1385

Table 7.3: Period time and WCET for all component to task allocations in Figure 7.4.

Allocation	$\mathbf{p_A}$	$\mathbf{s_A}$
A	4.6%	7644
B	3.5%	6832
C	2.4%	6020
D	1.3%	4696
E	1.1%	2348

Table 7.4: CPU overhead and memory usage with respect to the different allocations presented in Figure 7.4.

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 in terms of memory usage and CPU-overhead. The best feasible allocation for the usage independent (ui) WCET prediction is C, for usage profile U_1 the best feasible allocation is also C, and, for usage profile U_2, the best allocation is E. Refer to Table 7.3 and Figure 7.4 for the different allocations, and Table 7.4 for the resource usage for respective allocation.

In this example we have illustrated different allocations and we have shown that tighter WCETs can increase the possibilities for finding allocations with lower memory consumption and lower CPU overhead.

7.7 Summary

Resource efficiency is important in ERTS, both with respect to performance and memory. Schedulability, considering resource efficiency, has gained much focus, however the allocation from components to tasks has gained very little attention. 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 iteratively 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 $\approx 30\%$ reduced memory size and $\approx 50\%$ 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.

We demand rigidly defined areas of doubt and uncertainty!

-Hitchhiker's guide to the galaxy

Chapter 8

Empirical results

In this chapter we present the evaluations and results of the technical contributions presented in Chapters 6 and 7. Section 8.1 presents the evaluations of the methods outlined in Chapter6 while Section 8.2 presents the evaluations of the component to task allocation framework outlined in Chapter 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.

8.1.1 Prototype tool

Many of the methods presented in Chapter 6 have been implemented in a prototype tool. The prototype tool also performs *program slicing* for deriving the input variables whose different values may cause the program execution time to vary. This allows the input data search space to be reduced when deriving WCET input values for any type of measurement-based or static WCET analysis. It is also possible to provide a pre-defined set of inputs, overriding the slicing. Furthermore, the user may have to provide limitations for inputs that the tool can not determine automatically.

Tool overview

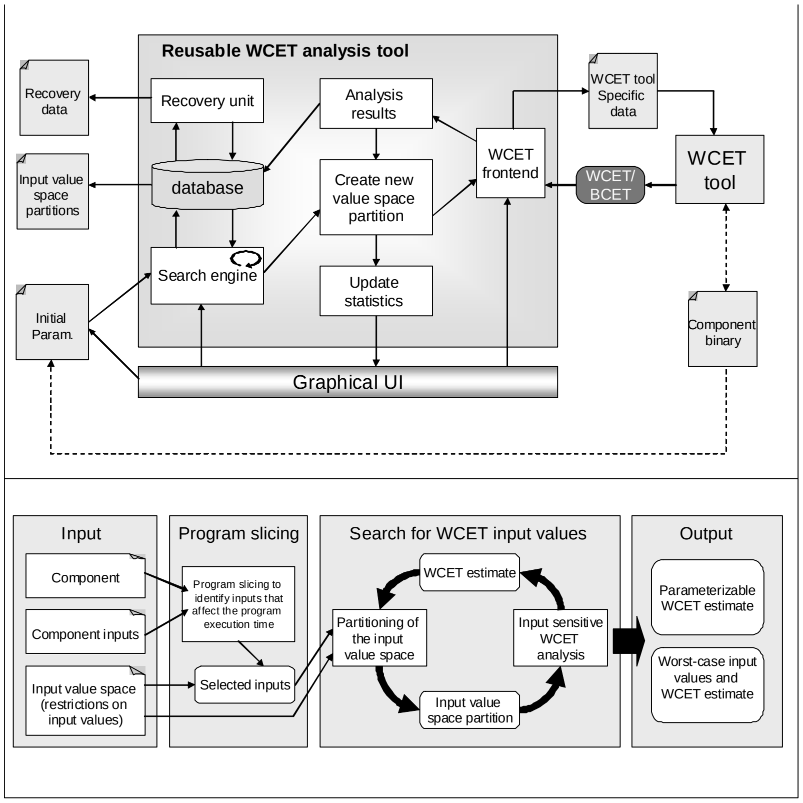

Reusable WCET analysis tool
Recovery data
Recovery unit
Analysis results
WCET tool Specific data
Input value space partitions
database
Create new value space partition
WCET frontend
WCET/ BCET
WCET tool
Search engine
Update statistics
Initial Param.
Component binary
Graphical UI
Input
Component
Component inputs
Input value space (restrictions on input values)
Program slicing
Program slicing to identify inputs that affect the program execution time
Selected inputs
Search for WCET input values
WCET estimate
Partitioning of the input value space
Input sensitive WCET analysis
Input value space partition
Output
Parameterizable WCET estimate
Worst-case input values and WCET estimate

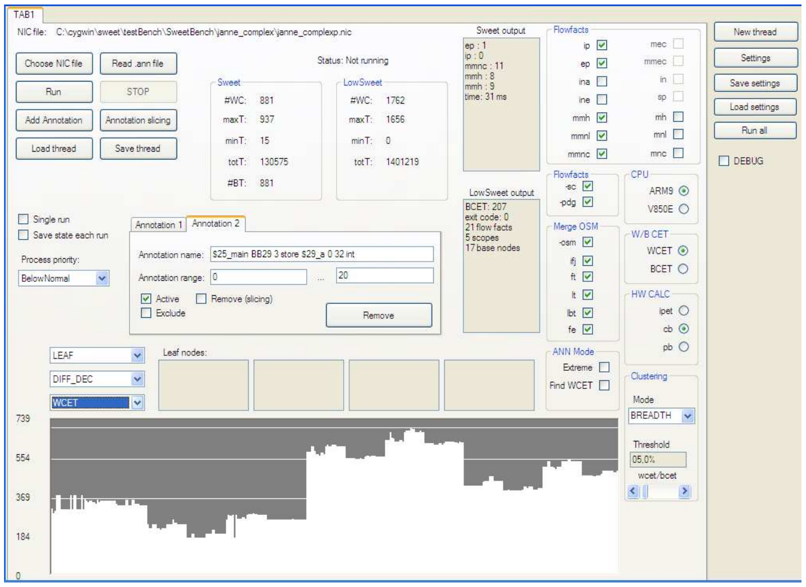
Figure 8.1: Tool architecture and a logical view of the information flow.

The tool architecture and logical work-flow are presented in Figure 8.1. A component with a specification of the inputs and an initial value space are input to the tool. Slicing is used for removing inputs that do not affect the execution time. A value space is partitioned and a database in the form of a kd-tree [Ben75] is used for storing information about each evaluated value space partition. A value space partition is chosen with the binary tree search heuristics; from the chosen value space partition two new partitions are created according to one of the heuristics presented in Chapter 6, using a static

WCET analysis tool for deriving the WCET and (if required) BCET for each value space partition. The new value space partitions and their corresponding WCET and BCET are stored in the database. The results from the tool are (i) a parameterizable WCET to be reused with the component, and, (ii) the worst-case input combination and its respective WCET.

A WCET tool front-end provides the possibility to change WCET tool. The Graphical User Interface (GUI) shown in Figure 8.2 provides the user with options for different tool settings, and graphically presents the WCETs and BCETs for the derived value space partitions. Statistics other than execution time are also logged and presented such as, e.g., flow analysis time, low-level analysis time and number of flow facts. The analyses can be length and therefore the tool has a recovery unit storing the current state in the event of an unscheduled stop in the analysis.

From the GUI the user can start and stop the analysis, or, load a recovery state after an unscheduled stop; also WCET tool options and heuristics are chosen via the GUI. A graphical panel in the bottom of the tool window shows the progress of the tool, and the value space partitions.



NIC file: C:\cygwin\sweet\testBench\SweetBench\janne_complex\janne_complexp.nic

Status: Not running

Sweet
#WC: 881
maxT: 937
minT: 15
totT: 130575
#BT: 881

LowSweet
#WC: 1762
maxT: 1656
minT: 0
totT: 1401219

Sweet output
```
ep : 1
ip : 0
mmnc : 11
mmh : 8
mmh : 9
time: 31 ms
```

LowSweet output
```
BCET: 207
exit code: 0
21 flow facts
5 scopes
17 base nodes
```

☐ Single run
☐ Save state each run

Process priority: BelowNormal

Annotation name: $25_main BB29 3 store $29_a 0 32 int
Annotation range: 0 ... 20
✓ Active ☐ Remove (slicing)
☐ Exclude

LEAF
DIFF_DEC
WCET

Flowfacts: ip ✓, ep ✓, ina ☐, ine ☐, mmh ✓, mmnl ✓, mmnc ✓, mec ☐, mmec ☐, in ☐, sp ☐, mh ☐, mnl ☐, mnc ☐

Flowfacts: -sc ✓, -pdg ✓

Merge OSM: -osm ✓, ifj ✓, ft ✓, lt ✓, lbt ✓, fe ✓

ANN Mode: Extreme ☐, Find WCET ☐

CPU: [✓ ARM9] V850E

W/B CET: [✓ WCET] BCET

HW CALC: ipet [✓ cb] pb

Clustering
Mode: BREADTH
Threshold: 05,0%
wcet/bcet

☐ DEBUG

Figure 8.2: Graphical user interface of the evaluation tool.

The tool uses the WCET analysis tool SWEET (SWEdish Execution time Tool) [SWE06]. 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. The AE options can be set in the GUI together with calculation method and limitations on the input variables.

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.

Benchmarks

To evaluate the methods in Chapter 6, programs from the Mälardalen WCET Benchmark suite [SWE06] have been used together with components provided by industrial partners [BEG^{+}08], to evaluate our methods. Benchmarks which may by run with many different input values are included. 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. For the industrial benchmarks named "task" we do not have access to the source code. Because of this, all benchmarks have been treated as black-box components, for which we only have considered information about their inputs. Furthermore, the benchmarks named "task" are designed according to the Rubus [HMTN^{+}08] component model.

8.1.2 Reusable WCET

Many benchmarks' inputs have been divided until the point where (i) all value space partitions are *single valued*, or (ii) the accuracy[1] reach 100%. For some benchmarks this has not been possible due to limitations on the WCET analysis tool or limited search heuristic.

All benchmarks have been run with four different strategies for partitioning the input value space partition. These strategies are; firstly, **Last used** (L) where the next variable to be divided in the value space partition is the same variable as was divided last. The L strategy is similar to a *depth first* tree search algorithm. The second strategy is **Next** (N) where the next variable to

[1]Accuracy is defined in Chapter 6.

Program	Description	#LOC	#Vars	|D|
`crc`	Cyclic redundancy check computation on 40 bytes of data.	128	4	81
`edn`	Finite Impulse Response (FIR) filter calculations.	285	3	2^{16}
`inssort`	Insertion sort on a reversed array of size 10.	92	10	10^{320}
`jcomplex`	Nested loop program.	64	2	2^{9}
`lcdnum`	Read ten values, output half to LCD.	64	2	2^{10}
`ns`	Search in a multi-dimensional array.	535	1	2^{9}
`nsichneu`	Simulates an extended Petri net. Generated code with more than 250 `if`-statements.	4253	6	2^{18}
`esab_mod`	Industrial code developed by CC-Systems and Esab to control welding machine.	3064	17	2^{72}
`task1`	Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers.	55	4	2^{6}
`task3`	Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers.	58	7	18
`task4`	Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers.	72	18	2^{57}
`task5`	Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers.	86	8	2^{10}
`task7`	Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers.	123	26	2^{37}

Table 8.1: Benchmark programs used.

be divided is the next variable in their annotated order. The N strategy is similar to a *breadth first* tree search algorithm. Both strategies are also analyzed with eXtreme value heuristics **Next eXtreme** (NX) and **Last used eXtreme** (LX). 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 in three new value space partitions the first time each variable is divided, as suggested in Section 6.4.2.

