Mälardalen University Dissertations
No.5

Architectural Modeling and Analysis of Complex Real-Time Systems

Anders Wall

September 2003

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISBN 91-88834-05-0
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Most automation systems and other large industrial software systems have long lifetimes, and customers expect these systems to be supported as long as they are in operation. Furthermore, software components in these systems may be reused in different products, e.g. using a software product line approach. Hence, the lifetime of software in individual systems may be very long; several decades or even longer.

Software that is used for a long time will be exposed to frequent changes as the system evolve over time, e.g. due to adding new functionality, error corrections, or changing the hardware platform. The larger and older the system is, the harder it becomes to foresee the consequences of changes.

In this thesis we present three different techniques for managing the evolution of large and complex real-time systems. The techniques are based on analytical modeling, predicting different quality properties, e.g. temporal correctness, by analyzing a model of the software. The first technique is a component model with analytical interfaces (ReFlex) that allows us to predict different properties of a component assembly, the second is a probabilistic modeling language which is analyzed by simulations (ART-FW), and the third technique is an extension of classical timed automata with a notion of real-time tasks (TAT).

Ideally, the analytical models should evolve together with the software. However, since new features are often added and the implementation is often changed without updating the model, the model becomes obsolete and predictions based on the model are no longer valid. By applying the techniques proposed in this thesis, we can re-introduce analyzability; Using ReFlex we can update the analytical aspects while redesigning the system. Unless ReFlex has been used in the earlier design, this will require a costly redesign of the complete system, but consistency between the analytical model and the implementation will be ensured. Using ART-FW or TAT the implementation will be kept untouched by

introducing a separate model. The drawback is that an extra effort is required to keep the model consistent with the implementation.

We have applied ART-FW in the re-engineering activity of a large industrial system. The results indicate that the approach is indeed applicable on real systems.

To Pernilla and Gustav

Acknowledgments

I would like to thank my supervisor Christer Norström for giving me the opportunity to enter deeply into interesting aspects of computer engineering. Christer has been a great support and is always a source of inspiration and motivation.

I would also like to thank Kristian Sandström, Hans Hansson, Jukka Mäki-Turja, Mikael Nolin, Dag Nyström, Wang Yi, and Jan Gustafsson for their valuable comments and suggestions that have increased the quality of this thesis.

Thanks to Johan Andersson and Jonas Neander for the implementation of the ART-ML parser and the simulator, which was done as part of their master thesis project.

Last but not least, thanks to the colleagues at the department of computer science and engineering that all contribute to an inspiring and creative working environment.

This work has been supported by MRTC (Mälardalen Real-Time research Centre), and ARTES (A network for Real-Time research and graduate Education in Sweden). ARTES is supported by SSF (Swedish Foundation for Strategic Research).

Publications

I have authored/co-authored the following publications:

Articles in collections

- Henrik Thane, Anders Wall, Testing Reusable Software Components in Safety-Critical Real-Time Systems, In the book: Building reliable component-based software systems, 2002. Artech House Publishers ISBN: 1-58053-327-2

Conferences and workshops

- Christer Norström, Anders Wall, Johan Andersson and Kristian Sandström, Increasing Maintainability in Complex Industrial Real-Time Systems by Employing a Non-intrusive Method, Accepted for publication at the NetObjectDays Conference in the Workshop on Migration and Evolvability of Long-life Software Systems, Erfurt, Germany, September 2003

- Anders Wall, Johan Andersson, and Christer Norström, Probabilistic Simulation-based Analysis of Complex Real-Times Systems, In proceedings of the 6th IEEE International Symposium on Object-oriented Real-time distributed Computing, Hakodate Hokkaido, Japan, May 2003

- Anders Wall, Johan Andersson, Jonas Neander, Christer Norström, and Martin Lembke, Introducing, Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems, In proceeding of the 9th Conferance on Real-Time Computing Systems and Applications, Tainan, Taiwan, February 2003.

- Anders Wall, Magnus Larsson, Christer Norström, Towards an Impact Analysis for Component Based Real-Time Product Line Architectures, In Euromicro Conferance on Component Based Software Engineering, September 2002.
- Anders Wall, Magnus Larsson, Christer Norström, Ivica Crnkovic, Using Prediction Enabled Technologies for Embedded Product Line Architectures, In the 5th International Conference on Software Engineering Workshop on Component-Based Software Engineering, May 2002.
- Anders Wall, Christer Norström, A Component Model for Embedded Real-Time Software Product-Lines, In the 4th IFAC conference on Fieldbus Systems and their Applications Nancy (France) , November 2001.
- Anders Wall, Kristian Sandström, Jukka Mäki-Turja, Christer Norström, Verifying Temporal Constraints on Data in Multi-Rate Transactions, In proceeding of the 7th Conferance on Real-Time Computing Systems and Applications, Korea, December 2000.
- Christer Norström, Anders Wall, and Wang Yi, Timed Automata as Task Models for Event-Driven Systems, In proceeding of the 6th Conferance on Real-Time Computing Systems and Applications, Hong Kong , December 1999.

Technical reports

- Anders Wall, Markus Lindgren, and Tage Tarkpea, Experiences from Introducing UML and OO in an Organization, MRTC Technical Report, December 2002.
- Anders Wall, Joakim Fröberg, Christer Norström, Using Analytical Models of Complex Real-Time Systems for Temporal Impact Analysis, MRTC Technical Report, November 2002.
- Anders Wall, Kristian Sandström, Christer Norström, Product Line Architectures for Embedded Real-Time Systems, MRTC Technical Report, December 2000.
- Anders Wall, Software Architectures for Real-time Systems, MRTC Technical Report 00-20, May 2000.

- Henrik Thane, and Anders Wall, Formal and Probabilistic Arguments for Reuse and Reverification of Components in Safety-Critical Real-Time Systems, MRTC Technical Report, January 2000.
- Anders Wall, Software Architectures: An overview, MRTC Technical Report, October 1999.

Licentiate Thesis

- Anders Wall, A Formal Approach to Analysis of Software Architectures for Real-Time Systems, Licentiate thesis, Dept. of Computer Systems, Uppsala University and Dept. of Computer Engineering, Mälardalen University, September 2000.

Contents

Chapter 1

Introduction

In this chapter we motivate the work presented in this thesis and briefly describe the specific contributions.

1.1 Motivation

Software reuse and component based software engineering are considered to be the potential solution for developing software based product faster, with higher quality, and to a lower price. This is true especially in domains where time to market is very critical such as for consumer products, e.g. cellular telephones. Delaying an introduction of a new cellular telephone model on the market may cause huge losses in revenue, losses of market shares and goodwill. Besides shortening development time, properly handled reuse will also improve the reliability since code is executed for longer time and in different contexts [FP96].

However, to make software reuse deliver, we must not only focus on the development of reusable software components. The architecture must also be taken into consideration, as well as processes and organizational issues. By treating the architecture as a reusable asset in itself, it may be reused across a product line. We will refer to such reusable architectures as product line architectures [DKO+96][Bos00][CN01].

We define a product line to be a set of software products that share a common technology platform as well as having common functionality. A generic software architecture that constitutes the base on which all products in the line are built, is called a product line architecture (PLA). The major part of a product in a product line is typically based on reusable assets, e.g. software components, architecture, requirements.

For each product a product architecture (PA) is derived from the PLA upon product instantiation. Tailoring and adopting the architecture and its components are used to construct products that belong to a particular product line. The tailoring can be achieved by, e.g. parameterization of generic reusable components and product-specific implementations of software components. The way in which we develop products is a strategy that belongs to, and is developed together with, the product line.

Applying a product line approach has more implications than just on the software in itself. It will influence also the software tools, configuration management (CM), and the complete organization of the software development departments. Thus, besides developing strategies for developing a product's software, organizational strategies, strategies with respect to development tools and CM, and education of engineers within the organization, must be developed. Questions such as ownership of product assets and education of engineers that develop generic software and engineers that build products is essential. If the fundamental software strategies are not adhered to, the lifetime of the investment in a product line will be reduced. Consequently, the payoff due to reuse and reduced maintenance will not come about, or at least it will be reduced.

1.1.1 Real-time systems

In this thesis we focus on component based development and product line architectures for a particular domain of software systems: *real-time systems*. Real-time systems are characterized by their temporal requirements. Besides being functionally correct, i.e. exhibit the correct functional behavior, they must also be *temporal correct*. By temporal correct we mean the correct function is provided at correct time. Correct time is not necessarily as fast as possible, but sufficiently fast, or slow. A classical example of a real-time system is a control system for an airbag. Inflating the airbag to late when a collision is detected results in the driver hitting the steering wheel. Moreover, inflating the airbag to early may have the same result as the airbag is deflated when the driver hits the steering wheel and the dashboard. Consequently, temporal correct in this context is, neither too fast, nor too slow. It is the *temporal requirements* of the system that defines exactly what a temporal correct system is. Traditionally, we divide real-time systems into two classes: *hard real-time systems*, and *soft real-time systems*. Hard real-time systems are those where the temporal correctness is critical. If they are violated the consequences may be catastrophic. The airbag system described above is

in this class. Hard real-time systems are often considered *safety related*, or *mission critical*. Hence, the ability to analyze and predict the systems behavior is crucial.

Soft real-time system on the other hand are those where the consequences of an incorrect temporal behavior are not catastrophic. Typical examples of such applications are cellular telephones and video streaming. Violating a temporal requirement only results in poor audio, or video quality.

Regardless of the system's criticality, i.e. hard or soft, we would like to provide means for analyzing the system with respect to temporal behavior and resource utilization (CPU, communication buses, memory). This is obvious when it comes to hard real-time systems, but the ability to analyze a soft real-time system is also important. A typical example is that we would like to analyze the effect of a maintenance activity, i.e. adding new functionality, or changing existing functionality, with respect to the resources in the system.

1.1.2 Analytical models and analysis

In order to enable system analyses we must have system models that facilitate such analyses. In this thesis we refer the such models as *analytical models*. The analytical model of a system supplies the information necessary for performing the required analyses. The information required for analyzing a system's temporal behavior are, for instance, execution times of a piece of software and the frequency with which that piece of software executes. We refer to an entry in the temporal analytical model as a *temporal attribute*. The exact appearance of the temporal part of a system's analytical model is decided by the scheduling strategy that is adopted by the real-time operating system.

A system's temporal behavior is implemented by assigning temporal attributes, e.g. by assigning period times and priorities to tasks in a fixed-priority system. Consequently, in order to be temporal correct the temporal attribute assignment must fulfill the temporal requirements. Temporal requirements may come in various forms and may be *implicitly*, or *explicitly* expressed. Implicit temporal requirements are derived from the application and the environment in which it operates. Typical examples of such requirements are precision, control performance, and system dynamics that eventually will be converted into temporal requirements on the software. For instance, a control system that samples a particular process must do so with a certain frequency in order to get a

correct view of the process values of interest. Explicit temporal requirements are those expressed directly as temporal properties, e.g. a latency requirement.

There exists an abundance of methods for analyzing a real-time system's temporal behavior, i.e. whether or not the system comply with its temporal requirements. However, those methods often assume too simplistic analytical models. Systems that have grown in complexity during many years often exhibit complex temporal behavior which can not easily be captured by existing analytical models and analysis methods. For instance, assuming worst-case scenarios, e.g. worst-case execution times, may be a far too pessimistic assumption. Applying such a method on some systems may gives a result that indicates that the system is not schedulable, even though it is working properly. The reason for this is that there are semantical couplings between different software components in the system which inhibit them from all executing their worst-case execution time simultaneously. An example of such a dependency is components that communicate with each other. Imagine two communicating components c_1 and c_2. In the cases where c_1 sends a messages to c_2, it exhibit its WCET. However, c_2 exhibit an execution time less than its WCET when receiving the message from c_1. Taking this dependency among c_1 and c_2 into consideration when analyzing the system's temporal behavior give as a result a feasible system which may not be the case when always assuming WCET for all components. Moreover, the correctness of a real-time system may not necessarily be expressed in terms of temporal requirements. Such properties are not handled by traditional real-time analysis. Another limitation of traditional analyses is that they produce results that are of binary nature, i.e. schedulable or not. They do not give any numbers on probability of failure, which would be useful in many cases.

1.1.3 Maintaining long-lived real-time systems

Large and complex systems usually have long life cycles, i.e. system or sub-systems live over decades. This is especially true when software is reused as in a product line approach where components are reused over and over again. During their lifetimes, systems are exposed to error corrections and functional improvements. We refer to this activity as *maintenance*. The complexity of the system makes it virtually impossible to foresee the consequences of altering the behavior or adding new functions to it if no means for analysis exist. Consequently, the analytical

models allow us to analyze the *impact* of a maintenance activity.

The long lifetime of a system also brings the problem of keeping the analytical model consistent with the implementation. While studying several complex real-time systems we have identified two different types of analytical models: the ones that are part of the components' specifications, and the ones that live in parallel with the actual system. Component models that facilitates, or requires, that analytical properties are specified obviously enforce consistency of the analytical model as it is updated as part of the maintenance activity. However, analytical models that live in parallel with the implementation are exposed to the risk of not being updated when the implementation is changed (compare with system documentation which are often carelessly updated). Hence, we may arrive at a point where we have to re-introduce analyzability, e.g. re-constructing the analytical model. Such a re-introduction can be either *intrusive* or *non-intrusive*. The most extreme form of intrusivity is a complete *re-design* and *re-implementation* of the system and its analytical model. Such an activity is costly and associated with many risks, e.g. the new design may exhibit other deficiencies, the introduction of new technologies that can not be properly used. Nevertheless, systems may arrive at a point where such costs and risks are justified. A typical scenario is when the system grows out of its architecture due to, e.g. new features that was not foreseen when the initial architecture was constructed and which has been introduced by force rubbing out the initial design rationales. Examples of such design rationales are maintainability, reliability, performance, timing.

A not so drastic form of an intrusive approach is to *restore* the system and its model. By restoring a system we re-construct the analytical model and make as few changes in the implementation as possible in order to get an analyzable system. Even though this is an intrusive approach, the risks are minimized.

In a non-intrusive approach we re-engineer the system in order to develop an analytical model of it. Hence, we do not touch the implementation of the system in order to increase maintainability, we only provide means for analyzing the impact of maintenance. The most imminent risk associated with this approach is that the analytical models may become obsolete if the system evolves but the models do not. A non-intrusive approach is, typically, applicable on systems that behave reasonable well but where the impact of maintenance activities is hard to predict.

The problems we are targeting in this thesis are:

- How to develop complex embedded real-time systems using the product line approach.
- What should a component model that facilitate the development of complex embedded real-time product lines look like?
- How to construct, and re-construct, analytical models of complex embedded real-time systems.
- How to analyze real-time systems that have a complex temporal behavior which can not be captured and analyzed by existing analytical methods.
- How to increase maintainability of complex real-time systems by introducing analytical models.

1.2 Contribution

The contributions in this thesis are a flexible component model (ReFlex), for embedded real-time product lines and two different approaches to modeling and analysis of real-time systems: *probabilistic modeling*, and *Timed Automata with Tasks* (TAT).

The main scientific contributions are in essence:

- A component model for predictable real-time systems with mechanisms supporting a product line approach, allowing specification and analysis on the general product, on the architecture level, and on the specific product level,
- A method and a framework for re-introducing analyzability in complex real-time systems by the use of probabilistic models, a probabilistic query language, and simulations,
- An extension of the classical timed automata with real-time tasks.

1.2.1 Component model with analytical interfaces

ReFlex facilitates analysis of the temporal behavior as well as product consistency, i.e. that a valid product is assembled in terms of component versions and variants. The information necessary for the proposed analyses resides in the *analytical interfaces* of the components. The analytical interfaces are the analytical model of such a component assembly. Even

though we present a couple of concrete examples of analyses that apply to a real-time product line approach, we consider the analytical interface approach a general framework in which we can provide any information that we might find necessary for a particular analysis. By defining new analytical theories, and specify the analytical interfaces that provide the information required by these theories, we can extend the current framework. The component model is memory efficient in the sense that it do not require a large, and complex infrastructure, e.g. request brokers.

ReFlex facilitates development and temporal analysis of embedded real-time system product lines. The component model is based on the concept of prediction enabled component technologies and port-based objects [HMSW01][SVK97]. The model has been developed based on the requirements from the embedded real-time systems domain. The most important requirements that we have considered are:

- specification of temporal attributes
- memory usage
- analyzability and predictability
- variability

The temporal attributes are typically period times with which a component executes, priorities, etc. Basically, the temporal attributes define the constraints under which components execute and is input to the analysis of a systems temporal behavior. Resources such as memory are usually scarce in embedded systems. Hence, we can not afford to implement the system with a component model that requires a large and complex infrastructure, such as request brokers. Mechanisms such as request brokers that aim at providing flexibility by late binding will also introduce unpredictable temporal behavior. Consequently, it is hard to foresee and guarantee the temporal behavior of a system, i.e. the analyzability of the system is influenced negatively.

We have to live without the flexibility through, for instance, late binding. Nevertheless, our component model must provide some mechanisms for variability since it is to be used in product lines. Real-time systems may exhibit two different kinds of variability: *variability in the functional domain*, and *variability in the temporal domain*. The model we propose provide functional variability by parameterization of the services provided by the components, as well as allowing for specifying interfaces

that must be implemented when making an instance of a component. Temporal flexibility is supported by separating the temporal constraints from the component.

The price one usually has to pay when requiring an analyzable system is that it is difficult to add software that do not completely conform with the existing architecture. However, this may also be a benefit since resistance in the architecture itself will ensure that the initial design objectives implemented in the architecture is kept intact. There is a fundamental tradeoff between analyzability and flexibility. The more open and flexible the system is, the less analyzable it becomes and vice versa.

1.2.2 Probabilistic modeling and analysis

The probabilistic modeling and analysis approach is targeting modeling analysis of large and complex real-time systems. A complete framework called ART-FW has been developed which includes a modeling language (ART-ML), a probabilistic requirements language (PPL), a simulator, and a set of tool for system measurement and data processing. The analyses are based on simulations of models where execution times are specified as statistical distributions. The simulator produces statistical distributions that describes the tasks temporal behavior, e.g. start times, stop times, response times. The simulation approach also allows us to specify starvation requirements on message queues in a system. PPL is a requirement language which allow us to specify temporal requirements which are verified against the simulation results. The stringency of, and confidence in, a simulation approach in contrast to an analytical approach is also discussed.

ART-FW provide a *non-intrusive* approach to the introduction of analyzability in a real-time software system as the analytical model is separated from the implementation. In this thesis we present a method for introducing analyzability using ART-FW which covers measuring of the system, constructing models based on the measurements, validating the model, and analyze the simulation results.

One of the mentioned risks with a non-intrusive approach is that the model may, eventually, become obsolete as the system evolves. In order to manage this risk we need to incorporate the modeling as a part of the development process. The process of developing and maintaining analyzable systems through models is discussed in this thesis as well.

The probabilistic modeling and analysis approach has been success-

fully applied in a case study from which we report in this thesis. This case study was performed at ABB Robotics in Västerås, Sweden. ABB Robotics develops, and manufactures industrial robots and it is their complex control software that has been modeled and analyzed.

1.2.3 Timed automata with tasks

Timed Automata with Tasks is an extension of the classical timed automata [AD94]. The extension consists of the possibility to model real-time tasks as timed automata processes. Timed automata models can be mathematically verified by model checking. We have shown that we can transform the schedulability problem, i.e. the problem of proving that all tasks in a real-time system submit to their specified deadlines, into a reachability problem which has been proven decidable for timed automata. With TAT we can model complete systems, i.e. both the control software and the environment that the software controls. This is a desired property of a modeling language for real-time systems as the systems often interact, and control a physical process. Moreover, the temporal requirements on a real-time system is often derived from the temporal characteristics of the physical process. TAT allows us to verify the complete system, i.e. the temporal requirements on software in interaction with the physical environment.

However, TAT do not scale properly which makes it less feasible for complete models of very large systems. The scalability problem is not due to the modeling technique as such but rather a matter of tools and representations available today that can not cope with the large state-spaces.

1.3 The relation between the contributions

The modeling methods proposed in this thesis aim at providing models of embedded real-time systems that allow us to analyze different properties, e.g. temporal correctness. As a synergistic effect of providing analyzability we can improve the maintainability of the system as it permits us to analyze the effect of changing the system, i.e analyzing the *impact* of the changes. In Table 1.1 the modeling languages are compiled in a way that emphasize their features.

The problem of predicting the impact of a maintenance activity usually becomes evident late in the system's life cycle. Consequently, an implementation exists but no correct analytical models are present. Even

	ART-FW	ReFlex	TAT
SW architecture model	ART-ML	component assemblies	Timed automata
HW architecture model	simple	no	no
Environment model	no	no	yes
Requirements language	PPL	no	Temporal logic
Analysis method	Probabilistic simulation	analytical methods	Model checking

Table 1.1: The modeling languages and their features.

though there initially where correct models of the system, the models may become obsolete if they do not evolve with the system. This is similar to the problem of keeping documentation consistent with a system. The problem is rather insidious since the consequences in terms of handling the system's complexity often arise after many evolutions when the system has grown considerable. Finally, the consequences of changing the system can not be easily foreseen without a model. In order to promote analyzability and, consequently, provide the ability to analyze the impact of changing the behavior, we must *re-engineer* such a system. By re-engineering we mean identifying and modeling the system's architecture, measure its temporal behavior, and construct a valid system model. We say that such models are constructed *by re-engineering*. ART-FW and TAT provide non-intrusive methods to the re-engineering approach. Both ART-FW and TAT is similar in the way that models produces a state-space which is checked by expressing properties that reflect the requirements, and verifies the properties against the state-space. However, ATR-FW generates the state-space for a particular model by running many simulation scenarios which results in distributions that describes the temporal behavior of the system, while in TAT, the complete state-space is explored, i.e. every path of execution through the network of timed automata processes.

Note that the re-engineering approach is a continuous activity. To use this approach successfully the analytical model must be maintained during the complete life cycle of the system. If the analytical models are not continuously maintained, the effort of making the model can not be

justified.

ReFlex, on the other hand, provides an intrusive method since the system must be re-implemented using the ReFlex component model. On the other hand, since the analytical model is part of the implementation in ReFlex, the problem of ending up with inconsistent and obsolete models is minimized.

Ideally, software systems are designed through models. Designing systems through models is the ideal software development approach since it promotes means for early assessment of the design as well as means for managing systems' complexity. Moreover, by specifying components and interfaces early in the development phase, the integration of the system becomes very smooth, given that the implementation conforms to the restrictions specified in the component specification phase. We refer to this as *by construction*. The different methods proposed in this thesis can all be used in a by construction approach. However, ReFlex enforces that the models are kept intact through out the lifetime of the system.

1.4 Research method

Research is a continuous and iterative process of observing a phenomenon, finding relevant questions, formulating hypotheses, test whether or not the hypotheses holds, and evaluate the results. There are different ways, or methods, for carrying out the research activities. Often the results raise new relevant questions.

In this work the questions arise from both in existing results from academia and from the industrial point of view. Basically, by studying real-world systems and identify the problems we have found our questions. Note that the question that we find are dependent on what we are looking for. Consequently, we have no intention of finding, and solving all existing problems. However, we must make sure that the questions are general in the sense that they have a broader relevance than just for the studied system. We have assured the relevance, and the generality of our research questions by examining several systems in order to see that the questions apply, and by carrying out literature studies. Studying existing literature will also establish the academic relevance of the research questions.

Given the set of relevant research questions we have formalized hypotheses which are possible solutions to the problems that are being solved. This activity have in our case included implementation of tools

and methods. Again, by studying existing literature we can establish the approach in an academic perspective as well as its uniqueness.

Finally, we have verified our hypotheses by testing them on systems and analyzing the results, i.e. we have performed case studies. We say that we verify the hypotheses, not the research results. The reason for this distinction is that we consider the result from verifying the hypotheses as part of the research result. As research is an iterative process we may find new relevant and interesting research questions when evaluating the verification activity, or we may narrow the scope of the questions.

1.5 Outline

Chapter 2 provides a state-of-the-art description of the software architecture domain. The software architecture domain is rather broad so we have chosen to focus on architectural description (models), and architectural analysis. In Chapter 3 we discuss software product lines in general, and software product lines for embedded real-time systems in particular. It also briefly discuss organization, processes and business matters which are very important issues in a product line approach.

The contributions in this thesis is presented in Chapter 4 through Chapter 6 which are all organized in a similar way: an introductory discussion, related work, the contribution and finally a discussion and comparison between the contribution and the related work.

ReFlex is presented in Chapter 4 which introduce a constructive approach to provide analytical models of a system. The ART-FW is presented in Chapter 5 which also discuss the re-engineering approach to analytical models. In Chapter 6 the TAT extension to ordinary timed automata is described.

Finally, chapter 7 and Chapter 8 provide conclusions and future directions of the research presented in this thesis.

Chapter 2

Software architectures: modeling and analysis

In this chapter we present a state-of-the-art description of the software architecture area. Emphasis is on architectural modeling and architectural analysis.

2.1 Introduction

The number of projects in industry developing software is constantly increasing. Software is not only replacing old and well-established technologies, but also increasing in size and complexity. To manage the complexity, engineering methods for constructing software are needed, i.e. software engineering. Software engineering has been established as a broad discipline that covers topics ranging from requirement engineering, design, implementation, maintenance, and verification and validation. An established engineering practice is taken for granted in many engineering disciplines but not in the software community. In order to be considered an engineering practice, it must be possible to construct models that can be analyzed and verified. Moreover, design methods are needed including established techniques that have been proven successful as well as tools supporting the methods. The part of software engineering that focuses on high-level design and analysis is called software architectures.

Edsger Dijkstra pointed out, in a paper from 1967 [Dij67], the importance of partitioning and structuring software, in contrast to just focusing on programming to produce the correct functionality. This is

what software architecture, and software architectural analysis are all about. It deals with how to structure a software system and how to evaluate that structure with respect to different *non-functional requirements* which expressed in terms of *quality properties*. Typical examples of quality properties are, maintainability, reliability, performance, etc. The interest in the software architecture field has increased lately due to the increased functionality provided by software systems, the increased size and complexity, and the increased cost of developing and maintaining software products. Today, industry is aware of the benefits of being able to analyze and verify software constructions in an early phase of the development process. If a software development project diverges from the functional requirements or the non-functional requirements, and if those divergences are not detected early, the cost of revising the design in the end of the project will be significant due to redesign. Almost 80 percent of the cost for developing a software product are spent after the initial design and implementation phases [CN96]. These 80 percent are spent on maintenance, which includes error detection, correction and evolutionary development.

Not only does a structured description of a software system constitute a basis for architectural analysis, it can also improve the productivity of new members in a project. The architecture provides a simple and holistic view of the whole system. This is very important since complex system usually engage a lot of people, all with unique competencies, at different stages of the development process. Since designing real-time systems usually require multi-disciplinary knowledge, it is very important to have an architectural description that can be understood by software engineers as well as control and mechanical engineers. Furthermore, many software projects employ a lot of consultants. Consultants may have little knowledge of a company's product line and need a quick briefing in order to get productive and cost efficient.

The complexity of software systems also causes problems when maintaining and correcting errors in a software product. It is seldom possible to, in advance, be aware of all the side effects that a particular correction may give rise to. If an architectural description is at hand, it could give some guidance on what modules are most likely to be affected by the correction. This is highly related to evolutionary development. If the architecture of the software construction is violated, it ceases to exist in its former shape. The construction still has an architecture, but as long as the architecture is not explicitly, and correctly described, it is of little, or no use. Consequently, the architectural description may, and should,

evolve as the construction that it describes evolves.

2.2 Towards a definition

There are almost as many definitions of software architecture in the literature as there are software architects and designers. We mention a few examples:

The IEEE has the following definition of architecture [IEE00]:

> Architecture: the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

Bass et al. propose the following definition [BCK97]:

> The software architecture of a program or computing system is the structure or structures of the system, which compromise software components, the external visible properties of those components, and the relationships among them.

In [Pau94], the following definition is given:

> Software architecture not only reflects how the functional requirements are met, but addresses:
>
> - non-functional requirements
> - design rationale
> - architecture style

Yet another definition is provided in [Cle96]:

> A view of a system that includes the system's major components, the behavior of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the system's mission.

One property that seems to be common among almost every proposed definition is that the software architecture describes a system by a composition of its components and their interrelationships. However, by component in this context we do not mean a software component as in

COM or CORBA, but rather a software entity. This software entity may very well be a traditional software component, but is could also be a subsystem or a software module. In addition, software architectures should provide a high level description, i.e. a more abstract level than the level that algorithms and data structure provides. However, defining a software architecture only as a syntactical representations of software entities and their interconnections in the software systems is not sufficient. To be useful, additional information must be present in the description, in particular the semantics of software entities and connections. Different domains of software systems have different semantics of their software architectural description. A domain defines the class of applications0 to which a product belongs, e.g. desktop applications and industrial control applications. As a consequence, there will be variations in the definitions of software architectures depending on the domain. Furthermore, the definition also depends on the aim of the architectural description, e.g. support for architectural analysis, representation or description of the designed system. It is probably impossible to unify software designers in one single definition as it depends on the aim of the architecture and the domain in which it is used. What we can state is that software architecture is a description of the software structure and methods to evaluate and compare design solutions.

2.2.1 Software architectures, frameworks, styles and patterns

Architectural patterns, also referred to as *architectural styles*, are identified and named due to being successful architectural solutions to particular design problems [BMR+99][SG96]. Examples of architectural patterns are pipes-and-filters, client-server, model-view-controller, and layered architectures.

Design patterns on the other hand, are small collections of objects and classes which solve common problems in the design, applied on a slightly lower level than the architecture itself [GHJV94]. Moreover, a design pattern must specify the problem it solves, the way it is constructed and the consequences of using it. Usually, a software architecture is implemented with a lot of different design patterns. These patterns appear within the components in the overall architecture.

The advantage of both architectural patterns and design patterns is that they allow engineers to communicate design solutions in a common language as names and behavior of the patterns are agreed upon.

