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

A Context Aware Approach

Johan Fredriksson

June 2008

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

Real-time and embedded systems are integrated into products in many technology areas, e.g., in different kinds of automation systems controlling production and machines. As the complexity of software intensive systems grows, and more software controls these systems, more emphasis is put on producing dependable software. Dependability is not only important in safety-critical systems such as cars, nuclear plants and heart-lung machines; however, dependability also become increasingly important for non-safety critical systems, whose unreliability may be detrimental to their business value, e.g., consumer electronics. One important class of dependability requirements in embedded systems is real-time requirements because embedded systems typically react on the environment and have to respond within a bounded interval in time. At the same time as developers of these systems want to keep production and development costs to a minimum, more functionality is added for increasing market competitiveness. Thus, the increasing number of functions along with added complexity places a demand on better development methods, models and tools.

Component-Based Software Engineering (CBSE) is a development strategy where engineering aims at systematic reuse of pre-built software components. It is one of the fastest growing development strategies in industry because of its promise to, lower costs, increase maintainability and shorten time-to-market. However, CBSE comes with the cost of greater initial investments for creating more general and reusable components, and software components only justifies its investment when they are deployed in several environments. The performance and resource usage of software components heavily depends on the environment of the component. Because of lack of models and tools for analysing predictability and resource consumption in relation to components, CBSE has not yet been as successful in the embedded domain as in, e.g., the desktop systems domain.
To tackle these problems, this thesis present two novel approaches for improving predictability and utilization of resources in embedded component-based systems. The first approach we present is 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 by means of component contracts over the input domain. The second approach we present is a framework for finding resource-efficient mappings between component-models and real-time systems. Few component technologies today consider the mapping between components and run-time tasks, while maintaining stipulated real-time constraints. Effective mappings can reduce memory usage and CPU-overhead considerably.

We have implemented two tools and evaluated the techniques with both industrial software and academic benchmarks.
Foreword
List of publications (short)

The full list of publications is presented in Appendix B. The following is the list of publications that are specially used in this thesis. The papers are categorized in publications that have a direct relation to the contributions of the thesis, and papers that contribute the thesis. The papers are presented in reverse chronological order.

Publications related to the thesis contributions

1: Deriving the Worst-Case Execution Time Input Values, Andreas Ermedahl, Johan Fredriksson, Peter Altenbernd, submitted to the 29th IEEE Real-Time Systems Symposium (RTSS’08).


Publications related to the thesis


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

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


## Contents

### I  Thesis

#### 1 Introduction

1.1 Embedded real-time systems .......................... 3
1.2 Component-based software engineering ............... 6
1.3 Informal problem formulation .......................... 7
   1.3.1 Components and real-time ....................... 7
   1.3.2 Resource efficiency ............................. 8
   1.3.3 Developing embedded real-time systems ......... 9
1.4 Overview of our solutions ............................ 10
   1.4.1 Reusable WCET analysis ......................... 10
   1.4.2 Derivation of WCET input combinations .......... 11
   1.4.3 Allocation to tasks .............................. 11
1.5 Summary of results .................................. 12
1.6 Contributions ....................................... 13
1.7 Thesis overview ..................................... 13

#### 2 Embedded real-time systems

2.1 Embedded systems - general concept ................. 17
   2.1.1 Embedded systems characteristics .............. 18
2.2 Real-time systems - general concept ............... 19
   2.2.1 Classification .................................. 21
2.3 Real-time model ........................................ 22
   2.3.1 Task model ..................................... 23
   2.3.2 Resource model .................................. 24
   2.3.3 Scheduling policies .............................. 24
2.4 Real-time analysis .................................... 25
2.5 Schedulability analysis ............................... 25
## Components

### 2.5 Utilization based analysis
- 2.5.1 Utilization based analysis  
- 2.5.2 Demand based analysis  
- 2.5.3 Response time analysis

### 2.6 Worst-case execution time analysis
- 2.6.1 Classification of WCET analysis

## Component-based development for ERTS

### 3.1 Motivation
### 3.2 Component reuse
### 3.3 Basic Definitions in CBSE
### 3.4 CBSE development process
### 3.5 Component model
### 3.6 Component definition
  - 3.6.1 Component interaction
  - 3.6.2 Component interface
  - 3.6.3 Component composition
  - 3.6.4 Component contracts
### 3.7 Component technology
### 3.8 Component frameworks
### 3.9 Discussion

## Research problem

### 4.1 Research introduction
### 4.2 Specific research goal
### 4.3 Research method
### 4.4 Industrial problem
  - 4.4.1 General industrial problem
  - 4.4.2 Specific industrial problem
### 4.5 Research setting
  - 4.5.1 Limiting the problem
  - 4.5.2 Research problems
### 4.6 Requirements
### 4.7 Partial solution proposals
  - 4.7.1 Resource aware development
  - 4.7.2 Input sensitive WCET analysis
  - 4.7.3 Transformation from components to real-time tasks
### 4.8 Validation
  - 4.8.1 Resource aware development
  - 4.8.2 Input sensitive WCET analysis
# Contents

4.8.3 Transformation from components to real-time tasks 68  
4.9 Discussion 69  

5 Resource aware development 71  
5.1 Component-based software engineering 72  
5.2 Reusable analysis 72  
5.3 Mapping components to tasks 76  
5.4 System model 77  
5.5 Discussion 82  
5.5.1 Contributions 83  
5.5.2 Related Work 83  

6 Input sensitive execution-time analysis 85  
6.1 Input sensitive WCET analysis 86  
6.1.1 Input value space partitioning 86  
6.1.2 Analysis assumptions 89  
6.2 Reusable WCET analysis 90  
6.2.1 Component WCET analysis 92  
6.2.2 Algorithm description 95  
6.2.3 Approaching parametric WCET 99  
6.2.4 Algorithm complexity 102  
6.3 Finding WCET input combination 103  
6.3.1 Algorithm description 103  
6.3.2 Example 105  
6.3.3 Algorithm complexity and back-tracking 107  
6.4 Approaches for faster termination 107  
6.4.1 Extreme value heuristic 108  
6.4.2 User interaction 109  
6.5 Discussion 111  
6.5.1 Contributions 112  
6.5.2 Related Work 112  
6.5.3 Future work 114  

7 Transforming components to real-time tasks 115  
7.1 Introduction 116  
7.2 Allocating components models to real-time models 117  
7.3 Allocation framework 117  
7.3.1 Constraints on allocations 118  
7.4 Using the framework 122
## Contents

7.5 Evaluation ........................................ 123  
7.5.1 Fitness function ................................. 123  
7.5.2 Simulation set up .............................. 124  
7.6 Discussion ......................................... 126  
7.6.1 Contributions .................................. 126  
7.6.2 Related work ................................. 127  
7.6.3 Future work ................................. 127  
8 Empirical results .................................. 129  
8.1 Input sensitive execution-time analysis ......... 129  
8.1.1 Reusable WCET .............................. 129  
8.1.2 Finding WCETs .............................. 139  
8.2 Allocating components to tasks ............... 141  
8.2.1 Results ................................... 142  
8.3 Evaluation tool ................................. 146  
9 Summary and Conclusions ................. 149  
9.1 Summary ..................................... 149  
9.2 Applicability .................................. 150  
9.3 Conclusion .................................... 150  
9.4 Contributions .................................. 151  
9.5 Possible future research directions .......... 153  
9.5.1 Implications for WCET analysis .......... 154  
A Complete list of tables and graphs .......... 157  
A Complete list of publications ............. 159  
Bibliography ...................................... 167
I

Thesis
Chapter 1

Introduction

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

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

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

1.1 Embedded real-time systems

Many modern products have impressive capabilities, take as example a modern car with functions like Electronic Damper Control (EDC) and Adaptive Cruise Control (ACC). Only two decades ago such functions were impossible to achieve, only relying on mechanical solutions. New advanced functions are possible because mechanical systems are being replaced by electro mechanical systems controlled by software. Consider the Electronic Stability Control (ESC) which is an advanced function in modern cars [vZELP98]. ESC is a
technology that improves the vehicle’s handling by detecting and preventing skids. This function is possible because mechanically controlled brakes are replaced by computer controlled brakes where each wheel can be individually braked. The power of software controlled systems has lead to modern cars having up to 90 onboard computers controlling the different functions in the car. Such computer systems that are embedded in apparently non-computerized electrical and electro-mechanical devices are known as embedded systems, and constitutes more than 99% of all computers in the world [Tur02, htt06]. 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.

Embedded systems comprise electronics and software operating to adapt to, or control, its environment. Embedded systems are different from desktop computers in the sense that they do not commonly have a screen or keyboard for interaction, but rather have different inputs for analog and digital sensors, and, different types of communication buses. The embedded computers are often refered 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. The sensors are typically 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 (sideway) acceleration of the car. The ECU continuously reads the sensors to determine if the car is understeering ((B) in Figure 1.1) or oversteering ((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 oversteering or understeering the correct brake action is calculated and applied for each wheel individually together with reducing the engine power.

The ESC system relies on that observation and action are performed with a predictable timing pattern. The timing pattern usually comprise an exact periodicity \( (T) \) and a last finish time, i.e., a deadline \( (D) \) when all calculations and actions need to be finished. Typically an ESC system is triggered periodically observing the environment every 40 milliseconds, and the brake force should be applied within a few milliseconds. In order to prove that the system really fulfills 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. The WCET is used for calculating if the software will finish before its stipulated deadline.

Systems that rely on time to function correctly are called Real-Time Systems (RTS). A definition that is commonly cited in literature is one by Stankovic [SR88]:

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.

Development of software for embedded RTSs is considered a complex and difficult task, both due to the additional requirements imposed by such systems but also because of the inherent inobservability of embedded systems as they normally lack human machine interfaces (screen and keyboard). Software gives an increasing possibility for advanced functions and adaptive behaviour and has become the primary means for creating added value for customers, for instance software in cars help reduce gas consumption as well as increase performance, comfort and safety. As a result systems become increasingly software intensive, for example, the next generation of premium auto-
mobiles are estimated to carry around one gigabyte of binary code [ABBP98].
This is comparable to what a typical desktop workstation runs today. Reasons
for this tremendous increase include the demand for new functionality on the
one hand, and the availability of powerful and cheap hardware on the other
hand [PBKS07, ABBP98].
In addition customers expect new embedded systems to enter the market
faster, at lower prices, and the competition for customers is tough. The decreasing
time to market and increasing product differentiation leads to that software
is required to be flexible enough for rapid reuse, extension and adaptation of
system functions. As a result the trends in the embedded systems sector are:

- embedded RTSs become increasingly software-intensive [FdN08].
- costs shift from hardware to software [CAPD02].
- individual functions integrate increasing functionality over different projects [CL01].

1.2 Component-based software engineering

To cope with the decreasing time-to-market and the increasing software com-
plexity, designers are looking for new ways of building systems. The notion
of reuse has gained increasing interest as software complexity grows. How-
ever, 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 notion of the structured reuse of-
fered, and the possibility of integrating software from other vendors, i.e., third
party composition. One 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 com-
posed 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. With
other words, the component model is a set of rules governing how the compo-
nents may or may not be used. The composition of components is the process
of assembling components to form an application. Components are composed
to constitute systems by connecting their interfaces according to the rules de-
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 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.

1.3 Informal problem formulation

As software complexity increases, software reuse becomes interesting. Because software is “soft”, it is easy to create tailored software that exactly fulfills 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 fulfills all component 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 each 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].

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

1.3.1 Components and real-time

One of the most important activities for RTA is WCET analysis. There are many theories and tools for performing such analysis [SWE, Rap, aiT, Br, 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 us inherent in that the analysis is context and usage unaware; and
components are typically deployed in different contexts with different usage, and the usage can vary a lot between different contexts. Therefore, for components that are reused in different systems it is today often not very meaningful to perform WCET analysis before the complete system has been designed, and each components’ usage has been determined.

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

Summarizing the above:

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

### 1.3.2 Resource efficiency

In order to further support resource efficiency it is important to consider the transformation from components to real-time tasks. Components are often directly transformed to real-time tasks partly due to the ease of directly mapping timing constraints from components to tasks, and, partly due to that it is not trivial to find an allocation between components and tasks such that the stipulated timing requirements can be fulfilled. Transforming components to 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 transformed to a single task the WCET prediction error of each task is the same as the error of the component. When several
components are mapped to one task, the error scales with the number of components, and the error can become quite large. The total system error stays the same but greater errors of individual tasks have a greater impact on properties like input jitter and output jitter, just to mention a few.

In other words:

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

- Transformations from components to tasks must maintain the component architecture.

- Transformations from components to tasks must consider temporal constraints.

1.3.3 Developing embedded real-time systems

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

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

- Prediction of execution-time of components with respect to component usage.

- Classification of execution times with respect to usage.

- Transformation of components to tasks with respect to real-time constraints.

- Optimization of system properties with respect to resource efficiency.
1.4 Overview of our solutions

In this thesis we study the problems of reusable WCET analysis and and optimization of resource utilization for software components for embedded RTSs.

The first problem is that we need reusable predictions in order to carry on with the second problem, which is in relation to resource usage, where the inaccuracy of WCET accumulates when several components are allocated to one real-time task.

We present three solutions that help tackle this problem. 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 real-time property WCET for reusable software components. The second solution is an application of the first solution, where we derive the input combination that triggers the WCET path.

These two problems are (i) reusable and tight prediction of the real-time property WCET, and (ii) the transformation of components to real-time tasks for improved resource utilization and maintained real-time constraints.

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:

Component WCET analysis Analyzing the WCET of the component with respect to many different general usage scenarios (inputs).

Clustering WCETs Clustering inputs that lead to similar execution times.
1.4 Overview of our solutions

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.

1.4.2 Derivation of WCET input combinations

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

WCET analysis Analyzing the WCET with respect to a large set of input combinations

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

Backtracking To explore all possible WCET input combination candidates.

1.4.3 Allocation to tasks

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

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

System property calculation The properties stack usage and CPU-overhead are calculate for each allocation.

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

This problem is difficult by its nature and 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 48% and 32% respectively, compared to allocating one component to one task.

1.5 Summary of results

In this thesis we show...
1.6 Contributions

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

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

C1 Models and techniques for increasing accuracy with respect to WCET analysis.

C2 Derivation of WCET input values.

C3 A framework for predicting WCET of software components with respect to usage, and derive WCET input combinations.

C4 Methods and a framework for mapping components to tasks with respect to minimizing resource consumption while maintaining real-time constraints.

C5 Integration of the proposed methods in the CBSE development process.

C6 Validations of the two frameworks with respect to increased resource utilization.

1.7 Thesis overview

The thesis structure is depicted in Figure 1.2. Note that the chapters 6 and 7 are shown as parallel in the thesis outline. This is because they are two separate solutions, that are collaborative in the sense that they tackle the same problem. However, they can also be seen as, and used as, two separate solutions.

Chapter 1, introduce the reader to the particular questions this thesis seeks to answer. We discuss the motivation and objective of this thesis: to research and develop methods to facilitate component-based development in embedded real-time systems by increasing resource efficiency and analyzability.
Chapter 1. Introduction

<table>
<thead>
<tr>
<th>Contrib.</th>
<th>CBSE for ERTS</th>
<th>Resource efficiency</th>
<th>Development of ERTS</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1, C3</td>
<td>We create techniques for predicting WCET that can be reused through parameterization, yet with high accuracy.</td>
<td>Higher accuracy of predictions allows the developer to dimension hardware correctly.</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>C2</td>
<td>Higher accuracy of predictions allows the developer to dimension hardware correctly.</td>
<td>n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>Helps to systematically transform components models to real-time models, something that otherwise often is performed ad-hoc.</td>
<td>Minimizing system overheads through efficient transformations from components to tasks, while at the same time maintaining both real-time requirements component architecture.</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>C5</td>
<td>Validation of the methods.</td>
<td>We integrate both WCET analysis and model transformation in the component based development process.</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td>Validation of the methods.</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: The relation between contributions and the problems
Chapter 2, provides the reader a theoretical background to real-time systems 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 gives a critical survey of the current state of research.

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

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

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

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

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

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

Appendix A, presents an extended set of data and graphs.

Appendix B, summarizes all publications, and reports written during the phd studies.
Figure 1.2: An overview of the chapters in the thesis
Chapter 2

Embedded real-time systems

In this chapter we give an introduction to embedded and real-time systems. We describe the terminology and definitions, used throughout this thesis.

2.1 Embedded systems - general concept

We do not need to search far to find an example of an embedded real-time system in a modern everyday appliance. For example in a modern vehicle, the engine is controlled by a real-time system, 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) [Ros01] are controlled by a real-time system, continuously monitoring and controlling the brakes to ensure the maximum braking effect. In the unlikely case of a collision, an embedded real-time system will detect the impact and deploy the airbag at exactly the right point in time [Cha02]. What is common to all these systems is that they are parts of a bigger system and their actions have to be delivered at specified instants in time. If they fail to deliver their services at the right time, the consequences can lead to low performance, material damage or in the worst scenario, loss of human life.
There are several definitions of embedded systems. IEEE [IEE92] has provided a common definition of embedded systems as follows:

\[ A \text{ computer system that is a part of a larger system and performs some of the requirements of that system; for example a computer system used in an aircraft or rapid transit system. } \]

This definition is quite vague since it only states that an embedded system is part of a larger system. Li and Yao [LY03] defines an embedded systems as follows:

\[ \text{Embedded systems are computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function.} \]

This definition includes the additional information that software is tightly coupled with hardware and that both are designed to perform a dedicated function.