For each benchmark we show the number of analyses required to reach a specific accuracy, with the number of runs on the y-axis, and the accuracy on the x-axis. Each benchmark has been evaluated with respect to ARM9 hardware and the cluster-based (CB) calculation [Erm03]. 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.

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[2] 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.

Program	**Flow facts**						
	ip	**ep**	**ina**	**ine**	**mmh**	**mmnl**	**mmnc**
`crc`	X	X			X	X	X
`edn`	X	X			X	X	X
`inssort`	X	X			X	X	X
`jcomplex`	X	X			X	X	X
`lcdnum`	X	X			X	X	X
`ns`	X	X			X	X	X
`nsichneu`	X	X			X	X	X
`esab_mod`	X	X	X	X	X	X	X
`task1`	X	X			X	X	X
`task3`	X	X			X	X	X
`task4`	X	X			X	X	X
`task5`	X	X			X	X	X
`task7`	X	X			X	X	X

Table 8.2: Generated flow fact for each benchmark.

To be able to reach an accuracy of 100% the algorithm consider value space partitions with any difference in either WCET or BCET to be different value space partitions. In practice it is likely to have a certain range where value space partitions with similar execution times are grouped.

For many benchmarks we see a trend with rapid improvement for the first few runs, and a slower improvement after the evaluation has reached an accuracy of $\approx 50 - 70\%$. There is often a clear increase in runs where the improvement slows down that we call the *break point*.

The desired behaviour is to have the break point as late as possible, and to always have the rapid improvement before the slow improvement. If the slow improvement is before the rapid improvement it becomes more difficult to trade runs against accuracy.

[2]A header node is the first node in a scope, determining if any nodes in that scope will be executed.

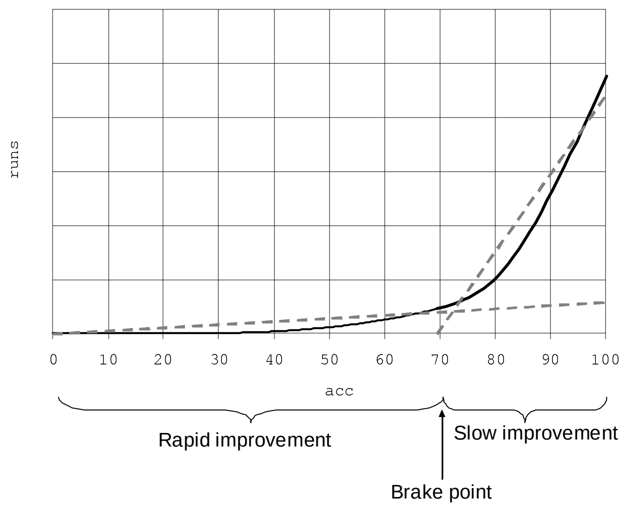


Figure 8.3: Trend of accuracy with respect to #runs.

`crc` and `insertsort` benchmarks

The evaluations show no improvement for the `crc` and `insertsort` benchmarks. 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 is very small. These benchmarks are therefore not representative to the proposed methods.

`jcomplex` and `lcdnum` benchmarks

For the benchmarks `jcomplex` and `lcdnum` we see a rapid improvement before a slow improvement with a break point at $\approx 50\% - 60\%$. For LX there are two break points, which is not desirable. The other strategies performs well. Figure 8.4 and Figure 8.6 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 8.5 and Figure 8.7.

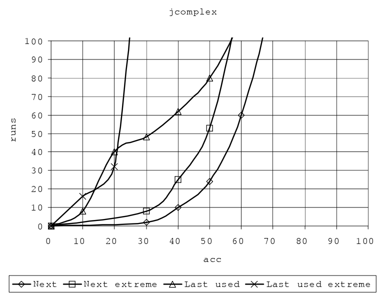


Figure 8.4: `jcomplex` first 100 runs.

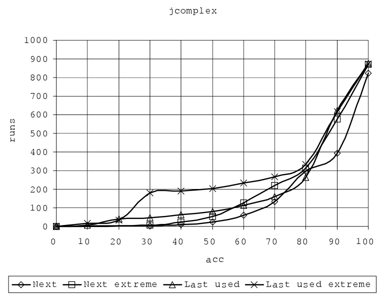


Figure 8.5: `jcomplex`.

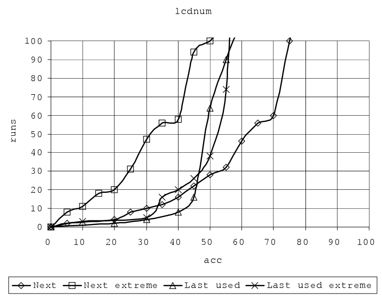


Figure 8.6: `lcdnum` first 100 runs.

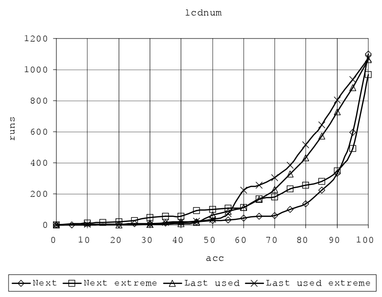


Figure 8.7: `lcdnum`.

`nsichneu` benchmark

`nsichneu` has a rapid improvement first and a break point between 50% and 70% depending on strategy. The L and LX strategies outperform the N and NX strategies as can be seen in Figure 8.8. Despite the complexity and size of input domain the `nsichneu` benchmark reach 50% accuracy after $\approx$ 25 runs and

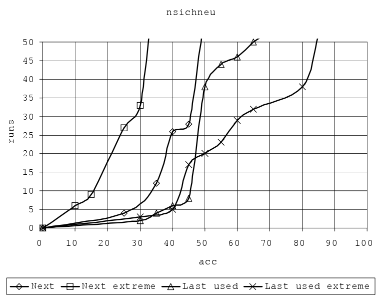


Figure 8.8: `nsichneu` first 50 runs.

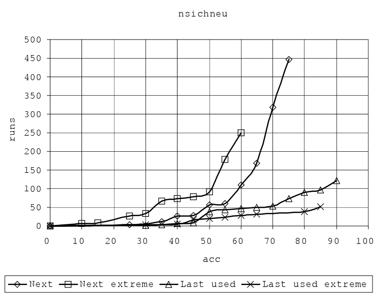


Figure 8.9: `nsichneu`.

90% accuracy after $\approx$ 150 runs. Various reasons to why not 100% accuracy is reached may be to few runs, inaccuracy of the WCET tool or patterns difficult to find with binary search.

`edn` and `ns` benchmarks

The `edn` benchmark shown if Figure 8.10 exhibits a different behaviour than previous benchmarks in that it has a threshold for the L strategy. The slow improvement precedes the rapid improvement making it more difficult to trade effort against accuracy. The `ns` benchmark shown in Figure 8.11, has the rapid improvement before the slow improvement and the break point at $\approx 70\%$. The `ns` benchmark only has one input variable, thus resulting in the same behaviour independent of using the N or L strategies. For the `ns` benchmark a small penalty in using the extreme value heuristic is observed.

`task1` benchmark

`task1` exhibits different behaviours depending on strategy. The L strategies exhibit the desired behaviour with the rapid improvement before the slow improvement and a late break point, while the N strategy has an earlier break point. Notice that the accuracy is still comparably high for a low number of WCET runs for the L and LX strategies. The results for `task1` are depicted in Figures 8.12 and 8.13. `task1` reaches an accuracy of 100% after $\approx$ 20 runs, and 60% already after 2 runs. `task1` reaches 100% accuracy.

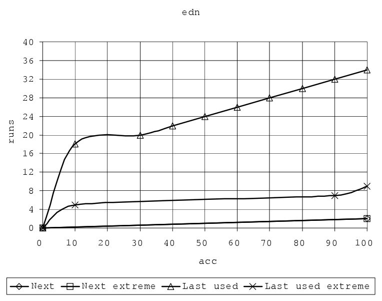


Figure 8.10: edn.

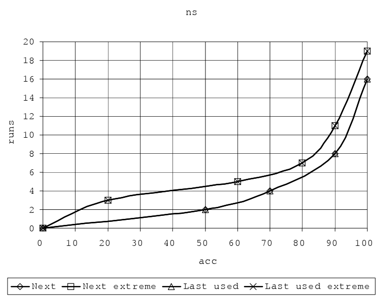


Figure 8.11: ns.

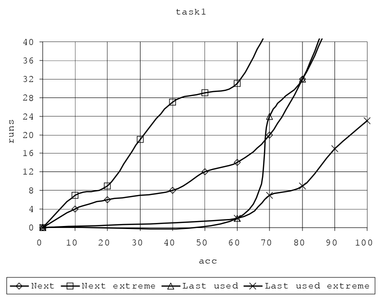


Figure 8.12: task1 first 40 runs.

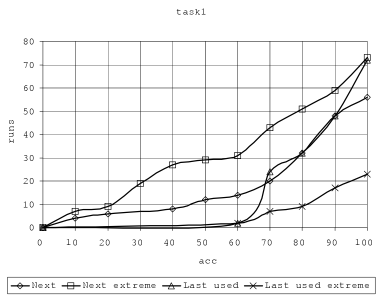


Figure 8.13: task1.

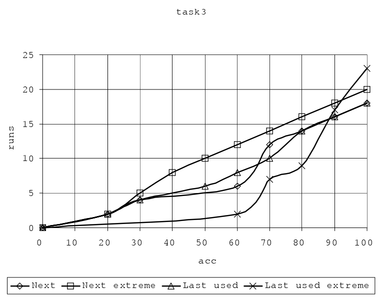


Figure 8.14: task3.

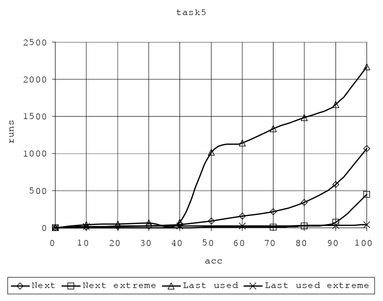


Figure 8.15: task5.

`task3` and `task5` benchmarks

`task3` and `task5` shown in Figures 8.14 and 8.15 both exhibit the behaviour with a rapid improvement before a slow improvement and a late break point for at least one strategy. The different strategies performs very differently and for `task3` the LX strategy perform well, while the other strategies perform worse. For `task5` all strategies except L performs well. The L strategy has an early break point. Both benchmarks reach 100% accuracy.

`test4` benchmark

`task4` has despite its input value size reached an accuracy of 90%. The results also show that a low effort give a large effect with a break point as late as $\approx 80\%$ for the L strategy. The results are shown in Figures 8.16 and 8.17. `task4` reaches 90% accuracy after 20 runs and 80% already after 5 runs.

`task7` benchmark

`task7` has reached 30% accuracy after 700 WCET runs. Various reasons to why not 100% accuracy is reached may be to few runs, inaccuracy of the WCET tool or patterns difficult to find with binary search. Still, a low number of runs reach and accuracy of 15-20%. Note in Figures 8.18 and 8.19 that the graph stops at 40%. There is a big difference between the strategies and NX performs best. However, the rapid improvement is before the slow. The break point is between 20% and 25%.