A framework can be seen as an "architectural mould" for a particular product domain, or a part of a complete architecture, e.g. the user interface for a software system. It is a library, for instance, a class library, which when properly reused results in a particular architectural pattern, or style. Frameworks are common in the human-machine interface community where the designer, for instance, inherits a dialog class and customize the concrete sub-class to get the desired behavior. An example of such a frameworks is Microsoft Foundation Classes (MFC) and Visual C++. Frameworks can also be designed for a variety of domains such as control systems, database applications, etc.

Apparently, there are commonalities between software architectures, frameworks and patterns. There are however distinct differences as well. Patterns are language independent descriptions, i.e. they can be implemented using any desired programming language. Frameworks, on the other hand, are partially abstract implementations in a particular programming language. To use a framework in a reasonable way, that particular language has to be used in the concrete implementation of the application.

2.3 Architecture description languages

Communication among software engineers is crucial. Without means for communication, important information into- and from the design phase might accidentally get lost, resulting in misinterpretations. Moreover, a system designer must be able to communicate with customers, other project members and management in an unambiguous way. An unambiguous architectural description is also a necessary condition for performing architectural analysis. A parable is the building trade, where building architects transform the customer requirements into a design. This design must be described in a way the building constructor understands in order to do mechanical strength calculus and for building workers to use as a blueprint. When developing software, a software engineer formalizes the customer requirements. Based on the requirements, a high-level design is described in a language that is commonly understood by customers and designers. The common language is a necessity in order to communicate and discuss design solutions. As output from the high-level design phase, one or several candidate architectural solutions are produced.

To verify that the non-functional requirements of the system are met

by the architectural solutions, the architecture has to be analyzed. Hence, the description language used in the high-level design must support the required analysis methods. Once a software architecture is constructed that fulfills the requirements, the architectural description is used as a blueprint when implementing the system. In addition, an architectural description makes maintenance easier since it facilitates the understanding how parts of software systems cooperate. Thus, the parts of a software system, i.e. components and sub-systems, affected by a correction are detected in advance.

2.3.1 Desired properties of an architecture description language

Languages for architectural description are called *Architecture Description Languages* (ADLs). There is an abundance of ADLs, each of them with its own specific syntax, semantics, expressiveness and purposes [EHL+94] [LKA+93] [Ves94] [BRJ98a]. An ideal ADL should however, provide six classes of properties: composition, abstraction, reusability, configuration, heterogeneity and analysis [SG96].

By *composition* is meant that a software system should be described as a composition of software entities and connections. Furthermore, software entities and connections must also be described in a way that clearly and explicitly describes the exact role of each element.

Any model is only justified if it provides some level of *abstraction*. If not, we would simply use the most exact model instead, i.e. the implementation. Nevertheless, even if the level of abstraction should be as high as possible, it must provide the details necessary for communicating and analyzing the architecture.

As software entities may be *reused* in different applications that are described using different description languages, the architectural description must be able to adopt to reuse. That is, it should be possible to reuse descriptions of entities, connectors and architectural patterns in different architectural descriptions.

Heterogeneity provide the possibility of combining different heterogeneous architectural patterns in a single system. For instance, it must be possible to have both a pipes-and-filter pattern where the components also can access a shared data base. Another example is that every layer in a layered architecture might be implemented by using any other architectural structure. Heterogeneity also means that software components that are implemented in different programming languages can be com-

bined in an architecture. This is a result of the abstraction requirement on an ADL described above.

Configuration means that the architectural structure among software entities in the system should be separated from the structure within the software entities. This enables us to understand, reason, and change the architecture without having to examine each individual component in detail. Moreover, the language should also support the specification of dynamic reconfiguration if such is possible during runtime.

Finally, as high-level *analysis* is one of the primer justifications for using software architectural description techniques, the architectural description must support different kinds of analyses. Thus, the architectural description must provide the level of details necessary for performing the required analyses.

Considering the desired properties of an architectural description above, how can a software architecture be described? One possibility is a plain textural description in a natural language. However, natural languages tend to be ambiguous, making them really hard to interpret in a consistent manner. By using a formal language an unambiguous description is obtained. With formal languages it is possible to use mathematics when modeling, analyzing, and verifying the architecture. The disadvantage of using formal languages as architectural descriptions are that most of them requires a lot of experience and mathematical skill. Consequently, such a description may be sufficient and useful at some stage in the design process but not for communication with partners in a project without a computer science background.

Graphical representations are usually intuitive and relatively easy to understand. Even very inexperienced engineers can get a feeling for how a system is constructed by interpreting a graphical representation of it. Such a description also permits analyses and quality predictions to be made as described later in this report. However, it is very important that the semantics of each graphical construction is clearly, unambiguously defined. This approach has been adopted by many of the available ADLs, where the software design is constructed using components and their interconnections in a 4th generation language manner as illustrated in Figure 2.1.

2.3.2 Semantics of an ADL

The architectural description in Figure 2.1 provides only the information that there are three software entities in the system, which are connected

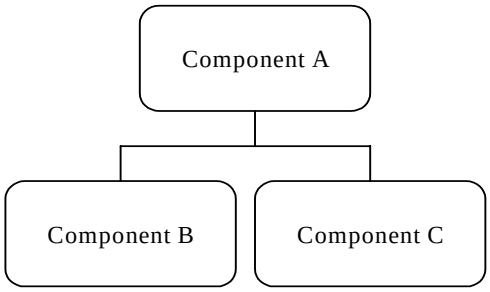

Figure 2.1: A graphical software architecture description.

to each other. The connections could indicate a class hierarchy or a network communication link over a distributed hardware architecture. As stressed by Clements and Northrop [CN96], it must be known exactly what the software entities are, what the connections mean and what the position of the components imply, i.e. a *well-defined semantics*. If the semantics is not clear the architectural description is quite useless.

One single architectural description language can not fit the desired level of abstraction for every different software domain and application. There is for example a big difference between designing a real-time system with hard- and soft temporal requirements compared to designing a desktop application such as a word processor. Consequently, we need a different description language for every application domain, all with their unique constructions.

Even though there must be differences in the architectural description depending on the application domain, there might exist a least common denominator. Such a least common denominator could, for instance, consist of software entities and connections. But the significance of a connection or a software entity could be domain specific.

If the ADL has an unambiguous semantics, design tools for architectural analyses can be developed [SEG97][LPY97a]. However, analysis of quality properties usually requires more information than just the architectural structure. This additional information is provided by the architectural views and is discussed in 2.4.

2.3.3 Examples of existing architectural description languages

There exist several architectural description languages for real-time systems. Typically they differ in their expressiveness and formality.

One example of an ADL for real-time systems is MetaH [Ves94]. MetaH provides means for specifying real-time processes, referred to as tasks, that can be either periodic or aperiodic, communication among tasks, modes and composites of processes and modes that are called macros. Furthermore, the hardware allocation of processes and characteristics of the hardware such as channels that are used for communication among processors can be specified. As the temporal properties of tasks and modes are provided in the models, MetaH support different kinds of real-time analyses such as schedulability analysis. There exist a graphical tool that supports the modeling in MetaH and analysis of real-time software architectures described in MetaH. The schedulability analysis in this tool is based on rate-monotonic [LL73].

Even though UML was not initially designed to be an ADL it is used as such in industry today. UML is an acronym for *Unified Modeling Language*, and is a language primary intended for the description of object-oriented design, e.g. classes, objects, use-cases, sequence diagrams [BRJ98a]. However, now there are constructions in UML for subsystems and components as well. The language is constantly under development and among the most recent arrivals is the scheduling and performance profile that is intended for the design of real-time systems. The language is quite flexible as it allow an user to define the semantics by using *stereotypes*. A stereotype defines the exact meaning of a particular construction.

In Section 5.4 we propose an ADL called ART-ML that is suitable for large and complex real-time Systems [WAN+03]. ART-ML has constructions for describing tasks as well as communication and synchronization among them. Tasks can be triggered by events in the system or be periodic. Furthermore, ART-ML allows execution times of tasks to be specified as distributions, i.e. a set of different execution times and their probability of occurrence. Simulation is used as a method for analyzing different properties of an architecture modeled in ART-ML. Typically, temporal correctness and performance is analyzed.

What ADL to use is determined by the type of system that is developed and the type of analyses that are of interest. For instance, if the temporal behavior is of vital importance, the ADL must provide mecha-

nisms for modeling the temporal aspects of an architecture, and in way that makes analysis of the temporal behavior possible. In Section 2.4 the views and aspects of an architectural description is discussed. Quality properties, and analysis of these are further discussed in Section 2.5.

2.4 Architectural views

An architectural view is a description that is seen from a given perspective and omits entities that are not relevant to this perspective. It is like a slice cut through the architectural description that brings out particular information. The different perspectives are important for different stakeholders in a software development project. For instance, a project manager requires different information about the software architecture than a software engineer.

There exist no agreed upon standard set of views, nor a standard that define their names. Moreover, a view is not justified if it is not useful in the development process. Hence, we can define and name any view that make sense in our own software development project. Examples of possible and relevant views are:

- the structure of the main software entities, e.g. sub systems, components
- the allocation of software to threads of execution
- communication among software entities
- temporal related information, i.e. period times, release times
- allocation of software to hardware
- data flow among software entities
- description of the software entities' functional behavior

The list is not by any means complete, it only gives examples of potential views among which some are of importance for real-time systems, e.g. information about the temporal behavior.

One of the most widely accepted model for architectural views are variants of the *4+1 view model* [Kru95] [HNS99] [Kru99]. The variants differ slightly in terms the views' names and contents. The success of the 4+1 model is mainly due to its close relation to the *Unified Modeling*

Language (UML), which has become a de facto standard in the industry [BRJ98a]. The variant of 4+1 view model in the Rational Unified Process (RUP), defines 5 views: *Logical view*, *Implementation view*, *Process view*, *Deployment view*, and finally the "+1" view which is called *Use-case view*.

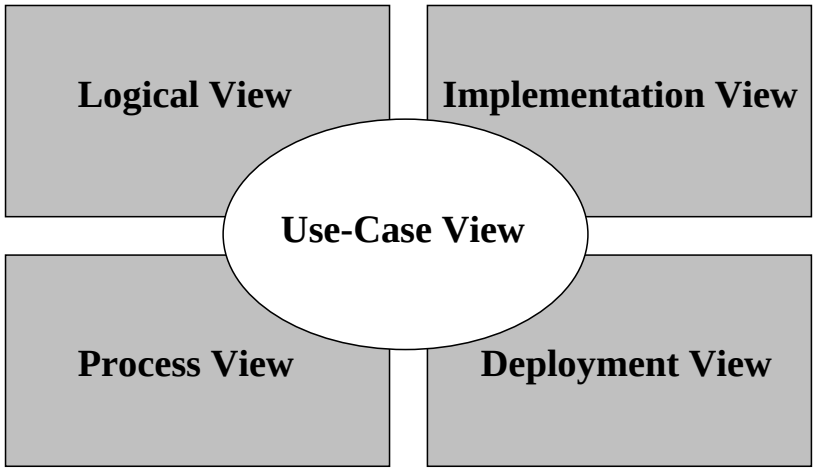

Figure 2.2: The 4+1 view model

It is not easily, or maybe even not possible, to define and decompose all important issues of a software system into distinct views. It seems that there are always issues that cut across all, or several, views. This is also true for the 4+1 model. For this reason we introduce the notion of *view aspects*. A view aspect emphasizes a particular important issue of a view, and may very well exist in different shapes in several views. For instance, communication among software entities in the logical view of the architecture has a different meaning than communication between the same software entities in the process view or the deployment view.

This thesis deals with modeling and analysis of real-time systems. Hence emphasis is on the logical view and the process view. However, we will also touch on issues related to the deployment view. In relation to this we will present aspects that are important in the real-time systems domain. More precise, we define three different aspects:

- temporal aspect
- communication aspect
- synchronization aspect

2.4.1 Logical view

The logical view addresses the functional requirements of a system. It identifies subsystems, components, and classes. The logical view can be hierarchically specified, i.e. what components are part of which subsystem. Moreover, dependencies among the entities in this view is also visible here.

The overall architectural structure and style is provided, i.e. the highest level of abstraction in an architectural description. This is the natural starting point for designing a software system.

As design on the highest level of abstraction is rather rapid, it is possible to design several competing architectures for evaluation and comparison. Once a software architecture satisfying the quality requirements is selected, it is settled. Depending on the required analyses, more views might have to be modeled in order to make a correct design decision.

The hierarchical decomposition in the logical view may, eventually, reach its bottom in some kind of state diagrams that provide an abstraction of the implementation, i.e. a model of the behavior. Such a description may constitute the basis for formal verification. Some possible descriptions on this level are state machines, e.g. Statechart [HN96]. There exist tools that provide automatic verification as well as code generation facilities based on these types of diagrams.

In the design methodology called Module Approach to Software Construction, Operation and Test (MASCOT), the logical view is modeled with a diagram called the decomposed component level view [Mas]. This view provides a decomposition of a sub-system into its main constituents, i.e. its tasks.

The object-oriented methodology for real-time systems called Hard Real-Time Hierarchical Object-Oriented Design (HRT-HOOD), also has a logical view that is provided by the so-called parent-objects [BW94]. A parent-object is a component on its highest-level that may be further decomposed.

UML have a notation for describing *packages* which basically is a collection of model elements that constitute a sub-system. Moreover, UML provides a component diagram in which components and dependencies among them can be visualized. The contents of components and subsystems are modeled with class diagrams and object diagrams. For the lowest level of abstraction in the logical view we can use statechart diagrams in UML.

Temporal aspect of the logical view

In real-time systems, the temporal behavior of the software entities are of vital importance. The temporal aspect of the logical view includes execution times of the components. The execution times are required to being able to guarantee the temporal correctness. The execution time of a software entity my be given as a requirement, i.e. a time budget that must be kept during implementation, or it can simply be the product of a particular implementation [NSG+00].

Communication- and Synchronization aspect of the logical view

Dependencies among entities modeled in the logical view indicate that there exist a relation among them. The semantics of the dependencies decides the exact meaning of the dependency. The data flow constitute the communication aspect in the logical view.

2.4.2 Implementation view

This view describes how to organize source code, data files, executables and other accompanying artifacts in the development environment in terms of packaging and configuration management. Typically, ownership and release strategies are addressed here.

Moreover, requirements on e.g. distributed development, may require a particular mapping of software entities onto executables and and source code files in order to minimize the conflicts that arise when a file is updated simultaneously by different engineers. This can be compared with a requirement on the software that result in a particular functional decomposition.

The aspects we are interested in for real-time systems provides no information in the implementation view. Hence, we describe it briefly.

2.4.3 Process view

The process view addresses the concurrent behavior of a system at run-time. In the real-time community we like to think of this as assigning tasks to software components. The tasks determine the execution of the software. Moreover, interactions between processes and tasks are visible in this view. Interaction in this view may be in the form of synchronization and/or interprocess communication.

Temporal aspect of the process view

The *temporal aspect* of the process view is concerned with modeling of the temporal behavior of tasks. As the correctness of a real-time system not only depends on correct function, but also *temporal correctness*, the temporal constraints must be present in the architecture. By temporal correct we mean not too early and not too late. In order to verify whether or not tasks in a real-time application will be schedulable, i.e. all temporal requirements are fulfilled such as all deadlines are met, we need an aspect that defines the temporal attributes, *the temporal aspect*.

The temporal aspect of the process view contains data such as release time i.e. the earliest start time of a task, the deadline i.e. the latest completion time of a task, the period time (the frequency) of a task, etc. We say that a *task model* determines the exact content of the temporal view. The exact appearance of a task model varies depending on the execution strategy. The execution strategy defines the rules that determine what task to execute.

As an example of different task models consider a periodic task that samples a sensor in a process. As the sampling should be performed with some specific frequency in order to obtain a correct view of the process, a period specifying the interval between two consecutive executions of the sampling must be specified. In contrast, if the application is purely event triggered, i.e. tasks have arbitrary release times, there is no need for specifying period times. Instead, the minimum inter-arrival times must be specified for the tasks.

HRT-HOOD has a temporal aspect that is divided into two parts, one that describes the execution strategies for a class and one that provides the temporal attributes. The execution strategy can be either cyclic or sporadic. Depending on the execution strategy, classes can be assigned, e.g. period times, minimal inter-arrival times, and deadlines.

It is possible to annotate sequence diagrams with timing information in UML. Components or objects can also be stereotyped in order to model temporal properties of software. For instance, the stereotype *active*, may indicate that component or object has its own thread of execution and the naming convention of an instance defines the temporal attributes. For example, an object with the name Task-10-1 means that task runs with the frequency 10 ms and has priority 1. Note that this is not part of the standard, but rather defined by individual development projects. This is only one example out of many possible way of how to specify temporal attributes in UML.

Synchronization aspect of the process view

As real-time systems are multi-tasking systems having several tasks running concurrently, it is necessary to synchronize access to shared resources in order to avoid inconstancy. This is an aspect of the process view that we refer to as the *synchronization aspect*. Tasks that uses a shared resource must mutually exclude each other, i.e. only one task can use the resource at the time. Synchronization is also required in order to fulfill some of the temporal requirements, e.g. precedence requirements. Precedence requirements are concerned with the order in which tasks executes in a system.

The solution domain provides several techniques for handling mutual exclusion in real-time systems, e.g. semaphores, signals, or separation of task in time.

Communication aspect of the process view

Allocating software entities that exchange data between each other to different tasks introduce the need of *interprocess communication* (IPC). This is modeled in the process view as a *communication aspect*.

In MASCOT, communication aspect and synchronization aspect is modeled using paths along which entities communicate. A path can indicate a dependency to commonly used data, or a dependency to another entity that results in a sending/receiving of messages. UML uses sequence diagrams for describing communication and synchronization among objects. Relations and dependencies among components and objects can also be stereotyped such that they indicate a communication, or synchronization relation.

Examples of formal languages that can be used for modeling communication and synchronization among processes are CCS or Timed automata [Mil89] [AD94]. Timed automata can be used for real-time systems as it provides a notion of time as well as concurrency. We have also developed an extension of timed automata that defines the concept of a task which we call timed automata with tasks (TAT) [NWY99]. In this work we can reason about schedulability problems of systems modeled as timed automata which now has resulted in a new tool called TIMES [AFM+02]. CCS is a process algebra with which it is possible to model concurrent systems. Such algebra is useful when modeling communication and synchronization, which is essential when designing real-time systems.

2.4.4 Deployment view

The deployment view shows how software is allocated to hardware resources in a system. If the system is distributed, i.e. a set of interconnected and geographically separated CPUs, or a multi-processor system, i.e. a set of interconnected and geographically collected CPUs, there might be requirements of pre-allocated functionality among the nodes in the system. Moreover, performance requirements may result in allocation of software close to the physical process it controls. Hence, the deployment view may also contain I/O units. The allocation of software in a system will affect the final architecture and the performance of the application.

Yet another reason for describing the hardware in the software architecture is the issue of portability. If software should be easy to move between different types of platforms, the dependencies to the hardware and the operating systems must be encapsulated from the rest of the software system. One can discuss whether this is a software architectural view or not, it is more associated with a system architecture. However, as long as hardware has an impact on the software architecture, we consider it important in the system description.

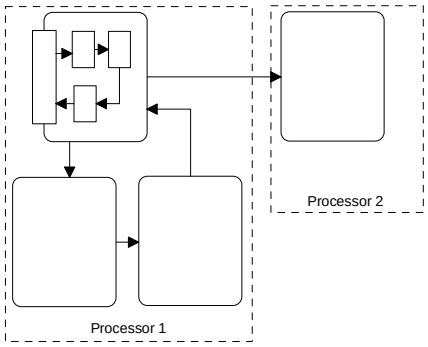

Figure 2.3: The processor allocation in the deployment view

Communication aspect of the deployment view

Communication buses are also important resources in the deployment view. The presence of a communication bus indicates the need for interprocessor communication, i.e., communication and synchronization

among software entities becomes evident. Hence, this is the *communication aspect* of the deployment view.

In the Yourdon Structured Method (YSM), the allocation of functions to hardware processors is called the processor environment model [Coo91]. Besides the function allocation, this view reveals the data that will be communicated among the processors. The *deployment diagram* in UML describes the allocation of software onto hardware.

2.4.5 Use-case view

This view contains the key requirements on the architecture. The key requirements are used to drive the design of the architecture. A particular structure in the logical view of the architecture is usually the result from fulfilling a key requirement, e.g. reusability, maintainability. It is desirable that the intention of an architectural solution can be visible in the description, i.e. tracability between the solution domain and requirements [N.G00].

In UML there exists use-case diagrams which is used for this view.

Temporal aspect of the use-case view

Temporal requirements can origin from many sources, and they may take many forms. The most intuitive ones are explicitly stated in a requirements specification and may very well have their origin in user issues such as responsiveness of a feature.

When it comes to control systems, the temporal requirements are often derived from the processes they control. A typical example of such a requirement is sampling frequencies. Moreover, temporal requirements may be due to control performance. For instance, in order to achieve good control performance it is desirable that the *jitter* is minimized, i.e. minimizing the variations in the periodicity of task [T98].

2.4.6 Architectural views: An example

As an example of how an model architecture and by utilize the architectural view and the view aspects, consider a real-time system that controls the water level in a tank. The system samples a water level sensor, takes a decision whether to let water out, or pour water into the tank. The system actuates a pump or a valve if the level has to be adjusted. As it is a real-time system, the temporal constraints on the system must be modeled, i.e. the temporal aspect of the views. The ADL used in this

example is invented for this example and has no claims, what so ever, to fulfill the required properties of a complete ADL.

First, the highest level of abstraction in the logical view of the architecture is modeled by identifying the components in the system and how they relate to each other interconnections. In this case, the arrows between the components represents a data flow. Hence, that data flow constitute the communication aspect of the logical view. The logical view is shown in Figure 2.4. While portability is crucial, the operating system is modeled as a component as well in the diagram. The dotted arrows indicates a dependency.

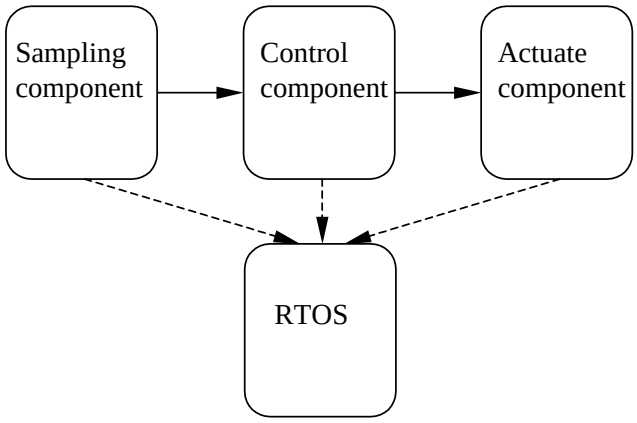

Figure 2.4: The first candidate architecture for a water tank controller

Moving on to the process view of our small control system architecture, we decide that every component is assigned its own thread of execution, i.e. task. Moreover, in order to verify the temporal correctness of the system we must populate the temporal aspect of the process view. In this case we assume a periodic task model, i.e. we assign period times and deadlines. The execution time of a software entity is determined by the source code. Hence, execution times are part of the temporal aspect of the logical view. In this example we use a table for modeling the tasks and the task allocation, (Table 2.1).

The components communicates their data between each other using the communication mechanisms provided by the tasks. This is the temporal aspect of the process view. The diagram for showing this aspect of the process view has been omitted in this example.

The implementation view is of less interest in this small example.

Task	Period	wcet	Deadline	allocation
Sampling task	1 ms	50 μs	60 μs	sampling component
Control task	2 ms	200 μs	1 ms	control component
Actuate task	2 ms	50 μs	1 ms	actuate component

Table 2.1: The temporal aspect of the process view

However, we have to allocate the software onto a hardware architecture. The is rather simple in our system since it is a single processor system. Consequently, the communication aspect of the deployment view is superfluous. The diagram is depicted in Figure 2.5

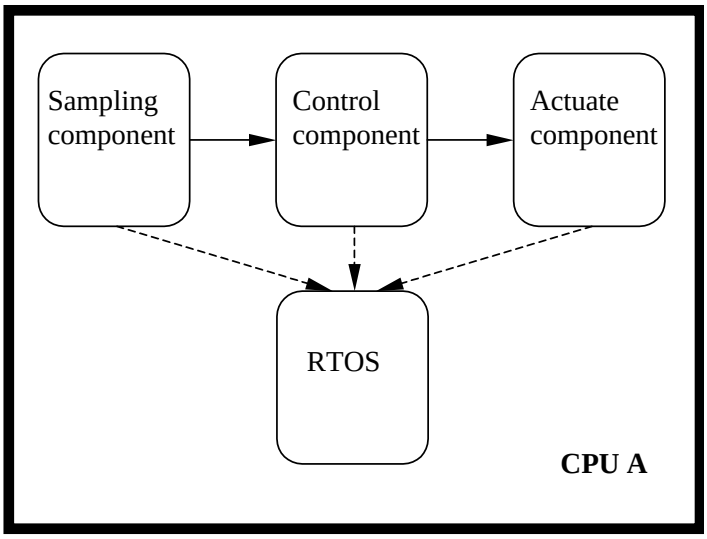

Figure 2.5: The deployment view of our architecture

In Section 2.7.1, we will revisit this example in order to analyze whether or not the proposed architecture complies with the requirements, e.g. temporal correctness.

2.5 Architectural analysis

One of the main incitement for using software architecture notation when designing a software system is the ability to analyze and verify the design in an early stage of the development process. By comparing dif-

ferent candidate architectures, confidence in early design decisions can be achieved. Such a comparison is done by comparing the results from analyzing each architectural solution with respect to the architectural requirements. Furthermore, architectural analysis enables the possibility to get software metrics based on the high-level design, e.g. the level of coupling and cohesion within and between the different modules that constitute the software system [FP97].

We have divided the quality properties of a software system into two different classes: *operational* and *non-operational*. Operational quality properties are those concerned with the runtime behavior of the software, e.g. performance or reliability, whereas non-operational quality properties are concerned with the quality of the software itself, e.g. maintainability or reusability.

2.5.1 Methods for architectural analysis

An architectural analysis process is divided into two stages, questioning and measuring. The questioning phase generates questions that are answered by the measuring phase. Len Bass et. al. [BCK97], have categorized the questioning stage in architectural review and evaluation into three different classes, namely *Scenario-based*, *Checklist-based* and *Questionnaire-based*.

Scenarios are a set of cases where the software architect asks a lot of "what if" questions that reflect the requirements. Scenarios make quality properties concrete in the view-point of the system. For instance, the requirement that a system should have high performance is very general; too general in order to make sense as a requirement. However, specifying that it is high performance in terms of data latency, and even further, minimizing storage latency in a certain data base to 500 ms, is a scenario that makes the requirement analyzable.

It is, however, not a trivial task to construct the right questions and to know when to stop generating scenarios. This requires a lot of experience and knowledge, which can be achieved by being involved in many design projects. A scenario is always system specific, i.e. tailor-made for a particular application in a domain, whereas questions that are valid for all architectures in a particular domain resides in a checklist. The items in the checklist can either generate scenarios or be verified in the measuring stage directly. As an example, consider the domain of safety-critical real-time systems. The checklist probably contains the following items among several other:

- Is the system schedulable?
- Is there error recovery code in the system to clean up after error detection?

The first item is verified directly by performing a mathematical schedulability analysis. The second item is too general and therefore it must be further decomposed into a set of specific scenarios before it can be answered. As scenarios are system specific, they can stress different types of errors in specific modules residing in the system. One possible scenario that origin from the second bullet above is:

> What happen when division by zero occurs in the control task?

The scenarios can than be verified by, for instance, simulation or scenario execution, both described later in this chapter.

The questionnaire-based questioning typically stresses general logistical software architecture issues. These questions have usually very little to do with the quality of the software itself, but are rather focused on issues such as documentation and project organization. Although the logistical questions do not examine the quality of the software product itself, it has impact on the quality since good quality requires a mature development process. Examples of such questions are:

> Is a standard architectural description language used?

or

> Is the intended work distribution supported by the architecture?.

The measuring techniques available for architectural analysis are: *scenario execution*, *simulation and prototyping*, *mathematical methods*, and *experience based knowledge reasoning*.

The idea with scenario execution is to "execute" a question stated by a scenario on the architecture. By executing a scenario is meant that the effects on the architecture imposed by a scenario is investigated. This method is particular well suited for analysis of non-operational quality properties.

Simulation techniques often require a prototype implementation of the architecture. Such a prototype should be as "thin" as possible, containing only the information needed for the analysis to be performed. The thinner the prototype, the earlier can the analysis be performed. Instead of using a prototype as the base for simulations we can use a model, i.e. a description of the software system in a modeling language with a well defined semantics. In Chapter 5, we describe our language for modeling of real-time systems and how we can use simulation for analyzing the correctness. Simulation is a method targeting analysis of operational quality properties.

Experienced-based reasoning can be used for any of the two classes of quality properties. Actually, experienced-based reasoning is usually how the software architecture evaluation is done in industry today, although in a relatively unorganized manner. Senior programmers and experts have the knowledge and know what a good design is. As an organization and its development process mature, more of the formal evaluation techniques will be adopted.

Mathematical methods can be used provided that a mathematical model of the architecture can be delivered. Such a model could be in the form of e.g. timed automata. More examples of mathematical measuring techniques are the schedulability test for real-time systems and statistical reliability modeling. These methods give a clear yes- or no answer, or a quantitative value that is comparable among all different types of software applications.

Figure 2.6 provides a schematic picture of how the different evaluation techniques relate.

Although measuring techniques might provide quantitative values, these values must be treated carefully. The quantitative values should be used as relative values when comparing competing software architectures. Moreover, if scenarios or experienced reasoning was used to obtain the measurements, the exact same set of scenarios and reasoning must be used when evaluating the competing, or refined architectures. Otherwise, the measures are not comparable. For instance, the result from measuring, e.g. the reusability of a system, says nothing in isolation. It only makes sense when comparing it with the results from measuring competing architectures.