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

2.1.1 Embedded systems characteristics

As processors become more powerful and inexpensive they become attractive for use in new areas. Computer controlled systems replace mechanical or electro-mechanical systems. The systems also become more sophisticated and complex since software allows for new advanced functionality that could not be implemented with mechanical solutions.

However, embedded systems can not simply be seen as a scale down version of desktop systems. This is due to requirements that are not regarded in desktop applications, such as low memory utilization, low processor overhead and predictability. Also, embedded systems are unique in the sense that they interact with the “real” physical world that we live in. Classical computer systems are designed in a classical “the faster the better” approach. For embedded systems, time is usually divine in
the sense that if the timing is wrong the whole system function may be wrong. Many embedded systems are safety-critical because they control critical applications in our society. If these applications malfunction they can have disastrous consequences. Some examples are heart-lung machines, or an air-bag system in a car. These types of systems are usually referred to as embedded *real-time systems*.

### 2.2 Real-time systems - general concept

Embedded real-time systems usually controls its environment by:

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

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

![Figure 2.1: Simple real-time model](image)

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

A common given example of a hard real-time system where predictability is very important, is an airbag in a car. An airbag system 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 Airbag control unit, and this determines the angle of impact, the severity, or force of the crash, along with other variables. Depending on the result of these calculations, the ACU may also deploy various additional restraint devices, such as seat belt pre-tensioners, and/or airbags (including frontal bags for driver and front passenger, along with seat-mounted side bags, and "curtain" airbags which cover the side glass). [MA95]

The greater the collision impact the earlier the airbag should be deployed. Because, the airbag has to be deployed before an occupant moves forward 125 mm relative to the car. Normally it takes 30 milliseconds for an airbag to be deployed after it gets a trigger signal from the airbag sensor. Thus, an airbag sensor is to be designed in such a way that it can send a trigger pulse to the airbag deployment circuit 30 ms before the time when the occupant's head moves forward five inches with respect to the car. For a crash at 50 km/h the airbag should be triggered between 10 ms to 20 ms after the crash. [Cha02]

Timing is decisive to achieve maximum protection. The airbag must be opened in the right millisecond. 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. The timespan for inflating the airbag correctly depends on many variables, and the timespan is quite small, a few milliseconds [WU93].

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

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 [SR88]:

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 real-time systems that are classified according to the criticality of a failure. Mission critical real-time systems, where a failure is considered to be a fault are denoted hard, while real-time systems where failures can be accepted are denoted soft.

**Hard real-time systems**

Hard real-time systems are systems where the consequences of a failure to meet all constraint is a fatal fault. For hard real-time applications, the system must be able to handle all possible scenarios, including peak load situations. Thus, the worst-case scenario must be 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 real-time systems are systems where some requirements may be violated to some defined extent. Late completion is undesirable but generally not fatal. Occasional missed deadlines or aborted execution is usually considered tolerable. The constraints are often specified in probabilistic terms.
A RTS is said to be soft if missing a deadline decreases performance of the systems, but does not jeopardize its correct behaviour [But97].

According to these definitions most RTS are soft, except some safety critical systems, such as airbags in cars. It is sometimes proposed that catastrophic consequences can be from a business perspective, causing several other systems under the hard RTS category, such as multimedia systems, where poor quality of service is catastrophic for the business.

2.3 Real-time model

Real-time software is often divided into so called tasks that execute a piece of software. Each task have some properties and requirements. The requirements are typically a periodicity, a longest allowed delay (often referred to as deadline) and a priority, defining its execution order in a system with mutiple tasks. Each task also have 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 is decided by the scheduling policy and the operating system. To be able to schedule the components they must be transformed into tasks conforming to the specified rules of the scheduler and properties must be set, e.g., period, priority, deadline etc. To ensure that the software fulfills the stipulated requirements, real-time analysis have to be performed.

So, 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 is 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 and embedded systems domain there exist many theories, methods and tools. These methods use a number of real-time properties, such as worst-case execution time (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 $\tau_i$ is described by (and depicted in Figure 2.2):

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

![Figure 2.2: Task model](image)
• 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.

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 needs to allocate at least one active resource to execute, and the execution may depend on zero or more passive resources.

2.3.3 Scheduling policies

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

Off-line scheduling

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

**On-line scheduling**

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

### 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, before run-time, to 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 fulfills all timing requirements and constraints can be constructed, the system is, by definition, schedulable.

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

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

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

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

\[
U \leq 0.69 \text{ as } N \to \infty
\]

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

2.5.2 Demand based analysis

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

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

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

2.5.3 Response time analysis

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

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

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

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

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

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

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

Starting with $R_i^0 = C_i$ and iterating until $R_i^{n+1} = R_i^n$ is guaranteed to yield the smallest possible solution and thus the response time for $\tau_i$ [SH98].

In order to guarantee convergence either 1) one must ensure a total task utilization is not greater than 100% or 2) one can stop iterating when $R_i^{n+1} > D_i$, i.e., a deadline violation has occurred.
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].

![Figure 2.3: Execution time analysis](image)

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 longest possible time for executing a program on a specific hardware platform. The different types of analyses are divided into static WCET analysis that performs a static analysis of the source code, producing an estimate of the execution time that is a sure to be longer than the actual WCET - it is said to produce a safe overestimation. Measurement based WCET analysis that measures the execution time during program execution. Measurement based analysis will report the longest observed execution time. It is not sure, however, that the absolute worst-case has occurred (Figure 2.3). Hybrid WCET analysis that uses both static and dynamic analysis to get a safe, yet tight WCET.
Finally, some approaches to parametric WCET exist, however, many of them suffer greatly from exponentially increasing complexity.

### 2.6.1 Classification of WCET analysis

We classify the different types of execution time analysis.

**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 the three phases of [Erm03, WE08+]:

- **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 sometimes may result in a cache hit.

Some static WCET analyses are input-sensitive, meaning that they are able to take constraints on possible input variable values into account when calculating the WCET estimate. In general, such analyses should be able to be more automatic and 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 mea-
measurement facility, such as oscilloscopes, logical analyzers, or hardware trace mechanisms to derive the timing of the program or parts of the program [BCP03b, 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 hybrid methods. An advantage of the hybrid approach may be that selection of test cases and control of coverage are well-known techniques in software engineering.

Hybrid WCET analysis

Hybrid analysis methods combine measurements and static analysis. The tools use measurements 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 [Rap06] and SymTA/P [Sym]. 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. Actually, hybrid methods may also overestimate the WCET, since measurements from mutually exclusive parts of the program may be combined in the final WCET. RapiTime is able to either analyse source code, adding instrumentation points on the source code level, or, otherwise use binary readers and instrument the generated code.

Parametric WCET analysis

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

There are a few approaches to parametric WCET where the WCET is expressed as a formulae with respect to loops, e.g., Vivancos [VHMW01]
and Bernat [BB00, CB02] presents techniques that mainly parameterizes loop bounds. However, in many cases it is necessary to be able to express excluding and infeasible paths in order to get a tight and parametric WCET. Some attempts have been made to express the WCET fully parametric [Lis03, BL08, AHLW08, Hum06], but they suffer from exponential complexity with respect to the program size, because of state space explosion.
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 [CL02]. It is a promising approach for efficient software development, enabling well defined software architectures as well as reuse. Component technologies have been developed addressing different demands and domains, the most common technologies are perhaps Enterprise Java Beans and Java Beans from SUN, COM and .Net from Microsoft, and technologies implementing the CORBA standard as is from OMG. These technologies are used for desktop and distributed enterprise applications all over the world. However, these technologies are in general not used for all classes of systems. They are not used for (i) resource constrained systems; they are simply 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. Embedded systems can often be classified as combinations of (i), (ii) and (iii).

Adoption for the development of embedded real-time systems is significantly slower. Major reasons are that such systems 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 are inherently heavyweight and complex, incurring large overheads on the run-time platform; they do not in general address timeliness, quality-of-service or similar extra-functional properties that are important for embedded real-time systems.

Component technologies have been developed for particular classes of embedded real-time systems. Often, these have been done within development organizations, and their adoption outside these organizations is limited. To avoid heavy-weight run-time platforms, they mostly do not support run-time deployment of components and lack many services. Composition of components into a (sub)system is rather performed in the design environment, prior to compilation, thus enabling static prediction of system properties and global optimizations. Examples of such models include the Koala component model for consumer electronics [vOvdLK00], PECOS for industrial field devices [WGC+02], and PBO for robotics [SVK97b]. They have often been tightly coupled to a specific operating system or a specific domain and only few of them consider non-functional properties, and they have not been general enough to be adopted for use in other domains [MmFN03].

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

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

![Popcorn Hour](image)

**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. When software become complex, software reuse becomes more interesting due to the fact that software is
“soft”. 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 Component-Based Development (CBD) is that individual components are not specified and laid out according to existing other components that are supposed to integrate their services. Every single component is specified according to a more or less general requirements profile, so it can be reused and integrated in a number of different contexts. Generality is a key feature of components because they should be reused in many different contexts.

Some of the benefits of reuse are [BBB+00, BBD+00]:

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

**More stable code** a quantitative analysis has shown that the amount of modified code between releases is less in reused software components, than in non-reused.

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

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

**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 basis of Component-Based Systems (CBS) is naturally the component. A software component is a software entity that conforms to a component model and can be composed without modification [CL02]. 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 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, e.g., scheduling, and finally, 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 and CBD is that the CBD 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 surveyed 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 CBD processes.

### 3.5 Component model

The only way that a component can be distinguished from other forms of packaged software is through its compliance with a component model. However, no agreement on what should be included in a component model exists, but a component model should specify the standards and conventions imposed on developers of components. Common is that
Figure 3.2: CBSE development process
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

### 3.6 Component definition

The key concept of CBSE is that of software components, e.g., those pieces of software that may 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 does not say anything about requirements on the component model. The two definitions principally agree, since the requirement that components can be modified without modification can only be satisfied if interfaces and context dependencies are well defined and that compliance with a standard naturally supports composition by third parties.

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

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

Lüders [Lüd06] states that the definition is somewhat broader than the previous two, as “replaceable within its environment” is a weaker requirement than “subject to independent deployment and composition by third parties”. This terminology is also used by Crnkovic and Larson [CL02], 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:
3.6 Component definition

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

**Independent** A component should be independent - it should be possible to compose and deploy it 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.

**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.

**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 of all component interfaces have to be specified.

**3.6.1 Component interaction**

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

**Pipes and filters** The components in this style are called filters and each have a set of inputs and a set of outputs. The outputs of a filter can be attached to inputs of other filters via simple connectors called pipes. Typically, the filters transform streams of input data...
to streams of output data in an incremental fashion. An important constraint is that filters should be independent in the sense that they do not share state and each filter is unaware of the identities of the other filters it is connected to.

**Black board** The basic model of a blackboard system is composed of three main entities: the blackboard, a set of knowledge sources and a control mechanism [CL92]. The blackboard is a globally accessible database, which is shared by knowledge sources. It contains the data and intermediate solutions. The blackboard is structured as a hierarchy of abstraction levels, which determine where data is input and where solutions are collected. Partial solutions are associated with each level and may be linked to information on other levels.

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

### 3.6.2 Component interface

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

A component realizes an interface by providing services or entry points for data and control, this interface is called “provide 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 too in a “required interface”. A required interface specifies services that are required by, or data and control passed to, other components.
Provided interface  Defines the services or data and control entry points that are provided by the component.

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

3.6.3 Component composition

Composition is to bring together components so that they give the desired behaviour. The possibilities for composition should be defined by the component model. 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 probably at this moment predictions of run-time properties can be done by supporting tools.

3.6.4 Component contracts

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

Syntactic:  Conformance of functionality.

Behavioural:  Behavioural contracts describe the behaviour of the component with respect to the interfaces. Behavioural contracts can be realized with textual descriptions, formal methods or component pre and post conditions.

Synchronization:  Synchronization between components.

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

Behavioural contracts describe the functionality of the component, and are often realized as pre and post conditions allowing or disallowing certain in and outputs of the component. However, they can also specify
certain context dependencies. *Quality of Service contracts* describe the performance of the component, and guarantees certain quality of a service given a specific context.

### 3.7 Component technology

A component technology is a concrete implementation of a component model and consists of tools and models for supporting assembling of, and interoperation between components. A component technology should provide necessary run-time support for the components and mature component technologies often offer different development tools simplifying the engineering process. Roughly speaking, there are three lines of widely adopted component technologies: JavaBeans/EJB from Sun. COM/DCOM/COM+/.NET from Microsoft, and and CORBA/CCM from OMG.

### 3.8 Component frameworks

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

### 3.9 Discussion

During the last decade advances have been made in component-based development for desktop and internet applications. A few de-facto standards have completely transformed the way such software is developed. These standards are mainly Microsoft’s .NET, SUN’s Enterprise Java Beans and OMG’s Corba Component Model (CCM). Component models for embedded systems are usually designed with very domain specific requirements in mind [MÅFN05a]. 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 [WG+02], Rubus [LLL03], SaveComp [ÅCF+06] and many more. None of these however have yet been successful outside of their intended domain. Thus, for embedded systems it seems difficult to define de-facto standards due to highly diverging requirements on different industrial segments [Crn04b].

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 embedded real-time systems 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 behavior 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. There are very few component models that support general component properties at the same time as they are highly resource and run-time aware, and vice versa. Component models that are specifically designed for a particular group of systems are often not adaptable and general enough to be used in other systems or other domains. In order to achieve reuse, most component technologies of today intentionally do not consider the system context, e.g., inputs, hardware and run-time system. As a result performance prediction is often inaccurate.
Chapter 4

Research problem

In this chapter we describe the industrial and research problems, and we discuss our solutions and the research methodology we have used in order to solve the research problem.

The research problem stems from the general observation that CBSE has not been as successful for embedded systems as for, e.g., desktop systems. The general and overarching question that we ask is

\textit{Why has not CBSE been as successful for embedded real-time systems as for desktop systems, and what can be done to make CBSE more successful for the embedded systems area?}

We do not aim to answer this question, but we will explore this problem domain and provide knowledge to contribute to the answering of the question.

4.1 Research introduction

One of the most important aspects of embedded real-time systems is that they must exhibit a predictable timing behaviour; furthermore, these systems are often embedded in larger systems where resources are scarce. In order to meet the challenges of predictability and resource efficiency
in the increasing complexity of embedded software, developers need appropriate development and analysis tools at hand. One of the development strategies that industry is interested in, is CBSE, and especially the notion of reuse. However, because of lack of models and tools for analysing predictability and resource consumption in relation to components, CBSE has not yet been as successful in the embedded domain as in, e.g., the desktop systems domain.

There has been several attempts to develop component technologies for real-time systems. Examples of component technologies that have been developed for particular classes of real-time systems systems are PECOS [WG+02], PBO [SVK97b], RubusCM [LLL03], SaveCCT [ACF+06]. 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 is the future development strategy for software. The increasing complexity of software is an incitament for reuse, because the effort to re-create the software become higher and higher. For most systems in the industrial segment of embedded systems it is not enough to only consider functionality. Even if the functionality is reused it is still a lot of work to re-analyze the software in order to be able to make predictions on the software. Thus, there is potentially a lot of gain to be able to reuse also the extra-functional parts.

The specific goals of this thesis are

- 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 transformations from component models to real-time models.
  - reduce system CPU and mempry overhead while maintaining real-time requirements.
4.3 Research method

In this section we start with the 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 classify in two different categories, i.e., 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 in our research has been motivated by practical industrial problems. Thus, early on in the research we performed qualitative studies [MmFN04, ÅFSC04], trying to understand the problems in industry. As with most real problems they are complex, difficult to grasp and near impossible to solve. 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, i.e., to know that we really solve the problem we define.

Good research requires both a clearly stated problem, results, but also convincing evidence that the results are sound. In order to fulfil these criteria and approach the problem methodically, our research method is divided it into the five following steps inspired by Shaw [Sha01]:

1. Identify industrial problem (Section 4.4).
2. Transfer the industrial problem to an academic setting where we formulate three specific academic research problems P1, P2 and P3, as depicted in Figure 4.2 (Section 4.5.2).
3. Figure out a general solution to the different problems. (Section 4.6).
4. Describe a specific solution (Section 4.7).
5. Validate the specific solution (Section 4.8).

The structure of this chapter follows a logical reasoning that leads from an industrial problem → research problems → requirements → solution proposals → validation, this logical chain is depicted in Figure 4.1.
In the first two sections we describe our research methodology and the industrial problem. In Section 4.5.2 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.

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

### 4.4 Industrial problem

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

#### 4.4.1 General industrial problem

The industrial problem we discuss exists in the realm of embedded and real-time systems, slanted towards the automotive and vehicular domains. 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...
Due to these issues, many companies are looking for new development strategies that can handle these problems [But06]. CBSE is an approach that promise 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 in use in several domains, such as desktop, internet and business systems. Compared to these domains, the embedded systems segment struggles with different requirements. This gap between the current state of CBSE and the embedded systems requirements has to be lowered [HKK04].

**Problem statement S1:** The efficiency of reuse has been proven only outside of the embedded and real-time systems domain.

One of the classes of requirements that are imposed on embedded real-time systems are real-time requirements. Real-time systems have historically mostly been used in mission critical systems, such as automotive, factory automation or aerospace. However, also in machinery and con-
sumer electronics the use of real-time systems have become widespread to satisfy increasingly demanding customers [Bro06, PBKS07]. Real-time requirements originate from the fact that embedded software needs to interact with a physical environment. This has created a need for analytically being able to reason about the software behavior. For safety-critical systems in particular it is important to analytically show that the system will behave correctly. Examples of well-established uses of such analytical properties is the increasing use of WCET analysis, which is a critical activity for proving the correctness of real-time systems [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.