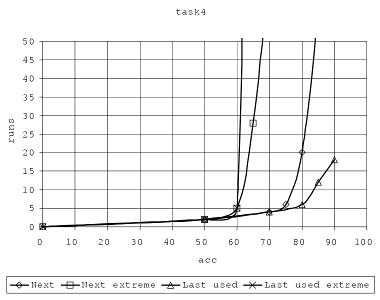


Figure 8.16: `task4` first 50 runs.

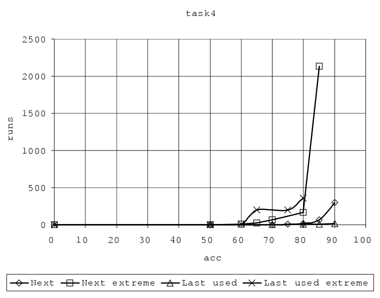


Figure 8.17: `task4`.

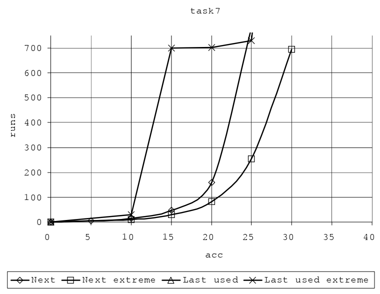


Figure 8.18: `task7` first 700 runs.

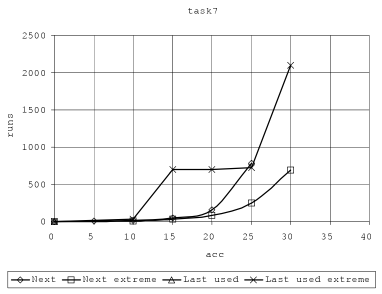


Figure 8.19: `task7`.

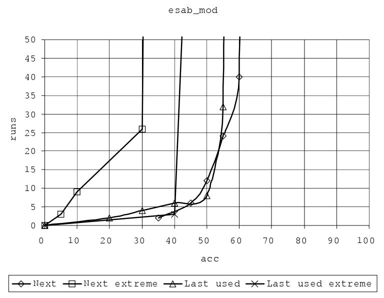


Figure 8.20: `esab_mod` first 50 runs.

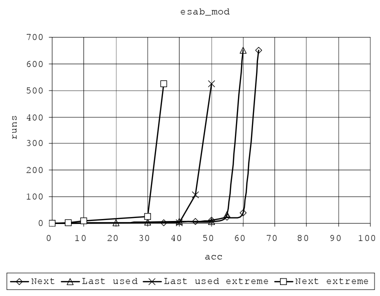


Figure 8.21: `esab_mod`.

esab_mod benchmark

esab_mod has reached an accuracy of 65%, as shown in Figures 8.20 and 8.21. Various reasons to why not 100% accuracy is reached may be to few runs, inaccuracy of the WCET tool or patterns difficult to find with binary search. The esab_mod benchmark exhibits different break points for different strategies and the N strategy performs best. All strategies has a rapid improvement in the beginning and a slow improvement at the end; thus a low number of runs give a comparably high accuracy. An accuracy of 50% is reached already after less than 10 WCET runs.

Conclusions

We notice a big difference in many of the tests between the different approaches. Hence we draw the conclusion that it is important to choose the right strategy. We propose that this is automated in future work by monitoring the difference in accuracy between each run and change the strategy if the accuracy is not improving.

It is interesting that even though the different strategies performs very differently, all benchmarks, within 10 WCET runs give an improvement greater than 20% compared to classical WCET analysis. Most benchmarks give an improvement of 50% within ≈ 20 WCET runs.

Benchmark	Best strategy		Worst strategy	
	Up to 50%	Up to 100%	Up to 50%	Up to 100%
crc	n/a	n/a	n/a	n/a
inssort	n/a	n/a	n/a	n/a
edn	N/NX	N/NX	L	L
jcomplex	N	N	LX	L
lcdnum	N	NX	NX	N
ns	L/N	L/N	LX/NX	LX/NX
nsichneu	LX	L	NX	NX
task1	L/LX	LX	NX	NX
task3	N	L/N	LX/NX	LX/NX
task4	L/LX/N/NX	N	L/LX/N/NX	NX
task5	NX	L	L	L
task7	NX	NX	L	L
esab_mod	L	N	NX	NX
Sum L	4	4	4	4
Sum LX	3	1	4	2
Sum N	6	5	1	1
Sum NX	4	3	7	5

Table 8.3: Best and worst strategies.

We have categorized the different strategies with respect to their performance. They are categorized in best and worst up to 50% accuracy, and, best and worst up to 100% accuracy. We do not draw any final conclusions from the findings in Table 8.3, but the results are slightly slanted towards that the N strategy performs slightly better than the others.

One conclusion that we draw from the results in general is that it indeed is possible to create input parameterizable WCET. The results show that, for both academic and industrial components, a low effort in terms of WCET runs gives rise to a comparably high accuracy. Reaching very high accuracy requires a comparably higher effort, and, in many cases it is possible to reach 100% accuracy.

We finally note that for our results to be representative for all type of input-dependent embedded systems, more evaluations need to be performed. The included code may be quite complex, however several of the benchmarks are quite small.

8.1.3 Finding WCETs

Program	**#Vars**	$\lvert D \rvert$	**Basic**				
			#WC	**MinT**	**MaxT**	**#BT**	**TotT**
`crc`	4	81	7	6922	7421	0	49934
`edn`	3	2^{16}	12	5063	5531	0	63512
`inssort`	10	10^{320}	619	344	765	0	326894
`jcomplex`	2	512	455	32	828	155	116541
`lcdnum`	2	1024	7	31	859	0	3436
`ns`	1	512	6	1093	38172	0	155205
`nsichneu`	6	2^{18}	25	17733	95203	0	1716109
`esab_mod`	19	2^{72}	309	9836	97153	92	4126380
`task1`	4	64	11	16359	19564	0	188593
`task3`	7	18	9	14468	18968	0	147326
`task5`	8	2^{10}	53	47	157	7	5434
`task7`	26	2^{37}	167	375	719	2	72165

Table 8.4: Finding WCET analysis results for benchmarks without program slicing.

Tables 8.4 and 8.5 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 mini-

Program	**#Vars**	$\|D\|$	**Extreme**				
			#WC	**MinT**	**MaxT**	**#BT**	**TotT**
`crc`	4	81	10	6813	7562	0	71872
`edn`	3	2^{16}	7	4687	4984	0	33636
`inssort`	10	10^{320}	619	485	780	0	326894
`jcomplex`	2	512	455	47	922	141	97078
`lcdnum`	2	1024	4	63	734	0	2157
`ns`	1	512	1	35256	35256	0	35256
`nsichneu`	6	2^{18}	20	13968	95203	0	1309785
`esab_mod`	19	2^{72}	227	9881	99443	54	3740033
`task1`	4	64	8	14203	20234	0	121687
`task3`	7	18	10	14125	21890	0	153654
`task5`	8	2^{10}	43	62	187	8	4514
`task7`	26	2^{37}	174	328	625	3	72912

Table 8.5: Finding WCET analysis results for benchmarks with extreme value heuristics and without program slicing.

Program	**#Vars**	$\|D\|$	**Basic**				
			#WC	**MinT**	**MaxT**	**#BT**	**TotT**
`nsichneu`	5	2^{10}	24	17453	92780	0	1638990
`esab_mod`	15	2^{58}	216	6750	61012	17	3956334
`task7`	17	2^{37}	167	328	656	2	70164

Table 8.6: Analysis results for benchmarks affected by program slicing.

Program	**#Vars**	$\|D\|$	**Extreme**				
			#WC	**MinT**	**MaxT**	**#BT**	**TotT**
`nsichneu`	5	2^{10}	17	17087	46702	0	644249
`esab_mod`	15	2^{58}	90	7668	61601	10	1185105
`task7`	17	2^{37}	174	328	625	3	72893

Table 8.7: Analysis results for benchmarks affected by with extreme value heuristics and program slicing.

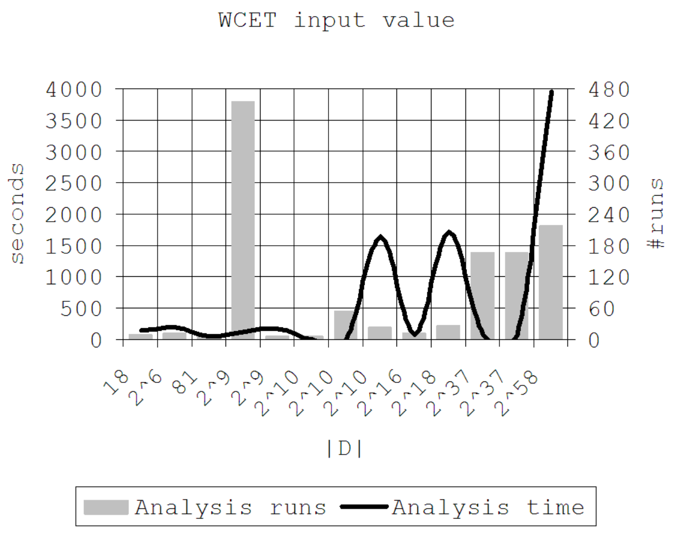


Figure 8.22: Number of runs and analysis time with respect to input space.

mum and maximum analysis time (in milliseconds) used for any of the WCET calculations. **#BT** gives the number of back trackings performed in the input-value search analysis. **TotT** gives the total analysis time (in milliseconds) 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, (`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. Furthermore, the two different heuristics (**Basic** and **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 8.6 and 8.7 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. **#Vars** gives number of input variables left after the slicing and $|D|$ gives the size of the corresponding input value space.

Only the benchmarks (`nsichneu`, `esab_mod` and `task7`), that had their value space reduced by the slicing are included in the Table 8.6. We see that the reduction in analysis time, as well as the size of the input value space, when using slicing, are significant for `nsichneu`. However, for `task7` the effects are more moderate. This is inherent in that most input data removed for `task7` were relatively small in value size, or static pointers holding only a single abstract pointer value.

Note that our results may not be fully representative for all type of input-dependent embedded system programs. Some of the included components are quite complex; however, 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 Allocating components to tasks

8.2.1 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 transaction 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 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 than 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. Table 8.8 outlines all values and distributions, of the system generation values. 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.

Parameters on component level		WCET in % of period	Distribution (%)
Number components	Distribution (%)	2	45
40	1,25	4	50
50	6,25	8	5
60	10	Stack size	Distribution (%)
70	6,25	256	10
80	2,75	512	25
100	7,5	1024	25
120	13	2048	35
140	7,5	4096	10
150	5	**Parameters on system level**	
160	2,5	transactions size	Distribution (%)
180	8	% of num. comp.	
200	5,25	10	10
210	5	13	25
240	9	17	25
250	1,25	21	25
280	5	25	15
300	2	Laxity % of ctr.dl	Distribution (%)
320	1	110	33
350	1,25	130	33
400	0,25	150	33
Isolation % of all components	Distribution (%)	Utilization %	Distribution (%)
0	20	30	25
10	30	50	25
20	30	70	25
30	20	90	25
		GA parameters	
Period time (μs)	Distribution (%)	GA property	Value
10000	20	Population	300
25000	20	Generations	500
50000	40	Elite rate	5%
100000	20	Cull rate	40%
		Mutation rate	1%

Table 8.8: Data used for generating systems, and GA parameter.