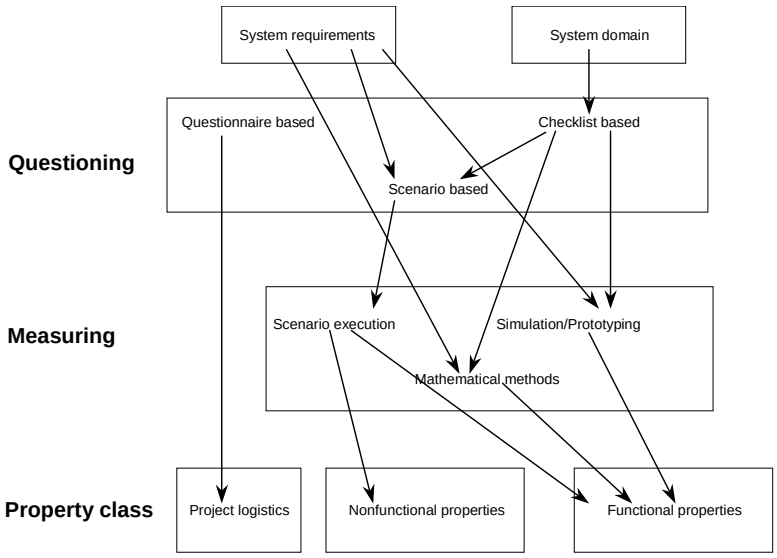

Figure 2.6: Evaluation techniques and property characteristics

2.5.2 Analysis of operational quality properties

There exist operational quality properties in abundance, among which properties of particular interest when designing safety-critical real-time system is listed in Table 2.2.

Besides describing the operational quality properties we also give recommendations of how to design a system with emphasis on a particular ability. However, the design recommendations are not exhaustive, they only suggest one or a few out of many possible design solutions or design recommendations.

Performance

Certain operational properties of a software system, including performance, are tricky or even impossible to predict using the architectural description level only. Performance estimations must have the algorithmic solutions as input. As discussed in the introduction, software architecture is a description of the system on a higher level of abstraction than algorithmic solutions and data structures. However, by using prototyping and simulation techniques, performance in terms of, for instance, event throughput or queuing length for events in a system, can be estimated

Performance	The systems capacity of handling data or events.
Reliability	The probability of a system functioning correctly over a given period of time.
Availability	The probability of a system functioning correctly at any given time
Safety	The property of the system that it will not endanger human life or the environment
Security	The ability of a software system to resist malicious intended actions
Timing	The temporal behavior of a system.

Table 2.2: Operational quality properties

[GB98]. Since such a performance measure is not absolute, it can only be used when comparing different architectural solutions.

Reliability

There are mathematical methods based on probability theories, such as Markov models, for assessing reliability [Tra95]. However, these theories are developed for hardware where failures often are caused by physical wear such as corrosion, overheating, etc. Such failures are probabilistic in nature whereas software failures are mistakes (errors), made in the specification, the design or in the implementation. These types of failures are certainly not probabilistic according to some distribution over time. Furthermore, software can never be worn out. Attempts have been made to apply the methods from the hardware community to software. In software, the statistics are the numbers of errors in the program or the likelihood of a failure in a point of time based upon the error distribution in the past [FP97]. To get such failure estimations, there must be an implementation of the application or at least a prototype. Anyhow, a description of the application on a lower level than the architecture is needed. With heuristics from similar applications developed earlier, experienced engineers can estimate the expected number of errors in the components. Such estimations are very complex, giving rough metrics. An alternative to directly measure the reliability of the architecture is to measure the testability. The testability is a function of the effort required in order to assure the required level of reliability or availability.

Design recommendations: Reliability

There are three different approaches to handle faults in order to achieve a reliable system [Lap92]:

- Fault avoidance
- Fault removal
- Fault tolerance

Fault avoidance is about designing error free systems. This implies the use of structured design methodologies such as formal methods or semi-formal methods. Formal methods are based on mathematical models of the software system and the requirement specification. These models form the basis when proving correctness of the model with respect to the system specification. There exists a wide range of formal methods and formal modeling languages, each supporting different system domains. Semi-formal methods are, as the name suggests, less formal, i.e. they do not support techniques to exhaustively prove correctness of the models. Instead, they offer a structured way of reasoning, both when designing models of the system and when analyzing the models. The methods are usually based on some "formal" notation, e.g. Unified Modeling Language (UML) [BRJ98b], ADLs, etc., representing the system model. Examples of such methods are object-oriented analysis and design (OOA/OOD), and software architecture techniques in general.

No matter how accurate the models are analyzed, there may still be errors in the implementation. These errors usually originate from the specification and from the mismatch when mapping the models to the source code. In order to improve reliability in the program, fault removal techniques can be applied. Fault removal is basically the task of finding the errors by testing and removal of them by error correction. Under the assumption that no new errors are introduced, the reliability will grow as errors are corrected. This assumption is, unfortunately, seldom true, implying that the whole system has to be re-tested after each increment. The results from testing and re-testing can be used for statistically forecasting of the failure rate (and consequently the reliability), of a software system. Such a method is the reliability growth model, first proposed for software by Jelinsky et. al. [JM72]. There exist an abundance of different approaches to model reliability growth; they are all based on data collected during testing, but differ in the way the statistical model is made.

Some faults are impossible to avoid regardless of how accurate the design and the tests are performed. If it is particularly important that a certain module in the system does not fail, fault-tolerance can be introduced. Fault-tolerance is a technique which can be interpreted in two different ways: it could be the ability of a software system to tolerate faults from its environment, e.g. the operator, hardware errors, etc., or it could mean that the system should be tolerant against design faults in the software itself. The two different fault-tolerance approaches are, naturally, solved using different techniques. For instance, to be fault-tolerant against hardware errors such as electro magnetic distortion, redundant hardware can be used, each with equivalent software running on them. This solution will however not tolerate software faults. Different approaches to be tolerant against software faults are recovery blocks and N-version programming [Sto96] [CA78].

Recovery blocks are based on acceptant tests of the calculated values. If the processed value is not accepted the program tracks back to a recovery point where it is safe to continue the execution after having restored the system's state.

N-version programming is achieved by developing N different versions of the software; each developed by different and isolated design teams. All N different versions run in parallel at runtime and their respective results are voted upon. This technique has, however, been proven not so successful since all different versions of the software start out from the same specification, and since most design errors originate from the specification, they will contain common errors.

Even if the source code is absolutely correct, the compiler may still produce erroneous binaries. Faults introduced by the compiler can be tolerated by using the N-version approach. Each version has exactly the same code, but they are all compiled using different compilers. It is important to note that the different techniques discussed above can be applied at any stage in the development process. For instance fault removal can be used when verifying the designed architecture against the system specification. Fault-tolerance is also a matter of architectural design. The techniques for fault-tolerance discussed above are all achieved using different architectural solutions.

Safety

Safety seems, at a first glance, very similar to reliability. There is however a clear distinction as safety is only concerned with failures that endangers

human life and the environment, i.e. hazards, whereas reliability deals with all failures regardless of their consequences. Moreover, safety is a property of a system, not only the software. Hence, a reliable software is not a guarantee for a safe systems. For instance, consider an industrial robot. No matter how correct and reliable the software is, humans can still get injured if they get close enough to the robot.

Before any safety analysis of the architecture can be performed, the hazards must be identified. This is done in a hazard analysis that is a reasoning based method for finding all hazards in the system that is going to be designed [Lev95].

There exist several techniques for assessing safety properties in software designs. Most of them are scenario based and work either backward or forward. If the method works backwards, the analysis starts with the hazard as a scenario, trying to trace down the responsible component. On the contrary, if the method works forward, the effects of an error in a component is investigated.

Some of the most well known forward methods are Failure Mode and Effects Analysis (FMEA) [IEC], and Hazard and Operability studies (HAZOP) [Kle95]. Both methods analyze the consequences of failures in the components. One commonly used backward technique is called Fault Tree Analysis (FTA) [Sto96]. FTA starts with a hazard, trying to determine its origin among the components. This kind of analyses give an understanding of where in the architecture fault-tolerance techniques should be introduced, or if already introduced, verifying whether the intended fault-tolerance is achieved or not.

Design recommendations: Safety

Depending on the results from the safety analysis, changes in the design may have to be performed. Different design approaches to avoid catastrophic failures can be applied based on the severity of an accident caused by the hazard. The different approaches are [Lev95]:

- Hazard elimination
- Hazard reduction
- Hazard control
- Damage minimization

The severity is a quantified value that makes it possible to compare and rank hazards. Typically, the severity is given in terms of the cost or, lost lives, for the stakeholder if the accident occurs.

Substitution, decoupling, and simplifications achieve hazard elimination. By substituting a dangerous design possibility by a functionally equivalent, but not dangerous solution, the hazard itself is eliminated. For instance, if the system involves a very toxic chemical liquid, substituting the liquid with a non-toxic one eliminates the hazard. Moreover, by decoupling safety-critical parts of the software from non-critical software, the risk for an error in the non-critical part to propagate into the safety-critical parts is eliminated. There exist some known architectural solutions based on decoupling, e.g. safety kernels, firewalls, hierarchical architectures [Sto96].

Hazard reduction reduces the likelihood of the occurrence of a hazard. It might not be feasible or even possible, to eliminate the hazards. Then the designer has to design the system in such a way that the hazard is not very likely to occur. An example of hazard reduction is to erect a fence around an industrial robot, preventing humans to come close enough to get hurt.

Hazard control is applied in order to reduce the likelihood of an accident if a hazard arises. This can be achieved using fail-safe design, i.e. the system should be designed to detect the hazard and then transfer it into a safe state if it exists. There are, however systems where no safe state exists. A typically example of such a system is airplanes. These systems must keep operating even if something goes wrong. This is achieved using fault-tolerance such as redundancy. It is essential that an airplane keeps flying even if one engine breaks down, i.e. the second engine should be switched to operate the airplane. The performance will of course be reduced, but the airplane can still be maneuvered to its safe state on the ground.

Yet, if an accident still occurs, the consequences and losses must be reduced. This is achieved with damage minimization that strives to minimize the exposure of the accident to the environment or human beings.

Availability

Reliability and availability are strongly correlated. According to the definitions given in Table 2.2, reliability is the probability of a software system functioning correctly over a given period of time and availability

is the probability of a software system functioning correctly at any given time. The availability is given by the displayed equation below:

$$Availability = 1 - \frac{MTTR}{MTBF} \tag{2.1}$$

where MTBF is the Mean-Time-Between-Failure and MTTR is the Mean-Time-To-Repair, i.e. time spent on service. The relation between MTBF and MTTR is shown graphically in Figure 2.7. If any point of time is picked randomly along the y-axis, there is a probability of having correct functionality, i.e. the availability of the software system.

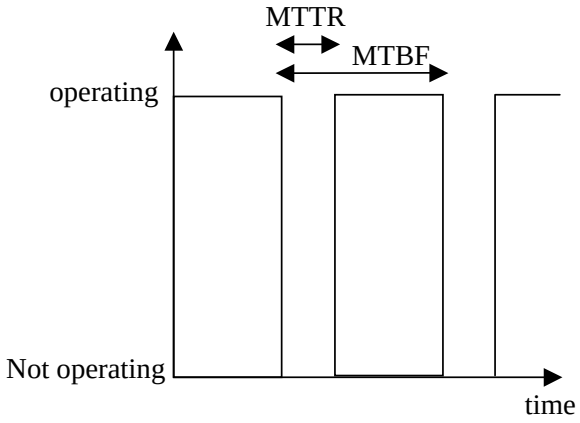

Figure 2.7: Availability and reliability

Design recommendations: Availability

The recommendation given for reliability also apply to availability. Moreover, process related issues are important. Proper strategies for validation and verification are crucial in order to maximize the MTBF, but also designing the system for high testability. In order to minimize the MTTR we must design an maintainable architecture, i.e. adding new functionality as well as changing existing functionality should be as effortless as possible. Testability and maintainability is further discussed in Section 2.5.3.

Security

Security is concerned with protecting a software system from malicious intended actions, e.g. intrusion by unauthorized users or locking out unintended accesses to safety-critical parts of the system. The importance of security is constantly increasing as more and more systems are becoming accessible via mobile terminals or have web-based interfaces.

A scenario-based method can be used for analyzing the security property. Typically, such a scenario could reason about what happens if an operator or a sub-module tries to access a protected region of the system. Another possible way of analyzing software architectures from the security point of view, is simulation, provided that the logical view of the software architecture contains sufficient information regarding rules for authorization and identification.

Design recommendations: Security

A secure software system can be achieved by different architectural solutions: safety/security kernels, firewalls, etc., which all are different ways of restricting the access to the system or sub-systems [Sto96] [Lev95]. As security can be achieved using different architectural solutions, it can be assumed that security is assessable by architectural analysis.

Timing

Real-time systems are characterized by their correctness criteria. Not only should a real-time system deliver correct functionality, but correct functionality at correct time. We refer to this criteria, or property, of a software system as *temporal correctness*. A system is considered temporally correct if it complies with all its temporal requirements, (discussed in Section 2.4.5). Typical examples of temporal requirements are deadlines and jitter.

We classify real-time systems according to the importance of complying with the temporal correctness. We refer to the classes as *hard real-time systems* and *soft real-time systems*. A real-time system is considered hard if it is of vital importance that no temporal requirement is ever violated. The consequence of violating a temporal requirement in a hard real-time system may be catastrophic. Applications that belong to this class includes aircrafts, airbags, and control systems in nuclear power plants.

In soft real-time systems it is allowed to sometimes miss some of the temporal requirements. For instance, we can allow the system to violate a particular deadline every once in a while without catastrophic consequences. Examples of application in this class are video streaming, cellular telephones, and toys.

The information necessary for the verification of temporal correctness is provided by the temporal aspect of the software architecture. The attributes in the temporal aspect reflect the system's characteristics which often is referred to as *execution strategies* or *scheduling strategies*. Example of such characteristics are event-driven systems, periodic system, and fixed priority systems. We refer to the exact appearance of the temporal aspect of the process view as a *task model*.

Several mathematical methods for analyzing the temporal correctness exist. Each method differs in the assumptions they make about execution strategies, and the task model. [LL73][ABD+95][Bur93][XP00].

One of the most commonly used analysis methods are *Fixed Priority Analysis* (FPA). FPA in its simplest form assumes a task model which consists of a period time and a priority that is assigned according to some algorithm, e.g. rate-monotonic. Rate-monotonic assigns the task with the shortest period time the highest priority [ABD+95]. Moreover, FPA assumes that the worst-case execution time (WCET), of the software is known. By calculating the response time for each task, we can verify that no deadline requirement is violated.

Simulation can also be used for analyzing the temporal correctness of a real-time system. In Chapter 5 we will present a simulation based analysis framework that targets temporal correctness. This framework also aims at analyzing other correctness criterion such as the non-emptiness of message queues in a system. This is one of the biggest advantages with the simulation approach that we can monitor any resource in a system, not only the CPU.

Design recommendations: Timing

There are no architectural design issues that make a system more temporal correct. Either the system is temporal correct or is it not. The choice of a particular architectural solution in the temporal domain, i.e. the scheduling strategy, is dependent on the temporal requirements. For instance, if the system is purely event-drive it makes no sense to specify period times for the task set. Instead we can chose an earliest deadline first strategy (EDF), which executes the task that has least time left until

deadline. EDF assumes that deadlines are specified as well as WCET.

In Figure 2.8, a classification of different scheduling strategies is illustrated.

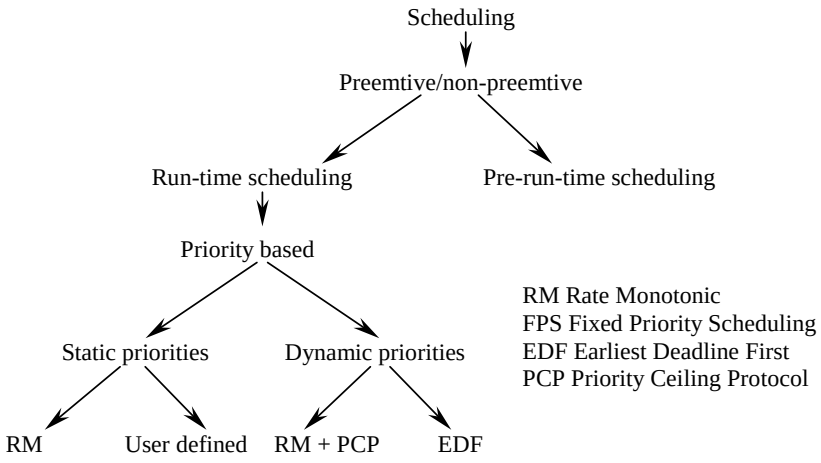

Figure 2.8: Classification of scheduling strategies

The choice of scheduling strategy can also affect other properties of a software architecture. For instance, as the execution order among tasks are fixed when using time-triggered solutions, i.e. tasks are executed according to a schedule that has been defined pre-runtime, it is possible to reproduce an execution scenario which is essential when designing a system with high testability [TH01].

2.5.3 Analysis of non-operational quality properties

The number of non-operational quality properties is, as the operational quality properties discussed in the previous section, very large. In Table 2.3, a subset of all such quality properties is listed, all being important in a mature and modern design process for real-time systems.

A very simple but yet powerful method for analysis of non-operational quality properties is the execution of scenarios. Many of the quality properties listed in Table 2.3 can be examined and analyzed by using scenarios. The design recommendations given in this section is, as the case with the design recommendations given in Section 2.5.2, is not exhaustive. They are only examples and other possible solutions exist.

Cost	The cost for performing any action such as development, evolution and verification
Testability	How easy it is to prove correctness of the system by testing
Reusability	The extent to which the architecture can be reused
Portability	How easy it is to move the software system to a different hardware- and/or software platform
Maintainability	The aptitude of a system to undergo repair and evolution
Modifiability	How sensible the architecture is to changes in one or several components

Table 2.3: Non-operational quality properties

Cost

Cost estimations are probably one of the hardest tasks for every large software development project. Accurate cost estimation for the design of a completely new system is extremely hard to make. Usually such estimations are based on historical experiences with similar systems. If no such experience is available, the estimation gets even more imprecise.

The software architecture description could help illuminate the cost of developing a system or adding new functionality to an existing system. Partly by being a structured description of the application, helping the designer to get a full perspective of the application scope, but also by providing techniques for analyzing the effects of adding new features to an existing software system.

Testability

Testing is an essential activity in order to establish confidence in that the software is correct with respect to the specification. Testing is also used for obtaining some confidence in operational quality properties such as reliability, performance, etc. A lot of time, and consequently, money, is spent in the testing phase of software development. To reduce the amount of time needed for testing of the software, the architecture can be designed so that it is easy to test, i.e. having high testability. The testability is dependent on three individual properties: *observability, controllability*, and for concurrent systems and systems dependent on time, *reproducibility* [Bin94].

In order for a test case to be useful, the result of it must be observed. If the software entities in the architecture are considered as "black boxes", i.e. only the interfaces are observable. The bigger interface, the more visibility. Apparently, bigger interfaces give higher observability, thus higher testability.

When performing a test, a particular input stimuli is given to the system or a sub-system. This stimuli is the only way in which the test engineer can control the path taken in the program. If the path taken only depends on the input itself, maximum controllability is achieved. This is of course not the case in general. There are often data dependencies between different modules such as global variables etc. If those data dependencies, which are not controllable by the test input data, affect the control flow, the controllability is decreased, giving lower testability.

Finally, when testing concurrent system or real-time systems in general, the order in which different processes in the system are executed will influence the observed result from a test. For instance, in a system controlling the water level in a tank, there is one process sampling the actual water level and one process calculating how to adjust the water level based on the measured value and some desired value. If the control process executes twice without any intermediate execution of the sampling process, the result of the control decision will be different in the second invocation than if the water level was re-sampled in between. To get high testability, the order in which processes execute must be controllable or deterministic, i.e. high reproducibility [TH99].

Reusability

Reusing a software component to its full extent, without any modifications, is extremely difficult if not the domain in which the reuse is intended is the exact domain of the component's origin. When a component or architecture is reused in the same application domain we call it a domain-dependent reuse. When containers are reused, i.e. lists, arrays, sets, etc., they can be reused across different application domains. An example of such reuse is the Standard Template Library (STL) for the object-oriented language C++. Reuse, which is possible across the application domains, is consequently called domain-independent reuse.

When analyzing the level of reusability of a software component or a part of the architecture, one must consider not only the original application domain, but also how isolated and independent it is from rest of the system. The less dependencies, the more reusable, and vice versa.

The focus on reuse, in industry, has been intensified due to the potential cuts of cost. The time spent on implementation decreases when reusing components. Furthermore, components can be bought from third-party developers. Such components are called Commercial-Off-The-Shelf components (COTS). However, constructing reusable software is expensive. The business case must be clear and the extra cost must by justified by planned reuses.

Portability

To be able to analyze software architectures with respect to portability, the infrastructure on which the system is going to run on has to be modeled as well. This to unveil the dependencies between the software components in the system and the infrastructure. As infrastructure we consider the hardware, e.g. processors, A/D converters, as well as software such as operating systems and communication protocols. If the amount of direct dependencies, i.e. the number of components having a direct connection to the infrastructure, is low, then the architecture as whole is quite insensible to a change of infrastructure. Thus, having a high degree of portability.

Design recommendations: Portability

Designing an architecture that is portable is all about isolating the dependencies to the entity that is subject for being exchange. By introducing an *abstraction layer* that hide, for instance, the operating system, we only have to rewrite the abstraction layer when moving a software system to another operating system. If the system uses a particular communication component that provide a communication protocol, we may introduce a *proxy*, that acts as a communication component for the system by providing an interface that remains unchanged even though the communication component is exchanged.

Modifiability

The Architecture-Level Modifiability Analysis Method (ALMA) has been developed by Bengtsson et. al. [Ben02]. It is a scenario-based analysis method that, given a set of change-scenarios, identifies the affected components, the impact on those components, and the ripple effect. The input to such an analysis is a high-level architectural description in any form as long as components, their interrelations, and their relations to the

systems environment is present. The result from applying this method is, as for any scenario-based method, dependent on the selection of scenarios. It is important that the correct, and important scenarios are elected. The result can be presented qualitatively, e.g. a description of the changes that are required, or quantitatively by ranking the effects, e.g. a five level scale.

Design recommendations: Modifiability

It is impossible to give any general recommendations on how to construct a modifiable software system. It depends heavily on the predicted changes. However, low coupling between software components minimizes the ripple effect when changing a component.

Maintainability

Maintainability and modifiability are strongly related. Maintenance in software corresponds to a modification in contrast to a mechanical construction where maintenance could mean exchanging a broken part by a new, equal part.

Kazman et. al. [KABC96], have proposed a methodology for visualizing the amount of changes required in the modules or in the architecture when adding or changing functionality in the system. The amount of changes in the software architecture enforced by adding new functionality or error corrections, are referred to as maintainability. By using scenarios developed from the requirements of the new function, the existing architecture is analyzed. The concept of direct scenarios were introduced meaning scenarios that are directly supported by the existing architecture i.e. no major architectural changes are required. In contrast, an indirect scenario exposes the need for architectural changes, which is more difficult and costly to achieve.

After having mapped the scenarios on the architectural structure and determined if the scenario is direct or indirect, scenario interaction should be revealed. Two or more indirect scenarios are said to interact if they affect the same module. To make the potential architectural violations and changes in the system visible, graphical representation of modules were scaled graphically in the ADL according to the amount of indirect scenario interactions.

Design recommendations: Maintainability

As for modifiability it is impossible to give any concrete recommendations on how to design a maintainable software system. It depends on the types of maintenance that are foreseen in the design phase.

2.6 Existing analysis methods

A method called *Software Architecture Analysis Method* (SAAM), has been developed at the Software engineering institute (SEI) at Carnegie Mellon university. The purpose of SAAM is to analyze software quality attributes by examining competing architectures [KBAW94]. To do so, they partitioning the functionality in the architecture, i.e. identifies where in the different architectures the functionality of the system is allocated. The functional partitioning is system domain specific. Some domains already have a well-defined functional partitioning; a typical example of such a domain is compilers. Compilers are built with a front-end, a parser, a code generator etc. However, nothing is assumed about how functions are organized and structured, i.e. the architecture of the compiler. This partitioning gives a common description and common modules, each with the same functionality but organized in different ways. The communal description is an absolute condition for the comparison, which aims to unveil how well a certain quality attribute, is adopted by the architecture. Again, the analysis is based on scenarios, constructing input for a tradeoff analysis.

A method for tradeoff analysis called *Architecture Tradeoff Analysis Method* (ATAM) is also developed at SEI [KBK+99] [CKK02]. In fact, ATAM is a continuation of the SAAM method described above. Basically, ATAM defines four phases: *presentation*, *investigation and analysis*, *Testing*, and *reporting*. In the presentation phase different stakeholders present their knowledge in the evaluation group. The evaluation leader presents the ATAM method, the project leader presents the business goals that motivates the development effort and the evaluation, the architect describes, and motivates the architecture. The architectural styles, or as they are called in ATAM, the architectural approaches are identified in the investigation and analysis phase. Moreover, the quality properties that are important for the system is identified and specified down to the level of scenarios with their individual importance. This is presented in a *utility tree*. The architecture is then analyzed based on the high-priority scenarios in the utility tree. The result of this activity is

the architectural *risks*, the *non-risks*, the *sensitivity points*, and the *tradeoffs*. Risks are potentially problematic architectural solutions/decisions, whereas non-risks are good architectural decisions. Sensitivity points are properties of one or more components, or component relationships, that are crucial for achieving a particular quality property. They indicate that changing such a property can potentially endanger the architecture. A tradeoff is a relation between two or more quality properties such that a raised level of one, results in a lower level of another. Hence a tradeoff is a sensitivity point that affects several quality properties. For instance, a system may have requirements on performance and security. Raising the level of encryption will give a more secure system, but as it requires more computation, i.e processing time, the performance will go down. In the testing phase are three different classes of scenarios identified and analyzed: *use case scenarios* that express in which way the system is to be used, *growth scenarios* that represent ways in which the architecture is expected to change, e.g. modifications, porting to a new OS, and *exploratory scenarios* that represent extreme forms of growth, i.e. dramatic new requirements. The last phase in ATAM, i.e. reporting, basically presents the results, i.e. the identified approaches, the scenarios, the utility tree, risks, non-risks, sensitivity points, and tradeoffs.

Architecture-Level Modifiability Analysis (ALMA) is developed at the department of Software Engineering and Computer Science at Blekinge Institute of Technology [Ben02]. ALMA consists of five steps: *goal selection*, *software architecture description*, *scenario elicitation*, *scenario evaluation*, and *interpretation*. The goal selection activity establishes the goal of the analysis. ALMA can target the following goals:

- Maintenance prediction, i.e. the effort required to modify the system to accommodate future changes
- Risk assessment, i.e. identifying the type of changes for which the architecture is inflexible
- Software architecture selection, i.e. comparing two candidate architecture with respect to modifiability.

In the software architecture description step the architecture is described in the level of details required by the analysis. Typically an ADL should be used. Scenario elicitation is the process of finding and selecting the change scenarios that are to be used in the evaluation step of ALMA. The scenario evaluation step identifies the components affected

by the change scenarios, the impact on those components, and the ripple effect. Finally, the interpretation step aims at drawing conclusions from, and interpret the results from the evaluation step.

2.7 Architectural design

Architectural analysis can, and should, be used as guidance when designing a software system. A software system can be implemented in several ways, all having different architectural solutions. By using architectural analysis, the architecture that fulfills the requirements best can be chosen. The workflow for designing architectures for a system is shown in Figure 2.9.

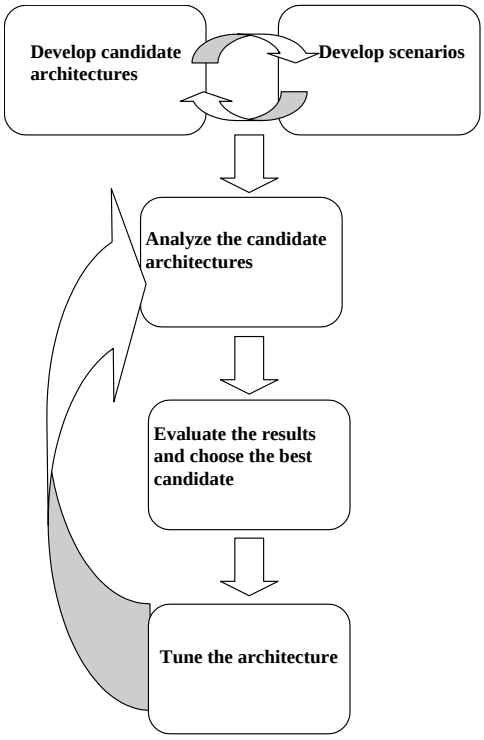

Figure 2.9: Architecture development and analysis process.

The first phase when developing a software system is to develop candidate architectures and a set of scenarios that reflects the requirements on the system. The number of scenarios to develop is related to the

generation of ordinary test cases. Eventually, a state is reached where the added value of a new scenario is less then the effort required to develop the scenario itself. When this point in time is reached the scenario generation should stop.

Now we have the candidate architecture and a set of scenarios. By executing the scenarios on the architecture a table with the desired quality attributes can be constructed. In the table, all requirements are marked with plus signs and minus signs to represent how well the architecture fulfills the requirements. If the result from the analysis is satisfactory, the next phase is to do low-level design and implementation. However, if the analysis results are not satisfactory, an alternative architecture must be developed, or the best candidate must be modified or tuned. It is important that exactly the same scenarios are executed all over again on the architecture in order to verify the new, or changed architecture. The work of finding a sufficient architecture is highly iterative, meaning that the architecture can evolve by small steps until a reasonable solution is found. Consequently, changes suggested by the analysis may result in a complete redesign using a completely different architectural style or minor modifications in subsystems only.

The table produced in the analysis phase containing all the analyzed quality properties constitutes the input to a tradeoff analysis. In a tradeoff analysis the competing architectures are compared or the result from a refined architectural solution is compared with the result from the analysis of the preceding generation of the architecture. The objective of the tradeoff analysis is to choose the architectural alternative that best complies with the ranking among the quality properties.