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 issues are not solved in relation to CBSE and the demanding requirements industry is facing on resource consumption, timeliness and reliability [BCC+03].

New legislation for machinery (2006/42/EEC) [SAE] has lead to that several domains that have been exempted the *machine directive* needs to follow this. The machine directive is ultimately about product safety, and the safety aspects must be integrated in all aspects, from construction to use. Machinery that are controlled by software need to prove correctness of the software with different type of software analysis, e.g., WCET analysis.

### 4.4.2 Specific industrial problem

In this section we further limit the general industrial problem to form a more narrow industrial problem.
When developing embedded RTS it is of main concern to make accurate predictions of component properties such as timing and memory consumption. At the same time it is desirable to gain from the development benefits offered by CBSE. The key activity that will bring the CBSE benefits is reuse. In order to support reuse in the CBSE development process, components are made general. Reuse traditionally only considers functional parts, leaving timing prediction to the system developer. 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 to much context-specific dependencies.

By reusing traditional WCET analysis results of a component, reuse is increased, but the analysis must be performed with respect to all possible contexts, resulting in inaccurate estimations. On the other hand, by analysing a specific system with the knowledge of the system context, the analysis results are more accurate estimations, but the key property reusability is decreased. Also timing prediction is a difficult and time-consuming activity, resulting in a higher development effort [KP, HK07].

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

An obstacle to combine predictability with correctly dimensioned hardware is the inaccuracy of the system analysis. Real-time analysis is based on worst-case assumptions, and the composition of overestimated worst-cases make the system impractically oversized and under utilized [Dur06]. Few component technologies methodically considers the transformation from components to run-time entities, leading to worse than possible resource utilization. A faulty transformation from components to runtime entities can even lead to violated real-time requirements [HMTN06, FSM05a].

**Problem statement S5:** Poor transformations from component to run-time entities may lead to inefficient resource utilization or violated real-time requirements.
Thus, there are several trade-offs (i) between the generality required by components for efficient reuse in the CBSE process, and the specific usage required by embedded RTS for achieving accurate predictions required by real-time analysis; and (ii) between resource consumption and real-time constraints when transforming components to real-time entities.

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. 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 to increase 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.

All research is built upon assumptions since we are limited in what we can test at one time. Some variables may not be measurable until later. By making assumptions the industrial problem is reduced to a simpler problem that can be tackled easier. In subsequent research, assumptions can be relaxed to make the problems and solutions more generally applicable.

4.5.1 Limiting the problem

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

Partial solution proposal 1
Use the CBSE development process for combining WCET analysis with structured reuse of both functionality and WCET.

Partial solution proposal 2
Combine WCET analysis with structured search over the input space to create input parameterizable component WCET contracts.

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

Requirements:
R4: Accurate analysis
R2: CBSE Process
R3: Reusable WCET analysis
R5: Resource efficiency
R1: Automation

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

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

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

Validation: Do the reusable analysis and component to task transformation we propose achieve sufficient resource utilization and predictability for reusable components in embedded real-time systems?

Industrial and academic problem formulations
Refinement
Innovation
Implementation
Many small embedded systems do not have complex interaction models, but rather require models that are simple and analyzable. Many component technologies for embedded systems therefore have chosen to limit the interaction model to the pipes and filter model. Thus, we limit the problem to only consider component technologies that use the pipes-and-filter interaction model.

Assumption: Pipes-and-filter component interaction.

In this research we do not aim to develop yet another WCET analyzer. Therefore we use existing tools. We also assume that a usage is always known for a component in a specific system, and the analysis is performed with respect to this usage. The analysis should be accurate with respect to that usage; however, we do not make any assumptions on the accuracy of the usage though.

Assumption: Input-sensitive WCET analysis is available.

Assumption: Components usage is known for each specific context.

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.

Assumption: Known and predictable context switch time 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.

Context is a word of many meanings, it may incorporate (from a component centric view) physical environment, input stimuli, hardware, collaborating components and much more. In this thesis we only consider some parts of the context, one thing that we do not consider is different variants of hardware. This, of course limits the reuse for systems with the same hardware.

Assumption: Invariant hardware.

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.

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. 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.

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 fulfil 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 industrial problem lies in the component-based development of real-time systems, where general and context-unaware software components meets requirements on accurate timing predictions and low resource consumption.

The research problems describe one of many possible views of the industrial problem. We do not claim that our view is more correct than any other or that we cover all aspects of accurate timing predictions or low resource consumption. Huwever, the research problems reflects the identified industrial problem.

The statements S1-S5 describe 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 is reason to find a solution to the trade-off. We continue by deriving a set of research problems from the statements S1-S5.

There are many incitaments to reuse software components in a structured way to lower, among many things, time-to-market and development costs. Reuse has been proven efficient for some domains within software engineering, however, the domain of embedded and real-time
systems has 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 the embedded and real-time domain, it is necessary to consider the development of components. ...

It is necessary in the development 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 and 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 have 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 behavior of a component without knowing its intended use may lead to very inaccurate predictions.

To provide evidence of predictable behavior for real-time systems, 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 usage, i.e., how it is used in that particular setting. If WCET is predicted without respect to context the predictions become much too inaccurate and pessimistic. 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 [GESL06a, KP05]
• Accurate analysis is required for correctly dimensioned hardware, and correct system behaviour [Dur06].

Although predictability is one of the most important aspects of embedded and real-time systems, it is also important with resource efficiency in order to not use over dimensioned 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 transformation from components to run-time system must be considered.

Components are reused at design and development of a system. However, transformations between components and run-time systems, is not reused. Each system needs to be transformed to a run-time system. The component-based system must also be transformed to fit a specific run-time system with its specific attribute assignments. Thus efficient methods and tools are required for this specific task. Improper transformations from components to run-time entities may both reduce resource efficiency, and violate system properties.

Problem P3 Inefficient transformation between components and run-time system may reduce resource efficiency.

Statements : S2 and S5.

Motivation :
• It is often desired to keep resource consumption low in embedded real-time systems [Crn04a].
• 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 run-time tasks [MG02, KWS95].

4.6 Requirements

From the formulated problems we break down the problems and identify a set of key requirements. The rationale for defining a set of requirements from the research problems is to validate the solutions with respect to
the requirements in order to increase the confidence that our solutions actually reflects 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 embedded real-time systems, 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 considers to be important.

It is possible to tackle the problems P1-P3 with 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 in earlier studies that we have presented in, e.g., Möller et.al [MÅFN05b], 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

**Useable:** 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.

WCET analysis is both time consuming and difficult. By having tools that automatically derives a reusable WCET it increases the development is supported by increasing the useability, making WCET analysis simpler and more accessible.

**Requirement R1**  **Automation:** The WCET analysis shall automated as much as possible, requiring a minimum of human interaction.
Derived from: P1, Simplicity, Useable

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; however, to make sure that it is a safe estimation all possible uses must be considered. This will potentially lead to very inaccurate predictions. Thus, we want to reuse the WCET analysis, but we also want it to be as accurate as possible.

(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 run-time system conforming to a specific real-time model. We must be able to separate an efficient transformation from an inefficient. A common approach for transforming components to tasks is simply to view one component as one task.
\(\text{Requirement R5}\) \textbf{Resource efficiency}: Component shall be transformed to real-time tasks such that the resource efficiency is never lower than for a system that is mapped with one component to one task.

\textbf{Derived from:} P3, Resource efficiency

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

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

\textbf{Derived from:} P3, Predictable

The requirements are not exhaustive in the sense that they cover the problems, as depicted in Figure 4.5. The problems are too big and complex and it is only possible to find partial solutions to the problems.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.5.png}
\caption{Figure 4.5:}
\end{figure}

\section*{4.7 Partial solution proposals}

In [ÅFSC04, MmFN04] we have surveyed several component technologies for embedded and real-time systems, and investigated different methods that are commonly used with embedded real-time systems. Often predictable behaviour and resource efficiency are conflicting properties because many methods for lowering resource consumption are simply not suitable for embedded systems. 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 embedded system 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 fulfill 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 ... contract ... 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 increase resource consumption because of the generality needed for efficient reuse. However, how much lower prediction accuracy can we accept? 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., economics, 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:** Parameterize 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 previously stated assumptions, the component WCET can be reused for the same hardware. In future work it may be possible to also parameterize the contracts with respect to hardware to facilitate reuse over hardware boundaries.

*How does this solution fulfill requirements R4?* By gradually explore the program state space by using automatically derived annotations it is possible to perform as well as a fully 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 Transformation from 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 transformed and mapped to a run-time system. In real-time systems 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 mapping where one component is mapped to one task. This mapping is very simple because real-time analysis can be made early on in the development. However, the efficiency of the mapping is not very high due to high overhead.

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

**Fulfils requirements:** R5, R6

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

*How does this solution fulfill requirements R6?* The transformation framework has a set of rules for evaluating a mapping if it is feasible. In those rules lies also 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 mapping.

*How does this solution contribute to the overarching question?* By providing a framework for structured transformation from components to tasks resource utilization and real-time requirements are maintained from design to synthesis. Synthesis from components to real-time tasks is one step in the usage of CBSE in embedded real-time systems. Facilitating this step means facilitating the use of CBSE from embedded real-time systems.

### 4.8 Validation

One of the things that distinguishes good software engineering research from other 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 research with
respect to the industrial problem by validating the results with respect to the requirements that were derived from the industrial problem.

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

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

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

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

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

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

### 4.8.1 Resource aware development

**Validation of partial solution 1:** We develop extensions for the CBSE development process and reasons about the usage of the process together with the proposed methods. We are using extensive use of 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 other variables are considered we cannot guarantee external validity.
Internal validity: We disregard the internal validity; we do not see any threats because we have control over all variables, and we know their relationships.

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

Original research: Paper A, Paper B, Paper C
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 embedded real-time systems. 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.

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

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

Described in thesis: Chapter 6 and Chapter 8.

4.8.3 Transformation from components to real-time tasks

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

**Type of validation:** System implementation, empirical evaluation

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

**External validity:** The results are only generalizable with respect to our explicit assumptions. When other variables are considered we can not guarantee external validity.

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

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

**Original research:** Paper A, Paper D and Paper G

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

### 4.9 Discussion
Chapter 5

Resource aware development

As discussed in the previous chapters, an important research issue in the field of Component-Based Software Engineering (CBSE) for embedded and real-time systems is how to handle resources and predictability. This chapter positions the methods that are described in detail in chapters 6 and 7, in the component-based development process. We first introduce the CBSE process, and then describes the methods and their relationships to the process.

Traditional software development (e.g., the waterfall model [Roy70]) considers the system view, and develops a system where the system context and usage normally are 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 the component development process is focused on developing general components to be used in many different systems, while the system development process 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 embedded and real-time systems it is important to address, foremost, the issues of reusable predictability and low resource
consumption. Since the CBSE development process is significantly different from traditional development, it is important to address these issues in relation to the CBSE development process.

5.1 Component-based software engineering

The CBSE process is divided into two parts: a system development process and a component development process [CCL06]. To facilitate reuse, an important distinction between traditional software development and CBSE is that individual components are not specified and laid out according to existing other components that are supposed to integrate their services. Every single component is specified according to a more or less general requirements profile, so it can be reused and integrated in a number of different contexts. Generality is a key feature of components because they should be reused in many different contexts. However, the interface between the component development and system development processes may be fairly complex. Existing components are surveyed in the system development process during the requirements phase, influencing the entire scope and direction of the system because the system is easier built with existing components. 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. Components and systems are developed independently in CBSE. The system and component development processes are similar but separated. Most of the interaction between the two development processes are performed during the requirements and design phases, and the verification phase.

5.2 Reusable analysis

WCET analysis has been used in traditional software development for embedded real-time systems for a long time. However, to introduce reusable WCET analysis for CBSE, it must be assessed how the reusable
5.2 Reusable analysis

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

analysis can be incorporated in the CBSE development process. WCET analysis is usually performed in the verification activity of traditional software development processes. The analysis become 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 embedded software is of prime importance for verifying that the system will fulfil its real-time and resource constraints. Particularly important in real-time systems is to predict the worst-case execution times (WCETs) of tasks, so that it can be proven that task temporal constraints (typically deadlines) will be met.

The division between development of components and development of systems in the CBSE process implies that one efficient way of reusing WCET analysis is to position the analysis in the component development process, so that the results can be reused with the component in the system development process. By introducing an analysis process where part of the analysis is performed by the component developer the over-
all development process becomes arguably more efficient compared to traditional analysis.

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

On a high level the WCET analysis is divided into (i) the construction and analysis of the component where the component developer performs a reusable WCET analysis, and (ii) the development of systems and usage where the system developer uses the reusable WCET analysis on a specific system to find the specific WCET of each component for the system, as depicted in Figure 5.2.

Figure 5.2: Resource centric development process view.
Figure 5.4 shows how a general model for CBSE processes is extended with the reusable WCET analysis as part of the Verification activity, and the analysis results packaged with the component in the Release activity. These results are reused during the system WCET analysis, which is performed in the Verify phase of the component assessment process.

This development process allows for the effort of WCET analysis to be moved from the system development to the component development. In this way, not only the component itself but also the WCET analysis is reused several times. The system developer escapes the effort of learning and using advanced WCET analysis tools, and the overall process becomes more efficient. We deepen these discussions and discuss our results and findings in Chapter 6.
5.3 Mapping components to tasks

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.

The transformation from components to tasks depends on real-time properties in order to perform real-time analysis in guaranteeing that stipulated requirements are fulfilled. The transformation 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 mapping.

A problem in current CBSE development practices for embedded software is the mapping of components to tasks [KWS95]. Because of the real-time requirements on most embedded systems, it is vital that the
5.4 System model

We describe a general system model that is used throughout this thesis to reason about our methods.

Component characteristics

The component interaction model used throughout this paper is a pipe-and-filter model with transactions. Each component has a trigger; a time trigger or an event trigger or a trigger from a preceding component. A component transaction describes an order of components to be executed and defines an end-to-end timing requirement. In Figure 5.5, the notation mapping 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. Embedded real-time systems consist of periodic and aperiodic events, often with associated end-to-end timing requirements. Components triggered by periodic events 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 map components to tasks. A quite common way of mapping, which may have lead to the confusion between what are components and what are tasks, is the one-to-one mapping where one component constitutes one task. This mapping is used in many component technologies, e.g., Rubus[Lun, LLL03], PBO[SVK97a] and Autocomp[SFm04]. Components can be clustered, 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. 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. Special methods need to be used, and one such method is described in detail in Chapter 7.
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.

The component model chosen is relatively straightforward to analyse and verify. The pipe-and-filter interaction model is commonly used within the embedded systems domain. Many component models for embedded systems have the notion of transactions built in; however, if a component model lacks the notion of transactions, there are often possibilities 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 described with a tuple $\langle P_i, R_i, Q_i, m_i, f_i, \text{prog}_i \rangle$, where $P_i$ is the provided interface, which is a set of $\{p_{i,0}, p_{i,1}, \ldots, p_{i,n-1}\}$ input variables and $R_i$ is the required interface, i.e., a set $\{r_{i,0}, r_{i,1}, \ldots, r_{i,n-1}\}$ of output variables. An input variable can pass data or control. $Q_i$ represents the minimum inter arrival time (MINT) in the case of an external event and it represents the period in the case of a timed trigger. $f_i$ is a contract as a function with respect to a usage that returns the estimated WCET of the component with respect to a specific usage $U_i$ such that $f_i : U_i, pt_i \rightarrow WCET_i$, where $U_i$ is a usage profile and $pt_i$ is a probability threshold used for removing WCETs with low probability. $m_i$ is the amount of stack memory required by the component. $\text{prog}_i$ is the software behaviour of the component in some form, e.g., source code, graphical model or binary format.
Usage profile $U_i$ is a set of inputs and probabilities for those inputs. $U_i$ represents predicted inputs for the component in a specific context and usage. The usage profile is described in detail in Chapter 6.2.3.

Isolation set $I$ defines a relation between components that should not be allocated. It is described as a set of component pairs $I = \langle (c_a, c_b), (c_c, c_d) \rangle$ 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 a component has a highly uncertain WCET. The predicate $Iso(a, b)$ defines that components $c_a$ and $c_b$ has an isolation requirement, and should not be co-allocated.

Component Transaction $ctr_i$ is an ordered relation between components $N_i = c_a, c_b, c_c, ..., c_n$, 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. 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 the set $N = c_a, c_b, c_c$ can be formalized with the regular expression denoted in 5.1.

$$c_a \Sigma^* c_b \Sigma^* c_b \Sigma^* c_c$$

(5.1)

Where $\Sigma^*$ denotes all elements defined by $N$ 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 analysis it is required that a transaction consists of components with the same, or, harmonic period times.

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


Task characteristics

Our task model specifies the organization of entities in the component model into tasks. During the transformation from component model to task model, properties like schedulability and response-time constraints must be analysed in order to ensure the correctness of the final system. Components only interact through explicit interfaces; hence tasks do not synchronize outside the component model. The task model is for evaluating schedulability and other properties of a system, and is similar to standard task graphs as used in scheduling theory [DRW98].