The simulations are performed for four different utilization levels, 30%, 50%, 70% and 90%. For each level of utilization 1000 different systems are generated. 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.

8.2.2 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 ge-

Load	Laxity	1-1 allocation			GA allocation		
		Stack	CPU	Success	Stack	CPU	Success
30%	All	28882	4,1%	74%	17380	2,0%	87%
	1.1	25949	3,5%	39%	14970	1,6%	58%
	1.3	33077	4,4%	78%	21005	2,2%	97%
	1.5	26755	4,1%	95%	15503	2,0%	99%
50%	All	37277	4,8%	82%	24297	2,4%	90%
	1.1	35391	4,3%	49%	23146	2,3%	64%
	1.3	38251	4,8%	88%	25350	2,5%	96%
	1.5	37043	4,9%	98%	23740	2,3%	100%
70%	All	44455	5,1%	85%	30694	2,7%	91%
	1.1	44226	5,0%	58%	31638	2,7%	73%
	1.3	44267	5,1%	94%	30686	2,7%	98%
	1.5	44619	5,2%	98%	30232	2,6%	100%
90%	All	46943	5,6%	87%	37733	3,1%	93%
	1.1	54858	5,7%	65%	41207	3,4%	80%
	1.3	49607	5,5%	92%	35470	3,0%	98%
	1.5	53535	5,7%	98%	38260	3,1%	99%

Table 8.9: Memory, CPU overhead and success ratio for 1-1 and GA allocations.

netic algorithm, and no two tasks have the same priority. We compare the allocations to one-to-one allocations. Table 8.9 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 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.

The first graph of the simulations (Figure 8.23) 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 compared to 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 8.24) shows that the deadlines are relaxed with

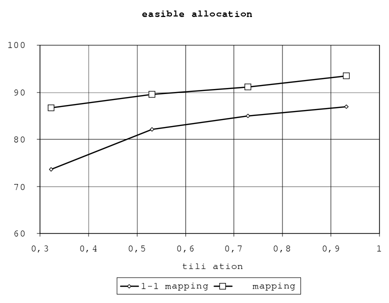


Figure 8.23: Average success ratio.

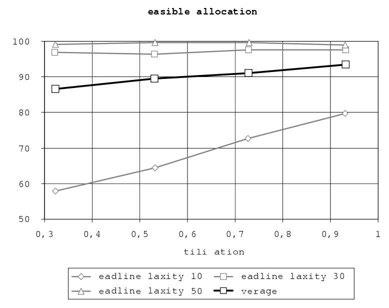


Figure 8.24: Success rate for allocations.

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 8.25) shows for both approaches the average stack size for the systems at different utilization. The comparison is only amongst allocations that are have been successfully mapped by both strategies. The memory size consists of the TCB and the stack size, and the TCB size is 300 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 8.26) shows the average task switch time in micro seconds 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.

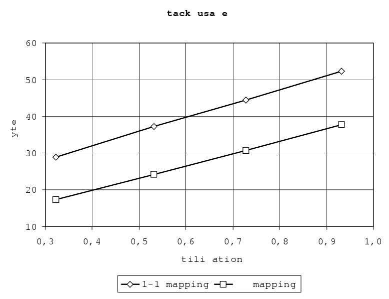


Figure 8.25: Average memory size.

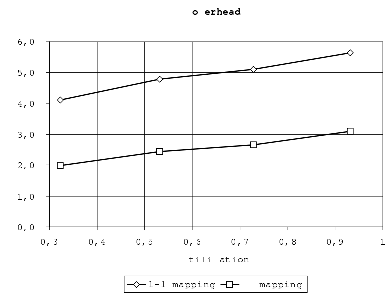


Figure 8.26: Average task switch overhead.

8.2.3 Summary

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.

I may not have gone where I intended to go, but I think I have ended up where I needed to be.

-The Long Dark Tea-Time of the Soul

Chapter 9

Summary and conclusions

In this chapter we summarize and conclude the thesis, and we discuss possible future research directions.

9.1 Summary

This thesis provides a link between Component-Based Software Engineering (CBSE) and Embedded Real-Time Systems (ERTS), presenting some of the difficulties in, and, providing several solutions for, developing predictable and resource constrained systems by reusing pre-fabricated software components.

We introduce our research starting with an illustrative example from the real world before giving an overview of the specific research and contributions. ERTS is introduced together with its terminology and definitions. The state-of-research is presented in terms of real-time scheduling and WCET analysis. We give an introduction to CBSE terminology and definitions and discuss its industrial motivations. The industrial and research problems are discussed and we outline the research methodology that has been used in order to approach the research problems. An extension to a CBSE development process is proposed and our methods are positioned in the CBSE process to support resource efficiency and accurate analysis in developing component-based ERTS. Two novel methods are outlined for deriving accurate WCET estimates of a component; the methods are based on a combination of static WCET analysis and systematic search over the value space of input variables. We also present a method for deriving allocations from components to real-time tasks that is optimized for low resource consumption, while maintaining stipulated real-time requirements; the method is based on real-time analysis, calculating resource consumption and genetic algorithms. Throughout the thesis we use an example of an Adaptive Cruise Controller application to illustrate our techniques. Finally we present evaluations of the technical contributions.

9.1.1 Contributions

We have introduced several novel methods for improving utilization and prediction accuracy for reusable software components. We have extended a 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 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 *Reusable WCET analysis*. The input space of a reusable component is partitioned with respect to execution time, creating parameterizable component WCET contracts. A WCET contract is parameterizable and produces a WCET that is more accurate with respect to the specific usage. The result is that the WCET analysis can be reused together with the components. The reusable WCET is evaluated with components from our industrial partners.

C2 *Methods for deriving WCET input values*. The input space of a component is divided into partitions 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 derivation of WCET input values 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 augmented with the methods outlined in this thesis. The WCET analysis is divided and positioned in both the component and system part. The component to task allocation is positioned after the reusable WCET analysis for providing a tight WCET to the allocation framework.

C5[1] *A prototype tool* implementing the ideas from contributions C1 and C2. The prototype tool graphically presents WCET and BCET connected to inputs and component contracts. The tool supports several different heuristics for creating WCET contracts.

9.2 Discussion

In Chapter 4 several industrial and academic problems were formulated, and solutions proposed. We revisit the questions identified from Shaw's classification of software engineering research [Sha01]; the questions related to characterization and methods. The first question is:

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 software development in the sense that it is divided into component development and system development. WCET analysis should be divided in two different parts, one component part, developed to be reusable by the component developer, and one system part, to be used by the system designer. This

[1]C5 is not a scientific contribution.

facilitates the adoption of the technique in the CBSE development process. Reusable WCET analysis should be performed in the component development part of the CBSE development process and should be parameterized in the system development part of the CBSE development process.

- Reusable WCET analysis results should reach a pre-defined accuracy, and it should be possible to reach higher accuracy compared to current state of the art static WCET analysis.
- A component-based system must be transformed to a real-time system conforming to a specific real-time model. A common approach for allocating components to real-time tasks is to view one component as one task. For better resource usage many components should be allocated to one real-time task. At the same time an allocation from components to real-time tasks must not violate temporal requirements. Thus, components should be allocated to tasks in such a way that the temporal correctness of the system is fulfilled.

The second question is related to methods and means from Shaw's classification of software engineering research.

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 comes with a penalty of increased effort in terms of calculation time for achieving the reusable WCET.

  The results show that the proposed method can express relationships between inputs and execution times with higher precision compared to traditional WCET analysis. The method has been evaluated with both industrial and academic software.

  Due to tighter WCET estimations when analyzing smaller partitions of a component the WCET become more accurate compared to traditional WCET analysis.

  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 evaluated components, the results scales very good up to around an accuracy[2] of $\approx 60 - 70\%$. The last $\approx 30 - 40\%$ accuracy generally scales worse than linear, requiring more effort. The algorithmic 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 (the path through a program that takes longest time). 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. However, rather than searching the entire input domain, we iteratively remove parts of the domain that do not generate execution of the WCET path.

  The evaluations show that it is possible to derive the input combination that generates the execution of the WCET program path within bounded time and with low algorithmic complexity, i.e., never higher algorithmic complexity than $O(n)$ where n is the size of the input domain.

- 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 produce a higher number of feasible allocations, 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% reduction in required memory size and 48% reduction of 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.

So, have we solved the outlined problems with these novel methods? No, we certainly have not solved the problems; however, we take one step in the right

[2]Accuracy is defined in Chapter 6.

direction. We underline the difficulty in reusing software components in ERTS, and provides methods and tool for facilitating CBSE in ERTS. The research show that it is possible to achieve efficient reuse of software components and at the same time have efficient use of resources and high accuracy of real-time predictions.

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 all research is based on assumptions, and some assumptions are impractical. For instance most real-time scheduling work is based on the assumption that tasks are independent. However, as research progresses assumptions may be relaxed, making the research more practically applicable.

9.3 Future research directions

In this section we present a selection of possible future research directions. We outline a list of possible directions and we will further discuss a few of those research directions:

- Parameterization with respect to hardware in the reusable WCET contracts.
- Parallelization of the algorithms used for finding input value space partitions in the reusable WCET method.
- More advanced real-time constraints in the allocation framework.
- More advanced search heuristics in the reusable WCET analysis.
- Incremental WCET analysis.

9.3.1 Varying hardware

In future work it may be possible to also parameterize 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 Distribute the search algorithms

The reusable WCET analysis is inherent suitable for parallelizing of the analysis executions. 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.

9.3.3 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. In this way it might be possible to always choose the best strategy depending on what part of the input space is divided.

Bibliography

[ABGP05] Colin Atkinson, Christian Bunse, Hans-Gerhard Groß, and Christian Peper. *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends (Lecture Notes in Computer Science)*. ISBN 3-540-30644-7. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Acc06] The embedded software industry: Challenges and successes. Whitepaper, May 2006. `http://www.accenture.com/`.

[ÅCF$^+$07] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The save approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[ÅFSC04] Mikael Åkerholm, Johan Fredriksson, Kristian Sandström, and Ivica Crnkovic. Quality attribute support in a component technology for vehicular software. In *Proc. 4th Conference on Software Engineering Research and Practice in Sweden*, Linköping, Sweden, October 2004.

[AHLW08] Sebastian Altmeyer, Christian Hümbert, Björn Lisper, and Reinhard Wilhelm. Parametric timing analysis for complex architectures. In *Proc. 15th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'08)*, Kaohsiung, Taiwan, August 2008. IEEE Computer Society Press.

[aiT] ait execution time analyzer. `http://www.absint.com/ait/`.

[Alt06] Sebastian Altmeyer. Parametric wcet analysis, parametric framework and parametric path analysis. Master's thesis, Saarland University, Department of Computer Science, Oct 2006.

[Aud91] Neil C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS-91-164, Department of Computer Science, University of York, December 1991.

[AZP03] Andreas S. Andreou, Andreas C. Zographos, and George A. Papadopoulos. A three-dimensional requirements elicitation and management decision-making scheme for the development of new software components. In *Proc. 5th International Conference On Enterprise Information Systems (ICEIS)*, pages 3–13, Angers, France, 2003. Springer Verlag.

[Baa88] Sara Baase. *Computer algorithms: introduction to design and analysis (2nd ed.)*. ISBN 0-201-06035-3. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[BB99] Ian Bate and Alan Burns. An approach to task attribute assignment for uniprocessor systems. In *Proc. 11th Euromicro Workshop on Real Time Systems*, pages 46–53, York, England, June 1999. IEEE Computer Society Press.