2.7.1 Architectural analysis: An example

Let us revisit the example in Section 2.4.6 in order to give an example of how to analyze and transform an architecture in order to better comply with its considered requirements.

Besides the already mentioned temporal correctness, the system should be easy to modify to run on different platforms (real-time operating system and hardware), i.e. portability is an important issue.

Verifying the temporal behavior requires the temporal aspect of the architecture. For this particular application, the period time, the estimated WCET, and the deadlines for the three tasks is shown in Table 2.4.

The temporal behavior is, in this case, verified using Fixed Priority

Task	Period time (T)	WCET (C)	Deadline (D)
Sampling task	1 ms	50 μs	60 μs
Control task	2 ms	200 μs	1 ms
Actuate task	2 ms	50 μs	1 ms

Table 2.4: The temporal view

Analysis (FPA), where the worst case response time for every tasks is calculated. If the response times are less than the specified deadlines for all tasks, the system is schedulable [JP86]. FPA requires priorities to be assigned to the tasks. In this particular example, priorities are assigned according to the rate monotonic algorithm where the task with the shortest period gets highest priority [LL73]. Rate monotonic gives the sampling task high priority, the control task medium priority and the actuating task low priority. The FPA formula is recursive and calculates the worst case response time with respect to interference of the execution of tasks with higher priorities. The recursion stops when two subsequent calculations result in the same response time, i.e. a fix-point is reached. The formula is shown below:

$$R_i^{n+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \tag{2.2}$$

where $\forall j \in hp(i)$ denotes all tasks j with higher priority than task i.

The response times for the sampling task is 50 μs as no other task interferes with it since it has the highest priority. The response time for the control task is 250 μs. Finally, the actuate task has a response time of 300 μs. If the calculated response times are compared to the specified deadlines, it could easily be verified that the system is schedulable as the response times for all tasks are less than corresponding deadlines.

To assess portability, scenarios can be used. For the matter of simplicity, only one scenario is used in this example, namely: *Move the system to another platform.* The idea is to execute this scenario on the proposed software architecture to estimate the number of component being subjects to changes. As portability is the issue, the number of affected components should be held to a minimum. In the architecture suggested in Figure 2.4, all the components interact with the real-time operating system. Consequently, there are a lot of platform specific system calls embedded in each and every component, giving poor portability since

every component has to be changed as a result of a changed platform. To increase the portability, architectural transformations have to be performed, i.e. the software architecture has to be refined. One possible transformation is to introduce a component that acts as an abstraction layer between the all components and the real-time operating system. This transformation is shown in Figure 2.10.

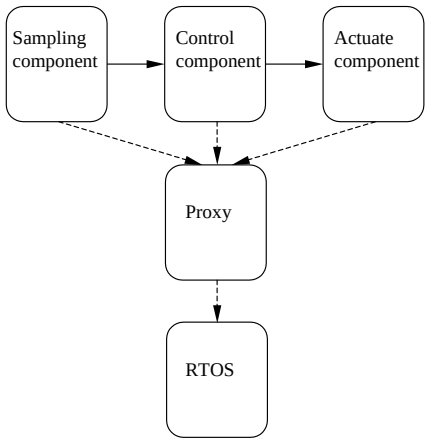

Figure 2.10: The logical view of the architecture after the proposed transformation

The abstraction layer component provides the tasks with all necessary services in order for them to perform their intended tasks, while hiding the actual system calls. To verify the new architecture according to the requirements, the scenario has to be re-executed. Now the abstraction layer component is the only one affected by a changed platform, i.e. a maximal portability is achieved. However, the portability is achieved at the expense of an increased overhead for system calls. Therefore, the worst-case execution times for the individual task components must be re-estimated and the FPA must be done all over again to verify the temporal behavior of the system. The phenomena that quality properties might affect each other in a negative manner, is referred to as *tradeoff*.

2.8 Discussion

Software architecture is part of what generally is referred to as software engineering. Software engineering also includes a lot of other techniques

like software metrics, formal methods, test methodologies, etc. Thus, software engineering is an umbrella for all techniques and methods needed to establish a "science of engineering" practice in the software community. Software architectures are an important part of software engineering since it deals with high-level modeling and evaluation. The software architecture community is still very young, but the recent interests from the industry have launched a lot of research activities in academia. Especially relevant are the software architecture analysis methods as the analysis provides the information for early design decisions.

Tool support for architectural development and evaluation is poor. It is possible to formalize knowledge in frameworks, guiding the designer in both architectural transformations and in the tradeoff analysis. There exist tools for some of the analyses, for instance tools for verifying the temporal behavior in a real-time system [SEG97], [WAN+03], but these tools are still islands in the ocean called software engineering. We need to discover, or build new islands and connect them to each other in order to get complete suits of tools, supporting the complete software development- and maintenance process. In mature engineering disciplines, such tool support is taken for granted. Software engineering tools will probably appear as the software community gets more mature, it is still very young, at least when compared to other traditional engineering disciplines.

Chapter 3

Product line architectures

In this chapter we will present software product lines in general by introducing current research and terminology. Moreover, we will discuss matters that are of particular importance when applying a product line approach to real-time systems. Non-technical issues such as business strategies and organization are also very important issues that are briefly discussed.

3.1 Introduction

Today the trend in computer-based products, such as cars and mobile phones, is shorter and shorter life-cycles. As a consequence, time spent on development of new products or new versions of a product must be reduced. One solution to this emerging problem is to reuse code and architectural solutions within a product family. Besides shortening development time, properly handled reuse will also improve the reliability since code is executed for longer time and in different contexts [FP96].

Basically, the product line approach is about maintaining and developing an architecture and a set of reusable software components that is common to a set of related products. By related products we mean a set of product that share a considerable number of features, and variants of the same *features*.

The notion of *platforms* also flourishes in the community. Platforms has been used synonymously with product line architectures, but it can also mean a subsystem that is used in several products which does not necessarily provide a particular architecture. A typical example of such a platform is a communication platform that provides the mechanisms

	dual band	wap	tetris	mp3	modem	Triple band
phone A	X			X		
phone B	X		X		X	
phone C		X	X		X	X

Table 3.1: A feature matrix for a product line of imaginary cellular phones

for data communication which is utilized by products that are developed in isolation. Henceforth we will distinguish platforms from product line architectures.

We say that features are the highest level of functional decomposition of a software system. A feature is typically a function as being grasped by an user of a system, or other stakeholders of the system. In general, a feature collects a set of functional requirements and quality requirements that together implement the feature. Quality requirements on features are sometimes referred to as non-functional requirements. Examples of quality requirements on a feature are reliability, availability, temporal correctness, etc. An example of a feature taken from the automotive industry is an automatic cruise control. This feature involves several requirements regarding turning the cruise control on and off, automatic control performance, etc. Furthermore, there are requirements concerning availability and timeliness.

There may also be a set of product-specific features. The software reuse in the product line approach is typically within the scope of an organization, or potentially, within a domain. In [Bos00], the author suggest that a product line could be described in terms of the products and their features. Such a description is called a *feature matrix*. In Table 3.1 the feature matrix of an imaginary product line of cellular telephones is displayed. Typically a product line for consumer products such as telephones, have products ranging from a low-end product to a high-end version.

3.2 Software product line architectures

A software product line architecture is concerned with a set of explicitly allowed variations, whereas with a conventional architecture almost any variation will do as long as the single system's behavioral and qual-

ity goals are met. A product line architecture is concerned with which variations that are instantiated, since an instance is a product. Thus, describing the allowable variations is part of a product line architecture's responsibility, as is providing built-in mechanisms for achieving them. Identifying the points of variation is not a trivial task. Not only must the variation points support the variations in the current product line, ideally they should also support features planned for the future and even unknown future features. Hence, designing a good product line architecture and reusable components requires engineers with great domain knowledge and experiences from developing similar systems.

We can view the creation of a product based on the product line approach as consisting of three different phases: the *product line architecture phase*, the *product architecture phase*, and the *product instance phase*. The phases are depicted in Figure 3.1.

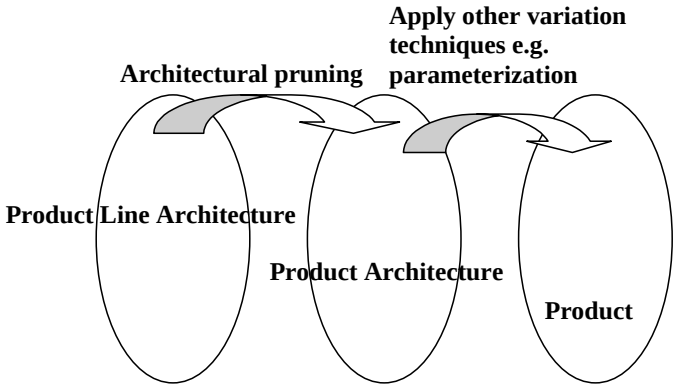

Figure 3.1: The three different phases of creating a product in the product line approach.

However, product line architecture and the product architecture can be identical. Hence the creation of a product may happen in two phases only.

Product lines introduce other pressures on the architecture that have to do with the variability that it must provide. Different products in a product line may have different quality property requirements. For example, one product may be highly secure but slow, whereas another may be fast but in-secure. The architecture must be flexible enough to support both. Moreover, products in a product line exist simultaneously

and may vary in terms of their behavior, quality attributes, platform, network, or in many other ways.

Support for variation can take many forms. Variation can be accomplished by introducing build-time parameters to a component, subsystem, or collection of subsystems whereby a product is configured by setting a collection of values. This type of variation assumes that all variants have been envisioned, a priori, and accommodated in the existing code. Conceptually, each combination's parameter values should correspond to a different product in the product line's scope. However, some parameter combinations may be disallowed as semantically meaningless or outside the scope. Conversely, some products in the scope may be achievable only by a means other than setting parameter. The following list includes some of the most usual mechanisms for achieving variability:

- Architectural pruning, i.e. removing superfluous parts of the architecture and extending the architecture with product specific components.
- inheritance can be used when a method needs to be implemented differently for each product in the product line
- extensions and extension points: used when parts of a component can be augmented with additional behavior or functionality
- parameterization can be used when a component's behavior can be characterized abstractly by a placeholder which is then defined at build-time.
- configuration languages can be used to define the build-time structure of a system, including selecting (or de-selecting) components
- code generation can be used when there is a higher level language that can be used to define a component's desired properties

In Chapter 4 our component model is described. It provides a set of variation mechanisms as well as means for predicting the temporal behavior and the correctness of the configuration of an assembly of such components.

3.3 Software product lines for real-time systems

In this section we describe how to employ the concept of product line to embedded real-time systems. The use of the product line concept

in the context of control systems for construction equipment vehicles is also presented. The main objective when designing a PLA is to make it *flexible enough* to incorporate and support all products defined as being part of the product line. Moreover, the design activity involves some degree of clairvoyance, since potential future products and features have to be taken into consideration as well. Developing a PLA is indeed an investment for the future, thus, *flexible enough* also embraces future requirements.

When instantiating a product, the PLA is tailored via different techniques such as parameterization, and by populating the architecture with components. The PLA approach has been used for soft real-time systems [Bos99], where the timeliness is of less, or no, importance compared to real-time systems in general. However, in systems where the temporal correctness is of vital importance for the reliability of the product, the temporal requirements and the temporal behavior must be included in the PLA. These temporal requirements must not be violated when instances of products are created.

We propose a set of methods that can be utilized in a design process suitable for developing product lines for real-time systems. Ideally, the process starts in a requirement-capturing phase where the requirements from all products in the line are collected. Commonalities in functional- and temporal requirements among the products will be considered when the actual PLA is designed. The PLA is then analyzed. The objective of analyzing the PLA is to gain confidence in that the PLA is flexible enough to be a base on which all products can be realized without violating any temporal constraints.

To enable the use of a PLA and derivation of product architectures from the PLA we need an adequate ADL. The ADL should have a precise syntax and semantics to enable architectural analysis, including analysis of, for instance, performance, maintainability, flexibility, and temporal properties. Moreover, the ADL must facilitate constructions for modeling of flexible components. The flexibility mechanisms specified on components constitutes the variation points that are used when product architectures are derived from the PLA, i.e. when the PLA is tailored for a particular product

David Stewart et al. have addressed the area of reusable components for embedded systems and how to create a framework for building systems based on components [SVK97][Ste00]. They have introduced a component model that provides flexibility in component behavior, hence reusability, through parameterization. In this paper, component reuse is

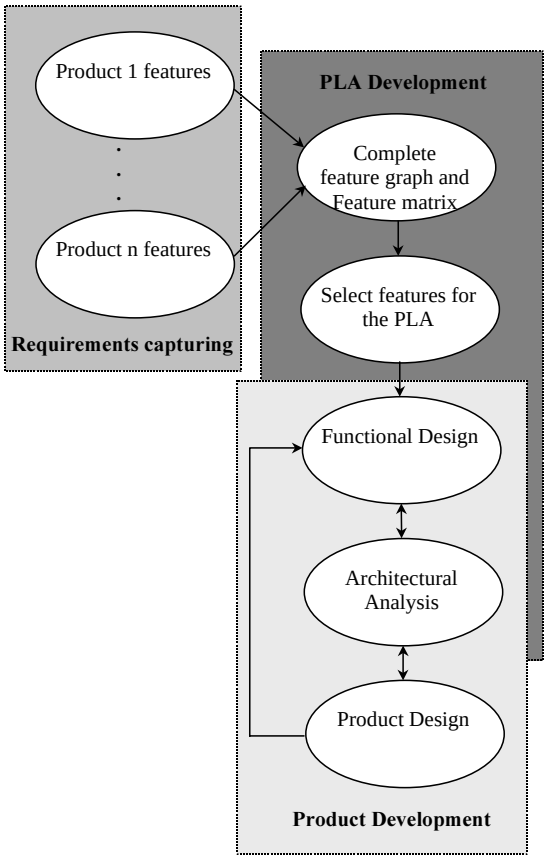

Figure 3.2: The process of developing a product line based on a product line architecture

accomplished by using additional techniques, not only through parameterization.

Moreover, we will discuss briefly the development process in which a product line architecture for embedded real-time products is constructed. The process is iterative and includes architectural analysis of properties that are of vital importance for a product line architecture, e.g. flexibility. Moreover, the derivation of products from a product line architecture is dealt with. The design process proposed in this paper is shown in Figure 3.2, where the process is divided into requirements capturing, product line architecture development, and product development.

3.3.1 Developing a product line architecture

Developing a product line architecture is done either in an evolutionary or revolutionary way [Bos00]. The evolutionary approach to product line architecture design is conducted by a generalization of existing products, whereas in the revolutionary approach, the common architecture is developed, rather than extracted from existing implementations. Independent of whether the evolutionary or the revolutionary approach is taken, the first step when developing a PLA is to capture the requirements for every product in the product line. One way to do this is to organize and group required functionality into features.

Scoping is performed on two distinct levels, *product scoping* and *feature scoping*. It is i vital importance that the products in a product line have sufficient level of resemblance. If the products are too sprawling in terms of functionality and behavior, the resulting product line architecture will become to general. Hence, in order to instantiate a product a majority of the components has to uniquely configured, i.e. very little is reused as it is. This is especially important when introducing a product line approach in an organization where the history of producing products is rather long. If this is the the case, experienced engineers tend to consider too many different systems when designing the architecture and the software components.

Once the product scope has been decided, the features in these products have to identified. As described in Chapter 3, features of a product line can be collected in a feature matrix. However, the feature matrix shown in Table 3.1 specifies the products and the features only. In embedded real-time systems are resources, e.g. CPU, memory, communication busses, limited. It is important that features do not over-utilize the resources available in the product line system architecture. Hence, we extend the feature matrix to also specify resource utilization per feature. In Figure 3.3 the multi-dimensional nature of a feature matrix for real-time systems is shown. We refer to such a matrix as the *resource-feature matrix*. The exact number of dimensions is decided by the resources in the system architecture. For instance, if it is a single node system, there is no need for specifying a communication bus utilization.

The features and their *explicit*- and *implicit* temporal requirements, which contributes to the resource utilization, are identified given the system's requirements specification. An explicit temporal requirement is a clearly expressed requirement such as an end-to-end deadline. An implicit temporal requirement is derived from a functional requirement

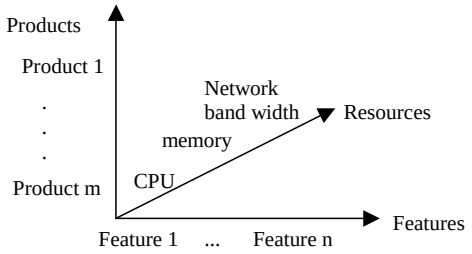

Figure 3.3: The multi-dimensional resource-feature matrix

or the controlled environment. For example, an accuracy requirement for a robot must be translated to timing requirements for the services involved and thus the accuracy requirement is implicit.

The utilization of any resource in a software system must not exceed 100 percent in order to be feasible. The resource utilization is, however, a very coarse specification. A CPU utilization less than 100 percent does not guarantee a temporal correct system. By temporal correct we mean that all temporal requirements are complied with. Examples of temporal requirements are deadlines the must be kept, jitter constraints, or latency requirements.

Consequently, the temporal correctness of the system must also be verified separately when more detailed information about the temporal behavior is available. There exist an abundance of methods for verifying the temporal correctness of a real-time system,as has been discussed in Section 2.5.2. All of them makes different assumptions about the task model provided by the infrastructure, i.e. the real-time operating system. Consequently, the selected method for schedulability analysis will vary.

Roughly, we divide the specification of a product line architecture into three different views that provide different level of abstractions. The views are: the *feature view*, the *component view*, and the *implementation view*. In Figure 3.4, we illustrate the different views of a specification as well as how information flows between those level of abstractions during the life-cycle of a product-line.

The source code constitutes the implementation view of the functional domain. In the temporal domain the implementation view is provided by the temporal attributes available in the infrastructure. Just as the functional requirements must be transformed into source code, the temporal requirements must be transformed into the temporal attributes of the

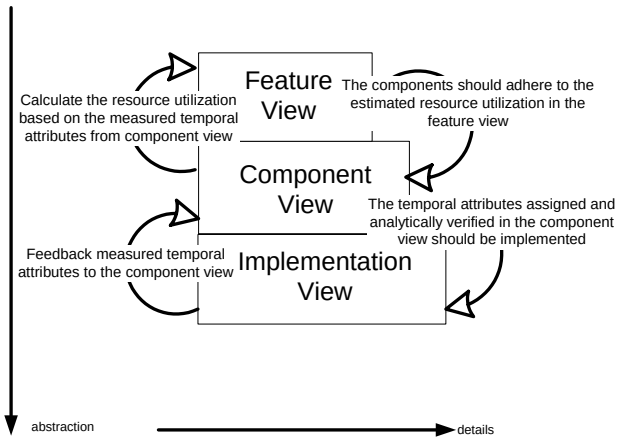

Figure 3.4: The different level of abstractions

chosen real-time operating system. Typical examples of such temporal attributes are, period times, and priority. A temporal requirement such as a specific deadline must be fulfilled by assigning, e.g. period times and priorities to tasks. The functional implementation also contributes to the temporal implementation as execution times are determined by the source code.

The component view provides a decomposition of features into software components and tasks. Tasks describes how components are to be executed, i.e. in the temporal domain. Typically, attributes such as offsets, deadlines, period times. etc, is specified here. Synchronization among tasks, and communication among components is also a part of this view.

Features are the highest level of functional decomposition in our framework. A feature must not necessarily correspond to one single component in the implementation domain. Several components typically interact in order to exhibit the specified behavior. A component may also be involved in the implementation of several features. The architectural language for real-time product lines proposed in Section 8.2.1 has support for all three of the level of abstraction discussed here.

A pervading characteristic of modeling real-time systems is that not only the functional domain has to modeled and analyzed, but also the temporal domain. On the component level, and the implementation level

the temporal domain is fairly straight-forward as described above. In the feature view we specify resource consumption by estimating, or giving an upper bound, of the computational resources that a feature, at maximum, can utilize. Such a specification is likely to be inaccurate. Thus feedback from the component view and the implementation view is important. Hence, a highly iterative development process is required.

Features are rarely independent from each other. A feature can for instance depend on other features in order to deliver the desired functionality. Another example of a relation between features is the mutually exclusive relation, implying that only one of the related features can appear in the final product. If mutually exclusive features must co-exist in the product, effort has to be made to resolve the conflict. Features in real-time systems will also exhibit dependencies related to the temporal domain. For instance, consider the lock-free break feature and the anti-slid feature in automotive vehicles. Both features need information about the wheels' velocities, thereby having a shared temporal dependency related to the freshness of the sensor data. Such relations on features are specified in a feature graph. The list below unveils some relevant relations between features:

- *Depends on*, a feature depends on the presence of another feature
- *Part-of*, a feature is part of another feature
- *Mutually exclusive*, two feature may not coexist in the same product
- *Conflicting*, trade-offs between quality requirements
- *Optional*, the feature is present in a subset of the product line

The relations among features can be visualized graphically in a *feature graph* [Bos00]. In 3.5, is a feature graph for the imaginary cellular telephone depicted. Note that not only functional dependencies are possible. For instance, two features may mutually exclude each other because the available memory in an embedded product is not big enough to encompass both, or the available computational resources can not sufficiently execute two of the features in the same product.

The feature matrix with resource specification, and the feature graph constitute the basis for deciding what features to include in the reusable part of the product line architecture, and what features to consider product specific. Typically, features that are common among a majority of the products in the product line are included, i.e. are developed as reusable

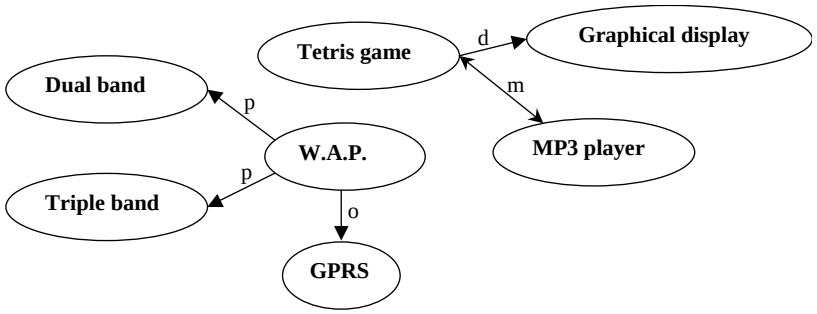

Figure 3.5: A feature graph for the imaginary cellular telephone, m denotes mutual exclusive, p denotes part-of, d denotes depend-on, and o denotes optional

software components. Consequently, the product line architecture may provide features that are superfluous for a specific product. Such features that are considered *product specific*.

The product line architecture must be flexible enough to incorporate product specific features upon product instantiation. Note that a product line architecture for real-time systems must provide sufficient flexibility in the temporal domain as well.

When the scope of the product line has been decided, the features should be mapped to components that, together with their interrelations, constitute the actual architecture. This part of the development process is referred to as functional design. Components can be implemented in both hardware and software. However, the focus of this thesis is on software components.

When performing functional design of a PLA, the designer must take into consideration that features may have different implementations for different products. Depending on how implementations differ, the correct mechanisms for obtaining the desired flexibility must be selected. In Section 3.5, some mechanisms for flexible software solutions are discussed. For real-time systems we also have to consider flexibility in the temporal behavior. A typical example of how the temporal requirements influence the functional design is the following. Consider features that are functionally equivalent between a set of products. If these products will run on different infrastructures, i.e. operating system and hardware platform, the functionality may be partitioned among components in different ways to fulfill the timing requirements. Moreover, in a high-end

product we can partition a feature in such a way that it will be easy to maintain while in a low-end product we have to make an architecture that is focused on performance to be able to fulfill the timing requirements given the limited resources.

After the initial design activity, the product line architecture must be analyzed in order to secure that the architecture is flexible enough to facilitate all products in the product line yet temporal correct. If not sufficiently flexible the architecture must be transformed. Thus, iterations between analysis and design are required. Since our focus is on real-time systems we would like to gain confidence in that the architecture is sufficiently flexible to be used in all products in the product line without violating the temporal constraints.

3.3.2 Product line architectural analysis

In this section we will focus on early analysis of temporal properties of an architecture as well as *flexibility* with respect to temporal correctness, i.e. the architecture must be flexible enough and still be temporal correct. Discovering that the real-time requirements cannot be fulfilled in a late stage of a product development often implies a delay in the release of the product on the market. Furthermore, such problems are often handled by ad-hoc optimizations, which in turn decreases maintainability and reusability. Positive experiences from making early analysis of temporal behavior using architectural descriptions and estimates of temporal properties have been reported in [NSG+00] where a new control system for construction equipment was developed by using a formal temporal model. As the temporal attributes were established early for every task in the system, e.g. execution time budget, period time, the integration of the complete system was very smooth.

By analyzing the temporal behavior, for each product, based on the product line architecture and the product specific features, we can extract traditional real-time measurements such as system utilization, response time for each feature, distribution of response times for a specific feature, and jitter information for features. However, this information is not only used for schedulability analysis, it is also used for analyzing the flexibility of the PLA with respect to implementation constraints.

Moreover, robustness with respect to internal errors and erroneous assumptions about the environment needs to be considered. Typical analysis is based on scenarios. Examples include:

- What will happen if a specific component overruns its time budget?

- What will happen if events from the environment are generated in a higher frequency than assumed?

To facilitate temporal analysis on the level of abstraction provided by the architecture, we must have information about the temporal behavior of the components that define the product line architecture. Such information can in principle be attained in two different ways, namely by earlier implementations of the same or similar functions or by intelligent guesses of the temporal attributes for completely new components. The execution time for a component is normally measured on the implemented component while the period time for a component is derived from the requirements. However, if we do not know the execution time for a component an estimate has to be made. This estimate is later used as an additional implementation requirement for the component.

The closer we get to the architecture for the specific products, the more confidence we get in the predictions. Depending on the confidence in the temporal information, different types of analysis can be performed, such as:

- Simulation, which can be used for exploring the system behavior when the confidence for the different parameters are low or when the system will operate in dynamic environment where the load is hard to estimate [LNP00] [WAN+03].
- Mathematical analysis which can be used when the product line architecture or parts of it has estimates of execution times, temporal properties such as period time, and the synchronization between the components specified. Mathematical analysis that can be used includes fixed priority scheduling [ABD+95][Bur93] and pre-runtime scheduling theory [XP00]. Especially different kinds of sensitivity analysis are of interest to predict the remaining capacity for product specific features [PDB97].

Based on the result of the temporal analysis of the products, different actions can be taken. If the analysis indicates that the requirements are fulfilled then the product line architecture is accepted. On the other hand, if the analysis indicates that the requirements cannot be fulfilled we have several options. As a result, the architectural has to be transformed. Typically, the temporal attributes for a particular component is changed, but it may also be necessary to reallocate features, i.e. moving them, completely or partially, to different CPUs. Transformations of

an architecture for embedded real-time product-lines can be any of the following:

- Modify the derived temporal attributes, e.g., deadline and period time. There are many ways to derive temporal attributes for the product line from the requirements, which may give different analysis results.
- tighten or relaxing resource specifications
- transform the architecture. For example, an architecture that is designed for portability could experience a trade-off situation, where we have to decrease the portability to gain real-time performance.
- change the hardware architecture, e.g. add more memory
- renegotiate the requirements. If none of the options above are possible we have to renegotiate the requirements with the stakeholders.

Following any transformation is a re-analysis of all products. We refer to this activity as *regression-analysis*. Regression-analysis is important in order to maintain the flexibility in the architecture.

From the perspective of adding new features to future versions of a product, we would like to have as much degree of freedom as possible with respect to system resources. Thus, we can strengthen the notion of flexible enough to be *as flexible as possible*. Consequently, we want the system's specification to be as tight as possible, i.e. leaving as much slack as possible on the resources for product-specific features/components and future development and maintenance. Feedback from the test- and verification phase to the system specification is consequently important. This ensures a specification that is indeed consistent with its implementation and that potential over-estimations made initially will be reduced. However, note that iteratively feeding verification results back to the design model in order to define an as flexible as possible, will delay time-to-market (TTM), and be more expensive than to settle with the first feasible architecture. Yet another approach to increase the resource slack available for future features is to increase capacity of the resources. Again there is a tradeoff between flexibility and the product cost.

When a stable generic architecture is found through subsequent analyses and transformations, the implementation phase can be initiated. Remember that the resource utilization specified for every feature must be

adhered to throughout the implementation. If they are violated the flexibility and the temporal correctness cannot be assured. If it turns out that the estimations of resource utilizations made in the specification phase are incorrect or too imprecise, the architecture must be transformed and regression-analyzed.

3.3.3 Product design based on a product line architecture

Requirements for the product architecture have already been addressed during the construction of the product line architecture. However, at that point the focus was on the commonality between several products belonging to the product family. The process of developing a product focuses on product specific requirements and functionality not covered by the product line architecture, i.e., the product line architecture must be tailored to cover the requirements of the product at hand.

The first step is to merge the product specific features with the features provided by the product line architecture consistently, i.e. merging the product's and the product line architecture's feature graphs. In this process conflicts can occur, e.g. overlapping features, superfluous features, temporal discrepancies.

When the feature conflicts have been resolved, the concrete product architecture can be derived from the product line architecture. The derivation involves architecture pruning. Although architectural analysis has been performed at the product line architectural level, the resulting product architecture provides more details. Thus we can make a more detailed analysis of the architecture with respect to, for instance, the temporal correctness.

The remaining step until we can release the product is to implement the components that make up the architecture in such a way that the temporal requirements are fulfilled. This step also includes component testing, integration, and verification of the integration, which have been covered elsewhere when considering temporal estimates [NSG+00].

3.4 An example of a successful product line

Volvo Construction Equipment (VCE), has built a distributed real-time system to control mainly the automatic gear box, the brakes and the hydraulic systems in a product line of construction equipment. There are five products in the product line: Backhoe loaders, Wheel loaders, Excavators, Articulated haulers, Motor graders. Each product has five

computer nodes. All the five products are different sized construction equipment vehicles.