Task \( \tau_n \) is a tuple \((Z_n, T_n, C_n, stack_n)\) where \( Z_n \) is an ordered set of components. Components within the same task are executed in sequence following the order of \( Z \) and with the same priority as the task. \( T_n \) is the period of the task. The parameter \( C_n \) is the estimated WCET. \( stack_n \) is the stack usage of the component. \( C_n, stack_n \) and \( T_n \) are deduced from the components in \( Z_n \). \( C_n \) is the sum of WCETs for all components in \( Z_n \). Hence, for a task \( \tau_n \), the parameters \( C_n \) is calculated with Equation 5.2. \( stack_n \) is the maximum of all components specified stack usage and is calculated with Equation 5.3.

\[
C_n = \sum_{i \in (c_i \in Z_n)} (f_i (U_i, pt_i))
\]

\[
stack_n = \max_{i \in (c_i \in Z_n)} (m_i)
\]

Task transaction \( ttr_i \) is a sequence of tasks \( O_i = \tau_1, \tau_2, ..., \tau_k \) and a relative deadline \( d_{ttr_i} \). \( O_i \) defines an ordered relation between the tasks, where in the case of \( O = \tau_1, \tau_2 \); \( \tau_1 \) is predecessor to \( \tau_2 \). The timing and execution order requirements of a task transaction \( ttr_i \) are deduced from the requirements of the component transactions \( ctr_i \). The task transaction \( ttr_i \) has the same parameters as the component transactions \( ctr_i \) but \( \tau_1, \tau_2, ..., \tau_k \) are the tasks that map the component \( c_1, c_2, ..., c_n \), as denoted in Figure 5.6. If several task transactions \( ttr_i \) span over the exact same tasks, the transactions are merged and assigned the shortest deadline. An event-triggered task may only appear first in a transaction. Two tasks can execute in an order not defined by the transactions. This depends on that
the tasks have different period times, and thereby suffer from period phasing; hence transactions cannot define a strict precedence relation between two tasks [WSMT+00].

Figure 5.6: Transactions from components to tasks

The task model specifies the organization of entities in the component model into tasks and transactions over tasks. During the transformation from component model to task model, extra-functional properties like response-time constraints must be respected in order to ensure the correctness of the final system.

System characteristics

The system consists of system parameters and a schedulable task set.

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

5.5 Discussion

CBSE in general is the emerging discipline of the development of software components and development of systems incorporating software components [CL02]. It is a promising approach for efficient software development, mainly through the activity reuse. In general, with some exceptions, component technologies are not used for (i) resource constrained systems; they are simply too demanding both in computing power and memory usage. They are not used for (ii) safety critical systems; it is hard to verify the functionality due to complexity and black box property of components. They cannot be used for (iii) real-time systems since they rely on unpredictable dynamic bindings and other complex run-time mechanisms. Embedded systems can often be classified as combinations of (i), (ii) and (iii).

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

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

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

In this chapter we have positioned our reusable WCET analysis, and the component to task mapping in the CBSE development process, in order to facilitate the usage of these techniques in relation to CBSE.

5.5.1 Contributions

The contributions presented in this chapter are

- The division of analysis in component analysis and system analysis.
- The extension of the CBSE process workflow for a context sensitive analysis and component-to-task allocation.

5.5.2 Related Work
Chapter 6

Input sensitive execution-time analysis

In this chapter we outline two novel methods, based on a combination of static WCET analysis and systematic search over the value space of input variables, for deriving the WCET behaviour of a software component. In particular:

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

We utilize the input sensitiveness of computer software components to express a relationship between their inputs and execution times.

The methods are evaluated in Chapter 8, and the results show that the relationship between inputs and execution times can be expressed with high precision. The method has been evaluated with respect to both industrial and academic software with different size and characteristics.
6.1 Input sensitive WCET analysis

A component may have inputs that affect the program flow (and thus the execution time), and inputs that do not affect the program flow (execution time). We only want to consider the inputs that do affect the program flow since this will leave us with a less difficult problem in the sense that there are fewer inputs to consider. 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 might cause the execution time to vary in two different ways:

(a) *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.

(b) *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 for example happen if the input variable values affect where in memory a certain load executes if 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 utilizes 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. 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.

6.1.1 Input value space partitioning

Given a set of component provided inputs \( \{p_{a,0}, p_{a,1}, \ldots, p_{a,n-1}\} \) for a component \( c_a \), the variables \( v_{a,0}, v_{a,1}, \ldots, v_{a,n-1} \) represent an input
value space (partition) as a set of input value combinations with respect to a set of constraints. The variables restrict the values that the component provided inputs may hold. An input variable is bounded by the type of the provided interface, i.e., if the provided interface input $p_{a,j} \in P_a$ is an 8-bit integer, then $0 \leq v_j < 255$.

**Definition 1.** An input variable $v_{a,j}$ is a set of possible values with a natural ordering for a component provided input $p_{a,j} \in P_a$.

Each variable can assume a number of values restricted by a constraint. The number of possible values is the domain size of the variable and is denoted $|v_{a,j}|$.

**Definition 2.** $|v_{a,j}|$ is the value domain size of the input variable $v_{a,j}$.

A value constraint $v_{c}$ can express any constraints, and several variables $v_{a,0}, v_{a,1}, ..., v_{a,n-1}$ can be affected. The value constraint $v_{c}$ limits the value domain of each variable such that the input variable $v_{a,i}$ with the constraints in value constraints $v_{c}$ is $v_{a,i}|_{c}$ and is a subset of the input variable $v_{a,i}$, i.e., $v_{a,i}|_{c} \subseteq v_{a,i}$. A value constraint $v_{c}$ for the variables $v_{a,0}|_{c}, v_{a,1}|_{c}, v_{a,2}|_{c}$ is, e.g., $v_{c} = \{v_{a,0} \leftarrow 0..0, v_{a,1} \leftarrow 0..0, v_{a,2} \leftarrow 0..1\}$ which restricts $v_{a,0}|_{c}$ and $v_{a,1}|_{c}$ to the single value 0, and $v_{a,2}|_{c}$ to the values 0 and 1.

**Definition 3.** A value constraint $v_{c}$ is a set of constraints on input variables.

The value space partition $D_a$ for the input variables $\{v_{a,0}, v_{a,1}, ..., v_{a,n-1}\}$ is defined as $D_a = v_{a,0} \times v_{a,1} \times ... \times v_{a,n-1}$ without value constraints on. A value space partition $D_{a|c}$ is the value space partition considering the value constraint $v_{c}$, such that $D_{a|c} = v_{a,0}|_{c} \times v_{a,1}|_{c} \times ... \times v_{a,n-1}|_{c}$.

**Definition 4.** A value space partition $D_a$ is the cross product of all possible input variable values, as formalized in 6.1

\[
D_a = v_{a,0} \times v_{a,1} \times \cdots \times v_{a,n-1} \tag{6.1}
\]

**Definition 5.** A value space partition $D_{a|c}$ is the cross product of the input variable values with respect to a value condition $v_{c}$, as formalized in 6.1.1
In $D_a$, all input combinations are represented, and in $D_{a|c}$ a subset of all input combinations are represented.

Consider three 8-bit integer input variables $v_{a,0}$, $v_{a,1}$, and $v_{a,2}$. The number of possible concrete input value combinations is the cross product of three 8-bit integers $|D_{a|c}| = 2^8 \times 2^8 \times 2^8 = 2^{24}$. Consider that we apply the input value constraint $vc_c = \{v_{a,0} \leftarrow 0..0, v_{a,1} \leftarrow 0..1, v_{a,2} \leftarrow 0..1\}$ then the resulting number of possible concrete input value combinations value space partition $|D_{a|c}| = 1 \times 2 \times 2 = 4$.

Note that a value space partition $D_{a|c}$, associated with a value constraints $vc_c$, is always a subset of $D_{a|c} \subseteq D_a$ such that $|D_{a|c}| \leq |D_a|$.

**Definition 6.** $|D_a|$ is the concrete number of input combinations in the set $D_a$, i.e., the product of the value domain size of each variable as formalized in 6.2

\[
|D_a| = \prod_i(|v_{a,i}|)
\]  

(6.2) \[
|D_{a|c}| = \prod_i(|v_{a,i|c}|)
\]

The number of possible non-empty value space partitions $|\mathcal{P}(D_a)|$ are then $2^{|D_a|} - 1$. The powerset $\mathcal{P}(D_a)$ contains all subsets $D_{a|c} \subseteq D_a$.

A WCET tool can produce a WCET considering that subset of inputs. Each value space partition $D_{a|c}$ can be analyzed and associated with two execution times, WCET and BCET (Best-Case Execution Time).

**Definition 7.** $WCET_{a|c} = est\_wcet(proga, D_{a|c})$ is an estimation of the WCET of an value space partition $D_{a|c}$ with respect to a software component behaviour prog_a of a component c_a.

**Definition 8.** $BCET_{a|c} = est\_bcet(proga, D_{a|c})$ is an estimation of the BCET of an value space partition $D_{a|c}$ with respect to a software component behaviour prog_a of a component c_a.
6.1 Input sensitive WCET analysis

Due to large number of possible program states, several program states must be merged during the WCET-analysis, producing a smaller number of states that can be handled. The effect of extensive merging is that precision is lost, generating increasingly large over estimations.

A smaller input domain generates fewer program states; limiting the input domain $D_{a|c}$ increases the precision of the analysis. Because of this we experience that the WCET become more accurate for the whole system due to less over approximations when analysing smaller parts of the behaviour by limiting the inputs:

$$est_{wcet}(prog, D_a) > \max_{\forall c (D_{a|c} \subseteq D_a)} (est_{wcet}(prog, D_{a|c}))$$

6.1.2 Analysis assumptions

We put some demands on the WCET analysis used. Let $est_{wcet}(prog_a, D_{a|c})$ be the WCET estimate calculated by a WCET analysis for a software component behaviour $prog_a$ and a value space partition $D_{a|c}$. Further, let $real_{wcet}(prog_a, D_{a|c})$ be the real worst-case execution time for the component behaviour $prog_a$ and the value space partition $D_{a|c}$. For the algorithm to work correctly the following assumptions should hold:

**Assumption 1.** The WCET calculation should never underestimate the WCET, i.e., $real_{wcet}(prog_a, D_{a|c}) \leq est_{wcet}(prog_a, D_{a|c})$ should be true for any $D_{a|c}$.

**Assumption 2.** A WCET calculation run with a single-valued value space partition should produce a WCET estimate equal to the time for running the program with these inputs, i.e., for $|D_{a|c}| = 1$ then $est_{wcet}(prog, D_{a|c}) = real_{wcet}(prog, D_{a|c})$ should hold.

**Assumption 3.** For any two input value space partitions $D_{a|1}$ and $D_{a|2}$ such that $D_{a|1} \subseteq D_{a|2}$ then $est_{wcet}(prog, D_{a|1}) \leq est_{wcet}(prog, D_{a|2})$ should hold.

**Assumption 4.** For any two input value space partitions $D_{a|1}$ and $D_{a|2}$ such that $D_{a|1} \subseteq D_{a|2}$ then $est_{bcet}(prog, D_{a|1}) \geq est_{bcet}(prog, D_{a|2})$ should hold.
**Assumption 5.** For any input value \( v \) there should be a natural ordering such that \( 0 \leq v < n \).

We claim that assumptions 1, 3 and 4 are sound and valid for most type of today’s input-sensitive WCET analysis tools. However, assumption 2 might not always be true. Moreover, we assume that each input variable only can be assigned a finite set of discrete values.

### 6.2 Reusable WCET analysis

For components that are reused in different systems it is today 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, but that prediction is only valid for that specific configuration, and the usage can vary a lot between different configurations. To support reuse of WCET predictions we need support for WCET analysis of different usage.

Our viewpoint is to make the analysis both (i) reusable, and (ii) 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, but often when using a component in a specific system, only parts of the component behaviour is used. Therefore generality and context-freeness leads to an increasing inability to make accurate predictions of the component behaviour for each specific use-case.

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

We augment the component 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 time it takes to execute
the longest execution part 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 both functional and extra-functional specifications since the main idea in component-based software engineering is to quickly assemble systems out of pre-fabricated components.

If a piece of software is scheduled with a *usage independent* WCET the predictions may be overly pessimistic and the system under utilized. Predicted with the correct behaviour allows for a considerably tighter WCET than would be predictable from the usage independent WCET analysis. In a large software system the usage independent WCET may be orders of magnitude inaccurate compared to the usage dependent WCET.

**Example**

Let’s revisit the component definition from Chapter 5. Let’s assume two components $c_a$ and $c_b$, where each component has a set of provided inputs $P_a = \{p_{a,0}, p_{a,1}\}$ and $P_b = \{p_{b,0}, p_{b,1}\}$. The variables $v_{a,0}, v_{a,1}, v_{b,0}, v_{b,1}$ and a value domains $D_a$ and $D_b$ represents the possible input value combinations. Analyzing these components considering the input domains $D_a$ and $D_b$ give the worst-case execution times $\text{est}_{\text{wcet}}(prog_a, D_a) = 1200$ and $\text{est}_{\text{wcet}}(prog_b, D_b) = 2400$.

Let’s assume that the components $c_a$ and $c_b$ are part of a system where their input values are limited due to a specific usage. These limitations can be described with value constraints $v_{c_i}$ and $v_{c_j}$, and the possible input combinations are the value space partitions $D_{a|i}$ and $D_{b|j}$ respectively. Thus, the resulting usage dependent WCETs are $\text{est}_{\text{wcet}}(prog_a, D_{a|i}) = 500ms$ and $\text{est}_{\text{wcet}}(prog_b, D_{b|j}) = 400ms$ as depicted in Figure 6.1.

The composite WCET of components $c_a$ and $c_b$ are 900 for the usage $D_{a|i}$ and $D_{b|j}$, as compared to the usage independent ($D_a$ and $D_b$) WCET 3600. The difference is quite large, and in a system with many components the difference between the context dependent and context-free WCETs can potentially be quite large, which leads to costly over dimensioning of the system resources.
We define a contract as a function of a set of a 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 value space partitions** Finding value space partitions in which all input combinations leads to similar execution times.

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

### 6.2.1 Component WCET analysis

Components are reused in different products and different usages. A different usage can substantially change the behavior of a component. To predict the execution time of a complex component with high accuracy, components must today 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 analyses become even more costly [Kor99].

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

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

When WCET analysis is performed with restrictions on the input parameters, not all single value input combinations are analyzed, but rather a set of value space partitions $D_{a|c} \subseteq D_a$, such that $D_{a[0]} \oplus D_{a[1]} \oplus \cdots \oplus D_{a[n-1]} = D_a$.

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

Finding value space partitions

We search the space $P(D_a)$ for value space partitions $D_{a|c}$ such that all input combinations have similar execution times, i.e., $\text{est}_{\text{wcet}}(\text{prog, } D_{a|c})$ is close to $\text{est}_{\text{bcet}}(\text{prog, } D_{a|c})$.

In most cases it is infeasible to separately analyze all possible input combinations. The challenge is to choose to analyze value space partitions $D_{a|c} \in P(D_a)$ in such a way that, in the ideal case every different execution time is mapped to every input combination that generates that specific execution time.

The search for value space partitions can be terminated when the following criteria are fulfilled with the following priority:

1. WCETs for all single input combinations are estimated.
2. A pre-defined accuracy is achieved.
The value space partitions $D_{a|c}$ should be chosen in such a way that similar execution times are grouped and can be expressed as restrictions on the inputs. A challenge is to find the right value space partitions $D_{a|c}$ such that accuracy of the WCET times is maximized.

**Definition 9.** Value space partition accuracy of a value space partition $D_{a|c}$ is the distance between the highest and lowest execution time within that value space partition, as formalized in Equation 6.3.

$$WCET_c - BCET_c$$  \hspace{1cm} (6.3)

The accuracy is an indication of how representative the highest value (WCET) is for each input combination in $D_{a|c}$. Consider the Figure 6.2, all “real execution times” will be represented by an estimated WCET. If there is a large discrepancy between the real execution times and the estimated WCET, then the estimated WCET is not representative for the value space partition $D_{a|c}$, and $D_{a|c}$ should be chosen differently.

![Figure 6.2: Value space partition accuracy over the input space partition $D_{a|c}$ and the execution times et.](image)

Theoretically, each single input combination has only one fixed execution-time. Thus, the difference between $WCET_{a|c}$ and $BCET_{a|c}$ of a value
space partition $D_{a|c}$ shows the greatest difference between two execution times within the value space partition. This is an indicator of how similar the execution times are in the value space partition. The sum of the difference between $WCET_{a|c}$ and $BCET_{a|c}$ of all 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 value space partition contains one element; a good solution is a trade-off between acceptable difference and the number of value space partitions. If the difference between $WCET_{a|c}$ and $BCET_{a|c}$ of each value space partition is larger than the required accuracy the value space partition should be chosen differently. The acceptable difference between $WCET_{a|c}$ and $BCET_{a|c}$ of the value space partition depends on the required accuracy of the value space partition. It is necessary to explicitly model states as derived inputs, in order to capture state variables.

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

**Definition 10.** Component WCET prediction accuracy is the sum of all weighted value space partition accuracy, divided by the number of value space partitions, as formalized in Equation 6.4.

$$\sum_{i=0}^{n} \left( (WCET_{a|c} - BCET_{e}) \cdot \frac{|D_{a|c}|}{|D_{a|c}|} \right)$$

The accuracy of the prediction itself is a termination condition for when the search for further value space partitions should stop.