[BB00] Guillem Bernat and Alan Burns. An approach to symbolic worst-case execution time analysis. In *Proc. 25th IFAC/IFIP Workshop on Real-Time Programming, Palma, Spain*. IFAC, Elsevier Science Ltd, May 2000.

[BB05] Jesal Bhuta and Barry Boehm. A method for compatible cots component selection. In *Proc. International Conference on COTS-Based Software Systems (ICCBSS'05)*, pages 132–143, Bilbao, Spain, February 2005. Springer Verlag.

[BBB$^+$00] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical concepts of component-based software engineering, volume ii. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie-Mellon University, May 2000.

[BBB04] Guillem Bernat, Ian Broster, and Alan Burns. Rewriting history to exploit gain time. In *Proc. 25th IEEE Real-Time Systems Symposium (RTSS'04)*, pages 328–335, Lisbon, Portugal, 2004. IEEE Computer Society Press.

[BBCD+00] Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume i: Market assessment of component-based software engineering. Technical Report CMU/SEI-2001-TN-007, Software Engineering Institute, May 2000.

[BCC+03] Ed Brinksma, Geoff Coulson, Ivica Crnkovic, Andy Evans, Sébastien Gérard, Susanne Graf, Holger Hermanns, Jean-Marc Jézéquel, Bengt Jonsson, Anders Ravn, Philippe Schnoebelen, Francois Terrier, and Angelika Votintseva. Component-based design and integration platforms: a roadmap. Technical Report IST-2001-34820, The ARTIST consortium, April 2003.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan M. Petters. Wcet analysis of probabilistic hard real-time systems. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 279–288, Austin, TX, USA, December 2002. IEEE Computer Society Press.

[BCP03a] Guilemm Bernat, Antoine Colin, and Stefan Petters. pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Technical Report YCS-2003-353, Department of Computer Science, York University, York, England, January 2003.

[BCP03b] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET, a Tool for Probabilistic WCET Analysis of Real-Time Systems. In *Proc. 3d International Workshop on Worst-Case Execution Time analysis (WCET'03) in conjunction with 13th Euromicro Conference of Real-Time Systems,* *(ECRTS'03)*, pages 21–38, Porto, Portugal, June 2003. IEEE Computer Society Press.

[BEG$^+$08] Dani Barkah, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Christer Sandberg. Evaluation of automatic flow analysis for WCET calculation on industrial real-time system code. In *Proc. 20th Euromicro Conference of Real-Time Systems, (ECRTS'08)*, pages 331–340, Prague, Czech republic, July 2008. IEEE Computer Society Press.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BJPW99] Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, 1999.

[BL08] Stefan Bygde and Björn Lisper. Towards an automatic parametric wcet analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time analysis (WCET'08) in conjunction with 18th Euromicro Conference of Real-Time Systems, (ECRTS'08)*. IEEE Computer Society Press, July 2008.

[BMdW$^+$04] Egor Bondarev, Johan Muskens, Peter de With, Michel Chaudron, and Johan Lukkien. Predicting real-time properties of component assemblies: a scenario-simulation approach. In *Proc.* 30th *Euromicro conference*, pages 40–47, Rennes, France, September 2004. IEEE Computer Society Press.

[BMdWC04] Egor Bondarev, Johan Muskens, Peter de With, and Michel Chaudron. Towards predicting real-time properties of a component assembly. In *Proc.* 30th *Euromicro conference*, pages 601–610, Rennes, France, September 2004. IEEE Computer Society Press.

[BNTZ93] Alan Burns, Mark Nicholson, Ken W. Tindell, and N. Zhang. Allocating and scheduling hard real-time tasks on a point-to-point distributed system. In *Proc. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'93)*, pages 11–20, Newport Beach, CA, USA, April 1993. IEEE Computer Society Press.

[Bou] Bound-t execution time analyzer. `http://www.tidorum.fi/bound-t/`.

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *Proc. 28th International Conference on Software engineering (ICSE'06)*, pages 33–42, New York, NY, USA, May 2006. ACM Press.

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

[But06] Giorgio Buttazzo. Research trends in real-time computing for embedded systems. *ACM Special Interest Group on Embedded Systems*, 3(3):1–10, July 2006.

[CA95] Sheng T. Cheng. and Ashok K. Agrawala. Allocation and scheduling of real-time periodic tasks with relative timing constraints. In *Proc. 2nd IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'95)*, pages 210–217, Tokyo, Japan, October 1995. IEEE Computer Society Press.

[CAPD02] Ivica Crnkovic, Ulf Askerlund, and Anita Persson-Dahlqvist. *Implementing and Integrating Product Data Management and Software Configuration Management*. ISBN 1-58053-498-8. Artech House Software Engineering Library, 2002.

[CB02] Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, pages 50–59, Washington, DC, USA, June 2002. IEEE Computer Society Press.

[CCL06] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based development process and component lifecycle. In *Proc. International Conference on Software Engineering Advances (ICSEA'06)*, page 44, Tahiti, French Polynesia, October 2006. IEEE Computer Society Press.

[Cha02] Ching-Yao Chan. A treatise on crash sensing for automotive air bag systems. *IEEE/ASME Transactions on Mechatronics*, 7(2):220–234, June 2002.

[CHMW07] Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. Generalizing parametric timing analysis. *Proc. ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'07)*, 42(7):152–154, 2007.

[CL92] Norman Carver and Victor Lesser. The evolution of blackboard control architectures. Technical Report UM-CS-1992-071, Department of Computer Science, University of Massachusetts Amherst, October 1992.

[CL02a] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. ISBN 1-58053-327-2. Artech House publisher, 2002.

[CL02b] Ivica Crnkovic and Magnus Larsson. Challenges of component-based development. *Journal of Systems and Software*, 61(3):201–212, April 2002.

[Crn02] Ivica Crnkovic. Component-based software engineering – new challenges in software development. *Software Focus*, 2(4):127–133, April 2002.

[Crn04] Ivica Crnkovic. Component-based approach for embedded systems. In *Proc.* 9th *International Workshop on Component-Oriented Programming (WCOP'04)*, Oslo, Norway, June 2004. Springer Verlag.

[CZM+03] Kendra Cooper, Jia Zhou, Hui Ma, I-Ling Yen, and Farokh B. Bastani. Code parameterization for satisfaction of qos requirements in embedded software. In *Proc. International conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03)*, pages 58–64, Las Vegas, NV, USA, June 2003. CSREA Press.

[Dou99] B. P. Douglas. *Doing Hard Time*. ISBN 0-201-49837-5. Addison Wesely, 1999.

[DP04] Laurent David and Isabelle Puaut. Static determination of probabilistic execution times. In *Proc. 16th Euromicro Conference of Real-Time Systems, (ECRTS'04)*, pages 223–230, Catania, Sicily, July 2004. IEEE Computer Society Press.

[Dur06] Marc Duranton. The challenges for high performance embedded systems. In *Proc.* 9th *Euromicro Conference on Digital Systems Design, (DSD'06)*, pages 3–7, Dubrovnik, Croatia, September 2006. IEEE Computer Society Press.

[Erm03] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Dept. of Information Technology, Uppsala University, Sweden, June 2003.

[EY97] Rolf Ernst and Wei Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. IEEE/ACM international conference on Computer-aided design (ICCAD'97)*, pages 598–604, San Jose, CA, USA, 1997. IEEE Computer Society Press.

[FÅDS03] Johan Fredriksson, Mikael Åkerholm, Radu Dobrin, and Kristian Sandström. Attaining Flexible Real-Time Systems by Bringing Together Component Technologies and Real-Time Systems Theory. In *Proc.* 29th *Euromicro Conference on EUROMICRO (EUROMICRO'03)*, pages 399–403, Belek, Turkey, September 2003. IEEE Computer Society Press.

[FB70] Alex Fraser and Donald Burnell. *Computer Models in Genetics*. ISBN 0-070-21904-4. New York: McGraw-Hill, 1970.

[FBH05] Viktoria Firus, Steffen Becker, and Jens Happe. Parametric performance contracts for qml-specified software components. In *Proc. 2nd International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA 2005)*, pages 73–90, Edinburgh, Scottland, December 2005. Elsevier Science Inc.

[FdN08] Peter H. Feiler and Dionisio de Niz. Assip study of real-time safety-critical embedded software-intensive system engineering practices. Technical Report CMU/SEI-2008-SR-001, Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, February 2008.

[FF95] Carlos M. Fonseca and Peter J. Flemming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary computation*, 3(1):1–16, Spring 1995.

[FH08] Christian Ferdinand and Reinhold Heckmann. Worst-case execution time – a tool provider's perspective. In *Proc. 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC08)*, pages 340–345, Orlando, FL, USA, May 2008. IEEE Computer Society Press.

[FPDF98] William Frakes, Ruben Prieto-Diaz, and Christopher Fox. Dare: Domain analysis and reuse environment. *Annals of Software Engineering*, 5(1):125–141, 1998.

[FSÅ05] Johan Fredriksson, Kristian Sandström, and Mikael Åkerholm. Optimizing Resource Usage in Component-Based Real-Time Systems. In *Proc. 8th International Symposium on Component-Based Software Engineering (CBSE8)*, pages 49–65, St.Louis, MO, USA, May 2005. Springer Verlag.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Systems Journal*, 17(2-3):131–181, 1999.

[Gan06] Jack Ganssle. Really real-time systems. In *Proc. Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, Boston, MA, USA, April 2006. Embedded Systems Conference.

[GCS$^+$02] Thomas Genssler, Alexander Christoph, B. Schulz, Michael Winter, Christian M. Stich, C. Zeidler, Peter O. Müller, A. Stelter, Oscar Nierstrasz, Stéphane Ducasse, Gabriela Arévalo, Roel Wuyts, P. Liang, Bastiaan Schönhage, and Reinier van den Born. Pecos in a nutshell. Technical report, The PECOS Consortium, 2002.

[GCW$^+$02] Thomas Genßler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. Components for embedded software: the pecos approach. In *Proc. 5th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'02)*, pages 19–26, New York, NY, USA, June 2002. ACM Press.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 57–66, Rio de Janiero, Brazil, December 2006. IEEE Computer Society Press.

[GHS94] Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Proc. 25th IEEE Real-Time Systems Symposium (RTSS'04)*, pages 192–203, Lisbon, Portugal, December 1994. IEEE Computer Society Press.

[Gil05] Helen Gill. Challenges for critical embedded systems. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05)*, pages 7–12, Washington, DC, USA, February 2005. IEEE Computer Society Press.

[GLN01] Paulo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multiprocessor systems-on-a-chip. In *Proc. 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, pages 73–81, London, UK, December 2001. IEEE Computer Society Press.

[Gom00] Hassan Gomaa. *Designing Concurrent Distributed, and Real-Time Applications with UML.* ISBN 0-201-65793-7. Addison Wesely, 2000.

[Gri93] Martin L. Griss. Software reuse: from library to factory. *IBM Systems Journal*, 32(4):548–566, July 1993.

[GSBC05] Stefan Valentin Gheorghita, Sander Stuijk, Twan Basten, and Henk Corporaal. Sharper wcet upper bounds using automatically detected scenarios. Technical Report ESR-2005-04, Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems, March 2005.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* ISBN 0-471-20284-3. John Wiley & Sons, 2004.