The requirements for a new generation product line were brought to the development department. These requirements were evaluated to derive the desirable features and their temporal requirements. The construction of the product line architecture was in this case done strictly in an evolutionary way. There was products and application knowledge already at the scene. Because of this, the process of extracting the features was quite painless. The feature matrix did already, to some extent, exist in the minds of some of the experienced engineers at the development department. Examples of features in this product line are the control of brake fluid pressure and automatic shifting of gears. The development process was essentially equivalent to the process proposed in Figure 3.2.

The mapping of features onto components, and the temporal attributes, is specified in configuration files. A configuration tool uses the configuration files in order to schedule the system and produce configuration data for the operating system. An examples of such configuration data is time granularity in the system. Moreover, the synchronization and communication between components are specified in these files. Thus, the configuration files constitutes the architectural description. In order to implement components with a correct temporal behavior, time budgets were estimated. A time budget is a specified maximum time that a component can execute to perform its function. The fact that this was done before the actual code implementation could have been a problem, but based on experiences from earlier products the estimates turned out to be quite good.

Each node of the distributed real-time system has separate configuration files and source code files. These files, however, are common between products in the product line. All products have identical hardware and software architecture and accordingly the configuration and source code files are identical for the products. Hence, the executable for each product is identical to its product sisters. This can be done in a relatively straightforward way since the nodes have identical computer hardware.

The unique behavior of each product is instead defined by *datasets*. A dataset is a set of parameters that control the program flow. The program checks the parameter values and calculates output accordingly. There could, for instance, be a conditional invocation of a function depending on a parameter value.

The temporal properties of this PLA would require great effort to test if each product/node should be tested independently of the next.

We would then have to test all the nodes in all the products and see that all temporal constraints are intact. By designing the most advanced system first, the tests could be simplified. The most advanced model is assumed to conform a superset of the functions of all systems, i.e. its tasks should be the hardest to schedule. Successful testing of the most advanced model indicates that the simpler models also will perform correctly time wise. This approach work well since the schedule is assuming a minimum execution time and a maximum execution time. Any variations that are within that range will still behave correctly temporally. In this particular case, the minimum execution time is assumed to be zero. Hence, if a task is not executed at all, the product is still temporal correct. However, the execution of tasks may vary within the limits specified by release times and deadlines if the execution of another task is excluded, i.e. jitter. If the jitter requirement is not explicitly specified in the temporal requirements the implementation in the temporal domain, i.e. the schedule, could exhibit undesirable behavior.

Note that this does not cover the testing of the systems functionality. Functionality was tested separately. In this respect, the mentioned method might not be perfect. The method is inflexible when we want to add products that include more functionality than the most advanced product in the product line.

Designing the system to run the same executable for different products could, however, also be too costly if we have a wider span of functionality between low- and high-end products. The low-end product would have to include hardware capable of executing the high-end program, even if most of the functionality is disabled. Given enough separation between low- and high-end products, this could prove costly in terms of memory, I/O and CPU. This could also be too expensive in product lines where large amounts of units will be produced or even in small series products where system resources are scarce. For instance, mobile phones are large series products where it would prove too expensive to include even the least unnecessary hardware. For a product line architecture of quite similar products, these extra resources would probably be acceptable.

The product line architecture at VCE can be illustrated by the leftmost figure in Figure 3.6. All product architectures are derived within the boundaries that the PLA define. This gives the advantages, discussed above, when testing the temporal behavior of the products. On the other hand, we are somewhat limited in developing new products. The product line architecture must be changed altogether if a product with more

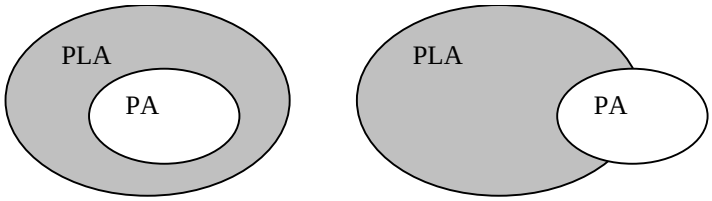

Figure 3.6: The relation between the functional sets of PLA and PA respectively

features is to be developed.

The ADL used in this project only allows parameter-based derivation of product architectures. This requires a very homogeneous product line. In Section 3.5, we will discuss desirable properties of an ADL for describing product line architectures.

3.5 Mechanisms providing flexible architectures

We need mechanisms, apart from parameterization, to obtain flexible architectures. Consequently, we need a design language that supports flexibility in component behavior, as well as flexibility in the systems architecture. We will refer to flexibility as optionability and variability. Optionability is the absence or presence of functionality, whereas variability is the possible variation in functionality. In real-time systems, there is also a need for variation of the temporal behavior, e.g. the period time of a control-loop might vary between products in the same line.

An ADL for product line architectures should not only have language primitives necessary for describing the structure of a software system. Such a language must also restrict, and guide the process in which an actual product is instantiated, to make sure that the product line architecture is not corrupted during implementation. Consequently, the primitives that describe the functional, and temporal variations must have a *well-defined semantics*. In Section 3.5.1 we propose language primitives needed for an architectural description language in order to describe variability and optionability in architectures for real-time systems.

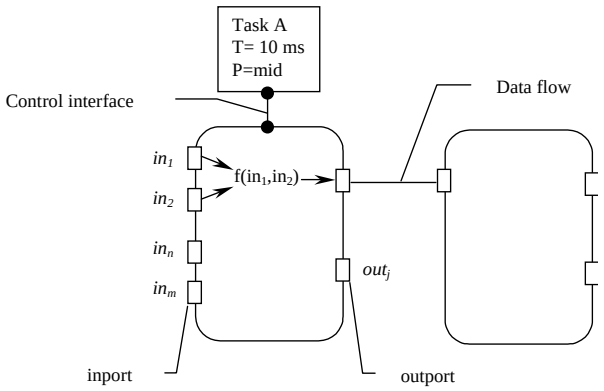

Figure 3.7: The data interface and the control interface

3.5.1 Language primitives for variability and optionability

In this section, language primitives supporting the specification of variability and optionability for product lines architectures for real-time system is discussed. The primitives that will be discussed are *components*, *interfaces*, and *tasks*.

A component is a computational unit having a data interface, a control interface, and a parameterization interface. Moreover, several components can be composed in a composite component. The interface of a component should be a separate language primitive so a specific interface can be used for different components as long as each component implementing the interface conforms to it.

The data interface of a component represents the data-flow to, and from the component and the control interface represents the control-flow (see Figure 3.7).

The control flow is defined by associating one or more tasks to a component. A task provides the thread of execution of a component. Further, one task can control several components, which is beneficial when for example two components should be executed serially, and several tasks can control one composite component which includes several parallel sub-components. A task also defines the temporal behavior of a component and can be either periodic or aperiodic.

The temporal parameters in the control interface are of two kinds; the ones derived from the requirements such as deadline and release time of the task controlling the component, and the execution time of the

components controlled by the task. However, the execution time of a component has to be specified for the component implementing the functionality and for a specific hardware platform. Such specification can either be based on real assessments or by intelligent guesses, as discussed earlier. In the latter case an intelligent guess becomes an implementation requirement of the component.

Furthermore, specification of synchronization can be achieved by using semaphores for mutual exclusion relationship between different components and signals can be used to trigger the execution of another component.

In the control interface of a composite component the temporal parameters of each composed component can be altered. As an example consider an interface for a multi-rate feedback controller component, composed by a sampler component, a controller component, and an actuator component. Such an interface can specify period times for each of the different sub-components as well as control jitter constraints.

The parameterization interface specifies the parameters that provide functional variation in pre-implemented software components. Our component model is described in more details in Chapter 4.

When defining an interface for a component that is not yet implemented, and for which we suspect that the implementation can include several sub-components, we can specify an utilization bound for each platform it shall run on. The utilization can be specified for any existing resource, e.g. CPU, communication busses, memory.

Our component model which was developed in order to provide the flexibility mechanisms discussed here, yet provide analyzability, is presented in Chapter 4.

3.5.2 Variability

Product line architectures for real-time systems can be varied functionally and temporally. Variations in component behavior and the systems architecture itself constitute the functional variation, whereas variation in the temporal behavior is related to tasks and their temporal attributes.

We have identified three possible techniques for obtaining variability of components. Basically, the techniques describe how interfaces and implementations alters a components behavior between products:

- The same interface, but different implementation
- Different interface and implementation

- The same interface and the implementation is varied by parameterization

In order to describe the techniques listed above, we use the language primitives discussed in Section 3.5.1, i.e. components and their different interfaces. To model a construction having the same interface but different implementations, we use the interface language primitive. Thus, in the architectural description only the interfaces are present. Each product instance of the architecture then has to implement the behavior by providing one or more components that comply with the given interface description. Hence, the components have to implement that particular interface.

If parts of the interface, as well as the implementation of the interface are different between products in the product line, we cannot describe this with the interface language primitive. For this purpose, we introduce a language primitive called abstract component. An abstract component indicates that the implementation of the component is tailored for each product. Note that although the component is abstract, several products, but not all, may share the same implementation of it.

The last identified technique for obtaining variation in components is parameterization. Here the same implementation is used throughout the complete product line, but the components' behavior is controlled by parameters. Consequently, the implementation must take all possible behaviors into account when the component is designed. Different behaviors does not necessarily correspond to different control-flow in the implementation; it can also be related to constants used in the calculations. For instance, a PID controller component could be provided with the proportional gain, the integration time, and the derivative time constants upon instantiation.

When the systems architecture is varied between product instances, the product-line architecture describes how the system is to be organized. As an example, consider a fire-alarm system where the number of smoke detectors varies depending on the size of the building in which the system resides. The smoke detectors in a product-line architecture for the fire-alarm system are described as a multiple component, i.e. the exact number of smoke detectors is decided upon instantiation of the product. In Figure 3.8, the sensor is a multiple component that represents one, up to n instances connected to the control component.

As discussed in the introduction of this chapter, real-time systems can also be varied through their temporal behavior. The control inter-

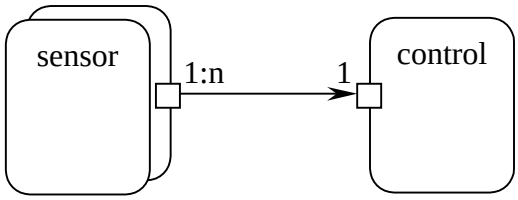

Figure 3.8: A Multiple component

face of components provides this variability. By changing the tasks that control a component through the control interface, the temporal behavior of that component is varied. For instance, the execution strategy and temporal attributes such as the period time can be altered. If such changes are made to a task, the temporal correctness of the system must be re-verified.

3.5.3 Optionability

Optionability is concerned with the absence or the presence of functionality in a system. We define components to be the smallest software entity subject to optionability, i.e. we can only add or remove complete components. Components in our description language that, for some products in the line, can be removed have a property called optional. The removal of a component could result in partial use of an interface in the product-line architecture. As a consequence of adding or removing components, the temporal domain is affected. For instance, tasks are removed, added, etc.

3.6 An example

In this section a small example is presented in order to make our discussion more concrete. Keep in mind that although we are targeting small, embedded systems, they are substantially larger and more complex than this small example.

The example system is a PID control application. The product-line currently consists of two different products, one cheap low-end product with an electrical sensor, and one high-end product featuring an optical sensor. The feature matrix is depicted in Table 3.2.

	electrical sensor	optical sensor	PID	Signal
Low-end product	X		X	X
High-end product		X	X	X

Table 3.2: The feature graph for our control product-line

Furthermore, we have a non-functional requirement on the PID controller that it must be reusable in several other applications where the controlled system's dynamics might be different. The process values read by the sensor must be processed. Thus we need a signal-process component that, according to the requirements, should base its computations on the ten latest sampled process values. Consequently, the sampling component will have a frequency ten times the signal-process, and the in-port to the signal-process must exhibit a buffered semantics. We must take into consideration that the sampling part of our system must be differently implemented for every product. Hence, the temporal constraints such as frequency and deadline must be relaxed so that the execution time of the different services can fulfill them.

Figure 3.9 shows the product-line architecture that fulfills all requirements. In order to support both products, the sampling component is specified as abstract. Thus, the low-end product must implement its own specific electrical senor sampling and the high-end must, consequently, implement an optical ditto. The in-port on the signal-process component has been specified as a buffer of size 15, which we suspect to be large enough to also support future possible uses. The PID component provides flexibility through parameterization of the control constants. The sensor component is also parameterized. The ratio between sampling and signal processing is set through the parameter S-S. This since the sampling component is responsible for executing the signal process component via a signal to task E on every S-S invocation. The number of invocations of the sampling component is kept track of in the state of the aggregated sensor component. Keep in mind that Figure 3.9 shows the product line architecture. In order to get an architecture for each and every product, we must derive the product architecture from the product line architecture (see Figure 3.1). In this particular case no architectural pruning is necessary. However, we have to utilize the specified points of variation in order to get a product. In this example we have to implement the sampling component and choose appropriate values for the

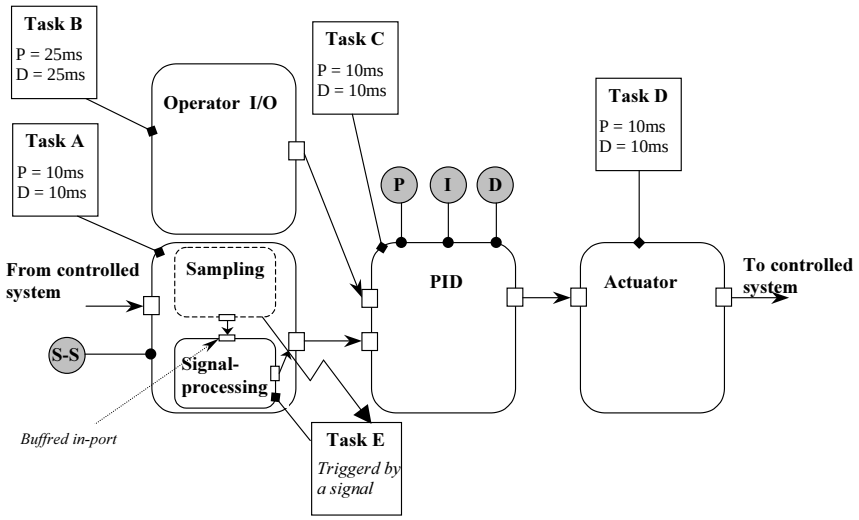

Figure 3.9: The example product-line architecture

PID parameters. Note that the different implementations of the sampling components may be reused in , e.g. a prospective new future product.

Finally the designed product-line architecture must be analyzed in order to verify, for instance, that the different products indeed can be implemented. Thus the architecture is flexible enough. Such an analysis will also include the verification of the temporal correctness. Hence, verifying the temporal flexibility.

3.7 Organization, process, and business

The product line architecture and the software components are, however, only two of the aspect of a product line approach. The software architecture is one *strategy* that has to be defined when moving towards a product line approach. However, we need a collection of strategies to successfully implement a product line. Issues that concerns organization of the development unit, processes, management education, and business strategies are equally important in order to success with the product line approach.

From the organizational point of view several implementations have been studied with respect to software reuse in [Faf94]. It is important

that strategies for the responsible of the reusable assets is established. In the list below some of the most crucial questions are listed:

- who owns the architecture?
- who is responsible for developing reusable assets?
- how is the reusable software development financed?
- what is the business relation between producers and consumer of in-house developed reusable software?

Ownership and responsibility are related issues. Responsibility of reusable architectures and software components is crucial in order to keep the architecture from degenerating. A person, a group of people, or a department have to be appointed as the ones taking all decisions regarding changes in the architecture and whether or not a component should be incorporated into the the reusable assets. Typically, but not necessarily, the set of reusable component grows as new features are developed and implemented.

It is also a well known fact that it is more cumbersome and expensive to develop reusable software than specific single product solutions. Studies have shown that producing a reusable component ranges from approximately 120 to 480 percent of the cost of creating a non-reusable version, and integration cost ranges from approximately 10 to 65 percent of the cost of creating a non-reusable version. The actual cost is dependent on the complexity of the implementation and the complexity of reusing it in an application [Fav90][Mrv97].

Certainly, no customer is willing pay the extra cost of developing reusable and generic software as no extra value for him/her is added. It has to be accepted on the manager level that the extra cost is well invested money that will pay back after a number of reuses. A customer should not be affected economically by the adoption of the product line approach, but rather the opposite. Nevertheless, customers may be affected, depending on how they are used to purchase systems, since the absolute functionality and behavior will be somewhat restricted to what the product line architecture can offer. The impact is related to the type of software system he/she is buying.

We classify different software based product along an axis ranging from pure a per customer based development to a pure consumer product

developer, illustrated in Figure 3.10. Organizations that have custom-based development produce an unique instance of a product for a majority of their customers, e.g. trains, or systems for control and supervision of power distribution. Moving along the axis we have products such as industrial robots which are, to a large extent, produced as consumer products. A company produces a fixed set of robot models, e.g. different sizes, and the customer themselves customize them as they program the movement of the robots on their own. However, a large customer can require a specific feature which is included on a customer basis in order to get share of the market even though this is not the way business is done in the general case.

The car domain is to be considered as consumer products but with a specific set of optional features which is build upon customer request. Typical examples of such options are automatic climate control, and navigation system. The product variation is more of optionality than variations in the behavior of common features.

Pure consumer products are typically presented as a product line with cheaper low-end models and more expensive high-end models. Typically the variations are in the set of provided features. However, the customers themselves can not decide the exact configuration of a product, but have to choose among the available ones.

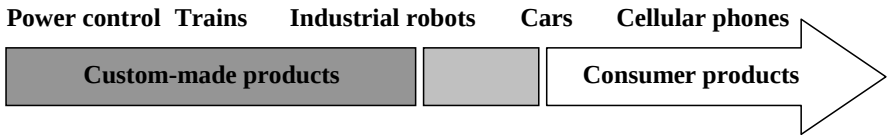

Figure 3.10: Examples of different approaches to sell and develop software based systems

Basically, the less variation between products in a product line and the less customer involvement in the production, the easier it is to adopt the concept of product line architectures. Nevertheless, stories of success exists from many different domains and product characteristics. Some of them have been described in [CN01] and [Bos00].

Moving from a pure a per customer production of systems towards a product line approach, will limit the variety of solutions that can be offered to a customer. A product that is captured by the scope of the product line and its intended use will be relatively cheap for a customer.

Any divergences will, however, cost more. Consequently, the business strategy in such a company, i.e. *the way in which it does business*, will change. The customer has to be convinced that a standard solution provides what they need, is relatively cheaper, and more reliable and well tested.

Chapter 4

Analytical models by construction

In this chapter we will present a component model for product line based development of embedded real-time systems (ReFlex). Moreover, we present a framework for analyzing properties of component assemblies based on analytical interfaces. The component model is specialized for developing embedded real-time systems taking the product line approach. The following requirements on a component model are in this context of particular importance:

- small infrastructure, i.e. memory efficient,
- predictability with respect to temporal behavior,
- mechanisms for functional, and temporal variability,
- specification of temporal attributes,
- mechanisms for concurrency control, e.g. synchronization, and
- analyzability

Memory is a scarce resource in most embedded systems. It does not permit large and complex infrastructures such as request brokers for dynamically finding components and interrogating components interfaces at runtime. Moreover, such runtime capabilities affect the predictability of a system. In order to have a predictable temporal behavior the execution times must be known and fixed within specified limits. Consequently, runtime mechanisms which are not predictable at construction

time can not be tolerated. Analyzability is required in order to establish the temporal correctness of a system, as well as other desirable quality properties.

4.1 Introduction

In order to analyze a software architecture with respect to a certain property, the information needed for analyzing that property must be provided by the architecture. From now on in this thesis we will refer to this information as the *analytical model* of a system. For instance, the information about period times, worst case execution times and priorities are necessary for deciding whether or not a fixed priority system is schedulable. Hence, period times, worst case execution times and priorities has to be a part of the system's analytical model.

In this thesis we present two different approaches to achieve analyzable architectures for real-time systems. We refer to the different approaches as by *construction* and by *re-engineering* Ideally, the information required by the analyses is provided in the construction phase, i.e. when designing the system's architecture. We refer to construction of analytical models in such early phases as by construction, in contrast to the re-engineering approach which we will describe in Chapter 5. In a constructive approach, the analytical model is a product of the development effort.

Moreover, we say that if the analytical model, i.e. the required information, is provided by the software components themselves, the components have *analytical interfaces*. The analytical interfaces of a component provide means for reasoning about properties of a set of assembled components. Hence, properties of a product that has been implemented with a set of components can be derived from the properties of the individual components.

Reuse is by far the most common argument for using component based software development. The ultimate vision is that components are mined and assembled into high quality software just as LEGO bricks are put together to form larger objects. Now a days, the component based software engineering community agrees on that this vision is more of a utopia. Reuse will not happen simply by introducing software components, it is far more complicated as it involves careful planning, management and mature development processes.

Nevertheless, software components is a concept which raise the level

of abstraction when constructing software based systems. The component abstraction defines components in term of their interfaces, i.e. the services they provide, and their contracts which describes the conditions under which the components provide their services. Hence, it is possible to reason about composition of different components independent from the implementations, as the component encapsulates and hide implementation issues from an user. Some argue that components should be binary entities specified by their interfaces [Szy02], while other claim that components is just an abstraction of an implementation that enables component based engineering of software, i.e. no matter if the components are *black boxes*, *gray boxes* or *white boxes* [WN01]. The component model proposed in this thesis belongs to the class of white boxes, i.e. we wish to develop our systems in a component engineering fashion, but still have control of the implementation.

A *component model* specifies the syntax and the semantics with which components are specified. It defines the type of interfaces available and how interfaces are specified. A language that specify interfaces are referred to as an *Interface Description Language* (IDL). Both CORBA and COM have their own IDLs. In Section 4.4 is our component model presented.

A *component technology* provides the infrastructure in which components can be deployed. For instance, CORBA components require the *Object Request Broker* (ORB), in order to execute.

4.2 Related work

In this thesis we use the concepts and vocabulary introduced in [HMSW01]. They have defined Prediction-Enabled Component Technology (PECT), which add analytical information to a component model. A typical examples of such an analytical information is WCET. However, the work is focused on how to establish confidence in the analytical information rather than on finding relevant analytical properties. They have applied PECT on a component model called COMTEK which is a new name of the component model previously called WaterBeans developed at Software Engineering Institute [PSW99].

Fioukov et al. have presented work where they apply the concept of analytical interfaces to the Koala component model in order to achieve predictable assemblies [FEHC02]. Their work is focusing on predicting the consumption of static memory.

There exist several component models for real-time systems. We have studied three existing component models for embedded systems: *port-base objects* [SVK97], *IEC 61131* [IEC95], and *Koala* [vOvdLKM00]. None of these models support early timing analysis and the support for explicitly describing the temporal behavior in the architecture is limited. Furthermore, neither port-based objects nor IEC 61131 are developed with the objective to support the product line architecture approach. However, all three models provide good support for structural reuse without considering the real-time behavior.

The Port-based object approach (PBO) was developed at the Advanced Manipulators Laboratory at Carnegie Mellon University. The model is based upon the development of domain-specific components that maximizes usability, flexibility and predictable temporal behavior. Independent tasks are the bases for the PBO model. Independent tasks are not allowed to communicate with other components, and thus components are loosely coupled and are consequently, at least in theory, easy to reuse. Although a system consisting of only independent components does not exist, minimization of synchronization and communication among components is a desired design goal. The data flow is specified through in- and out-ports. Whenever a PBO needs data for its computation, it reads the most recent information from its in-ports without knowing about the producer of that data. When a PBO component wants to make information available for other components in a system, it store data on its out-ports. In order to make PBO components more flexible and reusable a parameterization interface is provided. Through a parameterization interface several different application specific behaviors can be implemented by a single component. Besides the data interface and the parameterization interface discussed above, each PBO has an I/O interface which are defined in terms of *resource ports*. Resource ports are connected to sensors and actuators via I/O device drivers, which are not PBOs. Hence, the details of accessing sensors and actuators are encapsulated by a PBO. In the model presented in this thesis we make no difference between components that access external hardware, via device drivers, and ordinary components that interacts with other components. In Figure 4.1, a PBO is depicted.

IEC 61131 is standard for programmable control systems and a set of associated tools e.g., debuggers, test tools, programming languages. The part of IEC 61131 related to our work is concerned with the programming language and is referred to as IEC 61131-3. IEC 61131-3 structures an application hierarchically and provides mechanisms for executing an ap-

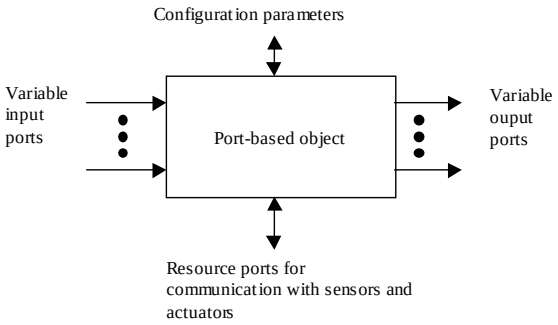

Figure 4.1: Port-based objects

plication and for communication between components. Components are implemented in any of the languages proposed in the standard: *instruction lists*, *assembly language*, *structured text*, *ladder diagrams*, or *function block diagrams* (FBDs). Structured text is a high-level language similar to Pascal, while ladder diagrams and function blocks are graphical programming languages. The most relevant and widely used language is FBDs. The data flow is specified in IEC 61131-3 function blocks by connecting in-ports and out-ports- Out-ports contain the result from a computation based on the input and the current state of the function block. Real-time tasks can be associated with a function block. Tasks can be either periodic or event-driven.

The structure of the IEC 61131-3 model is shown in Figure 4.2.

Koala is a component model that has been developed by Philips Research Laboratory for use in embedded consumer electronic products such as TV sets, DVDs, and VCRs. The Koala framework provides an ADL, an IDL, and a component description language (CDL). The ADL makes the architecture explicit when configuring components into products (see Figure 4.3). The IDL describes the interfaces while the CDL have constructions for describing the interfaces a component requires as well as provides. Koala also supports composed components, i.e. components that that encapsulate other components. Concurrency and execution of Koala components is implemented as a *pump*, which in essence is a queue of messages and a function that processes the messages. A *pump engine*, which corresponds to a real-time task manages a set of pumps and calls the appropriate pump function whenever there is a message in the queue

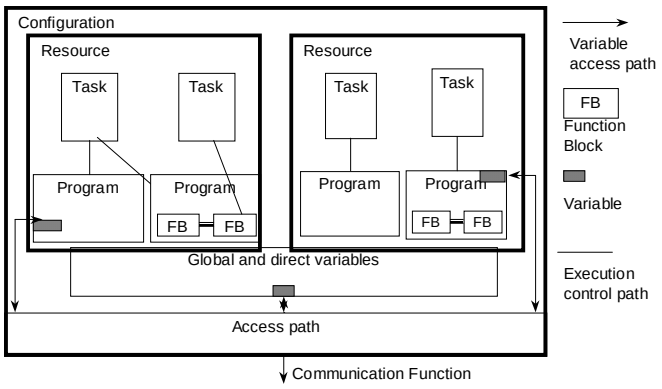

Figure 4.2: IEC 61131-3 function blocks

for one of the pumps. However, the temporal behavior of components is not explicitly expressed in the framework.

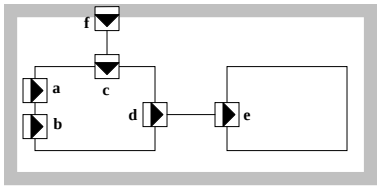

Figure 4.3: An example of the Koala component model

None of the component models above completely provide the characteristics that we require from our component model. They are especially weak in the specification of temporal behavior, e.g. period times, deadlines, execution times. Those attributes must be explicitly specified in a component model since they are required in order to analyze a system's temporal behavior, i.e. verifying that the temporal requirements are fulfilled. Moreover, in order for a component model to be suitable for a product line approach we must be able to specify- and provide mechanisms for variations. The variation points should also be visible in the architectural description as discussed in Section 3.3. This is explicitly dealt with in the Koala framework, but not in PBO, nor in IEC 61131.

4.3 Components, analytical interfaces and component assemblies

Applying the concept of product line architectures requires architectures and software components that are flexible, i.e. variations in features yet analyzable. Variations can, as discussed in Chapter 3, be obtained in different ways, e.g. applying variations in a flexible software architecture, parameterization of existing components, by using different implementations of components. In a product line architecture it is more likely that the software architecture is a constant, while flexibility is achieved through component variations. New functional, and non-functional requirements will be implemented by adding new components or by using different variants of existing components.

The flexibility is not only specified in the functional domain. Also non-functional properties may be subject for variability. For instance, in the real-time systems domain we are interested in the temporal behavior of a system as it is considered correct only if it performs correct function at correct time, i.e. temporal correctness. Consequently, by adding the temporal domain we must not only manage functional flexibility but also temporal flexibility. For instance, the frequency with which a particular component executes may vary between a high-end product and a low-end product due different demands from the controlled process.

One of the main problems in constructing and maintaining a product line architecture is to express and verify product properties derived from the properties of the individual components. To be able to predict the product properties from the component properties, we define a prediction-enabled component technology (PECT) similar to the one proposed in [HMSW01]. In a PECT there are both a constructive model and an analytical model. Examples of such analytical models on a component are different temporal attributes such as the frequencies with which a component executes and the version dependencies among components. While a constructive model deals with functional properties, analytical models describe non-functional properties (both operational and non-operational see Section 2.5). From the predictability point of view, obtaining new functional features of the products is straightforward as they come directly from the functional properties of components. On the opposite, the non-functional properties of products are hard to predict. For example, adding components with new functional features may affect the temporal correctness which in turn may degrade the quality of

services of a product. Moreover, a product line strategy can be focused on product families with the same functional properties, but different non-functional properties, e.g., scalability, flexibility, and safety. For this reason, the ability to derive non-functional properties from the properties of the components plays a significant role for product line architecture. Furthermore, as developing software products according to the product line architecture approach is based on reuse and repeatable processes, the findings and measurements from previously developed product versions can be taken as input to the method proposed in this chapter which may give more accurate predictions.