### 6.2.2 Algorithm description

To find accurate value space partitions with least effort and in bounded time we propose a binary tree search approach, recursively dividing the input space into two value space partitions until the required accuracy has been found for all branches, consider Figure 6.3. The only data initially known is the longest and shortest execution time for the entire search space (the WCET and BCET). There are several possible approaches to solve such search problems, where binary search, simulated annealing and evolutionary search, are a few possible candidates.
Figure 6.3: Binary search for WCET over the input domain represented by variables \( v_{a,0} \) and \( v_{a,1} \). Each gray block indicates that the desired accuracy has been achieved, and the figure shows how the search tree expands, and divides the inputs.
The arguments to the algorithm, described in Figure 6.4, are prog, the program Dₐ, the input variables’ value space partition. A WCET-list is built with the WCET and BCET as the keys. The current value constraint is denoted c. The input variables are iteratively chosen and split (in the basic algorithm they are split in half), and new value constraints lc and rc are created from c. The lowest(c) and highest(c) returns the lowest and highest input variable values considering the contraint c. The function new_vc() creates a new value constraint. The function new_D(D, v, c) creates a new value space partition by replacing the value constraint for the variable v in the value space partition D with the new value constraint c. The new value space partitions are added to the WCET-list.

As the tree is built, several value space partitions are eligible for further division. To achieve as high accuracy as possible in as few divisions as possible there are different possible strategies for choosing the next value space partition to divide.

A new value space partition is chosen with wcet_less_pop(strategy) such that:

1. depending on pop strategy:
   - **Pop strategy 1: worst accuracy** the value space partition with worst accuracy is returned and removed from the wcet_list.
   - **Pop strategy 2: best accuracy** the value space partition with greatest accuracy is returned and removed from the wcet_list.
   - **Pop strategy 3: highest WCET** the value space partition with highest WCET is returned and removed from the wcet_list.
   - **Pop strategy 4: last used** the value space partition last added is returned and removed from the wcet_list.

2. there exists at least one variable vₐ,i with a domain |vₐ,i| > 1
3. wcet_c - bcet_c > accuracy, where accuracy is a given positive integer representing the desired accuracy.
4. if there does not exist a value partition that fulfills these criteria, then wcet_list.pop(strategy) will return nil.

When a value space partition has been chosen, there may be several input variables in the value space partition eligible for division. Thus, there are different possible strategies for choosing which variable to divide.

A variable vₐ,i is chosen for division with select_var(D) such that:
1. the variable constraint is such that $|v_{a,i}| > 1$

2. depending on select_var strategy:

   **Select_var strategy 1: last used** same variable is chosen as last division iff (1) is fulfilled, else try strategy on next variable.

   **Select_var strategy 2: next** next variable is chosen iff (1) is fulfilled, else try strategy on next variable.

   **Select_var strategy 3: greatest domain size** the variable with the largest domain size is chosen iff (1) is fulfilled, else try strategy on next variable.

3. If no variable fulfills these criteria, then select_var(D) will return nil.

Finally, the termination condition for the algorithm are such that the algorithm terminates when:

1. Desired accuracy is achieved, OR
2. All value space partitions are fully divided.

```
1. BEGIN find_value_space_partitions(prog, Da)
2.  v ← select_var(Da, strategy);
3.  WHILE v ≠ nil DO
4.    r ← range(v, Da);
5.    l ← lower(r);
6.    u ← upper(r);
7.    lr ← new_range(l, l + ((u - l) / 2));
8.    ur ← new_range(l + ((u - l) / 2) + 1), u);
9.    Da[lr] ← new_D(Da, v, lr);
10.   Da[ur] ← new_D(Da, v, ur);
11.   WCET_a[lr] ← calc_wcet(prog, Da[lr]);
12.   WCET_a[ur] ← calc_wcet(prog, Da[ur]);
13.   wcet_list.insert(WCET_a[lr], BCETlr, Da[lr]);
14.   wcet_list.insert(WCET_a[ur], WCET_a[ur], Da[ur]);
15.   Da ← wcet_list.pop(strategy);
16.   v ← select_var(Da, strategy);
17.  END WHILE
18. RETURN Da;
19. END find_value_space_partitions
```

Figure 6.4: Basic WCET input 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. Our approach gives an arbitrarily granular WCET with respect to input parameters. The arbitrary granularity depends on the effort and time put into the clustering of value space partitions and the creation of WCET contracts.

Current problems with fully automatic parametric WCET relates to its very high complexity. Several simpler approaches only parameterize input variables that affect loops, which leads to potentially lower accuracy due to that they miss many other program effects (e.g., mutually excluding parts). Our approach is exact in the sense that it relates the execution time to their corresponding input values, and the complexity is related to the search depth, allowing differently accurate predictions (and complexity). Thus, in short time, a less accurate, yet still reusable parameterizable WCET may be achieved; while if time and resources are available it is possible to achieve high accuracy if desired.

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

Usage profile

Except the natural limitations given by the variable’s type, its possible input values can be further constrained, e.g., by the user or some code generating tool. For example, a variable speed declared as an 8-bit unsigned integer can hold integer values between 0 and 255. For example, 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 as \{speed \leftarrow 0..200\}. The number of possible values that the variable can assume is called the value domain size.

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. A value space partition $D_{a|U}$ described the possible values of a certain usage.

**Definition 11.** A usage profile $U = \langle D_{a|U}, P, pt \rangle$ is a set of value constraints $D_{a|U}$ connected with a probability $P$ distribution.

The usage scenario is associated with a probability mass function $P : D_{a|U} \rightarrow [0, 1]$ for the occurrence of the values in $D_{a|U}$ such that the sum of the probabilities of all input combinations is 1, as outlined by Equation 6.6. Consider the example in (Figure 6.5), where the probability mass function of a usage profile $D_{a|U}$ is depicted as the light gray area. The dark gray area is the probability of a subset $D_{a|c} \subset D_{a|U}$.

\[
P(D_{a|c}) = \int_{D_{a|c}} P(D_{a|U}) \, dD_{a,U} \tag{6.5}
\]

\[
\int_{D_{a}} P(D_{U,a}) \, dD_{a,U} = 1 \tag{6.6}
\]

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

**Component WCET contracts**

It is desired to create as few value space partitions as possible and yet acquire as high accuracy as possible. Too many value space partitions will result in an unhandleable amount of information. Consider two 32-bit integer inputs rendering $2^{64}$ possible input combinations, and equally many corresponding WCETs. The search algorithm may result in a large number of value space partitions if high accuracy is required. In the worst case, the number of value space partitions is equal to the number of input combinations.

To lower the number of value space partitions, if possible two value space partitions $D_{a|i}$ and $D_{a|j}$ are merged if their associated WCETs are the same, i.e., $D_{a|i} \cup D_{a|j} | WCET_{a|i} = WCET_{a|j}$. Actually two value space partitions can be merged if their corresponding WCETs are close
6.2 Reusable WCET analysis

Figure 6.5: Probability mass function of a usage profile $D_{a|U}$. The dark gray area is the probability of a subset $D_{a|c} \subset D_{a|U}$.

enough to maintain the desired accuracy, such that $D_{a|ij} = D_{a|i} \cup D_{a|j}$ and $WCET_{a|ij} = \max(WCET_{a|i}, WCET_{a|j})$

A component contract is used for acquiring a WCET with respect to a usage. The contract is parameterized with a usage profile. The goal of the parameterization is to match the input combinations of the usage with the input combinations of value space partitions and find only those partitions that have any input combinations in common with the usage. Those value space partitions are referred to as active value space partitions, i.e., for each value space partitions $D_{a|c}$ that have any joint members with the set $D_{a|U}$ such that $D_{a|c} \cap D_{a|U} \neq \emptyset$ are active and their respective WCETs are eligible for the component WCET. The active value space partitions are those that need to be considered for finding the usage-dependent WCET. The usage dependent $WCET_{U}$ is defined in Equation 6.7.

$$WCET_{U} = \max_{\forall c(D_{a|c} \cap D_{a|U} \neq \emptyset)} (WCET_{a|c}) \quad (6.7)$$

A component contract can be seen as a function of a usage $U$ that results
in a usage dependent WCET (Equation 6.8).

\[ f_U : \mathcal{D}_a|U \rightarrow \max \left( WCET_{a|c} \right) \quad (6.8) \]

∀, \left( \mathcal{D}_{a|c} \cap \mathcal{D}_a|U \neq \emptyset \right)

The usage dependent WCET together with the probability \( P \) of a WCET allows for a contract user to disregard usage dependent WCETs with low probabilities. Thus, the component contract can be seen as a function of a value space partition and a priority threshold \( pt \) disregarding inputs with low probability, as defined in Equation 6.9.

\[ f_U : \mathcal{D}_a|U, pt \rightarrow \max \left( WCET_{a|c} \right) \quad (6.9) \]

∀, \left( \left( \mathcal{D}_{a|c} \cap \mathcal{D}_a|U \neq \emptyset \right) \wedge \left( P(\mathcal{D}_{a|c} \cap \mathcal{D}_a|U) < pt \right) \right)

**Contract composition**

Each value space partition can be associated with a set of possible outputs. Abstract interpretation can be used to make a safe over-approximation of limitations on outputs given limitations on inputs by analyzing possible values of the output variables. Each component produces output given the input such that the required interface \( \mathbf{R}_i \) of component \( c_i \) is a function of the input \( f_i : \mathbf{P}_i \rightarrow \mathbf{R}_i \). By adding this information to the predicates the approach is composable since one component will automatically give a component usage scenario to the next connected component. SWEET [GESL06a] is one tool that can produce restrictions on the output with respect to the input.

**6.2.4 Algorithm complexity**

Our algorithm will have a worst-case behaviour of \( O(2 \log |\mathcal{D}_a|) = O(\log |\mathcal{D}_a|) \) where \( \mathcal{D}_a \) is the input value space. This is because in each step of the algorithm the size of the currently selected value space partition is divided by two.
6.3 Finding WCET input combination

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

This approach does not specifically target reusable software components, even though the method may prove valuable for combining static WCET analysis with measurements.

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

Similar to the clustering algorithm, we subdivide the 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 expose the worst-case execution time.

The algorithm is presented in Figure 6.6. It works by iteratively calculating WCET estimates for different partitions of the program’s input value space. In each iteration the part of the input value space with the largest calculated WCET estimate, which has not yet been subdivided, is selected and subdivided into two smaller partitions for which WCET estimates are calculated. The process continues until the selected partition corresponds to only one concrete input value combination. The partition then holds the WCET input value combination and is returned.

6.3.1 Algorithm description

The arguments to the algorithm are \( \text{prog}_a \), the behaviour of the component under analysis, and a value space partition, \( D_{a|c} \).
Figure 6.6: Basic WCET input search algorithm

1. BEGIN find_WCET_input_values(prog, I)
2. D ← I
3. priq_queue ← empty
4. v ← select_var(D)
5. WHILE v ≠ nil DO
6.  c ← vc(v,D)
7.  l ← lowest(c)
8.  h ← highest(c)
9.  lc ← new_vc(l, l+((u-1)/2))
10. uc ← new_vc(l+((u-1)/2)+1, u)
11. D_{lc} ← new(D, v, lc)
12. D_{uc} ← new(D, v, uc)
13. wct_{lc} ← est_wct(prog, D_{lc})
14. wct_{uc} ← est_wct(prog, D_{uc})
15. priq_queue.insert(wct_{lc}, D_{lc})
16. priq_queue.insert(wct_{uc}, D_{uc})
17. D ← priq_queue.pop_D_largest_wct()
18. v ← select_var(D)
19. END WHILE
20. END find_WCET_input_values
The functions \( \text{lowest}(c) \) and \( \text{highest}(c) \) return the lowest and highest value respectively in the range \( c \). The function \( \text{new}_D(D, v, r) \) creates a new input value space partition from the argument \( D \), by replacing the range assigned to \( v \) with the new range \( c \). The \( \text{prio\_queue} \) is a priority queue where inserted value space partitions are indexed upon their corresponding WCET estimates. When popping an item from the queue the value space partition with the largest WCET estimate will be returned and also removed from the queue. The \( \text{select\_var}(D) \) function selects a variable \( v \) with a multi-valued range, i.e., \( |v| > 1 \). If such a variable exists its corresponding range is extracted and divided into two subranges. These ranges are then used to create two new partitions. For each partitioning a WCET estimate is calculated, and inserted in the priority queue indexed on its corresponding WCET estimate. If the current partition does not include any multi-valued variables it is returned and the algorithm terminates.

To guarantee that all partitions simultaneously stored in the priority queue are disjunct, i.e., the same input value combinations should only occur in one partition, the variables should always be selected in some predefined order. Moreover, the currently selected variable’s range should be divided down into a single value before selecting the next variable.

### 6.3.2 Example

Figure 6.7 illustrates how the algorithm works. The component \( c_a \) has a provided interface \( P_a \) with three inputs \{\( p_{a,0}, p_{a,1}, p_{a,2} \)\} and has been given the initial value space of \( \langle v_{a,0} \leftarrow 0..15, v_{a,1} \leftarrow 0, v_{a,2} \leftarrow 0..1 \rangle \) which corresponds to \( 16 \times 1 \times 2 = 32 \) concrete input value combinations. Variable \( v_{a,0} \) is first selected to do range division upon. This produces two new partitions for which WCET calculations are made. The \( \langle v_{a,0} \leftarrow 0..7, v_{a,1} \leftarrow 0, v_{a,2} \leftarrow 0..1 \rangle \) partition gives the largest WCET estimate and the analysis therefore continues with this partition during the next iteration. This time \( v_{a,0} \)’s range is subdivided into \( 0..3 \) and \( 4..7 \), both producing partitions for which WCET estimates are calculated.

The division of \( v_{a,0} \)’s range continues until the value of \( v_{a,0} \) which produces the largest WCET estimate when \( v_{a,1} \leftarrow 0 \) and \( v_{a,2} \leftarrow 0..1 \) has been found. Since \( v_{a,1} \) only can hold a single value, the next variable selected is \( v_{a,2} \). The division of \( v_{a,2} \)’s range produces two partitions, for
Figure 6.7: Example of basic algorithm run
which $\langle v_{a,0} \leftarrow 5, v_{a,1} \leftarrow 0, v_{a,2} \leftarrow 1 \rangle$ gives the largest WCET. There are no other partitions in the priority queue with a larger WCET estimate, so the iteration stops and the partition is returned.

### 6.3.3 Algorithm complexity and back-tracking

Our algorithm will have a best-case behaviour of $O(2 \times \log |D_a|) = O(\log |D_a|)$ where $D_a$ is the input value space. This is because in each step of the algorithm the size of the currently selected value space partition is divided by two. In many cases this will also be the algorithm’s worst-case behaviour, since the worst-case input values are likely to be found in one of the two partitions originating from the currently selected one.

Unfortunately, due to over-approximations made in the WCET analysis, this is not always true, i.e., sometimes both partitions originating from the currently selected partition get a WCET estimate smaller than, in the priority queue already stored, WCET estimate. The analysis then has to back-track and continue with this partition. For example, assume that the $\langle v_{a,0} \leftarrow 5, v_{a,1} \leftarrow 0, v_{a,2} \leftarrow 1 \rangle$ partition in Figure 6.7 gave a WCET estimate of 65 instead of 70. The analysis should then continue with $\langle v_{a,0} \leftarrow 6..7, v_{a,1} \leftarrow 0, v_{a,2} \leftarrow 0..1 \rangle$ instead of terminating. In the worst-case this type of back-tracking gives that a WCET calculation must be made for each concrete input value combination plus the WCET calculation for the binary search. Thus, the algorithm has a worst-case behaviour of $O(|D_a| + 2 \times \log |D_a|) = O(|D_a|)$.

To reduce the amount of back-tracking, it is important that the WCET calculation is tight, i.e., $\text{est}_\text{wcet}(D_{a|c})$ is close to $\max_{v_c \in D_{a|c}} \text{real}_\text{wcet}(v_c)$ even though $|D_{a|c}|$ is large. However, if the analysis time for deriving a tight WCET estimate is significantly higher than deriving a less tight WCET estimate, some back-tracking might still be worth doing.

### 6.4 Approaches for faster termination

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

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

A general observation is that it is more likely that either the smallest or the largest value in an input variable’s value domain will be the value that gives the WCET. This is especially true if the outcome of loop conditions are dependent on this input variable. As an illustrative example consider the min and max input variables in Figure 6.8. 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 6.6 as follows: whenever a variable v with a range \( l \ldots u \) is selected for the first time it will be divided into three new ranges; \( l \ldots l \), \( l+1 \ldots u-1 \), and \( u \ldots u \), each producing a partition for which WCET estimates are calculated. If the medium range \( (l+1..u-1) \) is selected for division, normal binary range division is performed.

Figure 6.9 illustrates how the heuristic works. The program has two integer input variables \( x \) and \( y \), with a given input value range of \( 0..15 \) and \( 0..255 \) respectively. \( x \) is first selected to do input range division.
upon. This produces three new partitions for which WCET estimates are calculated. The \((x \leftarrow 0, y \leftarrow 0..255)\) partition gets the largest WCET estimate and the analysis continues with a division of \(y\)’s range. This produces three new partitions for which WCET estimates are calculated. The \((x \leftarrow 0, y \leftarrow 255)\) partition gives a WCET estimate which is larger than all other partitions in the priority queue. Thus, the WCET was given when \(x\) and \(y\) were assigned its minimum and maximum input values respectively.