[Har05] William C. Harris. *Frontier research: the European challenge; High-Level Expert Group report*. ISBN 92-894-9209-0. Office for Official Publications of the European Communities, febr. 2005 edition, 2005.

[HC01] George T. Heineman and William T. Councill. *Component-based Software Engineering, Putting the Pieces Together*. ISBN 0-201-70485-4. Prentice-Hall, 2001.

[HCDJ08] Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software*, 81(1):132–149, 2008.

[HK07] Trevor Harmon and Raymond Klefstad. Interactive back-annotation of worst-case execution time analysis for java microprocessors. In *Proc. 14th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'07)*, pages 209–216, Daegu, South Korea, August 2007. IEEE Computer Society Press.

[HKK04] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. Reuse of software in distributed embedded automotive systems. In *Proc. 4th ACM international conference on Embedded software (EMSOFT'04)*, pages 203–210, New York, NY, USA, September 2004. ACM Press.

[HKR06] Jens Happe, Heiko Koziolek, and Ralf Reussner. Parametric performance contracts for software components with concurrent behaviour. In *Proc. 3rd Workshop on Formal Aspects of Component Software (FACS)*, volume 167, pages 91–106. Elsevier Science Inc., sep 2006.

[HLB$^+$97] Hans Hansson, Harold Lawson, Olof Bridal, Christer Norström, Sven Larsson, Henrik Lönn, and Mikael Strömberg. Basement: An architecture and methodology for distributed automotive real-time systems. *IEEE Transactions on Computers*, 46(9):1016–1027, September 1997.

[HMTN06] Kaj Hänninen, Jukka Mäki-Turja, and Mikael Nolin. Present and future requirements in developing industrial embedded real-time systems - interviews with designers in the vehicle domain. In *Proc. 13th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 139–150, Potsdam, Germany, March 2006. IEEE Computer Society Press.

[HMTN^{+}08] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The rubus component model for resource constrained real-time systems. In *Proc. 3rd IEEE International Symposium on Industrial Embedded Systems (SIES'08)*, Montpellier, France, June 2008. IEEE Computer Society Press.

[HMTS^{+}08] Kaj Hänninen, Jukka Mäki-Turja, Staffan Sandberg, John Lundbäck, Mats Lindberg, Mikael Nolin, and Kurt-Lennart Lundbäck. Framework for real-time analysis in rubus-ice. In *Proc. 13th IEEE International Conference on Emerging Technologies and Factory Automation*, Hamburg, Germany, September 2008. IEEE Computer Society Press.

[Hol73] John H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105, June 1973.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[HS97] Chao-Ju Hou. and Kang G. Shin. Allocation of periodic task modules with precedence and deadline constraints in distributed real-time system. *IEEE Transactions on Computers*, 46(12):1338–1356, December 1997.

[Hüm06] Christian Hümbert. Parametric wcet analysis, parameter analysis and parametric loop analysis. Master's thesis, Saarland University, Department of Computer Science, Oct 2006.

[IEE92] IEEE. The new ieee standard dictionary of electrical and electronics terms. Technical report, IEEE std., 1992.

[JO95] Klaus Jansen and Sabine R. Ohring. Approximation algorithms for time constrained scheduling. In *Proc. 1st IEEE International Conference on Algorithms and Architectures for Parallel Processing*, pages 670–679, Brisbane, Australia, April 1995. IEEE Computer Society Press.

[Jor97] Kimberly Jordan. Software reuse term paper for the mjy team. Technical Report TP-KJ.1.0, George Mason University, April 1997.

[JWLQ06] Meng-Lou Ji, Ji Wang, Shuhao Li, and Zhi-Chang Qi. Automated wcet analysis based on program modes. In *Proc. International workshop on Automation of Software Test (AST'06) in conjunction with International Conference on Software Engineering (ICSE'06)*, pages 36–42, Shanghai, China, May 2006. ACM Press.

[KGV83] Scott Kirkpatrick, Charles D. Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Kop91] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *Proc. International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, 1991. Springer Verlag.

[Kor99] Bogdan Korel. Black-box understanding of cots components. In *Proc. 7th International Workshop on Program Comprehension (IWPC'99)*, page 92, Washington, DC, USA, 1999. IEEE Computer Society Press.

[KP03] Raimund Kirner and Peter Puschner. Discussion of misconceptions about wcet analysis. In *Proc. 3d International Workshop on Worst-Case Execution Time analysis (WCET'03) in conjunction with 13th Euromicro Conference of Real-Time Systems, (ECRTS'03)*, pages 61–64, Porto, Portugal, August 2003. Mälardalen Real-Time Centre, Mälardalen University.

[KP05] Raimund Kirner and Peter Puschner. Classification of WCET analysis techniques. In *Proc. 8th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 190–199, Seatle, WA, USA, May 2005. IEEE Computer Society Press.

[KWS03] Sharath Kodase, Shige Wang, and Kang G. Shin. Transforming structural model to runtime model of embedded software with real-time constraints. In *Proc. Design, Automation and Test in European Conference and Exhibition (DATE'03)*, pages 170–175, Munich, Germany, November 2003. IEEE Computer Society Press.

[Lar04] Magnus Larsson. *Predicting Quality Attributes in Component-based Software Systems*. PhD thesis, Mälardalen University, Department of Computer Science and Engineering, Västerås, Sweden, March 2004.

[Lau06] Audi Q7 Launch. Audi q7 launch (september 2006), September 2006. http://editorial.carsales.com.au/car-review/2051265.aspx.

[LBCC08] Rikard Land, Laurens Blankers, Michel Chaudron, and Ivica Crnkovic. Cots selection best practices in literature and in industry. In *Proc. 10^{th} International Conference on Software Reuse (ICSR'08)*, pages 100–111, Beijing China, May 2008. Springer Verlag.

[LGA$^+$07] Jingyue Li, Anita Gupta, Jon Arvid, Borretzen Borretzen, and Reidar Conradi. The empirical studies on quality benefits of reusing software components. In *Proc. 31^{st} International Computer Software and Applications Conference (COMPSAC 2007)*, pages 399–402, Washington, DC, USA, July 2007. IEEE Computer Society Press.

[Lis03] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. Technical report, Mälardalen Real-Time Research Centre, April 2003.

[LL73] Chung L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in hard-real-time environment. *Journal of the Association for Computing Machinery (ACM)*, 20(1):46–61, 1973.

[LLL03] Kurt-Lennart Lundbäck, John Lundbäck, and Mats Lindberg. Component-based development of dependable real-time applications, August 2003. Real-Time in Sweden (RTiS), Presentation of Component-Based Software Development Based on the Rubus Concept.

[LME98] Yau-Tsun Steven Li, Sharad Malik, and Benjamin Ehrenberg. *Performance Analysis of Real-Time Embeded Software*. ISBN 0-792-38382-6. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[LPB$^+$05] Jong-In Lee, Su-Hyun Park, Ho-Jung Bang, Tai-Hyo Kim, and Sung-Deok Cha. A hybrid framework of worst-case execution time analysis for real-time embedded system software. In *Proc. Aerospace Conference*, pages 1–10, Big Sky, MT, USA, March 2005. IEEE Computer Society Press.

[Lüd06] Frank Lüders. *An Evolutionary Approach to Software Components in Embedded Real-Time Systems*. PhD thesis, Mälardalen Univeristy, Department of Computer Engineering and Electronics, December 2006.

[Lun08] Kurt-Lennart Lundbäck. Rubus os reference manual – general concepts, August 2008.
`http://www.arcticus.se`.

[LY03] Qing Li and Caroline Yao. *Real-Time Concepts for Embedded Systems*. ISBN 1-578-20124-1. CMP Books, 2003.

[MA95] Syed M. Mahmud and Ansaf I. Alrabady. A new decision making algorithm for airbag control. *IEEE Transactions on Vehicular Technology*, 44(3):690–697, August 1995.

[MÅFN03] Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin. Software component technologies for real-time systems - an industrial perspective. In *Proc. WiP Session of Real-Time Systems Symposium (RTSS'03)*, Cancun, Mexico, December 2003. IEEE Computer Society Press.

[MÅFN04] Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin. Evaluation of component technologies with respect to industrial requirements. In *Proc. 30th Euromicro*

Conference, Component-Based Software Engineering Track, pages 56–63, Rennes, France, August 2004. IEEE Computer Society Press.

[MÅFN05] Anders Möller, Mikael Åkerholm, Joakim Fröberg, and Mikael Nolin. Industrial grading of quality requirements for automotive software component technologies. In *Proc. Embedded Real-Time Systems Implementation Workshop in conjunction with the* 26^{th} *IEEE International Real-Time Systems Symposium*, Miami, FL, USA, December 2005. IEEE Computer Society Press.

[MBD98] Yannick Monnier, Jean-Pierre Beauvis, and Anne-Marie Déplanche. A genetic algorithm for scheduling tasks in a real-time distributed system. In *Proc.* 24^{th} *Euromicro Conference (EUROMICRO'98)*, pages 708–714, Västerås, Sweden, August 1998. IEEE Computer Society Press.

[MBMB98] David Montana, Marshall Brinn, Sean Moore, and Garrett Bidwell. Genetic algorithms for complex, real-time scheduling. In *Proc. IEEE International Conference on Systems, Man, and Cybernetics*, pages 245–248, San Diego, CA, USA, October 1998. IEEE Computer Society Press.

[MG02] Kevin L. Mills. and Hassan Gomaa. Knowledge-based automation of a design method for concurrent systems. *IEEE Transactions on Software Engineering*, 28(3):228–255, March 2002.

[MGL06] Pascal Montag, Steffen Görzig, and Paul Levi. Applying static timing analysis to component architectures. In *Proc. International Workshop on Software Engineering for Automotive Systems (SEAS'06)*, pages 21–28, New York, NY, USA, 2006. ACM Press.

[MJ86] Paritosh K. Pandya Mathai Joseph. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[MMH$^+$05] Sibin Mohan, Frank Mueller, William Hawkins, Michael Root, Christopher Healy, and David Whalley. Parascale: Exploiting parametric timing analysis for real-time schedulers and dynamic voltage scaling. In *Proc. 26th IEEE Real-Time Systems Symposium (RTSS'05)*, pages 233–242, Miami, FL, USA, December 2005. IEEE Computer Society Press.

[Moh04] Parastoo Mohagheghi. *Impact of Software Reuse and Incremental Development on the Quality of Large Systems*. PhD thesis, Norwegian University of Science and Technology NTNU, September 2004.

[Moh06] Elsadig Yousif Mohamed. How to conduct scientific research? *Sudanese Journal of Public Health*, 1(2):144–147, April 2006.

[MT05] Jukka Mäki-Turja. *Engineering Strength Response-Time Analysis - A Timing Analysis Approach for the Development of Real-Time Systems*. PhD thesis, Mälardalen Univeristy, Department of Computer Engineering and Electronics, May 2005.

[MYZC06] Hui Ma, I.-Ling Yen, Jia Zhou, and Kendra Cooper. Qos analysis for component-based embedded software: model and methodology. *Journal of Systems and Software*, 79(6):859–870, June 2006.

[NAD$^+$02] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, P. Black, Peter O. Muller, T. Zeidler, Thomas Genssler, and Reinier van den Born. A component model for field devices. In *Proc. 1st International IFIP/ACM Working Conference on Component Deployment (CD'02)*, pages 200–209, Berlin, Germany, June 2002. Springer Verlag.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *Proc. Future of Software Engineering (FoSE'07)*, pages 55–71, Washington, DC, USA, May 2007. IEEE Computer Society Press.