We present a component concept that provides means for performing an *impact analysis*. The aim of impact analysis is to predict the consequences of altering a system, i.e. we want to analyze the impact of a change, e.g. installing new features in a product, maintaining existing components, construct a completely new product based on reusable assets within the product line.

This is especially important in a product line architecture perspective where components and architectures are reused and customized for different products. Moreover, this type of systems have long lifetime and are therefore exposed to a large amount of maintenance. The analysis presented in this chapter is based on the concept of PECT, which is integrated into our component model developed for use in real-time product line architectures. We demonstrate the analytical models by an example showing how they can be used to derive properties of an assembly and analyze the impact of, e.g. adding new features to a product. However, the intention of this work is to provide a framework in which analytical properties can be added to the model such that many other interesting property of an assembly can be expressed and analyzed. In particular we illustrate our approach by presenting how two different non-functional properties, temporal correctness, and version consistent, can be analyzed.

4.3.1 Components and Assemblies

An assembly is a specific configuration of a set of components that also defines the components interconnections. The union of all its component's states gives the state of an assembly. Formally we define an assembly as:

Definition 1. *An assembly A is a tuple $\langle C(A), R^* \rangle$, where C is the set of all components in the product line, $C(A) \subseteq C$ is the set of components in A, and R^* is the set of relations between components in $C(A)$.* □

Note that an assembly does not necessary correspond to a product. While in some cases we are interested in properties of the product, in some cases we may want to analyze properties of a sub-part of the complete product. In both cases we will refer to an assembly. An assembly is only a conceptual- and analytical view of a complete product that exists for the analysis of a particular property, and has not necessarily a constructive correspondence. A typical example of a relation valid between components in an assembly is data connection which will be further discussed in Section 4.4.

4.4 ReFlex: A flexible real-time component model

In this section we present our component model (ReFlex), that facilitates a component based development of embedded real-time systems, using the product line architecture concept. Such a component model constitutes the core entity in the architectural description language that facilitates the management and generation of component based real-time system. However, no syntax is presented. The graphical syntax used in this chapter is for the clarification of concepts only. The ReFlex model is shown in Figure 4.4. Ports, services, parameters, and tasks will be explained and formally defined in this chapter.

The idea behind product line architectures is to create a generic architecture that can be tailored for different members of a product family [DKO+96] [Bos99]. The tailoring can be achieved by parameterization of generic components or by providing specific implementation of selected parts. Components that must be specifically implemented for each product will be referred to as abstract components. To use this concept for non real-time systems is a challenge, and an even bigger challenge is to employ this concept for embedded real-time systems that most often are implemented on hardware that is very limiting in terms of memory- and computational resources. Thus, the component technologies provided by industrial state-of-the-art, e.g. CORBA, Jini, DCOM, cannot be used.

To be able to adopt the product-line architecture approach in real-time systems requires that the component model supports the specification of real-time attributes. Furthermore, the component model must also support specification of temporal attributes both on concrete and abstract components. The reason for requiring support for specification of timing constraints on abstract components is to facilitate early analysis

of the temporal behavior. Detecting design flaws early in the development and especially timing errors is important to avoid costly re-design in late phases of a project. A component model with a precise semantic does not only support analysis at different stages, it also enables the development of tools that can generate code automatically.

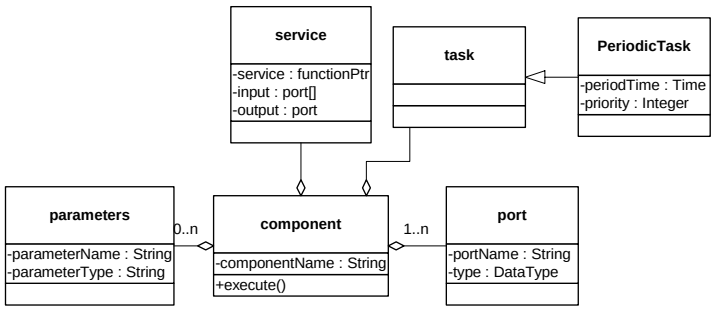

Figure 4.4: The constructive component model

ReFlex is a continuation of the component model developed for the Rubus RTOS [EMTP+96]. Rubus is a hybrid operating system in the sense that it supports both preemptive static scheduling and fixed priority scheduling (FPS). The statically scheduled part is concerned with hard real-time requirements and is referred to as the *Red part*, whereas the FPS part deals with soft real-time requirements and is referred to as the *Blue part*. The component model in Rubus was defined for the red part and is based on the same concepts as PBO, i.e. unbuffered in-ports and out-ports. Each component has one *entry function* that executes when the component is scheduled. The entry function base its execution on the values of the in-ports, and produces data on the components out-ports. The temporal attributes available in this component model is period time, release time, and worst-case execution time. The worst-case execution time is a static attribute that specifies, or constrains the time it takes to execute the entry function in the worst case. Deadline requirements are also specified for every component, i.e. they define constraints on the time between start and completion of the component's execu-

tion. Finally, the Rubus component allow for specification of precedence relations and mutual exclusion, i.e. the order of execution among tasks that control components, and synchronization among components. Given the temporal attributes, the precedence relations and synchronization requirements a tool generates a valid schedule in which every component fulfills its deadline requirement, if such a schedule exists. This component model has been successfully used in several commercial complex systems [NSG+01]. In Section 4.5.5, we will describe one of those systems in more details.

Both PBO and FBD can be considered as special cases of ReFlex. By parameterization and task assignment our component model can express the very same properties. Thus, it is more general and expressive. We also introduce the concept of abstract components and parameterization interfaces as a way to specify variability. Furthermore, more complex temporal requirements can be specified in our model.

We consider a component as a description of an encapsulation of services, defined in terms of its interfaces and its services. A service provides the logic that describes a function that is provided by a component. Encapsulated services can be implemented in any ordinary programming language, whereas the component is implemented in a specific component description language. Components can be hierarchically composed. Consequently, a component may encapsulate other components, sub-components. We will refer to such a construction as an aggregation, which is the very same terminology used in DCOM.

A component in our framework can reside in one out of three different states, *abstract component*, *concrete component*, and *component instance*. The different states are depicted in Figure 4.5. Concrete components and abstract components are both descriptions of encapsulations. However, an abstract component has an interface but no implementation of the behavior. The reason for having abstract components is to enable specification of components whose behavior must be tailored when reused across different applications. However, their interfaces are fixed.

A system is then generated according to the components and their interconnections. When generating a component instance, the component is dissolved into ordinary tasks and entry functions that can execute in any real-time operating system that supports the task model of the component. Thus, no special component infrastructure is required. Tasks are specified in the *control interface* of a ReFlex component (see Definition 10), whereas the entry functions correspond to the services in a ReFlex component.

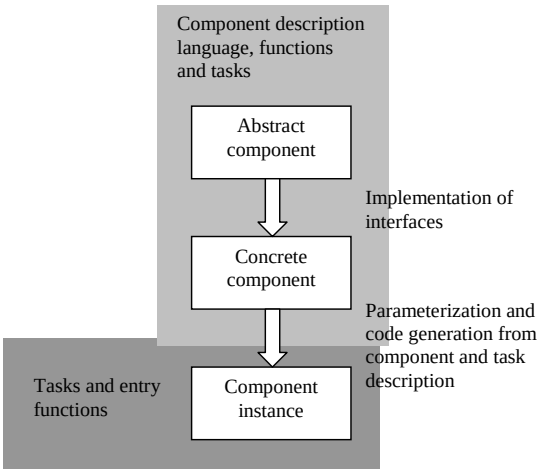

Figure 4.5: The component's states

A task defines the temporal constraints under which one, or several components executes. Concrete components together with tasks and parameterization defines the behavior, both functional and temporal, of a concrete component.

4.4.1 The component model

In this section we will describe our component model called ReFlex which is suitable for embedded real-time systems. ReFlex is based on the port-based object concept. Ports constitute the data interface for components as they define what data the component expects and the data it produces. However, port-based objects exhibit an overwrite semantics while our ports also can have buffered semantics. Besides having data interfaces, components in our framework have two additional interfaces, control interface, and parameterization interface. The execution of, and synchronization among services in a component is controlled through its control interface. The parameterization interface defines the points of variation of a component's behavior.

Moreover, components can be hierarchically composed, thus a component may encapsulate one or several other components. Furthermore, components can be either concrete or abstract. A concrete component, in its smallest constituent, encapsulates a service or other concrete components. An abstract component on the other hand, exists as a design

entity only. The abstract component indicates that when the component is reused it must be rewritten, i.e. tailored for its new context.

Definition 2. *A component c is a tuple* $\langle$ *I(c), O(c), C(c), P(c), F(c), A(c),* s_c $\rangle$*, where* $I(c)$ *is the set of in-ports,* $O(c)$ *is the set of out-ports,* $C(c)$ *is the control interface,* $P(c)$ *is the a parameterization interface,* $F(c) = f_1, ..., f_n$ *is the set of services encapsulated by c,* $A(c) = c_1, ..., c_m$ *is the set of aggregated components encapsulated by c, and* s_c *is the aggregated state.* □

A component's state is a persistent property that only can be changed by the services in a component. The state s_c of component c, is the recursive composition of all aggregated components states. Hence, $s_c = s_x \times S(A(c)$, where s_c is the composed state of component c, s_x is the state contribution from c and S(A(c)) is the set containing the states for all aggregated components. Typically, a state is defined by the internal variables whose values are kept intact between subsequent executions. The internal variable may change value due to being manipulated by the services encapsulated by the component.

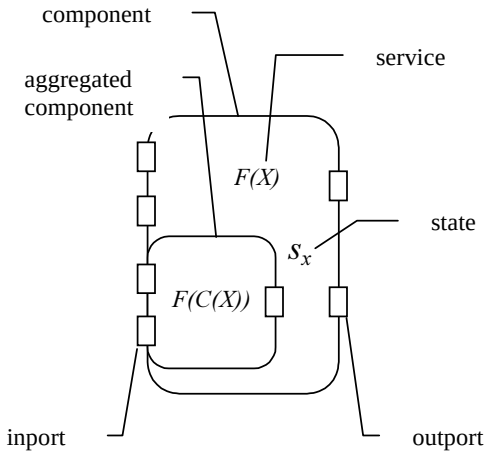

Figure 4.6: The component structure

Data interface

The data interface defines the input to, and the output from a component. We refer to input and output as ports. The data interface can consist of several such ports. Ports can exhibit two different semantics, overwriting semantic and buffering semantics. When overwriting semantics is specified, data consumers with a frequency lower than the producers might miss some data provided by the producer. On the contrary, if buffered semantics is specified data can be consumed in the pace of the consumer as long as the buffer is sufficiently large. Syntactically, the data interface specifies all ports in, and out from a component, each ports semantics, and the mapping from each port to the services in the component, or aggregated component, which require them.

Besides its semantics, a port is defined by its data type. A data type is defined by its name and the number of bytes it requires.

Definition 3. *A data type t is a pair $\langle name, size \rangle$, where name is the name of the type , and size $\in \mathcal{N}$ is the size of the data type in bytes* □

For instance, in the C programming language, characters are defined by the name *char* which is represented with one byte, i.e. $\langle char, 1 \rangle$.

Now we are ready to formally define a port:

Definition 4. *A port p is a pair $\langle$ buffer, type $\rangle$, where buffer $\in \mathcal{N}$, and type is a data type.* □

For instance, an integer port (4 bytes), with overwriting semantics is $\langle$ 1, $\langle$ int, 4 $\rangle$ $\rangle$, a real valued port with buffer size five is $\langle$ 5, $\langle$ real, 4 $\rangle$ $\rangle$.

A port acts as input to, and output from services encapsulated by components. Consequently, the value on an out-port is decided by the value of the in-port, the service associated with those in-ports and the state of the component. Consequently, for component c the out-port *out* can be described as: $out = f(in_1, ..., in_n, s_c)$. Formally we define a data interface for a component as:

Definition 5. *A data interface for component c, is a set of in-ports $I(c)$ and a set of out-ports, $O(c)$. Each service $f \in F(c) \bigcup F(A(c))$ is a function $in_n \times \ldots \times in_m \times s_c \rightarrow out_i \times s'_c$, where $in_n, \ldots, in_m \in I(c), out_i \in O(c)$, s_c is the state of component c before executing f and s'_c is the updated state.* □

A service can only acts as a producer of data for one out-port in a component instance. However, it is possible to specify more than on

producer for an out-port in a component. This is one of the possible ways in which we can specify variation points. However, the set of services specified as data producers to the same out-port are mutually excluding each other in the component instance. Component instances is further elaborated on in Section 4.4.2.

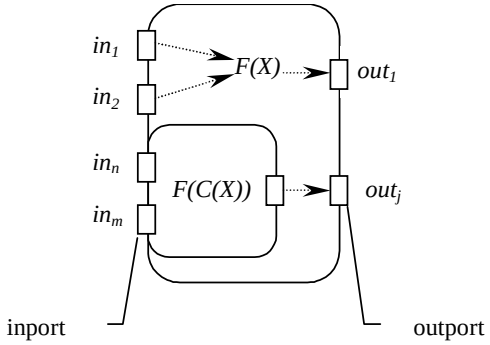

Figure 4.7: The component's data interface

Data ports interconnect components, i.e. they define the data flow through a set of components that constitute an assembly. We allow data connections from one out-port to several in-ports, but not the other way around, i.e. many out-ports to one in-port.

We define the data-flow relation as:

Definition 6. *A data flow connection, =, is a binary, anti-symmetric relation among ports on components, such that if* $c_i.i_x = c_j.o_y$ *then* c_i*'s in-port* i_x *is connected to* c_j*'s out-port* o_y*.* □

In Figure 4.8 is two components depicted that are interconnected through data ports. A's out-port, *out*, is connected to B's in-port *in*. Hence, B.in = A.out .

Control interface

A components control interface specifies the restrictions under which itself and its aggregated components execute. The control interface defines the execution in terms of their temporal behavior.

In order to control the execution of components and aggregated components we assign tasks to them. Tasks define the temporal attributes

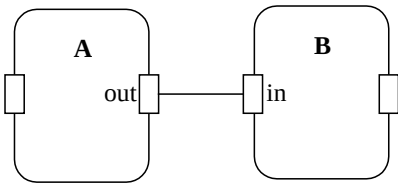

Figure 4.8: The data flow connection between two components.

that control the execution of components. Depending on the scheduling strategy, the actual attributes may vary. For instance, if the task is event-driven, no period time is specified. As components are "independent" from tasks, any scheduling strategy can be applied on a component. Thus, tasks that control the execution of components can be of any type, i.e. periodic, sporadic or aperiodic. A task can be associated with one or several components.

The services encapsulated by a component execute in a non-deterministic order but under the restrictions implied by the task connected to the component. However, if it is required that services in a component executes differently in the temporal domain, they can be encapsulated in an aggregated component which in turn can be assigned its own task.

We refer to the set of tasks in an assembly as a *task set* which defines the temporal view of an assembly:

Definition 7. *Taskset(A) is a tuple* $\langle T, R^*_{tasks} \rangle$*, where T is the set of tasks assigned to the components in the assembly A and* R^*_{tasks} *is the relations between tasks in T* □

As suggested above, executing constituents of a component, or components in an assembly in any order might not be sufficient. In order to specify the exact execution order, precedence relations are required. Moreover in order to restrict access to shared resources mutual exclusion relations among tasks are desirable.

A precedence relation defines the order in which two tasks may execute components. We say that if task τ_n precedes task τ_m then τ_m may start its execution earliest at the completion of τ_n. An example where such a constraint may be necessary is in a sample-control-actuate loop. The control component should only calculate new set-values after a new process-value has been sampled.

We have introduced the notion of component instances earlier in this chapter. We can also reason about *executional instances*. An executional

instance corresponds to an execution of a component enforced by a task. The first execution corresponds to the first executional instance, the second execution corresponds to the second executional instance, etc. Consequently executional instance n denotes the n^{th} execution of the task. When we say that a task τ_i precedes another task τ_j, we mean that τ_j's n^{th} instance may start earliest at the end of τ_i's n^{th} execution.

Formally we define a precedence relation as:

Definition 8. *A precedence relation, $\rightarrow$, is a binary, transitive relation among tasks such that if $\tau_i, \rightarrow \tau_j$, then τ_j may start its n^{th} execution earliest at the end of τ_i's n^{th} execution where $i \neq j$.* □

Note that two tasks that executes periodically must have the same period time in order to assure the precedence relation between them. If the execution of tasks are event-driven, the precedence relation must be maintained by other available mechanisms in the RTOS, e.g. signaling.

Another important synchronization constraint when designing concurrent real-time systems is *mutual exclusion*. Shared resources may be accessed by one component at the time only. Examples of such resources are global variables, databases, printers. We say that if τ_i mutually excludes τ_j, then τ_i is never allowed to execute while τ_j executes, and the other way around. Formally we define mutual exclusion as:

Definition 9. *A mutual exclusion relation, $\otimes$, is a binary, symmetric relation among tasks such that if $\tau_i \otimes \tau_j$, then τ_i is not permitted to execute while τ_j executes nor is τ_j permitted to execute while τ_i executes $i \neq j$.* □

The tasks are assigned to components through the control interface which controls the execution of the services in the component. Moreover, the control interfaces of the aggregated components is also part of the component's control interface. Formally we define a components control interface follows:

Definition 10. *A control interface, $C(c)$ for a component c, is $C(c) = Task \bigcup_{\forall c_i \in A(c)} C(c_i)$, where $Task = \emptyset$ if $F(c) = \emptyset$, else $Task = \{ \tau_c \}$.* □

So far, specification of the temporal behavior of components with tasks has been discussed. However, tasks do not specify the execution time of services in components. Thus the execution time is specified per service. As services are not always completely implemented, e.g. abstract components, the execution time may specify a budget that must

be adhered to by the service when implemented. The execution times are required when analyzing the temporal correctness of a software construction. We will discuss how to incorporate such analytical properties, such as execution times, into our component model later on in this chapter.

Parameterization interface

The parameterization interface defines the points where the behavior of an implemented component can be varied between uses. Such a point is referred to as a *variation point*. In Section 4.4.1 the control interface that provides variability through the task independence was discussed, i.e. the temporal constraints under which a component executes can be varied. Through a components parameterization interface, behavior and structure of a component can be varied. The constituents in a component instance, i.e. services and aggregated components that are present in the actual component instance, define a component's structure. The behavior of a component is defined by the parameterization of each aggregated component and service that is part of the components structure. As inports, discussed in Section 4.4.1, determines the dynamic behavior of a service in terms of the calculated result, the behavioral parameterization statically specifies the behavior.

As an example, consider a navigation component for an autonomous vehicle. Depending on the type of sensor, e.g. infrared sensor, bump sensor, or radio, the algorithm for calculating and presenting sensor values to the rest of the system will vary quite radically.

We have made a distinction between *behavioral parameterization* and *structural parameterization*. Behavioral parameterization is concerned with the behavior of services embedded by a component. Consequently, a behavior parameter decides the exact behavior of a parameterized service.

Definition 11. *A behavioral parameterization interface for component c, $P_b(c)$ is a set of tuples $\langle f, p_b \rangle$, where f is a service, and p_b is the set of parameters specifying the behavior of f.* □

The output, *out*, produced by a service f is, consequently, not only a function of the service's input *in* and the component's state s_c, but also the behavioral parameters $p_0, ..., p_n$ that controls the behavior of f.

$$out = f(in, s_c, p_0, ..., p_n) \tag{4.1}$$

Besides controlling the actual behavior of a service, we can control the structure of a component, i.e. which services and components should be present in a component instance. We refer to this parameterization as structural parameterization. This is similar to the *switch concept* in the Koala component model. A structural parameter specify which services and aggregated components that should be present in a component instance if there are optional services and aggregated components or *competing* services specified in the component. By competing we mean services that are specified as having the same out-port. We specify the exact structure of a component by associating a binary value with every optional service and aggregated component, as well as with every competing service. If the binary value is true the entity it is associated with will be present in the component instance.

Definition 12. *A structural parameterization interface for component c, $P_s(c)$ is a set of tuples $\langle x, p \rangle$, where x is a service or aggregated component, and p is binary true or false.* □

By using Definition 11 and 12, we can now define the complete parameterization interface for a component:

Definition 13. *A parameterization interface for component c, $P(c) = P_b(c) \cup P_s(c)$, where $P_b(c)$ is c's behavioral parameterization interface and $P_s(c)$ is c's structural parameterization interface.* □

4.4.2 Assemblies and component instances

Components are distinguished from instances of components. An instance of a component is a function of a concrete component, its parameters and task assignment. Consequently, component instances consist of fully implemented entities. Moreover, more than one service or aggregated component can be specified as data producer for the same out-port in a component. In component instances such conflicts are resolved through structural parameterization. Basically, every producer of data to the same out-port mutually excludes each other.

Definition 14. *An instance of a concrete component c, Instance(c), is a concrete component with parameterization and task assignments* □

An assembly consists of a set of component instances and their interconnections. The interconnection we have defined is data flow. An assembly describes the software architecture that implements functional-

and quality requirements of an application. Data flow is specified by connecting in-ports and out-ports in a consistent manner. By consistent is meant that all present in-ports are provided a data producer, i.e. an out-port. Just as for services that produces data to an out-port, there can only be one out-port connected to an in-port in an assembly. Rule 4.2 below formally specify that restriction for an assembly of components:

$$\forall c_i, c_j \in C(A) : c_i.i_x = c_j.o_y \Rightarrow \neg\exists c_k \in C(A) : c_i.i_x = c_k.o_z \qquad (4.2)$$

Synchronization among components is specified in the temporal view provided by the task set in an assembly. This is also a type of interconnections in an assembly but between the tasks in the task set.

4.4.3 ReFlex: An example

Consider a fuel level feature. In the high-end product low fuel level is indicated on a graphical display, while in the low-end product this is indicated with a lamp. We identify the need for implementing three components for this feature a sampling component, a filter component that pre-process the process value, and an actuator component that have two different functional behaviors, i.e. displaying low fuel level on a graphical display, and displaying low fuel level with a lamp.

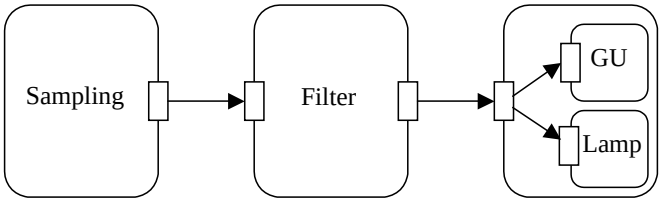

Figure 4.9: The components in the fuel level feature

The components communicate data through ports, which are defined by configuration language constructs. This allows for separation of communication from the implementation code. As presented in Section 4.4.1 we define a port by its data type and its semantics, i.e. buffered or overwriting. Below the specification for the ports in our fuel level feature is given. The TYPE defines the name of the data type and SIZE defines the size of the data type in bytes

```
PORT fuel {
  TYPE = FUELMEASURE;
  SIZE = 2;
SEMANTICS = OVERWRITING;
}

PORT filteredFuel {
  TYPE = FUELFILTERED;
  SIZE = 2;
SEMANTICS = OVERWRITING;
}
```

The data interfaces for out components is defined as:

```
DATAINTERFACE IFuelSampling {
  fuel outPort;

  outPort aFunction()
}

DATAINTERFACE IFuleFilter {
  fuel inPort
  filteredFuel outPort

  outPort anotherFunction(inPort)
}

DATAINTERFACE IFuleActuate {
  filteredFuel inPort

  yetAnotherFunction(inPort)
}
```

Tasks define the temporal behavior of the components. In our implementation of the feature we use three tasks, one for each component. The sampling component and the filter component is periodically triggered, while the actuator is triggered by the task that controls the filter component via a synchronization signal. This ensures that the precedence relation between the filter component and the actuator is maintained.

```
TASK fuelSamplingTask
{
  TRIGGER = PERIODIC
  PERIODTIME = 2000
  PREEMPTION = DISABLED;
}

TASK fuelFilterTask
{
  TRIGGER = PERIODIC
  PERIODTIME = 2000
  PRECEDES =
  SENDSIGNAL fuelActuatorTask
  PREEMPTION = ENABLED;
}

TASK fuelActuatorTask
{
  TRIGGER = SIGNAL
  RECSIGNAL = fuelFilterSignal
  PREEMPTION = ENABLED;
}
```

Finally, we specify the components and the parameterization interfaces.

```
COMPONENT FuelSamplingComponent(task)
{
  IMPLEMENTS = IFuelSampling

  IFuelSampling::outPort aFunction()
  {
    // Do something
  }
}

COMPONENT FuelFilterComponent(task)
{
  IMPLEMENTS = IFuelFilter
```

```
  IFuelFilter::outPort anotherFunction(IFuelFilter::inPort)
  {
    // Do something
  }
}
```

The actuator component has two different behaviors depending on the product in which it is used. For this reason we have a parameter that controls the exact behavior.

```
COMPONENT FuelActuatorComponent(parameter, task)
{
  IMPLEMENTS = IFuelActuate

  yetAnoterFunction(IFuelActuate::inPort)
  {
    if (parameter == PRODUCT_HIGH)
      // Do something
    else if (parameter == PRODUCT_HIGH)
      // Do something else
  }
}
```

Now we can specify the assembly that implements the fuel level feature by connecting the data ports between the component instances. First we create component instances by assigning tasks and parameters where required.

```
FuelSampling = FuelSamplingComponent(fuelSamplingTask)
FuelFilter = FuelFilterComponent(fuelFilterTask)
FuelActuator =
        FuelActuatorComponent(PRODUCT_HIGH, FuelActuatorTask)

Next we connect the ports between the components.

FuelSampling::IFuelSampling::outPort ->
        FuelFilter::IFuelFilter::inPort
FuelFilter::IFuelFilter::outPort ->
        FuelActuator::IFuelActuate::inPort
```

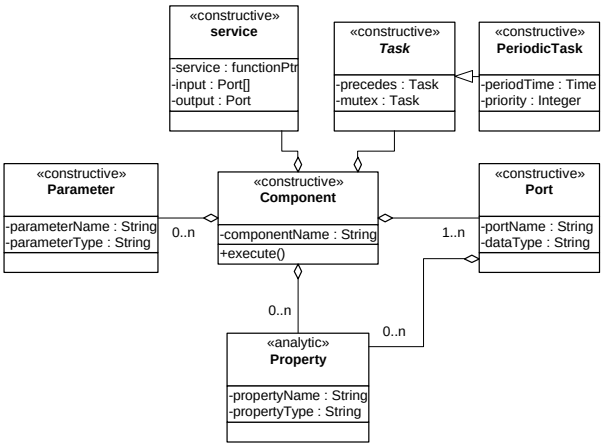

Figure 4.10: The constructive- and analytical model of a component

The assembly above implements the fuel level feature for the high end product. In order to create an assembly for the low end product we must make a different instance of the actuator component, i.e. another parameterization.

```
LowEndFuelActuator =
        FuelActuatorComponent(PRODUCT_LOW, FuelActuatorTask)
```

4.5 Analyzing assemblies

So far we have defined the constructive part of our component model, i.e. the mechanisms available for creating a component and interconnecting it with other components in an assembly. However, in order make analyses possible, e.g. the temporal behavior, we must add the information required by the analyses. We say that this information belongs to the *analytical model* of a component. The analytical model enables analyses of an assembly based on the information provided by each single component. In Figure 4.10 the complete component model is depicted, including both constructive and analytical parts. The difference from Figure 4.4 is that components and ports can be associated with *properties*. The analytical model of a component is defined by its analytical properties.

The property class that is stereotyped as analytic provides the information needed by the different analyses we are interested in performing on an assembly. We will refer to such a property as an analytical property. An analytical property usually does not have a correspondence in a component instance. A typical example of such a property would be the execution time of a service of a component. The execution time is derived from the source code, or by measurements, for the purpose of modeling and analysis of a system and has no correspondence as such in the execution at runtime. However, note that a task is a runtime mechanism and hence, it is a constructive part of a component. Still, some of the attributes of a task are required when, together with some analytical properties, e.g. execution times, analyzing temporal properties of an assembly. Properties on a port can be information about the data type, e.g. size.

4.5.1 Properties of an assembly

The intention of our work is to provide a framework in which new properties of an assembly could be taken into consideration and predicted for the purpose of analyzing the impact that the introduction of a new component in the system have. The general idea is that if the model has to be extended with a new predictable property, new analytic properties can be defined and new property theories be developed. The property theory defines how a particular property of an assembly is calculated, e.g. theories for verifying the temporal correctness. For instance, if we require an assembly to be type correct, i.e. the types of connected data ports are correct, we must add a method for checking this property and doing so requires an analytical property on data ports which carries the type information. Furthermore, we are using the prediction technologies in a product line perspective, i.e. we will discuss properties that are important when developing and maintaining product line architectures.

There are several realistic scenarios describing activities that a product line may undergo during its lifetime. We have not identified all possible scenarios but we will highlight some relevant cases and propose examples of properties that are interesting from those scenario's perspective.

Scenario 1: New features will eventually be added to a product line or a specific product within the product line. This new feature might be implemented by a set of new components as well as new versions of old components already existing as part of the reusable assets in the

product line. Doing this, there is a potential risk that components could end up being incompatible with components already used in the product, both with respect to version and variants. This scenario is also related to maintenance of a product that may alter the characteristics of a particular component. This change of characteristics is possibly acceptable for one particular product, but what are the consequences in the rest of the product line?

Scenario 2: As we operate in the real-time systems domain, we are also interested in predicting the temporal behavior of an assembly. Adding components to-, or changing components in a product or product line, may violate the temporal constraints in the system. The reason for violating the temporal constraints could be an over-utilization of the available resources in the system architecture. A big share of existing real-time systems are embedded systems, thus resources are usually limited.

Scenario 3: When an assembly of components is composed it is of importance to be able to predict if all component interactions are type correct. In a port based component model the components read the outputs from other components at the start of the execution. If the output type is not the same as the type of the input then we have a fault which can lead to a failure of the system. Hence we want to predict if an assembly is type correct before deploying it.

The scenarios discussed above also apply to the assembly of a new product, based on pre-existing reusable components. We have to make sure that the product is feasible both with respect to the functional behavior and the temporal behavior.