### 6.4.2 User interaction

Another option for improving the overall analysis time, is to let the user provide an input value space partition in which he/she believes hot spots are found. For example, in Figure 6.7 the user might believe that one worst-case is when \(k = 1\) and \(0 \leq i \leq 6\). The analysis can then be started with an initial set of partitions according to the user’s assumptions, e.g., \((i \leftarrow 0..6, j \leftarrow 0, k \leftarrow 1)\), \((i \leftarrow 0..6, j \leftarrow 0, k \leftarrow 0)\), and \((i \leftarrow 7..16, j \leftarrow 0, k \leftarrow 0..1)\), where the first partition corresponds to the...
Figure 6.10: division
user provided assumption.

6.5 Discussion

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 is important, it is neccessary 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 section we have introduced a reusable WCET analysis for reusable, yet accurate, predictions of the real-time property WCET.

... ...

As a side effect we see that the WCET become more accurate for the whole system due to less over approximaitions when analyzing smaller parts of the behaviour by limiting the inputs:

\[ est_{-}wcet\ (prog, D_a) > \max(WCET_{\alpha[a]})|D_a[0] \oplus D_a[1] \oplus \cdots \oplus D_a[n-1] = D_a \]  

(6.10)

We see the use for best-case prediction with best-case behaviour but worst-case hardware effects for a program and a given input set. This would make our predictions better.

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.

### 6.5.1 Contributions

In this chapter we have presented two novel methods and an evaluation tool. More specifically the contributions are:

- A method for reusable, input parameterized, WCET.
- A method for finding input combinations that cause the WCET path of a program to execute.
- An evaluation tool for both methods.

### 6.5.2 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, BEGL05].

Lisper et al. [Lis03] propose a fully automatic parametric WCET analysis. Fully parametric WCET is the strive, but there are many problems related to scaling. In [VHMW01, CHMW07] WCET is parameterized with respect to loops only. Ji et al. in [JWLQ06] divides the source code in modes depending on input, and only the parts that are used in the a specific usage are analyzed. 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 these works, but provides reusable parametric analysis for reusable software components.

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 dynamic voltage scaling depending on the timing requirements. Wenzel et al. [WRKP05] use both model checking and genetic algorithms to derive which input data makes a certain instrumented code part to be executed. Gross et al. [SESW05] use evolutionary testing with measurement based WCET analysis to find a context dependent WCET.
In [DP04] a framework for probabilistic WCET with static analysis is presented. The probabilities are related to the probability of possible values of external and internal variables.

A framework that considers the usage of a system has been developed in [LPB+05]; however, neither software components nor reuse is considered. In [BCP03a] each basic block execution times and probability distributions are measured. The results are transformed into execution time profiles and the resulting execution time profiles are then combined with probabilistic methods.

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.

One approach to solve similar problems is parametric WCET. This has been proposed by many researchers within the WCET community but there is still very few parametric WCET methods developed. In [Lis03] Björn Lisper outlines a technique for fully automatic parametric WCET analysis, which is based on known mathematical methods. In a MSc thesis [Alt06, Hum06] a method inspired by Lisper’s work has been developed and tested with the aiT tool [aiT]. However, the focus of this work is not reusable WCET analysis, and reanalysis is required for different usages. A program representation for parametric WCET analysis has been suggested by Colin and Bernat [CB02]. Vivancos et. al. [VHMW01] propose an iterative method for computing WCET for loops parameterized in the number of loop iterations.

In [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 comparison 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. In [JWLQ06] the source code is divided in modes depending on input, and only modes that are used in a given context is analyzed. 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. All mentioned methods have the drawback of requiring reanalysis for every new usage.

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, BEGL05].
There are several WCET tools that support assertions and conditions to make the WCET tighter, e.g., aiT [aiT], RapiTime [Rap], Bound-t [Bt] and SWEET [SWE].

In [LPB+05] a framework has been developed that considers the usage of a system; however, neither software components nor reuse is considered. In [DP04] a framework for probabilistic WCET with static analysis is presented. 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, NM01]; our work illustrates how the division into context-unaware and context-sensitive analyses would be integrated into these models.

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

6.5.3 Future work
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.

-Hitchiker’s guide to the galaxy

Chapter 7

Transforming components to real-time tasks

In this chapter we present one method for optimizing the resource usage in component-based real-time systems based on real-time analysis, calculating resource consumption and genetic algorithms for deriving mappings between components and tasks that are optimized for low resource consumption, while maintaining stipulated real-time requirements.

Allocating components to tasks, and scheduling tasks are both complex problems due to the exponentially growing search space. Simulated annealing and genetic algorithms 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 allocating 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 48% and 32% respectively.
7.1 Introduction

In many component-based real-time systems there is no explicit strategy for deriving real-time tasks from components. This has lead to that many systems use a one-to-one mapping between components and 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 are needed, that mapping between components and tasks need to be performed with an alternate approach. In such circumstances.

Let's consider the 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, and $c_B$ in one task.
2. $c_A$ and $c_B$ in the same task.
3. $c_B$ and $c_A$ in the same task (reverse order).

The allocations (2) or (3) will, e.g., result in lower memory consumption. If, e.g., the components are have different periodicity a co-allocation may lead to high processor utilization, which may be alright 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

Figure 7.1: Possible allocations of 2 components to tasks.
some allocations may not be possible. Another issue is the number of
 task switches and 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 feasibility.

In a system with only a small number of components it is trivial to find
the best mapping between components and tasks, but already in a system
with 10 components the number of possible allocations exceeds several
millions. In such systems it may be difficult to find a good mapping
manually. Many industrial systems are comprised by as many as 50 dif-
ferent components. Therefore we have defined a theoretical framework
for reasoning about the mapping between components and tasks. The
reasoning framework is well suited to be applied in relation to optimiza-
tion algorithms such as, e.g., genetic algorithms.

7.2 Allocating components models to real-time models

In RTS temporal constraints are of great importance and tasks control
the execution of software. Hence, components need to be allocated to
tasks in such a way that temporal requirements are met, and resource us-
age is minimized. Given an allocation we determine if it is feasible and
calculate the memory consumption and task switch overhead. To im-
pose timing constraints, we define end-to-end timing requirements and
denote them transactions. Transactions are defined by a sequence of
components and a deadline. Thus, the work in this paper 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 fulfil given requirements on memory consumption and CPU-overhead.

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

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

Each component $c_i$ has a pre-defined maximum stack size. The stack of the task is the maximum size of all components stacks allocated to the task since all components will use the same stack. The CPU overhead $p$, the memory consumption $m$ for a task set $A$ in a system $K$ are formalized in equations 7.1 and 7.2:

$$p_A = \sum_{\forall \tau_i \in A} \frac{\rho}{T_i}$$  \hspace{1cm} (7.1)

$$s_A = \sum_{\forall \tau_i \in A} (stack_i + \beta)$$  \hspace{1cm} (7.2)

Where $p_A$ represents the sum of the task switch overhead divided by the period for all tasks in the system, and $s_A$ represents the total amount of memory used for stacks and task control blocks for all tasks in the system.

### 7.3.1 Constraints on allocations

There is a set of constraints that must be considered when allocating components. These are:

- Component isolation
- Intersecting transactions
- Trigger types and period times
- Schedulability

Each constraint is further discussed below:
7.3 Allocation framework

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, therefore 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 jitter for some tasks, thus components with highly uncertain WCET should be isolated. There may also be integrity reasons to separate certain combinations of components. Hence it must be assured that two components that are defined to be isolated do not reside in the same task. This can be validated with equation 7.3:

$$Iso(a, b) : c_a \text{ has an isolation requirement to } c_b$$

$$\neg\exists i (\forall j \forall k (c_j \in Z_i \land c_k \in Z_i \land Iso(j, k))) \quad (7.3)$$

Where there must not exist any task $\tau_i$ that has two components $c_j$ and $c_k$, if these components have an isolation requirement.
Intersecting transactions

If component transactions intersect, there are different strategies for how to allocate the component where the transactions intersect. The feasibility is described in equations 7.4 and 7.5. A component in the intersection should not be allocated with any preceding component if both transactions are event triggered; the task should be triggered by both transactions to avoid pessimistic scheduling. A component in the intersection of one time-triggered transaction and one event-triggered transaction can be allocated to a separate task, or with a preceding task in the time-triggered transaction. A component in the intersection of two time-triggered transactions can be allocated arbitrarily. In Figure 7.2, two different allocations are imposed due to intersecting event-triggered transactions. In the left part of Figure 7.2 there is an intersection between a time triggered and an event triggered transaction. Then the intersecting component $c_3$ is allocated to the task triggered by the time triggered transaction. In the right part of the figure, where two event triggered transactions intersect, the component $c_3$ is allocated to a separate task, triggered by both transactions.

$$T_E(tr) : \text{transaction is event triggered}$$
$$T_T(tr) : \text{transaction is time triggered}$$
$$P(a, b, d) : c_a \text{ is predecessor to } c_b \text{ in the set } N_d$$
$$X^{bc}_{a} = c_a \in N_b \land c_a \in N_c$$
$$Y^{ce}_{ab} = c_a \in Z_c \land c_b \in Z_c$$
$$\neg \exists i (\forall j \forall k \forall l \forall m (X^{jk}_{il} \land Y^{lm}_{im} \land T_E(ctr_j) \land T_T(ctr_k) \land (P(m, l, k) \lor P(m, l, j))))$$
(7.4)

$$\neg \exists i (\forall j \forall k \forall l \forall m (X^{jk}_{il} \land Y^{lm}_{im} \land c_m \in N_k \land T_T(ctr_j) \land T_E(ctr_k) \land P(c_m, c_l, N_k)))$$
(7.5)

Where there must not exist any task $\tau_i$ that has two components $c_l$ and $c_m$ in a way that two component transactions $ctr_j$ and $ctr_k$ intersect in $c_l$, and $c_m$ precedes $c_l$ in the transactions $ctr_j$ or $ctr_k$, if $ctr_j$ or $ctr_k$ are event-triggered.
7.3 Allocation framework

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 7.3. Components with the same period time can be allocated together if they do not have any other explicit dependencies, see (2) in Figure 7.3. To facilitate analysis, a task may only have one trigger, so time triggered components with the same 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 in fact trigger on the same event, and have the same minimum inter arrival time, see (3) in Figure 7.3. Components with harmonic periods could also be allocated to the same task. However, harmonic periods create jitter. Consider two components with the harmonic periods five and ten that are allocated to one task. The component with the period five will run every invocation, while the other component will run every second invocation, which creates a jitter; therefore we have chosen not to pursue this specific issue.

![Figure 7.3: Component to task allocation considering triggers.](image)

Schedulability

Schedulability analysis is highly dependent on the scheduling policy chosen. Depending on the system design, different analyses approaches have to be considered. The task and task transaction meta-models are
constructed to fit different scheduling analyses. In this work we have used fixed priority exact analysis. However, the model can easily be extended with jitter and blocking for real-time analysis models that use 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. Different algorithms can be used to find a feasible allocation and scheduling of 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. The worst-case allocation is a one-to-one allocation where every component is allocated to one task. The best-case allocation on the other hand, is where all components are allocated to one single task. To allocate all components to one task is very seldom feasible. Also, excessive allocation of components may negatively affect scheduling, because the granularity is coarsened and thereby the flexibility for the scheduler is reduced.

Simulated annealing, genetic algorithms and bin packing are well known algorithms often used for optimization problems. These algorithms have been used for problems similar to those described in this paper; bin packing, e.g., has been proposed in [OS95] for real-time scheduling. Here we briefly discuss how these algorithms can be used with the described framework, to perform component to task allocations.

**Bin Packing** is a method well suited for our framework. In [JR95] a bin packing model 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 can solve, roughly, any problem as long as there is some way of comparing two solutions. The framework proposed in this paper give 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) is a global optimization technique that is regularly used for solving NP-Hard problems. The energy function consists of a schedulability test, the memory consumption and CPU-overhead. In [TBW92][CK95] simulated annealing is used to place tasks on nodes in distributed systems.

7.5 Evaluation

In order to evaluate the performance of the allocation approach the framework 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. We compare the allocations with respect to if the allocation is feasible (real-time analysis), memory consumption and CPU overhead.

The implementation is based on genetic algorithms (GA) [FF95], and as Figure 7.4 shows, each gene represents a component and contains a reference to the task 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 dependent on the validity of the allocation, the memory consumption and the CPU overhead.

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: I which represents component isolation, IT representing intersecting transactions, Tr representing trigger
Figure 7.4: The genetic algorithm view of the component to task allocation; a system with ten components, allocated to four tasks.

types and period times, and finally Sc represent scheduling. Consider that each of these feasibility tests is assigned a value greater than 1 if they are true, and a value of 0 if they are false. The parameter n represents the total number of components. Then, the fitness function can be described as with equation 7.6.

\[
\text{Fitness} = \left( (I+IT+Tr+Sc)F + \left( \frac{n}{s_{A}} + \sum_{\forall \tau_{i} \in A} \frac{p \cdot n}{T_{i}} \right)O \right) \cdot (I \cdot IT \cdot Tr \cdot Sc + 1)
\]

(7.6)

Where the fitness is the sum of all feasibility values times a factor F, added with the inverted memory usage and performance overhead, times a factor O, and \( F >> O \). 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.

7.5.2 Simulation set up

This section describes the simulation method and set up. For each simulation the genetic algorithm assigns 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. The component properties are deduced from the task they are allocated. Transactions are deduced the same way from the task set. In this way it 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 numbers of components in the systems are ranging in twenty steps from 40 to 400, with a main point on 120 components.
- The period times for the components are randomly selected from a predefined set of different periods between 10 and 100 ms.
- The worst case execution time (WCET) is specified as a percentage of the period time and chosen from a predefined set. The WCETs together with the periods in the system constitutes the system load.
- The transaction size is the size of the generated transactions in percentage of the number of components in the system. The transaction size is randomly chosen from a predefined set. The longer the transactions, the more constraints, regarding schedulability, on how components may be allocated.
- The transaction deadline laxity is the percentage of the lowest possible transaction deadline for the generated system. The transaction deadline laxity is evenly distributed among all generated systems and is always greater or equal to one, to guarantee that the generated system is possible to map. The higher the laxity, the less constrained transaction deadlines.

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

To get as realistic systems to simulate as possible, the values used to generate systems are gathered from some of our industrial partners. The industrial partners chosen are active within the vehicular embedded system segment. A complete table with all values and distributions, of the system generation values, can be found in [FSm05b]. The task switch time used for the system is 22 $\mu$s, and the tcb size is 300 bytes. The task switch time and tcb size are representative of commercial RTOS tcb sizes and context switch times for common CPUs.
The simulations are performed for four different utilization levels, 30\%, 50\%, 70\% and 90\%. For each level of utilization 1000 different systems are generated with the parameters presented above.

7.6 Discussion

Resource efficiency is important for RTS, both regarding performance and memory. Schedulability, considering resource efficiency, has gained much focus, however the allocation between components to tasks has gained very little focus. Hence, in this paper 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. It can also be used iteratively in case of design changes; however with some obvious drawbacks on the results. The framework can easily be extended to support other optimizations, besides task switch overhead and memory 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 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.

7.6.1 Contributions

Resource efficiency is important for RTS, both regarding performance and memory. Schedulability, considering resource efficiency, has gained much focus, however the allocation between components to tasks has gained very little focus. Hence, in this paper 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. It can also be used iteratively in case of design changes; however with some obvious drawbacks on the results. The framework can easily be extended to support other optimizations, besides task switch overhead and memory consumption.

7.6.2 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. The problem of allocating tasks to different nodes is a problem that has been studied by researchers using different methods [HS95, TBW92]. 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 [KWS95]. 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 [KWS95] 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 [SLS02] are discussing the code size, and how it can be minimized, but does not regard scheduling and resource constraints.

7.6.3 Future work

Future work includes adding other allocation criteria, e.g., by looking at jitter requirements, and blocking. By adding jitter constraints and blocking, trade-offs arise between switch overhead and memory size versus deviation from nominal start and end times and blocking times. Furthermore, a more efficient scheduling policy and priority assignment will be
applied. Due to the nature of GA it is easy to add new optimizations as the ones suggested above.
Chapter 8

Empirical results

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

8.1 Input sensitive execution-time analysis

We have used programs from the Mälardalen WCET Benchmark suite [WCE06], together with some codes provided by our industrial partner [BEG+08], to evaluate our methods. We have only included benchmarks which may by run with many different input values. Table 8.1 gives some details of the benchmark used. #LOC gives lines of C code. For the evaluations we used a high precision AE (generating much detailed flow information, but with a potential long analysis times), the ARM9 low-level analysis, and the clustered calculation method. We have also assumed that SWEET’s WCET calculation does not make any overapproximations when run with a one-sized input value partition.

8.1.1 Reusable WCET

We compare to traditional WCET mostly because there are no available parametric WCET tools to compare to.