[PD96] Rubén Prieto-Diaz. Reuse as a New Paradigm for Software Development. In *System Reuse: Issues in Initiating and Improving a Reuse Program*. London:Springer-Verlag, 1996.

[Pop08] Popcorn hour media streamer homepage, August 2008. `http://www.popcornhour.com/onlinestore/`.

[Pre01] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. ISBN 0-071-23840-9. McGraw Hill Higher Education, June 2001.

[Rap08] RapiTime WCET tool homepage, August 2008. `www.rapitasystems.com`.

[Ros01] William Rosenbluth. *Investigation and Interpretation of Black Box Data in Automobiles:A Guide to the Concepts and Formats of Computer Data in Vehicle Safety and Control Systems*, chapter A Review of Antilock Braking and Traction Control Systems, pages 72–79. ISBN 0-8031-2091-5. ASTM International, 2001.

[Roy70] Winston W. Royce. Managing the development of large software systems: concepts and techniques. In *Proc. IEEE WESTCON*, pages 1–9, Los Angeles, CA, USA, August 1970. IEEE Computer Society Press. Reprinted in Proc. Int'l Conf. Software Engineering (ICSE) 1989, ACM Press, pp. 328-338.

[RRC03] Michael Richard, Pascal Richard, and Francis Cottet. Allocating and scheduling tasks in multiple fieldbus real-time systems. In *Proc. Emerging Technologies and Factory Automation (ETFA'03)*, pages 137–144. IEEE Computer Society Press, September 2003.

[RT02] Ola Redell and Martil Törngren. Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter. In *Proc. 8th IEEE Real-Time Technology and Applications Symposium (RTAS'02)*, pages 164–172, San Jose, CA, USA, September 2002. IEEE Computer Society Press.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. ISBN 0-444-52726-5. Elsevier Science Inc., New York, NY, USA, 2006.

[SAE06] SAE J1939 Standards Collection, May 2006. `http://www.sae.org`.

[SEG^{+}06] Daniel Sehlberg, Andreas Ermedahl, Jan Gustafsson, Björn Lisper, and Steffen Wiegratz. Static wcet analysis of real-time task-oriented code in vehicle control systems. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, pages 212–219, Paphos, Cyprus, November 2006. IEEE Computer Society Press.

[SESW05] Jan Staschulat, Rolf Ernst, Andreas Schulze, and Fabian Wolf. Context sensitive performance analysis of automotive applications. In *Proc. Design, Automation and Test in European Conference and Exhibition (DATE'05)*, pages 165–170, Munich, Germany, March 2005. IEEE Computer Society Press.

[SFÅ04] Kristian Sandström, Johan Fredriksson, and Mikael Åkerholm. Introducing a component technology for safety critical embedded real-time systems. In *Proc. 7th International Symposium on Component-based Software Engineering (CBSE7)*, pages 194–208, Edinburgh, Scottland, May 2004. Springer Verlag.

[SH98] Mikael Sjodin and Hans Hansson. Improved response-time analysis calculations. In *Proc. 19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 399–408, Madrid, Spain, December 1998. IEEE Computer Society Press.

[Sha01] Mary Shaw. The coming age of software architecture resreach. In *Proc. IEEE 25th International Conference on Software Engineering (ICSE'01)*, pages 657–664, Toronto, Canada, May 2001. IEEE Computer Society Press.

[Sha02] Mary Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*, 4(1):1–7, November 2002.

[SLM02] Insik Shin, Insup Lee, and Sang Lyul Min. Embedded system design framework for minimizing code size and guaranteeing real-time requirements. In *Proc. 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, Austin, TX, USA, December 2002. IEEE Computer Society Press.

[Slo08] Neil J. A. Sloane. The on-line encyclopedia of integer sequences, number of "sets of lists", July 2008. http://www.research.att.com/˜njas/sequences/A000262.

[SR89] Jack A. Stankovic and Krithi Rammaritham. *Tutorial: Hard Real-Time Systems*. ISBN 0-8186-0819-6. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[Sta98] Jack A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, October 1998.

[SVK97] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12):pages 759 – 776, December 1997.

[SW00] Kang G. Shin and Shige Wang. An architecture for embedded software integration using reusable components. In *Proc. 4th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES'01)*, pages 110–118, Atlanta, GA, USA, November 2000. IEEE Computer Society Press.

[SWE] Swedish execution time tool. `http://www.mrtc.mdh.se/projects/wcet/`.

[SWE06] Wcet project homepage, August 2006. `http://www.mrtc.mdh.se/projects/wcet`.

[Sym08] Symta/p execution time analyzer, August 2008. `http://www.ida.ing.tu-bs.de/`.

[Szy98] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. ISBN 0-201-74572-0. Addison-Wesley, 1998.

[TBW92] Ken W. Tindell, Alan Burns, and Andy Wellings. Allocating hard real-time tasks (an np-hard problem made easy). *Real-Time Systems Journal*, 4(2):145–165, May 1992.

[Tin94] Ken W. Tindell. Adding time offsets to schedulability analysis. Technical Report YCS-94-221, Technical Report, Department of Computer Science, University of York, 1994.

[Tur02] Jim Turley. The two percent solution. Technical Report articleID 9900861, Embedded Systems Design, December 2002.

[VHMW01] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. Parametric timing analysis. In *Proc. ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems (LCTES'01)*, pages 88–93, Bird, UT, USA, June 2001. ACM Press.

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

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[vZELP98] Anton T. van Zanten, Rainer Erhardt, Klaus Landesfeind, and Georg Pfaff. Vdc systems development and perspective. In *SAE World Congress*, pages 424–444, Detroit, MI, USA, February 1998. SAE, SAE.

[WBS97] Wolfgang Weck, Jan Bosch, and Clemens Szyperski. Workshop report of 2^{nd} workshop on component-oriented programming. In *Proceedings Second International Workshop on Component-Oriented Programming (WCOP'97)*, pages 323–326, Jyväskylä, Finland, June 1997. Springer Verlag.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem — overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[Wei81] Mark Weiser. Program slicing. In *Proc. IEEE 5^{th} International Conference on Software Engineering (ICSE'81)*, pages 439–449, San Diego, CA, USA, March 1981. IEEE Computer Society Press.

[WKE02] Fabian Wolf, Judita Kruse, and Rolf Ernst. Timing and power measurement in static software analysis. *Microelectronics Journal*, 33(1):91–100, January 2002.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Design, Automation and Test in European Conference and Exhibition (DATE'05)*, pages 606–611, Munich, Germany, March 2005. IEEE Computer Society Press.

[WU93] Kajiro Watanabe and Yasushi Umezawa. Optimal triggering of an airbag. In *Proc. Intelligent Vehicles'93 Symposium*, pages 78–83, Tokyo, Japan, July 1993. Volvo AB.

[Yer96] Ramesh Yerraballi. *Scalability in Real-Time Systems.* PhD thesis, Computer Science Department, old Dominion University, Norfolk, VA, USA, August 1996.

[Zsc04] Steffen Zschaler. Formal specification of non-functional properties of component-based software. In *Proc. Workshop on Models for Non-functional Aspects of Component-Based Software (NFC'04) at UML conference 2004*. TU Dresden, October 2004. ISSN 1430-211X.

Appendix A

Complete list of tables

For each benchmark we show the number of analyses (**#Runs**) requried to reach a specific accuracy, and how many different value space partitions that resulted in (**#D**).

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%	8	5	16	11				
20%	40	17	32	19				
30%	48	22	180	53	2	2	8	6
40%	62	28	190	60	10	6	25	12
50%	80	37	204	64	24	13	53	26
60%	112	52	234	71	60	28	126	52
70%	160	75	268	87	132	59	220	68
80%	264	111	332	113	296	121	312	105
90%	622	127	616	132	392	323	578	139
100%	(96%)878	72	(96%)874	72	(96%)822	90	(96%)870	72

Table A.1: `jcomplex` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%			3	3			11	7
20%	2	2			4	3	20	10
30%	4	3	5	4	10	6	47	30
40%	8	5	20	13	16	8	58	28
50%	64	33	38	21	28	12	100	43
60%	114	58	226	104	46	23	112	44
70%	230	91	306	137	60	24	180	71
80%	432	141	518	172	138	55	256	102
90%	728	184	804	170	334	121	348	129
100%	1066	179	1072	169	1100	171	970	208

Table A.2: `lcdnum` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%							6	4
20%							27	10
30%	2	2	3	3	12		33	13
40%	8	4	5	4	26		73	24
50%	44	11	20	14	48	15	92	27
60%	46	13	29	20	98	25	250	69
70%	54	15	35	24	236	57	(64%)397	84
80%	90	23	38	26	(75%)448	94		
90%	122	25	(85%)52	32				
100%	(92%)304	50	(87%)328	95				

Table A.3: `nsichneu` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
20%			3	3			3	3
50%	2	2			2	2		
60%			5	4			5	4
70%	4	3			4	3		
80%			7	4			7	4
90%	8	3	11	4	8	3	11	4
100%	16	3	19	4	16	3	19	4

Table A.4: `ns` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%	18	3	5	3				
90%	32	3	7	3				
100%	34	2	9	2	2	2	2	2

Table A.5: `edn` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%					4	2	7	3
20%					6	3	9	5
30%							19	6
40%					8	5	27	10
50%					12	6	29	10
60%	2	2	2	2	14	6	31	9
70%	24	4	7	4	20	6	43	9
80%	32	4	9	4	32	10	51	9
90%	48	7	17	7	48	9	59	8
100%	72	5	23	5	56	7	73	7

Table A.6: `task1` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
20%	2	2	2	2	2	2	2	2
30%	4	3	5	3	4	3	5	3
40%			8	3			8	3
50%	6	3	10	3			10	3
60%	8	3	12	3	6	3	12	3
70%	10	3	14	3	12	3	14	3
80%	14	3	16	3	14	3	16	3
90%	16	3	18	3	16	3	18	3
100%	18	3	20	3	18	3	20	3

Table A.7: `task3` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
50%	2	2	2	2	2	2	2	2
60%			5	3			5	3
70%	4	3	69	11	4	3	198	10
80%	20	6	165	17	6	3	362	12
90%	298	20	(86%)2130	48	18	3		
100%					(94%)70	4		

Table A.8: `task4` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%	42	7	18	7	6	4		
20%	52	7			20	7		
30%	64	7			26	9		
40%	78	10	24	8	40	9		
50%	1018	10			92	11		
60%	1138	10	27	9	156	13		
70%	1332	10			216	14	9	5
80%	1486	9			344	15	21	7
90%	1662	11	30	8	584	15	77	12
100%	2166	9	39	6	1068	13	452	10

Table A.9: `task5` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%	(0%)2150	1	30	3	14	5	9	3
20%			702	11	160	36	83	26
30%			2097	14	(28%)2124	211	694	149
40%			(30%)2157	14			(33%)2125	358

Table A.10: `task7` benchmark

%Acc	Strategy Last Used				Strategy Next			
	Basic		Extreme		Basic		Extreme	
	#Runs	#D	#Runs	#D	#Runs	#D	#Runs	#D
10%							9	4
20%	2	2						
30%	4	3	3	3	2	2	26	11
40%	6	4	108	43	6	4	(35%)525	163
50%	8	5	525	52	12	8		
60%	650	13			40	10		
70%					(65%)650	13		

Table A.11: `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 fundamental for the thesis contributions, (ii) papers that are related to the thesis, but not directly to the thesis contribution, and, (iii) papers that are not related to the thesis. The papers that are not related to the thesis are results from cooperation with researchers from other research fields; often the contributions of these papers are somewhat outside my main research track. The papers are sorted chronologically under each category.