To illustrate predictability of assemblies for the specified component model, we shall discuss two concrete examples of an assembly's properties from a real-time product line point of view: consistent, and end-to-end deadlines. These properties are of completely different nature. Consistent is typically a property of a complete product. End-to-end deadline only concerns a subset of components in a complete product assembly. Moreover, there can be several end-to-end deadline requirements within the same assembly with respect to a subset of components from the full assembly.

4.5.2 The end-to-end temporal property

The temporal correctness is of vital importance in the real-time systems domain. Moreover, the temporal requirements on a real-time system are

seldom presented in terms of the temporal attributes provided by the RTOS or as simple deadlines for individual components. Typically they are considered on a higher level; for instance jitter constraints and latency for the control performance, end-to-end deadlines, response times, etc. Designing a real-time system is partly a matter of transforming such high-level temporal requirements to the attributes available in the task model at run-time, typically considering priorities and period times. In our approach the high-level temporal requirements are specified as properties on an assembly, e.g. end-to-end deadline, and the implementation of those requirements, e.g. period times, priorities, execution times, are specified as analytical properties of components and in tasks associated to components. A concrete example of a temporal property is end-to-end deadline. An end-to-end deadline, denoted as $A.e2e$, specifies a temporal requirement on a set of components. It defines the maximum distance between an input stimuli and the output response given. Typically, the end-to-end property requirements in hard real-time systems must be met, while in soft real-time systems a particular confidence of meeting the requirement may be sufficient. Statistical verification of an analytical properties can be performed to show how reliable the prediction actually is, e.g. the confidence in the estimated worst-case execution time.

Verifying that a temporal property of the assembly is feasible, we verify that our implementation is correct. However, this verification is correct under the assumption that all prerequisites are satisfied (For example, the execution time of a component, which is a component property). Consequently, the correctness of a property of an assembly depends on the confidence we have in analytical properties. The concept of credentials, as presented in [Sha96] includes a notion of confidence associated with a component property. The execution time can be statically analyzed given the source code, or empirically measured at runtime [LBJ+95]. The analytical method, e.g. the formula that calculates a property of an assembly, is referred to as a *prediction theory*. Empirical validation of the prediction theory is also needed to prove the soundness of the theory.

Figure 4.11 shows an example where four components have been instantiated from the model presented earlier in this chapter. The infrastructure in which those components will execute (the RTOS) has a scheduling policy based on fixed priorities. The task model consequently specifies the level of priority and the frequency of each task. When defining an assembly we also must specify how the assembly is build. There are not only the properties of the components that determine the properties

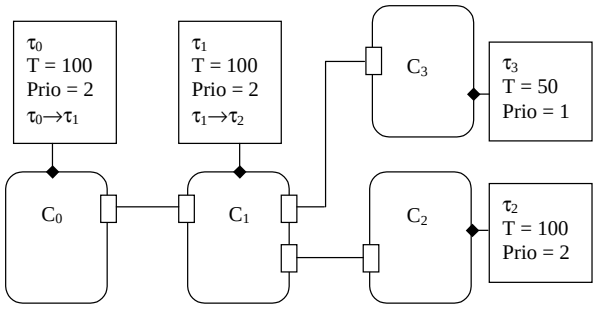

Figure 4.11: Four components with precedence and connection relations

of an assembly, but also the assembly architecture; we must define how the assembly is built. For example, in an architecture with a pipes-filter style the data flows between components, i.e. the precedence relations, must be specified. In this example we define the precedence property and ports connections. We also add an analytical property that specifies how many times components are supposed to be executed. Component c_1 has two preconditions, the first one express the precedence relation and the second the connection of ports.

Figure 4.11 shows four components where c_1 reads the out ports of c_0 and c_2, c_3 reads the out ports of c_1. The execution of component c_0 precedes the execution of component c_1 and the execution of component c_1 precedes the execution of component c_2, while c_3 can execute independently. Below is the components described according to Definition 2 on page 97:

$$c_0 = \langle f, P_0, \emptyset, \{o_1\}, f(\emptyset, \{o_1\}), \tau_0, s_0 \rangle$$
$$c_1 = \langle g, P_1, \{i_1\}, \{o_2, o_3\}, g(\{i_1\}, \{o_2, o_3\}), \tau_1, s_1 \rangle$$
$$c_2 = \langle h, P_2, \{i_2\}, \{o_4\}, h(\{i_2\}, \{o_4\}), \tau_2, s_2 \rangle$$
$$c_3 = \langle x, P_3, \{i_3\}, \{o_5\}, x(\{i_3\}, \{o_5\}), \tau_3, s_3 \rangle$$

In our example we have two views of the assembly, one for precedence among the tasks that control the components, and another that shows how the components are connected through ports. The assembly in our example according to Definition 1 on page 92 is:

$$A = \langle \{c_0, c_1, c_2, c_3\}, \{R_{Connection} = \{(o_1, i_1), (o_2, i_2), (o_3, i_3)\}\} \rangle$$

The data connection view of the assembly is the following:

$$A_{Connection} = \langle \{c_0, c_1, c_2, c_3\}, R_{Connection} \rangle$$

Moreover, the taskset for A is:

$$Taskset(A) = \langle \{\tau_0, \tau_1, \tau_2, \tau_2\}, \{R_{precedence} = \{\tau_0 \rightarrow \tau_1, \tau_1 \rightarrow \tau_2\}\} \rangle$$

A view of the task set is:

$$A_{Precedence} = \langle \{c_0, c_1, c_2, c_3\}, R_{Precedence} \rangle$$

We shell analyze a property of the assembly, namely *end-to-end deadline*, A.e2e. An end-to-end deadline in our framework defines the maximum distance between the earliest start time of a component c_{start} and the latest completion time, i.e. the response time, of a component c_{end}, where there is a transitive precedence relation between τ_{start} that control c_{start} and τ_{end} that controls c_{end}. There can, consequently, be any number of components participating in an end-to-end deadline requirement as long as their individual order of execution is defined through the tasks that control them. Since the execution order among the tasks in an end-to-end deadline requirement is explicitly defined, we can express this property as a function of the first component and the last component, taking all components in a the assembly into consideration. We have to consider the complete assembly as the execution of components that is not part of the end-to-end requirement may interfere with the execution of the components of interest. We have made a restriction in our interpretation of an end-to-end deadline as we require a transitive precedence relation between the first task and the last task. The reason for this is to keep the analytical calculations as simple as possible. After all, we intend to show how to specify properties of an assembly, not to solve the problem of verifying complex end-to-end deadlines.

We can define the property theory for our end-to-end property between a start component and an end component with a transitive precedence relation among the tasks that control them, A.e2e(c_{start},c_{end}) as:

$$A.e2e(c_{start}, c_{end}) = ResponseTime(c_{end}) - StartTime(c_{start}) \quad (4.3)$$

Both c_{start}, and c_{end} belongs to the assembly ($c_{start}, c_{end} \in C(A)$), ResponseTime($c_{end}$) is the maximum response time of c_{end},

StartTime(c_{start}) is the earliest start time of c_{start}, and τ_{start} that controls c_{start} precedes τ_{end} that controls c_{end} transitively.

In our example assembly we would like to verify an end-to-end deadline between c_0 and c_2, i.e. A.e2e(c_0, c_2). In the view $A_{precedence}$ of A we can see that τ_0 precedes τ_1, and τ_1 precedes τ_2. Hence, τ_0 precedes τ_2 transitively. From equation 4.3, we see that A.e2e(c_0, c_2) can be calculated as:

$$A.e2e(c_0, c_2) = ResponseTime(c_2) - StartTime(c_0) \tag{4.4}$$

Calculating the response time of components based on the attributes provided in a fixed-priority based RTOS is done with response time analysis
[ABR+93]. However, different methods must be utilized if a different scheduling policy is provided by the RTOS, e.g. earliest-deadline-first. Thus, the definition of a particular property of an assembly may vary due to mechanisms provided by the infrastructure in which the system will execute.

In our particular example we are using fixed priority scheduling in which we calculate the response time of component c_i controlled by task τ_i, R(c_i), as:

$$R^{n+1}(c_i) = c_i.wcet + B(\tau_i) + \sum_{\forall \tau_j \in hp(\tau_i)} \left\lceil \frac{R^n(c_i)}{\tau_j.T} \right\rceil c_j.wcet \tag{4.5}$$

,where B is the blocking time, hp(τ_i), is the set of task having higher priority than τ_i, $\tau_j.T$ is the period time of task τ_j, and $c_j.wcet$ is the worst-case execution time of component c_i.

The earliest-start time can also be calculated with the equation above by assuming that all components execute as fast as possible, i.e. with their *best-case execution time* (BCET). Furthermore, the start time will be approximately equal to the response time if we assume an execution time equal to zero of the component whose earliest start time is subject for the analysis.

The end-to-end property is a typical example of a property that may be defined on only part of a complete product. In Figure 4.11 it can be seen that τ_0, τ_1 and τ_2 are connected with the precedence relation but τ_3 can execute anytime when in the ready queue. It is of importance to be able to calculate the e2e property for c_0, c_1 and c_2 only. Our proposal is

that the property shall be defined for parts of the assembly with respect to a relation. In our example we can say that c_3 is independent from the other components with respect to precedence. Hence A.e2e over $\{c_0, c_1, c_2\}$ can be calculated with the response time of c_2. By having this notation it is possible to define properties that reflects parts of the assembly.

As discussed above, different task models will affect the set of analytical properties on components and how temporal properties of assemblies are calculated. Equation 6 shows how to calculate the response time for a system with periodic tasks and static priorities. However, if systems are event based and uses the earliest-deadline first scheduling algorithm new theories for verifying the temporal behavior are required. Thus, components, assemblies and the execution model affect the property theory. Hence, each of these has to be defined before we start reason about temporal properties of assemblies.

4.5.3 The version consistency property

In a product line approach the handling of consistency is a 2-dimensional problem. A component in a product line may be compatible with- or dependent on several different variants of other components. For instance, a GUI component for an embedded system could differ between products in a product line, e.g. high-end products with a color display and low-end products with monochrome displays. The color display and the monochrome displays are variants of the same feature, i.e. the feature of presenting information graphically to a user of the system. In turn, there can exist several versions of every variant of a component. Typically new versions emerge from error corrections and from new functionality being added.

A version of a component can be defined by having an analytic property on the component. Also dependencies between components are expressed through such a property. In our model we allow a component to depend on several different variants of a component but with only one distinct version of each variant.

The consistent property, *A.consistent*, is related to a capability to predict consistency of an assembly. An assembly is considered consistent if the versions of each component are correct according to the specification of a product in the product line. The specified features of a product determine which components, and in particular which components version should be included in a product. To be able to guarantee consistency

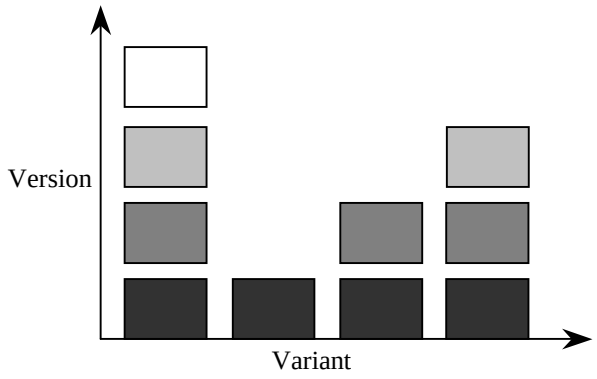

Figure 4.12: The 2-dimensional version-variant concept

we need to specify which versions of components a product depends on.

This idea of having version dependencies is very similar to how .NET assemblies use meta-data to describe dependencies to other assemblies [TL01]. Dependencies can be expressed and assured using OCL constraints for the components. A new constraint has been added to all components that state how the dependencies shall be evaluated and regarded analyzing the assembly.

For the purpose of predicting variant- and version consistency of an assembly, we must introduce the analytical property *depends on a component*, c.depends. The analytical property *c.depends* is a set containing all components and their variant and version, with which component c consistently can be assembled with. Each tuple is on the form $\langle C, variant, version \rangle$.

In many component models multiple versions of the same component may not coexist. In those cases there is a risk that components are assembled in an inconsistent way, by means of having the assembly include two or more different versions of the very same component. It is desired to prevent such invalid assemblies by being able to predict whether an assembly is consistent or not. The consistency of all variants and versions in an assembly can be calculated with the following formula. The property *consistent* is of type boolean.

An assembly A is variant- and version consistent, A.consistent if:

$$A.consistent = \forall\langle\langle c_i, variant, x\rangle, \langle c_i, variant, y\rangle\rangle \in V \times V : x = y \tag{4.6}$$

V is the union of all component's depends set described above in the assembly ($V = \bigcup_{\forall c_i \in C(A)} c_i.depends$), c_i is a component in the assembly ($c_i \in C(A)$), variant is a component's variant and x,y are versions.

That is, the assembly is consistent if a component does not appear twice with different version in the union set of all dependencies. Note that the property theory for consistency presented here only verifies that components are only assembled with compatible ones. To verify whether or not a valid product has been assembled is a different task which require information about what a valid product is in terms of components, versions, and variants.

4.5.4 Impact analysis

Before the new component is added we want to predict the impact it has to the system. For instance we want to calculate A.consistent and A.e2e over $\{c_0, c_1, c_2\}$ and $\{c_3, c_4\}$. We refer to such an analysis as impact analysis.

The e2e property, or any other temporal property of an assembly, may be affected by adding new components to a product. Assume, for instance, a fixed-priority scheduled system. The majority of the commercial available RTOS belong to this class. Moreover, assume that priorities are assigned to tasks according to the deadline-monotonic algorithm, i.e. the task with the shortest deadline is assigned the highest priority. Adding a component that has an unique deadline in such a system may require the rest of the system to undergo a new priority assignment, unless it has the latest deadline. Consequently, it is important to formalize the algorithm or strategy used for priority assignment as a property of an assembly. If such formalization does not exist, evolution and maintenance of the system may become expensive. Note that adding a component with lower priority than all existing components is no guarantee for a temporal correct system. Such a component can still affect the temporal correctness through, e.g. shared resources resulting in priority inversion.

In order to predict the need for reassigning priorities in a fixed-priority system we introduce a boolean property on an assembly expressing that the pre-existing priority assignment still will be valid after adding a new

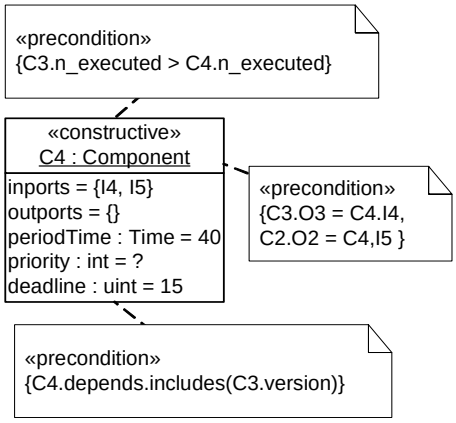

Figure 4.13: A new component c_4 is added to represent a new feature of a product.

component, *A.priority*. The theory for this property varies according to the strategy for assigning priorities; just as the theory A.e2e varies depending on the scheduling policy. Adding a component c_i will not affect the priority assignment according to deadline-monotonic if:

$$A.priority = \exists c_j \forall c_k \in C(A) : c_i.d = c_j.d \vee c_i.d < c_k.d \tag{4.7}$$

where $c_i.d$ and $c_j.d$ are the deadline for the task that controls the execution of component c_i and c_j respectively.

We illustrate the problem of adding a new component to a product line by continuing the example in Section 4.5.2. We introduce a new component c_4 which is dependent on the execution of c_3 and the output from c_3 and c_2. Such a component is presented in Figure 4.13. The component c_4 also expresses its version relation to other components. Component c_4 depends on a particular version of c_3. The dependencies are expressed using a precondition that asserts that the correct version of c_3 is in c_4's depends on set.

Applying the property theory for A.priority indicates that the priority assignment currently existing in the assembly must be revised as the new component has an unique deadline that is shorter than the deadline for component c_2. Note that the priority of a task can only be decided in relation to every other task in a system.

In a similar way as described in this chapter we can define, and apply, any other important property theory in order to analyze the impact of adding a new component to a system.

4.5.5 A successful constructive and component based developed system

The product line approach described in Section 3.4 is built using the Rubus component model presented in Section 4.4. The component model proposed in this thesis is a continuation of that model. The systems are also developed in a constructive fashion. Typically, a system in the VCE product line consists of approximately 80 components that have entry functions which execute between 10 μs and 1 ms. In the initial development of the system, or in the process of developing a new component, the temporal attributes and requirements are specified and the temporal behavior of the system, as well as synchronizations, is verified based on the information provided by the components. Moreover, an off-line schedule, i.e. a dispatch table, is constructed based on the components temporal attributes and requirements. Hence, the system can be temporally integrated and verified in an early stage even before any entry function has been implemented. As a result of adopting this approach, the time spent in the implementation phase and the integration phase was reduced both when it comes to initial development an maintenance activities such as adding new functionality. The method has been very successful. More about the findings from introducing this technology can be found in [NSG+00].

As described in Section 3.4, variations between different products was handled by datasets, i.e. parameterization. However, the parameterization was done by assigning constant values to in-ports. Consequently, there is no separation of data interfaces and configuration. The notion of composite components do not exist in the Rubus component model. Components can not hierarchically encapsulate other components. Components can only be encapsulated by a container which cannot have its own thread of execution.

4.6 A comparison of the component models

In this section the expressiveness of the proposed component model is compared to port-based objects, IEC 61131, and Koala in order to show that it is capable of specifying the same properties and in some cases,

show that the semantics is more expressive. The comparison is made based on the constructions for which we have specified a semantic, i.e. hierarchical composition, flexibility, temporal constraints, and synchronization. It unveils that the proposed model can express the same properties as both IEC 61131, the PBO model and Koala. However, our model is more expressive when it comes to specification of temporal attributes and synchronization. The notion of abstract components is also unique. When it comes to communication, IEC 61131, PBO, and Koala have some explicit constructions specified. In our model communication among components can be implicitly specified through, for instance, shared memory protected by a semaphore, i.e. mutual exclusion.

4.6.1 Hierarchical composition

Hierarchical composition of component is essential for building reusable components of convenient size. A hierarchical approach, i.e. the possibility to specify aggregated component, supports this by combining several smaller components with a unified and single interface to the rest of the system. Our model comprises this concept by aggregated components. In the IEC 61131 standard, a function block can be composed by several other function blocks. Thus, IEC 61131 also can express hierarchical composition. Koala also allows specification of hierarchical components called *compound components*, which corresponds to our aggregated components. Port-based objects on the other hand, have no such concept.

4.6.2 Specification of variation points

Beside the possibility of having components of suitable size, requirements on their behavior and characteristics may vary between uses in different products in a product-line. This variation is accomplished through the parameterization interface and the concept of abstract components in our model. Through the parameterization interface, the behavior can be varied without violating the encapsulating of the component, whereas abstract components specify the need for a possible specialized implementations in a reuse situation. The port-based object model also has a parameterization interface. But there is no equivalent to our abstract components. Thus, in cases where only a common interface can be specified in the product-line architecture, port-based objects will fail to do so. In IEC 61131, there is no means for specifying flexibility explicitly, although one may solve this by using ordinary input-data to a function

block as a constant that specify some variable property. Koala has several constructions that support diversity: parameterization through ordinary data ports which is referred to as diversity interfaces in the model, *optional interfaces*, and *switches*. An optional required interface need not to be connected, while an optional provided interface need not to be implemented. Basically, a switch decide which components, among several possible, that should be present in the deployed component. This corresponds to our structural parameterization.

4.6.3 Specification of temporal constraints

The temporal constraints on a real-time system is of vital importance since correctness of such systems is defined to include both functional- and temporal correctness. Furthermore, as many parameters as possible is desirable when tuning the temporal behavior since this will minimize the semantic gap between the high-level temporal requirements and the task model provided by a real-time operating system, i.e. the infrastructure. In other words, it is desirable to model tasks in a way that mimic the temporal attributes available in the infrastructure, e.g. period time, deadline, offset. If we consider component reuse across several different infrastructures we would like the model to support the union of the temporal attributes provided by those infrastructures.

ReFlex components are completely independent from the task models; there is only a relation between tasks and component or services in order to specify the temporal constraints under which it must execute. IEC 61131 have a similar approach where the task is separeted from the component. However, the temporal attributes are predifined and restricted to a very small number and they are quite simple which restricts the choice of infrastructure. Typically they specify a period time and priorities only. The port-based object model is equally weak on the ability to express temporal constraints. But there are temporal attributes, i.e. the period time in case of periodic execution, specified in the actual components. Thus, it is hard to use this model in an infrastructure that differs from the one intended for the component. In the Koala framework is concurrency and execution of components implemented as a *pumps*, which in essence is a queue of messages and a function that processes the messages. A *pump engine*, which corresponds to a real-time task manages a set of pumps and calls the appropriate pump function whenever there is a message in the queue for one of the pumps. However, the temporal behavior of components is not explicitly expressed in the framework.

Generally, both IEC 61131 and port-based objects have quite a tight coupling to a specific infrastructure, whereas our proposed model and Koala makes very few assumptions about the environment in which it will execute.

4.6.4 Specification of synchronization

Synchronization is an essential part of implementing the temporal requirements of a real-time system. In our model mutual exclusion between components and services can be specified. Moreover, precedence relations specify and control the order in which components are executed. In IEC 61131, there is a semaphore concept that can implement a mutual exclusion relation, but there is no equivalence to the precedence concept. In the port-based object model, the concept of synchronization among components is not defined. This is a major shortcoming of the models when they are used in large and complex systems. The pump-pump engine concept in Koala acts as dispatchers for events. Hence, synchronization among components operating on the same pump is not needed. However, if "inter-pump" communication is required, synchronization is to be solved by the tasks, which as previously discussed, have no correspondence in the component model or the ADL.

4.6.5 Predictable assemblies

The work presented in [HMSW01] is more focused on how to establish confidence in the analytical interfaces, whereas we have been focusing on finding analytical interfaces on components suitable for a product line approach. Many other relevant properties of a component assembly exists, e.g. memory consumption, bandwidth utilization. By identifying the information necessary in the analytical interfaces, and by defining correct property theories, such properties would easily integrate in our work.

4.7 Discussion

Another application of the ideas presented in this chapter is as a method for handling dynamically configurable systems. Consumer-products such as cellular phones may be configured/customized by the consumer himself, e.g. by downloading a new feature to the phone. Thus, the end-customer assembles products based on a product line architecture. By

distributing the analytical model together with the constructive software, the system can itself predict the impact the new feature will have on the system. Based on such an analysis the system can decide whether to accept the new product as valid or not.

Chapter 5

Probabilistic modeling and analysis

In this chapter we describe our probabilistic modeling and analysis framework (ART-FW). The simulations are based on analytical models of the system made in our probabilistic modeling language ART-ML (Architecture and Real Time behavior Modeling Language). By using simulations, we can define other correctness criterion than satisfying deadlines as mentioned in Section 2.5.2. Moreover, instead of always assuming worst-case scenarios, we can use execution time distributions. ART-ML also permits the behavior of tasks to be modeled, i.e. on a lower level than the software architecture. This permits a more precise model to be created as semantic relations among components can be introduced. Furthermore, we introduce a requirements language called PPL in which we can express statistical requirements which are verified given the simulation results. Hence, we have the possibility to feed more information back from the analysis than just *schedulable* or *not schedulable*.

5.1 Introduction

Large and complex distributed real-time computer systems usually evolve during a long period of time. The evolution includes maintenance and increasing the system's functionality by adding new features. Eventually, if ever existed, the temporal model of the system will become inconsistent with the current implementation. Thus, the possibilities to analyze the effect of adding new features with respect to the temporal behavior will be lost. For small systems this may not be that a big problem, but for large

and complex systems the consequences of altering the implementation cannot be foreseen. Introduce, or re-introduce, analyzability is the task of re-engineer the system and construct an analytical temporal model of it. We refer to this approach as providing analyzability by "re-engineering", in opposite to the constructive approach described in Chapter 4.

Some of the systems that we have studied is of the nature that the result from a FPA would be negative, i.e. by assuming worst-case scenarios, the system will not be considered temporal correct by the analysis in terms of meeting all its deadlines. Furthermore, a task may execute components with great variations in execution times sporadically. To be safe in FPA, the periodicity of sporadic tasks is modeled as having a frequency equal to the minimum inter-arrival time. Thus, more pessimism is possibly introduced.

FPA assumes a task model where deadline requirements are assigned to every task. In one of the systems we have investigated the temporal correctness is defined in terms of other criteria. Some of the tasks can have their deadlines derived from these criteria, but not all tasks can easily be assigned a deadline. An example of another correctness criterion is a message queue that must never be empty (starvation).

Since traditional temporal models and analysis do not apply to all complex real-time systems that we have studied, we have developed a simulation-based analysis approach.

The tool suit (ART-FW), in which the simulator is a part, also includes tools for measuring an existing system implementation, as well as tools for processing measurements and analyzing the results generated by the simulator. The analysis is based on probabilistic properties. Temporal requirement are specified in a query language, the *probabilistic requirement property language*. The result of such a query is the probability of complying with a temporal requirement.

The introduction of an analyzable model of a system brings a continuous activity of maintaining the model. The model has to be consistent with the current implementation of the system, i.e. the implementation should be a true refinement of the model. Consequently, our method must be an integrated part of a company's development process. An alternative strategy is to, once that the temporal analytical model of a system has been re-engineered, transfer to a component model that support a constructive approach, e.g. the model proposed in Chapter 4. This is, however, not always possible since such a transfer is associated with costs and risks that cannot be tolerated in an industrial perspective.

Figure 5.1 depicts the general activities required when creating and

maintaining an analytical model. Note that the process described here only concerns the method we are proposing. Important activities such as verification and validation of the implementation are omitted.

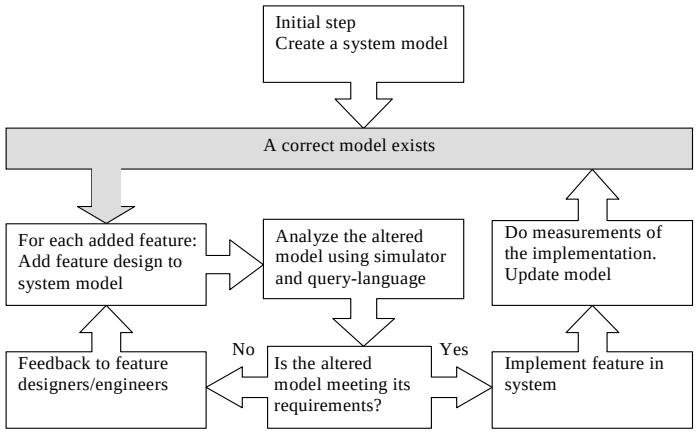

Figure 5.1: The process of constructing and maintaining an analyzable system.

The details of the process depicted in Figure 5.1 will be discussed in Section 5.3.

5.2 Related work

Simulation can also be used for analyzing the temporal correctness of a real-time system. A tool-suite called STRESS is presented in [ABRW94]. The STRESS environment is a collection of tools for analyzing and simulating the behavior of hard real-time safety-critical applications. STRESS contains a modeling language where the behavior of the tasks in the modeled system can be described. It is also possible to define algorithms for resource sharing and task scheduling. STRESS is primarily intended as a tool for testing various scheduling and resource management algorithms. It can also be used to study the general behavior of applications, since it is a language-based simulator.

Another simulation framework called DRTSS is presented in [SL96]. DTRSS is a high level simulation framework that allows its users to construct discrete-event simulators of complex, multi-paradigm, distributed

real-time systems. The DRTSS framework contains a set of algorithms and protocols from which one can pick the appropriate ones and build a simulator. New algorithms and protocols can be added to the original set. It has support for searching for extremes in the timing behavior of the simulated system. DRTSS has no language where task behavior can be specified, so the abstraction level of the simulation is high and fixed. DRTSS is a part of the PERTS tool-suite, which was developed at the University of Illinois at Urbana-Champaign. The PERTS tool-suite has been commercialized by Tri-Pacific Software Inc. [Tri].

Analytical methods for dealing with probabilistic temporal attributes have been proposed in the literature. In [MEP01], an analytical method for temporal analysis of task models with stochastic execution times is presented. However, sporadic tasks cannot be handled. A solution for this could not easily be found. Without fixed inter-arrival times, i.e. in presence of sporadic tasks, a least common divider of the tasks inter-arrival times can not be found.

Another analytical approach to probabilistic analysis is presented in [LN03]. Here they assume execution times and deadlines that both vary over time in an unpredictable manner, while their arrival times are fixed. Basically, the task model consists of a set of scenarios where every scenario is associated with a probability. For instance, a task may arrive with a certain execution time and deadline with a specified probability. Tasks execute probabilistically depending on several factors, e.g. the scheduling algorithm. The paper proposes solutions for Earliest Deadline First (EDF), and Least Laxity First (LLF). Even though the computational complexity of this solution has not yet been established, it seems, intuitively, that it is quite large. Moreover, none of the proposed analytical approach solve the problem of defining the temporal correctness in terms of starvation in message queues.

5.3 The process

The introduction of an analyzable model of a system brings a continuous activity of maintaining the model. The model should always be consistent with the current implementation of the system, i.e. the implementation should be a true refinement of the model. Consequently, our method must be an integrated part of a company's development process. In this section we will briefly describe the activities associated with the analytical model. Figure 5.1 depicts the general activities required in

our method. Note that the process described here only concerns the method we are proposing. Important activities such as verification and validation of the implementation are omitted.

The first activity in making an existing system analyzable with respect to its temporal behavior is re-engineering of the system. Typically, the re-engineering activity includes identifying the structure of the system, measuring the system, and populating the model. By comparing the result from analyzing the system using the analytical model with the temporal behavior of the real system confidence in the model can be established. This is exactly the same procedure as used in developing models for any kind of systems.