Sweet, full merge for all benchmarks.
### Table 8.1: Benchmark programs used

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LOC</th>
<th>Vars</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td>Cyclic redundancy check computation on 40 bytes of data</td>
<td>128</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>edn</td>
<td>Finite Impulse Response (FIR) filter calculations</td>
<td>285</td>
<td>3</td>
<td>2^10</td>
</tr>
<tr>
<td>inssort</td>
<td>Insertion sort on a reversed array of size 10</td>
<td>92</td>
<td>10</td>
<td>10^20</td>
</tr>
<tr>
<td>complex</td>
<td>Nested loop program</td>
<td>64</td>
<td>2</td>
<td>2^6</td>
</tr>
<tr>
<td>lcdn</td>
<td>Read ten values, output half to LCD</td>
<td>64</td>
<td>2</td>
<td>2^10</td>
</tr>
<tr>
<td>ns</td>
<td>Search in a multi-dimensional array</td>
<td>535</td>
<td>1</td>
<td>2^2</td>
</tr>
<tr>
<td>nsichneu</td>
<td>Simulates an extended Petri net. Generated code with more than 250 if-statements.</td>
<td>4253</td>
<td>6</td>
<td>2^10</td>
</tr>
<tr>
<td>esab_mod</td>
<td>Industrial code developed by CC-Systems and Esab to control welding machine</td>
<td>3064</td>
<td>17(15)</td>
<td>2^70</td>
</tr>
<tr>
<td>task1</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>55</td>
<td>4</td>
<td>2^5</td>
</tr>
<tr>
<td>task3</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>58</td>
<td>7</td>
<td>2^3</td>
</tr>
<tr>
<td>task4</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>77</td>
<td>18</td>
<td>2^2</td>
</tr>
<tr>
<td>task5</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>86</td>
<td>8</td>
<td>2^10</td>
</tr>
<tr>
<td>task7</td>
<td>Industrial task code developed by Volvo CE for the Transmission ECU of articulated haulers</td>
<td>123</td>
<td>26(17)</td>
<td>2^37</td>
</tr>
</tbody>
</table>

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

We have run all the benchmarks with four different strategies to dividing the input value space partition. These strategies are, firstly, **Last used**, where the next variable in the value space partition is the same variable as was divided last. This strategy is similar to a depth first tree search algorithm, since we use binary search to find the value space partitions. The second strategy is **Next**, where always the next variable is chosen (in their annotated order), giving a search similar to breadth first search. Each of these strategies are also analyzed with the **basic** and **extreme** heuristics determining how the value space partitions is divided. In the basic heuristics, the value space partition is divided in two, with respect to the chosen variable. In the extreme heuristics the value space partition is divided equally to the basic except for the *first time*, when it is divided in three new partitions, with respect to the chosen variable; those three partitions are the min-value, the max-value and the rest.
For each benchmark we show the number of analyses (\#Runs) required to reach a specific accuracy, and how many different value space partitions that resulted in (\#D).

The accuracy is in comparison to a single run, annotated with the initial value space partition (that has not been divided). The flow facts generated for each benchmark is shown in Table 8.2.

The flow facts in Table 8.2 are:

- **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 node can be visited.
- **mmnl**: minimum and maximum nested loop count, i.e., loop count for nested loops.
- **mmnc**: minimum and maximum node count, i.e., number of times a node is visited.

<table>
<thead>
<tr>
<th>Program</th>
<th>Flow facts</th>
<th>ip</th>
<th>ep</th>
<th>ina</th>
<th>ine</th>
<th>mmh</th>
<th>mmnl</th>
<th>mmnc</th>
</tr>
</thead>
<tbody>
<tr>
<td>crc</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>edn</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isort</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jcomplex</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lcdnum</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ns</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nsichneu</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task1</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task2</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task3</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task4</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task5</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task7</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8.2: Generated flow fact for each benchmark

The crc benchmark gives no improvement from the input division. The input domain is very small (only 81 input combinations). Even when
Chapter 8. Empirical results

Table 8.3: crc benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>% Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Runs #D</td>
<td>#Runs #D</td>
</tr>
<tr>
<td>crc</td>
<td>10%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
</tbody>
</table>

Table 8.4: jcomplex benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>% Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#Runs #D</td>
<td>#Runs #D</td>
</tr>
<tr>
<td>jcomplex</td>
<td>10%</td>
<td>8 5</td>
<td>16 11</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>40 11</td>
<td>32 19</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>48 22</td>
<td>180 57</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>62 28</td>
<td>190 69</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>80 37</td>
<td>204 64</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>112 52</td>
<td>234 78</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>160 75</td>
<td>268 90</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>264 111</td>
<td>332 113</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>642 127</td>
<td>616 132</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>(96%) 878.822</td>
<td>(96%) 874.874</td>
</tr>
</tbody>
</table>

The ns benchmark has

Let us for example study task1 and see what the resulting 5 value space partitions show. Task1 has 4 variables that we denote \{v_{task1,0}, v_{task1,1}, v_{task1,2}, v_{task1,3}\}. Our algorithm has divided the inputs in their respective value space partitions, as shown in Table 8.9. It is straightforward to extract 5 predicates from these value space partitions, that are used for the parameterization of task1.

When we have run the algorithm on these testbenches, we have considered the value space partitions with the slightest difference in either WCET or BCET to be different value space partitions. In practice it is likely to have a certain range or accuracy where value space partitions
### Table 8.5: ns benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs #D</td>
<td>Extreme #Runs #D</td>
</tr>
<tr>
<td>ns</td>
<td>10%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a n/a</td>
<td>3 3</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a n/a</td>
<td>4 4</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a n/a</td>
<td>5 5</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a n/a</td>
<td>4 4</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a n/a</td>
<td>4 4</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>8 8</td>
<td>11 11</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>14 14</td>
<td>19 19</td>
</tr>
</tbody>
</table>

### Table 8.6: edn benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs #D</td>
<td>Extreme #Runs #D</td>
</tr>
<tr>
<td>edn</td>
<td>10%</td>
<td>18 3</td>
<td>5 5</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>22 3</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>24 3</td>
<td>3 3</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>28 3</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>34 2</td>
<td>9 9</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>90 2</td>
<td>21 21</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>98 2</td>
<td>23 23</td>
</tr>
</tbody>
</table>

### Table 8.7: nsichneu benchmark - 138 runs

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs #D</td>
<td>Extreme #Runs #D</td>
</tr>
<tr>
<td>nsichneu</td>
<td>10%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a n/a</td>
<td>n/a n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>3 3</td>
<td>3 3</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>8 8</td>
<td>4 4</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>44 11</td>
<td>20 14</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>50 13</td>
<td>20 14</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>64 15</td>
<td>35 35</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>90 23</td>
<td>35 35</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>122 25</td>
<td>185%152</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>192%304</td>
<td>500%324</td>
</tr>
</tbody>
</table>

8.1 Input sensitive execution-time analysis
Table 8.8: task1 benchmark - 72 runs

Table 8.9: task1 resulting value space partitions

Table ?? and Table ?? gives 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. |I| 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. #D gives the number of value space partitions created. MinD and MaxD gives the minimum and maximum precision of the value space partitions. %Acc gives the total improvement of accuracy in percent compared to normal static WCET analysis. %WCET gives the share of input combinations that produce the WCET.

We notice a big difference in many of the tests between the different approaches. It is important to choose the right method for the right program. We propose that this is automated by monitoring the the difference in accuracy between each run and change the strategy if the accuracy is not improving.
8.1 Input sensitive execution-time analysis

Figure 8.1: edn ...

Figure 8.2: jcomplex ...

Figure 8.3: jcomplex ...

Figure 8.4: task5 ...

<table>
<thead>
<tr>
<th>Prog</th>
<th>% Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>Extreme #D</td>
</tr>
<tr>
<td>task3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>70%</td>
<td></td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>90%</td>
<td></td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 8.10: task3 benchmark
### Table 8.11: task4 benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>% Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic #Runs</td>
<td>D</td>
</tr>
<tr>
<td>task4</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Figure 8.5: Task 1 ...

Figure 8.6: Task 1 ...

Figure 8.7: Task 3 ...

Figure 8.8: ns ...

---

**Chapter 8. Empirical results**
### 8.1 Input sensitive execution-time analysis

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>task5</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 8.12: task5 benchmark

<table>
<thead>
<tr>
<th>Prog</th>
<th>%Acc</th>
<th>Strategy Last Used</th>
<th>Strategy Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Basic</td>
<td>Extreme</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lcdnum</td>
<td>10%</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>20%</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>30%</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>40%</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>50%</td>
<td>64</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>60%</td>
<td>114</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>70%</td>
<td>230</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>80%</td>
<td>432</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>90%</td>
<td>728</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>100%</td>
<td>1068</td>
<td>139</td>
</tr>
</tbody>
</table>

Table 8.13: lcdnum benchmark - ## runs

![Figure 8.9: lcdnum ...](image1)

![Figure 8.10: lcdnum ...](image2)
Figure 8.11: nsichneu ...

Figure 8.12: nsichneu ...

Table 8.14: task7 benchmark

Table 8.15: esab_mod benchmark
8.1 Input sensitive execution-time analysis

8.1.2 Finding WCETs

| Program  | #Vars | $|I|$ | #WC | Basic | #BT | TotT | #WC | Extreme | #BT | TotT |
|----------|-------|------|-----|------|-----|------|-----|--------|-----|------|
| ccc      | 4     | 81   | 7   | 6923 | 7441| 0    | 49934| 10    | 6813  | 7509 | 0    | 71874|
| edn      | 3     | 2$$^*$ | 12  | 5063 | 5531| 0    | 63512| 7     | 4687  | 4984 | 0    | 33636|
| insort   | 10    | 10$$^*$ | 619 | 344  | 765 | 0    | 326894| 619   | 485   | 780  | 0    | 326894|
| jcomplex | 2     | 512  | 455 | 32   | 838 | 155  | 116541| 455   | 47    | 922  | 141  | 97078|
| icdmux   | 2     | 1024 | 71  | 31   | 859 | 0    | 3438 | 4     | 63    | 734  | 0    | 21571|
| ns       | 1     | 512  | 6   | 1093 | 38172| 0    | 35256 | 35256 | 0    | 35256|
| nschne   | 6     | 2$$^*$ | 25  | 17733| 95203| 0    | 1716109| 20    | 13968 | 95203| 0    | 1309785|
| esab_mod | 19    | 2$$^*$ | -   | -    | -   | -    | -    | -     | -     | -    | -    | -     |
| task1    | 4     | 64   | 11  | 16359| 19564| 0    | 188593| 8     | 14203 | 20234| 0    | 121687|
| task2    | 8     | 10   | 9   | 14468| 18908| 0    | 147326| 10    | 14125 | 21890| 0    | 135654|
| task3    | 8     | 2$$^*$ | 53  | 47   | 137 | 0    | 5434 | 4     | 62    | 185  | 0    | 4314|
| task7    | 26    | 2$$^*$ | 167 | 375  | 719 | 2    | 72165| 174   | 328   | 625  | 3    | 72912|

Table 8.16: Finding WCET analysis results for benchmarks without program slicing

Table 8.16 gives 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. $|I|$ gives the size of the input value space. Basic gives the results for the basic input derivation method, while Extreme gives the corresponding ones for our extreme value search method. #WC gives the number of WCET calculations performed. MinT and MaxT gives the minimum and maximum analysis time (in seconds) used for any of the WCET calculations. #BT gives the number of back trackings performed in the input-value search analysis. TotT gives the total.
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 back-tracking, resulting in a fairly large number of WCET runs for relatively small input domains.

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

We also see that the two different heuristics (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.

Table 8.17 gives 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. 

Table 8.17: Analysis results for benchmarks affected by program slicing

<table>
<thead>
<tr>
<th>Program</th>
<th>#Vars</th>
<th>#I</th>
<th>Basic</th>
<th>Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MinT</td>
<td>MinT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MaxT</td>
<td>MaxT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#BT</td>
<td>#BT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TotT</td>
<td>TotT</td>
</tr>
<tr>
<td>nsichneu</td>
<td>5</td>
<td>2^1</td>
<td>24</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>92780</td>
<td>46702</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1638990</td>
<td>644249</td>
</tr>
<tr>
<td>esab_mod</td>
<td>15</td>
<td>2^3</td>
<td>216</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>67501</td>
<td>76687</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>610125</td>
<td>616016</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3956334</td>
<td>1185105</td>
</tr>
<tr>
<td>task</td>
<td>17</td>
<td>2^4</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>328</td>
<td>328</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>656</td>
<td>625</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>70164</td>
<td>72897</td>
</tr>
</tbody>
</table>

Table 8.17: Analysis results for benchmarks affected by program slicing
input value space, when using slicing, are significant for task7. However, for task7 the effects are more moderate. This was because most input data removed for task7 were relatively small in value size, or static pointers holding only a single abstract pointer value.

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

8.2 Allocating components to tasks

The system data is produced by creating a random schedulable task set, on which all components are randomly allocated. The component properties are deduced from the task they are allocated. Transactions are deduced the same way from the task set. In this way it 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 numbers of components in the systems are ranging in twenty steps from 40 to 400, with a main point on 120 components.
- The period times for the components are randomly selected from a predefined set of different periods between 10 and 100 ms.
- The worst case execution time (WCET) is specified as a percentage of the period time and chosen from a predefined set. The WCETs together with the periods in the system constitutes the system load.
- The transaction size is the size of the generated transactions in percentage of the number of components in the system. The transaction size is randomly chosen from a predefined set. The longer
the transactions, the more constraints, regarding schedulability, on how components may be allocated.

- The transaction deadline laxity is the percentage of the lowest possible transaction deadline for the generated system. The transaction deadline laxity is evenly distributed among all generated systems and is always greater or equal to one, to guarantee that the generated system is possible to map. The higher the laxity, the less constrained transaction deadlines.

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

To get as realistic systems to simulate as possible, the values used to generate systems are gathered from some of our industrial partners. The industrial partners chosen are active within the vehicular embedded system segment. The task switch time used for the system is 22 µs, and the tcb size is 300 bytes. The task switch time and tcb size are representative of commercial RTOS tcb sizes and context switch times for common CPUs. The simulations are performed for four different utilization levels, 30%, 50%, 70% and 90%. For each level of utilization 1000 different systems are generated with the parameters presented above.

In table 8.2 we show the specific data used for generating systems to the simulations. The GA was setup with an initial population of 300 individuals, and every simulation was run for 500 generations. The simulations were run on a 1.8 GHz Pentium 4m processor with 768 MB of RAM. The mean time for each simulation is 133 seconds.

### 8.2.1 Results

A series of simulations have been carried out to evaluate the performance of the proposed framework. To evaluate the schedulability of the systems, FPS scheduling analysis is used. The priorities are randomly assigned by the genetic algorithm, and no two tasks have the same priority. We compare the allocation approach described in this paper to one-to-one allocations. Table 8.2.1 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
### 8.2 Allocating components to tasks

#### Param on component level

<table>
<thead>
<tr>
<th>Number components</th>
<th>WCET in % of period</th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>2</td>
<td>45</td>
</tr>
<tr>
<td>50</td>
<td>4</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>70</td>
<td>6,25</td>
<td>10</td>
</tr>
<tr>
<td>80</td>
<td>2,75</td>
<td>25</td>
</tr>
<tr>
<td>100</td>
<td>7,5</td>
<td>25</td>
</tr>
<tr>
<td>120</td>
<td>13</td>
<td>35</td>
</tr>
<tr>
<td>140</td>
<td>7,5</td>
<td>10</td>
</tr>
<tr>
<td>150</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

#### Stack size

<table>
<thead>
<tr>
<th></th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>10</td>
</tr>
<tr>
<td>80</td>
<td>16</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>120</td>
<td>25</td>
</tr>
<tr>
<td>140</td>
<td>30</td>
</tr>
<tr>
<td>160</td>
<td>35</td>
</tr>
<tr>
<td>180</td>
<td>40</td>
</tr>
<tr>
<td>200</td>
<td>45</td>
</tr>
<tr>
<td>220</td>
<td>50</td>
</tr>
<tr>
<td>240</td>
<td>55</td>
</tr>
<tr>
<td>260</td>
<td>60</td>
</tr>
<tr>
<td>280</td>
<td>65</td>
</tr>
<tr>
<td>300</td>
<td>70</td>
</tr>
<tr>
<td>320</td>
<td>75</td>
</tr>
<tr>
<td>340</td>
<td>80</td>
</tr>
<tr>
<td>360</td>
<td>85</td>
</tr>
<tr>
<td>380</td>
<td>90</td>
</tr>
<tr>
<td>400</td>
<td>95</td>
</tr>
</tbody>
</table>

#### Stack size % of num. comp.

<table>
<thead>
<tr>
<th></th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
</tr>
<tr>
<td>30</td>
<td>22</td>
</tr>
<tr>
<td>35</td>
<td>25</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>45</td>
<td>31</td>
</tr>
<tr>
<td>50</td>
<td>34</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>65</td>
<td>43</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
</tr>
<tr>
<td>75</td>
<td>49</td>
</tr>
<tr>
<td>80</td>
<td>52</td>
</tr>
<tr>
<td>85</td>
<td>55</td>
</tr>
<tr>
<td>90</td>
<td>58</td>
</tr>
<tr>
<td>95</td>
<td>61</td>
</tr>
</tbody>
</table>

#### Laxity % of critical

<table>
<thead>
<tr>
<th></th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
</tr>
<tr>
<td>30</td>
<td>22</td>
</tr>
<tr>
<td>35</td>
<td>25</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>45</td>
<td>31</td>
</tr>
<tr>
<td>50</td>
<td>34</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>65</td>
<td>43</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
</tr>
<tr>
<td>75</td>
<td>49</td>
</tr>
<tr>
<td>80</td>
<td>52</td>
</tr>
<tr>
<td>85</td>
<td>55</td>
</tr>
<tr>
<td>90</td>
<td>58</td>
</tr>
<tr>
<td>95</td>
<td>61</td>
</tr>
</tbody>
</table>

#### Isolation %

<table>
<thead>
<tr>
<th></th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

#### Utilization %

<table>
<thead>
<tr>
<th></th>
<th>Dist. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

#### Period time (µs)

<table>
<thead>
<tr>
<th></th>
<th>GA property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>Population</td>
<td>300</td>
</tr>
<tr>
<td>25000</td>
<td>Generations</td>
<td>500</td>
</tr>
<tr>
<td>50000</td>
<td>Elite rate</td>
<td>5%</td>
</tr>
<tr>
<td>100000</td>
<td>Cull rate</td>
<td>40%</td>
</tr>
<tr>
<td></td>
<td>Mutation rate</td>
<td>1%</td>
</tr>
</tbody>
</table>

Table 8.18: Data used for generating systems, and GA parameter
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 has at least one solution. The stack and CPU values are only collected from systems where a solution was found.