With the papers that are fundamental for the thesis contribution, I also state my contribution, and how the paper is used in this thesis. Table B provides an overview of how the papers relate to the chapters in this thesis.

Chapter	Directly contributing publications	Related publications
Chapter 1		
Chapter 2		
Chapter 3		20,23,24,29,30
Chapter 4		17,21,22,27,28
Chapter 5	2,4	11,13,23,30
Chapter 6	1,2,3,5	8,9,10,14,15,16,19
Chapter 7	2,6,7	18,23,25,27,
Chapter 8		
Chapter 9		

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 fundamental for the contribution of derivation of WCET input value combination.

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.

2: **Contract-Based Reusable Worst-Case Execution Time Estimate**, Johan Fredriksson, Thomas Nolte, Mikael Nolin and Heinz Schmidt, In *Proc.* 13th *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, pages 39-46, Daegu, South Korea, August, 2007. IEEE Computer Society Press. **Awarded Best Real-Time Paper.**

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 component is selected be 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 fundamental 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 contributor of the paper. The work has been performed in close cooperation 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 Proc. 7^{th} *International Workshop on Worst-Case Execution Time Analysis (WCET'07), in conjunction with the* 19^{th} *IEEE Euromicro Conference on Real-Time Systems (ECRTS'07)*, Pisa, Italy, July, 2007. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany

Abstract: For component-based systems, classical techniques for Worst-Case Execution Time (WCET) estimation produce unacceptable overestimations 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 Engineering (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 fundamental for the reusable 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, Andreas Ermedahl and Mikael Nolin. The paper was initiated by Johan Fredriksson.

4: **Reusable Component Analysis for Component-Based Embedded Real-Time Systems**, Johan Fredriksson and Rikard Land, In *Proc.* 29th *IEEE International Conference Information Technology Interfaces (ITI'07)*, pages 615-620, Dubrovnik, Croatia, June, 2007. IEEE Computer Society Press.

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 the positioning of the proposed methods in the CBSE development process and is fundamental for Chapter 5.

My contribution: Johan Fredriksson has been the main author and contributor of the paper. The work has been performed in close cooperation with Rikard Land. 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 Proc. *Workshop on Model and Analysis Methods for Automotive Systems (WMAAS'06) in conjunction with IEEE Real-Time Systems Symposium (RTSS'06)*, Rio de Janeiro, Brazil, December, 2006. IEEE Computer Society Press.

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 fundamental for the reusable WCET analysis contribution.

My contribution: Johan Fredriksson has been the main author and contributor of the paper. The work has been performed in close cooperation with Thomas Nolte and Mikael Nolin and Heinz Schmidt. The paper was initiated by Johan Fredriksson.

6: Optimizing Resource Usage in Component-Based Real-Time Systems, Johan Fredriksson, Kristian Sandström, Mikael Åkerholm In *Proc.* 8^{th} *LNCS International Symposium on Component-based Software Engineering (CBSE8)*, pages 49-65, St.Louis, MO, USA, May, 2005. Springer Verlag.

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 fundamental for the component to task allocation framework.

My contribution: Johan Fredriksson has been the main author and contributor of the paper. The work has been performed in close co-operation with Kristian Sandström.

7: **Transformation of component models to real-time models**, Johan Fredriksson, *Licentiate Thesis*, Mälardalen University, ISBN 91-88834-55-7, April, 2005. Mälardalen University Press.

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 fundament 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 fundamental for the component to task allocation framework.

My contribution: Johan Fredriksson has been the sole author and contributor of this Licentiate thesis.

Publications related to the thesis

8: **Context Aware Optimizations for Embedded Real-Time Components**, Johan Fredriksson, *Ph.D. Proposal*, Västerås, Sweden, September, 2007.

9: **Worst-Case Execution Time Clustering for Software Components**, Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, Mikael Nolin, *Technical Report*, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2007

10: **Reusing Worst-Case Execution Time Analysis with Component Contracts**, Johan Fredriksson, Thomas Nolte, Mikael Nolin, Heinz Schmidt, In *Proc. 9th Real-Time in Sweden (RTiS'07)*, Västerås, Sweden, August, 2007.

11: **Packaging Component-Analysis for Reuse**, Johan Fredriksson, Rikard Land, In *9th Real-Tine in Sweden (RTiS'07)*, Västerås, Sweden, August, 2007.

12: **Contract-Based Reusable Analysis for Software Components with Extra-Functional Properties**, Johan Fredriksson, Thomas Nolte, In *Proc. Work-In-Progress (WIP) session of 19th IEEE Euromicro Conference on Real-Time Systems (ECRTS'07)*, Pisa, Italy, July, 2007.

13: **The SAVE approach to component-based development of vehicular systems**, Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, Massimo Tivoli, *Journal of Systems and Software*, 80(5):655-667, May, 2007.

[1]The Swedish Licentiate Thesis is a "half time thesis" in the Ph.D. studies

14: **Contract-Based Reusable Worst-Case Execution Time Estimate**, Johan Fredriksson, Thomas Nolte, Mikael Nolin, Heinz Schmidt, *Technical Report*, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2007

15: **Contract-Based Reusable Analysis for Software Components with Extra-Functional Properties**, Johan Fredriksson, Thomas Nolte, *Technical Report*, Mälardalen Real-Time Research Centre, Mälardalen University, April, 2007

16: **Increasing Accuracy of Property Predictions for Embedded Real-Time Components**, Johan Fredriksson, Thomas Nolte, Proc. In *Proc. Work-In-Progress (WIP) session of 18th IEEE Euromicro Conference on Real-Time Systems (ECRTS'06)*, Dresden, Germany, July, 2006.

17: **Industrial Requirements on Component Technologies for Vehicular Control Systems**, Anders Möller, Mikael Åkerholm, Joakim Fröberg, Johan Fredriksson, Mikael Nolin, *MRTC report ISSN 1404-3041 ISRN MDH-MRTC-195/2006-1-SE*, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2006.

18: **A component-based development framework for supporting functional and non-functional analysis in control system design**, Johan Fredriksson, In *5$^t h$ Conference on Software Engineering Research and Practice in Sweden*, Västerås, Sweden, October, 2005

19: **Component-Based Context Dependent Hybrid Property Prediction**, Anders Moller, Ian Peak, Mikael Nolin, Johan Fredriksson, Heinz Schmidt, in Proc. of Workshop on Dependable Software Intensive Embedded systems (ERCIM), pages 69-75, Porto, Portugal, September, 2005. ERCIM.

20: **Component-Based Development of Safety-Critical Vehicular Systems**, Ivica Crnkovic, DeJiu Chen, Johan Fredriksson, Hans Hansson, Jörgen Hansson, Joel Huselius, Ola Larses, Joakim Fröberg, Mikael Nolin, Thomas Nolte, Christer Norström, Kristian Sandström, Aleksandra Tesanovic, Martin Törngren, Simin Nadjm-Tehrani, Mikael Åkerholm, *MRTC report ISSN 1404-3041 ISRN MDH-MRTC-190/2005-1-SE*, Mälardalen Real-Time Research Centre, Mälardalen University, September, 2005.

21: **Quality Attribute Support in a Component Technology for Vehicular Software**, Mikael Åkerholm, Johan Fredriksson, Kristian Sandström, Ivica Crnkovic, In *4th Conference on Software Engineering Research and Practice in Sweden*, Linköping, Sweden, October, 2004.

22: **Evaluation of Component Technologies with Respect to Industrial Requirements**, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin, In *Proc.* 30th *Euromicro Conference, Component-Based Software Engineering Track*, pages 56-63, Rennes, France, August, 2004. IEEE Computer Society Press.

23: **Introducing a Component Technology for Safety Critical Embedded Real-Time Systems**, Kristian Sandström, Johan Fredriksson, Mikael Åkerholm, In *Proc.* 7th *International Symposium on Component-based Software Engineering (CBSE7)*, pages 194-208, Edinburgh, Scotland, May, 2004. Springer Verlag.

24: **A Sample of Component Technologies for Embedded Systems**, Mikael Åkerholm, Johan Fredriksson, *Technical Report*, Mälardalen Real-Time Research Centre, Mälardalen University, November, 2004.

25: **Calculating Resource Trade-offs when Mapping Component Services to Real-Time Tasks**, Johan Fredriksson, Mikael Åkerholm, Kristian Sandström, In *4th Conference on Software Engineering Research and Practice in Sweden Linköping*, Sweden, October, 2004

26: **Achieve consistent mappings between component models and real-time models - Licentiate Thesis Proposal**, Johan Fredriksson, *Technical Report*, Mälardalen Real-Time Research Centre, Mälardalen University, June, 2004

27: **An Industrial Evaluation of Component Technologies for Embedded-Systems**, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin, *MRTC report ISSN 1404-3041 ISRN MDH-MRTC-155/2004-1-SE*, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2004.

28: **An Industrial Evaluation of Component Technologies for Embedded-Systems**, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-155/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2004

29: **Software Component Technologies for Real-Time Systems - An Industrial Perspective**, Anders Möller, Mikael Åkerholm, Johan Fredriksson, Mikael Nolin, In *Proc. Work-in-Progress (WiP) session of 24th IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December, 2003.

30: **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 Larses (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.

Publications not related to the thesis

31: **Handling Subsystems using the SaveComp Component Technology**, Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, Mikael Nolin, Thomas Nolte, John Håkansson, Paul Pettersson, In *Proc. 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, December, 2006.

32: **Application of Built-In-Testing in Component-Based Embedded Systems**, Mikael Åkerholm, Irena Pavlova, Johan Fredriksson, *Technical Report*, Mälardalen Real-Time Research Centre, Mälardalen University, May, 2006.

33: **A component-based development framework for supporting functional and non-functional analysis in control system design**, Johan Fredriksson, Massimo Tivoli, Ivica Crnkovic, In *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 368-371, Long Beach, CA, USA, November, 2005. ACM.

34: **A component-based approach for supporting functional and non-functional analysis in control loop design**, Massimo Tivoli, Johan Fredriksson, Ivica Crnkovic, In *Proc. 10th International Workshop* *on Component-Oriented Programming (WCOP'05)*, Glasgow, Scotland, July, 2005.

35: **A component-based development framework for supporting functional and non-functional analysis in control system design**, Johan Fredriksson, Massimo Tivoli, Ivica Crnkovic, *Technical Report*, Mälardalen Real-Time Research Centre, Mälardalen University, June, 2005

36: **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.

37: **Attaining Flexible Real-Time Systems by Bringing Together Component Technologies and Real-Time Systems Theory**, Johan Fredriksson, Mikael Åkerholm, Radu Dobrin, Kristian Sandström, In *Proc. 29th Euromicro Conference, Component Based Software Engineering Track*, pages 399 - 402, Belek, Turkey, September, 2003. IEEE Computer Society Press.

38: **On the Teaching of Distributed Software Development**, Ivica Crnkovic, Igor Cavrak (external), Johan Fredriksson, Rikard Land, Mario Zagar (external), Mikael Åkerholm, In *Proc. 25th International Conference INFORMATION TECHNOLOGY INTERFACES*, pages 237-242, Dubrovnik, Croatia, June, 2003. IEEE Computer Society Press.