<table>
<thead>
<tr>
<th>Load</th>
<th>Laxity</th>
<th>Stack</th>
<th>CPU</th>
<th>success</th>
<th>Stack</th>
<th>CPU</th>
<th>success</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>30%</td>
<td>25949</td>
<td>4.1%</td>
<td>99%</td>
<td>14970</td>
<td>1.6%</td>
<td>58%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>26755</td>
<td>4.1%</td>
<td>95%</td>
<td>15503</td>
<td>2.0%</td>
<td>99%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>33077</td>
<td>4.4%</td>
<td>78%</td>
<td>21005</td>
<td>2.2%</td>
<td>97%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>37943</td>
<td>4.9%</td>
<td>98%</td>
<td>23749</td>
<td>2.3%</td>
<td>100%</td>
</tr>
<tr>
<td>50%</td>
<td>All</td>
<td>22727</td>
<td>4.8%</td>
<td>92%</td>
<td>22279</td>
<td>2.4%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>33591</td>
<td>4.3%</td>
<td>99%</td>
<td>21346</td>
<td>2.3%</td>
<td>64%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>38251</td>
<td>4.8%</td>
<td>88%</td>
<td>25350</td>
<td>2.5%</td>
<td>96%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>37043</td>
<td>4.9%</td>
<td>98%</td>
<td>23749</td>
<td>2.3%</td>
<td>100%</td>
</tr>
<tr>
<td>70%</td>
<td>All</td>
<td>24455</td>
<td>5.1%</td>
<td>85%</td>
<td>30894</td>
<td>2.7%</td>
<td>91%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>44226</td>
<td>5.0%</td>
<td>58%</td>
<td>31638</td>
<td>2.7%</td>
<td>73%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>44267</td>
<td>5.1%</td>
<td>94%</td>
<td>30868</td>
<td>2.7%</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>44619</td>
<td>5.2%</td>
<td>98%</td>
<td>30232</td>
<td>2.6%</td>
<td>100%</td>
</tr>
<tr>
<td>90%</td>
<td>All</td>
<td>46943</td>
<td>5.6%</td>
<td>87%</td>
<td>37933</td>
<td>3.1%</td>
<td>95%</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>54885</td>
<td>5.7%</td>
<td>65%</td>
<td>41207</td>
<td>3.4%</td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>49607</td>
<td>5.5%</td>
<td>92%</td>
<td>35470</td>
<td>3.0%</td>
<td>98%</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>53535</td>
<td>5.7%</td>
<td>98%</td>
<td>38260</td>
<td>3.1%</td>
<td>99%</td>
</tr>
</tbody>
</table>

Table 8.19: Memory, CPU overhead and success ratio for 1-1 and GA allocations

The first graph for the simulations (Figure 8.15) shows the success ratio, i.e., the percentage of systems that were possible to map with the one-to-one allocation, and the GA allocation respectively. The success ratio is relative to the effort of the GA, and is expected to increase with a higher number of generations for each system. Something that might seem confusing is that the success ratio is lower for low utilization than for high utilizations, even though, intuitively, it should be the opposite. The explanation to this phenomenon is that the timing constraints become tighter as fewer tasks participate in each transaction (lower utilization often leads to fewer tasks). With fewer tasks the task phasing, due to different periods, will be lower, and the deadline can be set tighter.

The second graph (Figure 8.16) shows that the deadlines are relaxed with higher utilization, since the allocations with relaxed deadlines perform well, and the systems with a more constrained deadline show a clear improvement with higher utilization.
8.2 Allocating components to tasks

The third graph (Figure 8.17) shows for both approaches the average stack size for the systems at different utilization. The comparison is only amongst allocations that have been successfully mapped by both strategies. The memory size consists of the tcb and the stack size, and the tcb size is 300 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 8.18) 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 smart methods to map components to tasks. A better strategy for setting priorities would probably lead to an improvement in the success ratio. Further we observe that lower utilization admits larger improvements than higher laxity of the deadlines; and since lower utilization in the simulations often gives tighter deadlines, we can conclude that the allocation does not negatively impact schedulability. However, regarding the improvements, the more components the more constraints are put on each transaction, and thereby on the components, making it harder to perform good allocations.

Results from simulations show that the 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% re-
duced 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.

### 8.3 Evaluation tool

Most of the outlined ideas have been implemented in our WCET analysis tool SWEET (SWEdish Execution time Tool) [WCE06]. Unlike most WCET analysis tools, SWEET is integrated with a compiler and performs the flow analysis on its intermediate code representation (NIC) [WPO01]. SWEET includes an input-sensitive flow analysis called Abstract Execution (AE) [GESL06b], which is a form of symbolic execution based on VA. The AE is capable of deriving various type of flow information and can handle almost full ANSI-C, including pointers, bit operations, and aggregate data structures such as structs and arrays [GESL06b, Tan06]. The AE has several options for trading precision and analysis time. SWEET also do a type of slicing to restrict the AE to only analyze those statements and variables that may affect the program flow [SEGL06].

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

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

As mentioned above, we have not implemented all of the outlined ideas. More specifically, we do not perform our slicing on the object code but on NIC, we only slice on whole statements.

We use program slicing for deriving the input variables whose different values may cause the program execution time to vary, allowing the input data search space to be reduced for any type of measurement-based or static analysis method to derive the WCET input values.
Chapter 9

Summary and Conclusions

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

9.1 Summary

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

In chapter 1 we give an introduction to our research, starting with an illustrative example from the real world before we give an overview of the specific research and contributions. Chapter gives an introduction to embedded and real-time systems and its terminology and definitions.

In chapter 3 we give an introduction to component-based software engineering terminology and definitions. We also discuss the industrial motivations for using component-based software engineering. In chapter 4 we describe the industrial and research problems and discuss the research methodology we have used in order to solve the research problem.
Chapter 5 positions our methods (that are described in chapters 6 and 7) in the component-based development process. In chapter 6 we outline two novel methods, based on a combination of static WCET analysis and systematic search over the value space of input variables, for deriving WCET behaviour of the program. Chapter 7 presents a method for optimizing the resource usage in component-based real-time systems based on real-time analysis, calculating resource consumption and genetic algorithms for deriving mappings between components and tasks that are optimized for low resource consumption, while maintaining stipulated real-time requirements. Finally in chapter 8 we present the evaluations of the technical contributions presented in Chapters 6 and 7.

9.2 Applicability

One of the contributions of the thesis is the method to acquire the input value combination that produces the program WCET for a piece of software. This was not a part of the initial research problem. In a joint work with Dr. Andreas Ermedahl and Dr. Peter Altenbernd we found that the methods for acquiring reusable WCET can be used also for finding the input value combination that produces the program WCET. This is an important contribution on its own as it can be used for strengthening measurement based and hybrid WCET methods.

We believe that there exist more applications, some of which we discuss as potential future research directions.

9.3 Conclusion

Embedded system software if compared to hardware, usually involves a significant overhead in terms of energy consumption and execution time. However, for flexibility reasons, hardware cannot be used in applications with changing requirements. In order to make a software implementation feasible, efficiency of embedded software is a must. Various approaches for achieving this efficiency have been explored.

Due to the efficiency requirements, using power-hungry, high-performance off-the-shelf processors from desktop computers is infeasible for many applications. Therefore, the use of customized processors is becoming
more common. The programs running on these processors must be optimized for resource utilization and performance.

Many embedded systems are real-time systems and have requirements on dependability. To fulfill such requirements, while maintaining high resource utilization it is important to have accurate predictions.

However, with increasing software complexity and more functions are integrated into the same component the software behaviour become increasingly varying. Properties of the component such as time and reliability are variable and usage-dependent, and the variance may be large. For software, in particular, a usage independent characterization of component properties is inadequate for accurately predicting the properties of the composite system constructed using these components.

Comment 1: In [FSm05a] we have shown that transformations from components to tasks potentially give high benefits in terms of increased resource efficiency. We have also shown that tighter WCET estimations (more laxity of the timing constraints) produces a higher number of feasible mappings, and hence a greater chance to find a better mapping compared to one-to-one mappings.

9.4 Contributions

In this thesis we have introduced several novel methods on improving utilization and prediction accuracy for reusable software components. We have extended the component-based software engineering (CBSE) development process with our methods, and we have also developed tools and evaluated the methods with both industrial code and academic benchmarks.

We begin with revisiting the requirements from Chapter 4, and discuss whether the requirements are fulfilled and to what degree.

(Requirement A1) WCET analysis shall be performed in the component development part of the CBSE development process.

To satisfy requirement A1, the reusable WCET analysis is divided into two different parts. Firstly, reusable component WCET analysis which is located in the component development part of the CBSE development process; secondly context sensitive WCET analysis, which parameterizes the reusable analysis with a system specific usage. This requirement is satisfied by partial solution 1.
(Requirement A2) It shall be possible to reuse a software component without re-doing WCET analysis.

To satisfy requirement A2, a component is augmented with a quality of service contract that can be parameterized with usage. This allows a component to be parameterized with usage to get a WCET instead of re-analyzing the component. However, if the reuse causes any changes to the component other than usage input parameters, e.g., any hardware changes, then it is required to redo the analysis. Possible future work could include adding, e.g., hardware dependencies to the quality of service contract.

(Requirement A3) Component WCET analysis shall be composable.

Component quality of service contracts are composable in the way that output from a component can be used as input to a subsequent component quality of service contract. However, we have not provided evidence that WCET is composable in the first place.

(Requirement A4) 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.

The reusable WCET analysis is based on systematic search over the input space and can produce accurate execution time for every possible input. The accuracy is limited by (i) the accuracy of the WCET tool at hand, and (ii) the search time allowed. The results show...

(Requirement M1) Component shall be mapped to tasks such that the resource efficiency is never lower than for a system that is mapped with one component to one task.

The allocation framework shows that an allocation where several components are allocated to one task can never have worse resource utilization than for an allocation where each component is allocated to one task. Also, the results show...

(Requirement M2) Components must be allocated to tasks in such a way that the temporal correctness of the system is maintained.

Components have several dependencies in terms of, e.g., precedence and timing constraints. These dependencies renders many allocations unfeasible. If there exists at least one possible allocation we can find it. If the search space is large it may require a long time. The results show...
The specific contributions of this thesis is a number of methods, tools and frameworks. The contributions are summarized below:

- Division of WCET analysis into component WCET analysis and system WCET analysis.
- Extension of CBSE process workflow, with reusable WCET analysis and transformation to real-time tasks.
- Methods and a framework for transformation of component models to real-time models.
- A tool based on genetic algorithms for allocating software components to real-time tasks.
- Methods for contract based reusable WCET analysis.
- Methods for deriving the concrete input combination that leads to the program WCET.
- A tool based on binary search for deriving reusable and absolute WCET in software components.

### 9.5 Possible future research directions

Regarding future work, we plan to investigate reuse of flow- and low-level analysis results between WCET calculations. For example, for many WCET analysis tools the low-level analysis result, which gives upper bounds on the execution time for individual instructions and basic blocks, are likely to be non-affected by the different input value constraints under which the analysis is run. Similarly, by performing a forward slice upon the input variables that had their input changed, we should be able to determine what code parts the may have had their flow analysis results affected by the change. Only those code parts may need to get their flow information recalculated. Thus, it should be possible to reuse flow- and low-level analysis results from already performed WCET calculations, thereby speeding up the WCET input values determination analysis.

Finally, we would also like to evaluate the outlined methods on more industrial codes available at our end-user company partners [WCE06].
Future work includes adding other allocation criteria, e.g., by looking at jitter requirements, and blocking. By adding jitter constraints and blocking, trade-offs arise between switch overhead and memory size versus deviation from nominal start and end times and blocking times. Furthermore, a more efficient scheduling policy and priority assignment will be applied. Due to the nature of GA it is easy to add new optimizations as the ones suggested above.

### 9.5.1 Implications for WCET analysis

In all software development processes we compile components or modules, but we don’t compile all modules again and again, but we compile only the modified ones and link them with the others. This should also be true for WCET analysis, however, today there are no efficient way of reusing WCET analysis results. If we analyze a piece of code, we should be able to store this information in a way that can be used later on and make the analysis easier. So that it is not required to do the whole complex analysis again and again, but rather do the complex analysis tasks only once, and reuse our results cheaply, so we don’t have to go through all the analysis and all the components again and again.

A big problem today is that when large systems that are analyzed with WCET analyzers are exposed to minor changes, the whole system is still re-analyzed. For large and complex system like, e.g., avionics, the WCET analyses may take several days. Therefore there is a strong desire to be able to do partial WCET analysis.

With the described method we propose that more information can be stored in the parameterizable contract. For instance, a promising approach is to store a list of all basic blocks that have been affected by the annotated analysis. If the WCET analysis is annotated in such a way that some parts of the source code is not affected, the basic blocks representing that code would not be in that list. Most state of the practice WCET tools can provide this information, and can also give a mapping between basic blocks and source code. This implies that if a component is changed, only those clusters that “contain” the affected basic blocks needs to be re-analyzed as depicted in figure 9.1. Effectively this gives us the possibility of incremental WCET analysis.

When re-compiling parts of the code it needs to be re-linked. This may in turn result in that the different pieces (basic blocks) may be aligned
Figure 9.1: Three different annotation scenarios w.r.t basic blocks.

different in the memory leading to, e.g., different cache behaviours. We propose that this can easily be solved with using a static link-map, where the linker is forced to align the different basic-blocks in a specified way. In many safety critical systems it is desirable to have control over the linker anyway.

It can also be used iteratively in case of design changes; however with some obvious drawbacks on the results. The framework can easily be extended to support other optimizations, besides task switch overhead and memory consumption.
Appendix A

Complete list of tables and graphs
Appendix A

Complete list of publications

Publications related to the thesis contributions

1: Deriving the Worst-Case Execution Time Input Values, Andreas Ermedahl, Johan Fredriksson, Peter Altenbernd, submitted to the 29th IEEE Real-Time Systems Symposium (RTSS’08).

Abstract:
Usage in thesis:
My contribution:


Abstract:
Usage in thesis:
My contribution:

3: Clustering Worst-Case Execution Times for Software Components, Johan Fredriksson, Thomas Nolte, Andreas Ermedahl and Mikael Nolin, in proceedings of the 7th International Workshop on

Abstract:
Usage in thesis:
My contribution:


Abstract:
Usage in thesis:
My contribution:


Abstract:
Usage in thesis:
My contribution:


Abstract:
Usage in thesis:
My contribution:


Abstract:
Usage in thesis:
My contribution:
Publications related to the thesis


15: A component-based development framework for supporting functional and non-functional analysis in control system de-
Chapter A. Complete list of publications


17: Reusing Worst-Case Execution Time Analysis with Component Contracts, Johan Fredriksson, Thomas Nolte, Mikael Nolin, Heinz Schmidt (external), Proceedings of the 9th Real-Time in Sweden (RTiS’07), Västerås, Sweden, August, 2007


23: Context Aware Optimizations for Embedded Real-Time Components, Johan Fredriksson, PhD Proposal


26: **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 Nystöm, 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

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

28: **Worst-Case Execution Time Clustering for Software Components**, Johan Fredriksson, Thomas Nolte, Andreas Ermedahl, Mikael Nolin, Technical Report, Department of Computer Science and Electronics, Mälardalen University, April, 2007


30: **Contract-Based Reusable Analysis for Software Components**


Publications not related to the thesis

33: A component-based approach for supporting functional and non-functional analysis in control loop design, Massimo Tivoli (former), Johan Fredriksson, Ivica Crnkovic, Tenth International Workshop on Component-Oriented Programming, Glasgow, Scotland, July, 2005


36: Handling Subsystems using the SaveComp Component Technology, Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, Mikael Nolin, Thomas Nolte, John Håkansson, Paul Pettersson, Workshop on Models and Analysis for Automotive Systems (WMAAS’06) in conjunction with the 27th IEEE Real-Time Systems Symposium (RTSS’06), Rio de Janeiro, Brazil, Editor(s):Marco Di Natale and Luis Almeida, December, 2006

37: On the Teaching of Distributed Software Development, Ivica Crnkovic, Igor Cavrak (external), Johan Fredriksson, Rikard Land,
Mario Zagar (external), Mikael Åkerholm, 25th International Conference INFORMATION TECHNOLOGY INTERFACES, IEEE, Dubrovnik, Croatia, June, 2003

38: **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


Bibliography


[AZP03] Andreas S. Andreou, Andreas C. Zographos, and George A. Papadopoulos. A three-dimensional re-


[CHMW07] Joel Coffman, Christopher Healy, Frank Mueller, and David Whalley. Generalizing parametric timing analy-


[KP] Raimund Kirner and Peter Puschner. Discussion of misconceptions about wcet analysis.


[LME98] Yau-Tsun Steven Li, Sharad Malik, and Benjamin Ehrenberg. Performance Analysis of Real-Time Embedded Soft-


[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holst, 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