As the system evolves, each new feature should be modeled and the impact of adding it to the existing system should be analyzed. This enables early analysis, i.e. before actually integrating the new feature into the system. Detecting flaws at an early stage is often more cost effective than discovering the problem late in the testing phase of the development process. Note, that such an approach requires a modeling language that support models on different level of abstractions. ART-ML has this property which will be further described in Section 5.4. Modeling of new features should be part of the company's design phase.

Finally, when the new feature has been implemented and integrated into the system, the model of that feature can be refined by feeding back information from the implementation into the model. Hence, a more precise model is implemented. This activity is typically performed in conjunction with the verification phase of a company's development process.

5.4 The method

When creating an initial model, M_0, of an existing system, S_0 several distinct activities which are depicted in Figure 5.2, are required. First, the structure has to be identified and modeled, *i.e.* the tasks in the system and synchronization and communication among them. In the next step, we measure the system and populate the structural model with data about the temporal behavior. Moreover, information needed in the validation phase is collected, *e.g.* response times. When tuning the model we simulate the initial model and compare the results with the validation data collected in the previous step. In this step we may have to introduce more details about the task behaviors in order to capture

the system's behavior accurately. There is a potential risk that we cannot model the system's behavior without introducing too many details. For instance, there are so many implicit relations among the tasks that we can not make a valid model without modeling the complete behavior of the tasks involved. This, however, unveils the complexity of the existing architecture. Consequently, the solution is rather to redesign the complex architecture. Up until this point, the work of making a model is quite straightforward.

To validate the usefulness of the model we have to perform a sensitivity analysis. The sensitivity analysis should be based on foreseen potential changes in the particular system. In the systems we have studied the following typical changes were identified:

- change existing behavior of a task which results in changes in the execution time distribution
- add a task to the system
- change the priority of an existing task

By introducing the changes in the model as well as in the system and comparing their behavior, we can increase the confidence in the created model. Any divergence between the behavior of the simulated model and the system indicates that more details must be introduced in the model. For instance, a change of the execution time in a task may result in a time-out for another task that waits for a semaphore. This could indicate that the semaphore behavior has to be introduced in the model as well.

Moreover, the accuracy of the model is dependent on the quality of the measured data. The measuring of the data should affect the system as little as possible. Too big probe effect on the system will result in an erroneous model and might cause wrong decisions regarding future developments.

A suitable notation is necessary for creating a system model. The language has to support both the architecture (i.e. nodes, tasks, semaphores, message queues), and the behavior of the tasks in different levels of abstractions. It should be possible to compare the behavior of the created model with the target system in an easy way in order to iteratively improve the model to satisfactory level, as illustrated in figure 5.3.

We use simulation in order to analyze the temporal behavior since our notation not only describes the architecture of the target system, but also the behavior of the included tasks. Simulation allows execution times

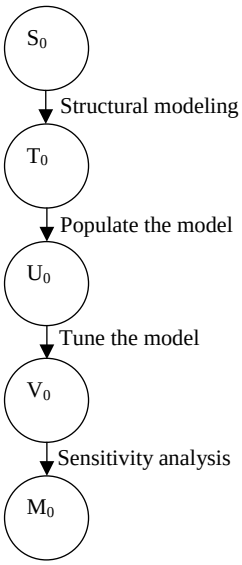

Figure 5.2: The process of constructing a model.

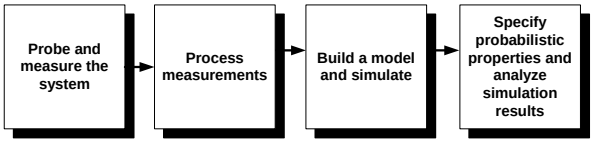

Figure 5.3: The work flow of making an analytical model

expressed as distributions. We analyze the output from the simulator by defining properties of interest. An example of such a property is the probability of missing a deadline requirement on a task. Moreover, the simulation approach allows us to define non-temporal related properties, e.g. non-empty message queues.

5.4.1 Measuring and processing data

Measuring data in a software system requires the introduction of software probes if no hardware probes are used [Sho02]. The data of interest is resource utilization, e.g. task execution times, memory usage or sizes of messages queues. We used software probes in order to log task switches and message queues. The measured data is stored in statically allocated

memory at runtime, in binary format. All formatting of the output is done offline, writing to a file at runtime is too time consuming. This minimizes the probe effect, i.e. the part of the execution time that is caused by the probe.

The output from the system is a text-file containing task switches, time stamps, and the number of messages in different queues. The size of the output can be very big, several hundred kilobytes per monitored second of execution. To manually analyze that data for developing a model would be too time-consuming. We have therefore developed a tool that extracts data from a log and computes the statistical distribution of each task's execution time. In table 5.1 the result of processing data from a task is shown.

In order to calculate the statistical distribution for a set of execution times for a task, we divide all execution times into *instance equivalence classes* (IEC), by stratifying the execution times with respect to a threshold. Formally we define an IEC as:

Definition 15. *An instance equivalence class IEC is a subset of execution time instances of a task E,* $IEC \subset E$*, defined by its upper bound* $\max(IEC) \in E$ *and its lower bound* $\min(IEC) \in E$ *and a threshold that specifies the interval between* $\max(IEC)$ *and* $\min(IEC)$*.* □

A task instance's execution time is a member of the IEC I_n iff it is larger or equal to $min(I_n)$ but less or equal than $max(I_n)$. All instances in an IEC in a model are represented as the average execution time of the IEC which have the probability of occurrence equal to the number of instances in the IEC divided by the total number of measured instances for a task. For example, consider the first entry in table 5.1 which express that, with the probability of 61.5 %, is the execution time for the task 360.097 time units. Consequently, the execution time of tasks in our method is represented as a set of pairs consisting of the average execution time of an IEC and its probability of occurrence.

Definition 16. *The execution time for task t, t.exe, is a set of pairs,* $\langle iec, p \rangle$ *where iec is the average execution time of an IEC and p is its probability of occurrence.* □

An algorithm was developed to automatically identify the boundaries $\min(I)$ and $\max(I)$ for all IEC:s given a set of execution times for a task and a threshold. The algorithm is recursive. Initially all instances are sorted by their execution time using the quicksort algorithm. The sorted

list constitutes the initial IEC, I_0 for the task. Next, the largest difference in execution time between two adjacent instances in the sorted list is located. If the largest difference is larger than a specified *threshold*, the list I_0 is split into two new IEC:s and recursive calls are conducted with each of the two new IEC:s. Consequently, the threshold specifies mathematically how big variations there can be in execution times belonging to the same IEC. From the system modeling point of view the threshold has two purposes. First, it can be used to filter small variations in execution times due to cache memories or branch prediction units, i.e. independent from the control-flow. Moreover, threshold can also specify the level of abstraction with which the temporal behavior is modeled. A large threshold results in a more coarse-grained distribution, i.e. less number of IEC:s for a task. Below the equation for finding distinct IEC:s, given a set of sorted execution times, is displayed.

$$\begin{aligned} &\forall\langle x_i, x_{i+1}\rangle \exists\langle x_j, x_{j+1}\rangle \in I_0 : \\ &\quad abs(x_j - x_{j+1}) > abs(x_i - x_{i+1}) \wedge \\ &\quad abs(x_j - x_{j+1}) \geq threshold \wedge i \neq j \end{aligned} \tag{5.1}$$

As a result from applying the equation above on a sorted set of execution time instances we may get two new potential IEC, I_k and I_{k+1} where $\min(I_k) = \min(I_{k-1})$, $\max(I_k) = x_j$, and $\min(I_{i+1}) = x_{j+1}$, $\max(I_{k+1}) = \max(I_{x-1})$. If no gap greater than the threshold is found, the final IEC is already found and the recursion is stopped. When the recursion is stopped, the largest and the smallest execution time in the list is considered to define the boundaries of an IEC.

This approach has been implemented in the ART-ML tool suit and worked well with the characteristics of our data. However, the distance between min and max in an IEC could be quite big if no gap greater than the threshold is found in the sorted list of execution times. Theoretically, all measured execution times may end up in the same IEC. We have three possible solutions for such a scenario:

- Reduce the threshold and try again
- Do not create any IECs (threshold = 0), use the entire set of instances and assign each of them the probability of $\frac{1}{no.of instances}$. This solution results in a very detailed model
- Model such a task as a linear distribution with a max, and a min execution time and uniformly assign probabilities in between them.

An alternative strategy when constructing an IEC is to define a threshold that constrains the distance between max and min for every IEC. This eliminates the problem discussed above. However, it still is an abstraction of the temporal behavior. Moreover, instead of representing an IEC by its average execution time we can choose to use the IEC's maximum execution time. This makes the model more safe but also more pessimistic.

The measured data can also be graphically visualized in a chronological order. In Figure 5.4 such a graph is depicted, where in this particular case, execution times (y-axis), are plotted in the order in which the task instances have executed. Studying such a graph may reveal executional dependencies among tasks. If regular patterns are present in the execution times there may be an relation to other tasks' executions. In Figure 5.4 we can see such a pattern. The execution time varies regularly between 5×10^{-8} and 5.5×10^{-8}. A typical example of such a dependency is communication among components. By examining the source code we can model the message passing in the ART-ML model. Introducing those dependencies will make the model more accurate with respect to the implemented system.

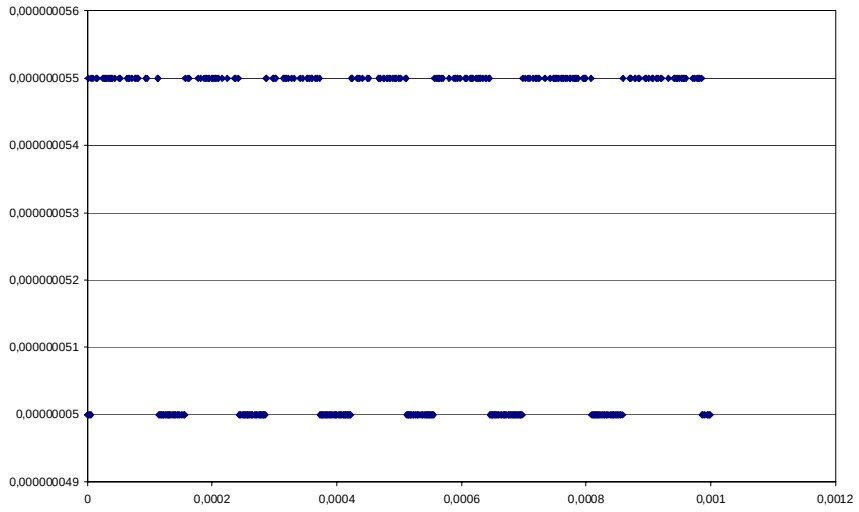

Figure 5.4: An example of measured execution times

Min time	Max time	Average time	n	n/N
287.265	420.876	360.097	131	61.5%
577.448	604.320	590.884	2	0.94%
4176.659			1	0.47%
4797.058	5024.122	4911.885	12	5.6%
5177.941	6829.881	5829.924	65	30.5%
11962.947			1	0.47%
12814.769			1	0.47%

Table 5.1: An example of statistical distribution of a task. $N = \sum n$, were n is the number of instances in an IEC.

5.4.2 Modeling on different levels of abstraction

When creating a model of the tasks in the target system, a level of abstraction has to be chosen. That level defines the accuracy of the model. The lower abstraction level, the more detailed and accurate model. There is no point in using the lowest possible level of abstraction, i.e. a perfect description. In that case, the actual code could be used instead. Using an extremely high level of abstraction results in a model that is not very accurate and therefore of limited use. The best result is something in between these two extremes.

In the ART-ML language, very detailed models of task's logical behavior can be made, theoretically perfect ones. By describing blocks of code only by their execution time, the abstraction level is raised to a higher level. The more code that is described by an execute-statement, the higher level of abstraction. The highest abstraction-level possible is if all code of the task is described using a single execute statement.

It is possible to use any level of abstraction when describing a task using the ART-ML language. It is therefore possible to describe different tasks at different levels of abstraction. This property of the language enables the model to be improved (in terms of level of detail) task by task.

The execution time distributions used also has different levels of abstraction. The measured data from the target system is somewhat filtered when creating the distributions. The recorded instances are grouped into equivalence classes. This causes data to be lost. The level of abstraction is in this case the number of intervals used to describe the execution time of the task. This level of abstraction impacts the accuracy of the model.

If there are multiple tasks in the system that is of no interest and do not affect the behavior of other tasks, they can be modeled as a single task at a maximum abstraction level, i.e. only by a single execution-time

probability distribution. We refer to such a group of tasks as a *composed task*. This reduces the complexity of the model without affecting the accuracy of the result regarding the tasks of interest. However, it is required that all tasks in a group has the same or adjacent priorities. Moreover, tasks can only be grouped in such a way that no other modeled task, i.e. task not being part of the group, has a priority within the range of a group. For instance, consider a composed task consisting of two tasks, Task τ_a with high priority, and task τ_c having low priority. Moreover, consider task τ_b which is also part of the system and runs at mid priority. Task τ_a should be able to preempt task τ_b, but not task τ_c should not. Thus, the composed task has to run on different priorities in order to reflect the control flow of the implemented system.

Moreover, tasks that exhibit dependencies to the tasks of particular interest may not be part of a composition if the dependency is required in order to make a valid model of the system. For instance, task τ_a sends a message to task τ_c which is a task that we are interested in analyzing. By composing task τ_a with a task τ_b the frequency with which task τ_a send its messages to task τ_c will be changed according to Equation 5.2. On the other hand, the composition of task τ_a task τ_b is valid if we do not need to model that dependency between task τ_a and task τ_c.

Formally, we can express the rules of grouping tasks into composed tasks, i.e. assigning execution time distribution, period time and priority, in a way that preserves the utilization of the CPU which the tasks in the group contributes to. First the set of tasks to compose, C, have to be normalized with respect to the period times. The composed task will run with the shortest period time among the participating tasks. Consequently, the period time of the composed task c is:

$$c.T = \min_{t \in C}(t.T) \tag{5.2}$$

Normalizing the tasks in such a way that the CPU utilization is preserved requires re-calculating the execution times for all IEC:s described in Section 5.4.1, for all tasks in C.

$$\forall t \in C \forall i \in t.exe : i.iec = \frac{c.T}{t.T} i.iec \tag{5.3}$$

The resulting execution time distribution for the composed task is obtained by calculating the cartesian product, V, of all $t.exe$ where $t \in C$,

i.e. $t_1.exe \times t_2.exe \times ... \times t_n.exe$. Every n-pair which is part of the cartesian product corresponds to an executional scenario. For instance, $\langle x_1, x_2, ..., x_n \rangle$ corresponds to the scenario where task 1 executes for $x_1.iec$ time units, task 2 executes $x_2.iec$ time units, and so on.

$$c.exe = \{\langle iec, p \rangle | \forall v \in V : iec = \sum_{\forall j \in v} j.iec \wedge p = \prod_{\forall j \in v} j.p\} \tag{5.4}$$

The final c.exe is obtained by merging pairs in c.exe that have equal iec:s (cmp. the generation of IEC:s described in Section 5.4.1). For the set of pairs, $\{\langle iec, p_1 \rangle, ..., \langle iec, p_n \rangle\} \subseteq c.exe$, of all pairs having the same execution time, the merged pair remaining in c.exe is $\langle iec, \sum_{i=1}^{n} p_i$, where $\sum_{i=1}^{n} p_i$ is the probability that task c, executes iec time units.

Finally, the priority of the composed task c, $c.p$, is assigned the highest priority of the tasks participating in the composition.

$$c.p = \max_{\forall t \in C}(t.p) \tag{5.5}$$

As an example consider the composition of two tasks: τ_a and τ_b. Task τ_a executes with the distribution $\tau_a.exe$={(1,0.75),(2,0.25)}, and $\tau_a.T$=10. Task τ_b executes with the distribution $\tau_b.exe$={(2,0.5),(3,0.5)} and $\tau_a.T$=5. Normalizing the execution of task τ_a, i.e. $\tau_a.exe$={($1\frac{5}{10}$,0.75), ($2\frac{5}{10}$,0.25)} gives the cartesian product, V, equal to {((0.5,0.75),(2,0.5)), ((0.5,0.75),(3,0.5)),((1,0.25), (2,0.5)),((1,0.25),(3,0.5))}. The cartesian product V results in an execution time distribution for the composed task, $\tau_c.exe$ equal to {(2.5,0.375), (3.5,0.375), (3,0.125), (4,0.125)}, $\tau_c.T$ = 5.

The assignment of temporal attributes to composed tasks described above is a coarse approximation of the system behavior. Ideally, all tasks are modeled individually. However, in order to limit the modeling effort, and to prune the state space, such approximations can be practical. The result of applying the proposed rules may lead to situations where execution times are longer than the period time. This corresponds to a system overload which is possible in the implementation.

5.4.3 Simulating the system behavior

The simulation-based approach used in this work allows correctness criterion other than meeting deadlines. An example of other correctness

criterion could be the non-emptiness of message-queues, i.e. a starvation property. One of the systems studied in this work had this criterion. If a certain message-queue got empty, it was considered a system failure.

Simulation also allows us to specify arbitrary system cycles. FPA assumes cycles equal to the Least Common Multiple of the period times in the task set (LCM). However, there exists systems, such as the robot controller investigated as part of this work, where the cycle times are determined by other criteria. For instance, in the robot case presented in Section 5.8, the system cycle is determined by the robot application, i.e. the cycle time of the repetitive task of robot which it is programmed to perform.

When designing the simulator, two different approaches were identified. The most intuitive was to let the simulator parse the model and execute it statement by statement. The other approach was to create a compiler that translated the high level ART-ML model into simple instructions and construct the simulator as a virtual machine that would execute the instructions. A test was made to compare the performance of the two approaches based on two prototypes. The virtual machine solution performed significantly better which is crucial for an analysis tool. Currently, work is going on to improve the simulator even further by compiling the code in the model.

The simulator engine is based on three parts, the *instruction decoder*, the *scheduler* and the *event-processing*. The instruction decoder executes the instructions generated by the compiler, i.e. it is the virtual machine. Some of the instructions generate events when executed. Those instructions are described in more details in chapter 5.6.1, but for the sake of explaining the event-processing part of the simulator we will already now introduce an instruction called *execute*. The execute statement describe the partial execution time of the code in the target system, i.e. the execution time for a complete task or part of a task. An event contains a time stamp, type of event, and an id of the source task. The time stamp specifies when the event is to be fired. Consequently, new decisions about what task to execute are taken upon an event. The scheduler decides which task that is to execute according to the fixed priority strategy.

It is the execute instruction that consumes time and drives the simulation forwards. First, an execution time is selected according to the modeled execution time distribution that is passed as an argument to execute. The current time is increased with that amount of time, or the time when another event interferes with the execution. If an event occurs during the execution of a task, the execution is suspended, the event is

taken care of and the scheduler makes a new decision. The next time the preempted task is allowed to execute, it will restart the execution of the execute-instruction, remembering how much time it has left for execution.

Since an execute instruction is necessary for pushing the simulation forwards, there must always be a task that is ready to execute and contains such a statement. Due to this, it is mandatory to have an idle-task in the simulation that consumes time if no other task is ready.

5.5 Model validity

In this section we will discuss how to assure model validity, i.e. the activity to establish confidence in the constructed model. This is an important and necessary part of constructing models.

5.5.1 Validity of the simulation approach

Existing analytical methods determines if the temporal behavior of a system is safe or not, given that the analytical model is correct, e.g. that the estimates of the WCET of the components are safe [But97][ABD+95] [LL73]. In order to be safe the WCET is assigned a value that is as tight as possible but slightly larger than the actual WCET. Such a method, however, tends to over-constrain the system as the worst-case always is considered. In Figure 5.5 a system's temporal behavior is depicted as sets. An analytical approach is pessimistic but safe, while simulation is realistic but not necessarily safe. Analytical models and analyses found in conventional scheduling theories are often too simple and therefore a real system cannot always be modeled and analyzed using such methods. Simulation is better from that point of view. By simulating the system with realistic distributions of the execution times, we can demonstrate that the system is correct. A disadvantage is that given the same correct analytical model, we cannot be confident in finding the worst possible temporal behavior through simulation.

There may be many valid models of a single system. Observing and measuring a systems behavior may give a system model that is valid given that the assumptions do not change. We will exemplify the fenomenon with a small physical experiment.

The experiment aims at deciding an equation (a model), for calculating how high a ball bounces of the ground the first time after being dropped from a certain hight. The equation is determined by repeatedly

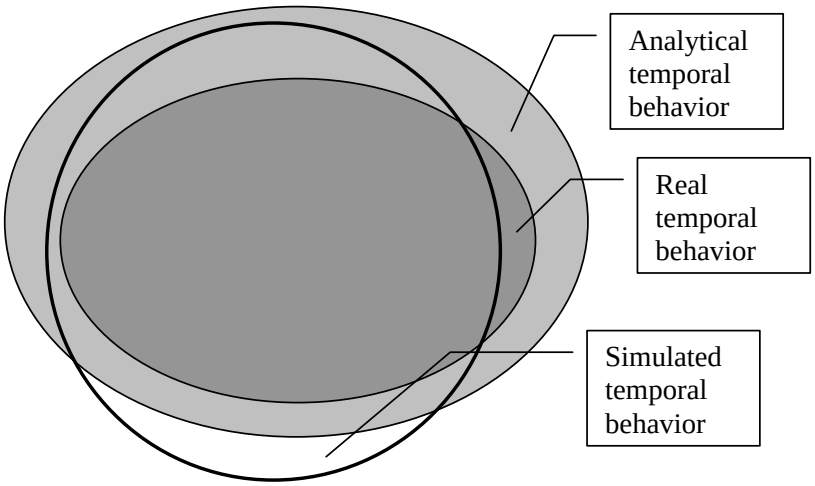

Figure 5.5: The confidence in analytical system analysis vs. simulation-based analysis

dropping the ball from different heights and measuring the hight of the bounce. The resulting equation could relate the bounce proportionally to the hight from which the ball is dropped. This is a completely valid model as long as nothing is changed. We can even change the size of the ball without making the model invalid. However, that model is too simple for capturing changes in, e.g. the material of the ball, or the material in the ground for that matter.

We can transfer the physical experiment onto our method for analyzing the temporal behavior of a complex system. We can convince ourself that the model is valid by comparing the output from the simulator with the values measured in the system. However, changing the model in order to analyze the impact of adding new features to a system can, potentially, invalidate the model. Whether or not the model is completely valid becomes evident only after implementing the new feature, i.e. when we have something to compare with. However, the more confidence we have in the model, the more confident can we be in the simulation results, i.e. before implementing the new feature in the system. Continuously maintaining and validating the model as part of the development process is the way in which the model is iteratively refined and kept consistent.

To exemplify the model validity problem consider a computer system with two tasks, τ_a and τ_b. Task τ_a sends a message to task τ_b and this

message passing is modeled in ART-ML and the simulation results indicates that the system is correct. Now task τ_c is added to the system which also sends a message to task τ_b. This changes the temporal behavior of task τ_b. However, we only model the task τ_c as an execution time distribution leaving the message passing out. As a consequence, the simulations of task τ_b diverge from what we can observe in the changed system. The model that initially was correct is now incorrect due to lack of details.

5.5.2 System identification

System identification is a technique used in the domain of control theory [Joh93]. By measuring and observing the input-output relationship between signals in the process a model can be determined in terms of a transfer function. Validating models based on the system identification approach is somewhat related to testing. Typically, output signals simulated using the model is compared with the output signals of the physical process. Hence, the model is regarded as correct if the simulations and the physical process generates approximately the same output.

Moreover, a method called *residual analysis* can be applied on models of continuous systems. By observing whether or not the errors in the prediction compared to the actual output, the *residual*, are independent from the input signal. If not, it indicates that there are dynamics in the system that is not yet in the model.

Testing the model with different input signals and comparing the prediction with the signals produced by the actual system is fine if the process is continuous in its nature. It is fair to assume that we can interpolate the behavior in between the tested signals. However, computer software is not continuous, they are discontinuous systems meaning that the behavior may change dramatically as a result of small changes in the system.

Our approach to model validation is similar to the one proposed in system identification. Our hypothesis is that potential discrepancies will be exposed if we introduce changes in the system and the corresponding changes in the model. Comparing the simulation results with the data measured in the system will give us the possibility to settle the validity of the model.

The model validation process we propose is depicted in Figure 5.6. The first activity is to develop a set of change scenarios which should reflect typical, possible, and foreseen changes in the system. This cor-

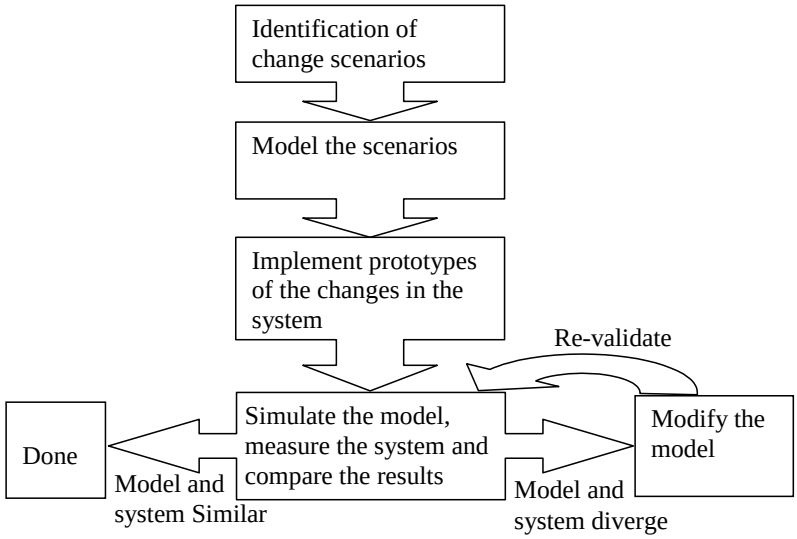

Figure 5.6: The process of validating the model.

responds to the scenario elicitation described in [Ben02]. The set of scenarios are system specific, i.e. they are valid only for the system for which they were developed.

After having selected a suitable set of appropriate and concrete change scenarios it is time to implement them in the model. By concrete we mean that scenarios such as:

change the priority of a task.

, has to specify exactly what task to alter and what the new priority is supposed to be.

Moreover, we have to introduce the proposed scenario in the system as a prototype that simulates the changes. Hence, we do not implement a complete functional change. We aim to mimic the temporal behavior of the changes. For instance, for the scenario where the functional behavior of an existing task is changed, we only need to inject code that simulate an increase or a decrease in execution times. Adding a new task is similar, we do not implement the task's behavior, we add the new task and implement a piece of code that mimic the temporal behavior only.

Finally, we compare the results from simulating the changed model to the behavior measured in the changed system. If they both behave

the same way for every identified change scenario we have establish confidence in the model. However, if the behavior of the simulation and the system diverge we must tune the model. Typically, more details has to be introduced in the model. Examples of such details are a more detailed model of the task's logical behavior and executional dependencies among task such as communication.

5.5.3 Validation recommendations

The abstraction introduced by composed tasks makes the model sensitive to changes, e.g. adding new tasks. If the new task has a priority within the priority range of a composed task, the composed task must be re-modeled. Moreover, changing the behavior of a task that is part of a composed task in the model requires re-modeling. Hence, it can be fruitful to consider the change scenarios when constructing the model. Likely changes should have as small impact as possible on the model.

The use of composite tasks in the model is also related to the intended use. Is the intention of the modeling effort only to analyze current status of the system, e.g. some tasks in the system, then large composite tasks are acceptable. On the other hand, a model that evolves with the system must utilize composite tasks carefully as we like to minimize the modeling effort when changing the system.

5.6 The ART-ML framework

5.6.1 The modeling language

The modeling language that we have developed for modeling complex real-time systems, ART-ML, is composed of two parts, the *architecture model*, and the *behavior model*. The architecture model describe the temporal attributes of tasks, e.g. period times, deadlines, priorities. The architecture model also describes what resources there are in the system.

The behavior model describes the behavior of the tasks in the architecture model. Thus the behavior is encapsulated by the architecture model. The behavioral modeling language is an imperative, Turing-complete language close to Basic and C in its syntax as illustrated by the following example:

```
mainbox TASK_C_MAILBOX 4;
mainbox TASK_C_MAILBOX 6;
```

```
const msgcode_ref_request 1001;
const msgcode_ack 1002;

task APERIODIC_TASK_C
  trigger mailbox TASK_C_MAILBOX
  priority 2

behavior{
  variable incoming;
  incomming = 0;

  recv(incoming, TASK_C_MAILBOX)
    timeout 100;
  if (incoming == msgcode_ref_request){
    recv(incoming, TASK_C_MAILBOX)
      timeout 10000;
    execute((60,6200),(40,6750));
    send(TASK_B_MAILBOX, msgcode_ack);
  }else{
    chance(80){
      execute((63,400),(37,470));
    }else{
      execute((100,1000));
    }
  }
}
```

Two constructs make ART-ML unique compared to other modeling languages that we have been studied: the *execute-statement* and the *chance-statement*.

The execute statement describe the partial execution time of the code in the target system, i.e. the execution time for a complete task or part of a task. The execution time for a task is represented by a statistical distribution. A probability distribution is implemented as a list of pairs that corresponds to the calculated IEC:s described in Section 5.4.1. Every pair has a probability of occurrence and an execution time. When a task performs an "execute" it supplies a probability distribution as parameter. An execution time is chosen according to the distribution and the task is put into "executing state". When a task has been allowed to execute for

that amount of time, the next statement, if any, in that task's behavior description is executed. In the example below, the execute statement will execute 10 time units with the probability of 19 % and 56 time units with the probability of 81 %:

```
execute((19,10), (81, 56));
```

The chance statement implements a stochastic selection. Stochastic selection is a variant of an IF-statement, but instead of comparing an expression with zero, the expression is compared with a random number in the interval [1-100]. If the value of the expression is less than the random number, the next statement is executed. If not, the else-statement is executed if there is one. Stochastic selection is used for mimicking task behaviors observed as a black box. For instance, we can observe that a task sends a message to a particular queue with a certain probability by just logging the queue. This can be model with stochastic selection such that we send a message with the observed probability. For instance, it is possible to specify that there is a 19 % chance of sending a message:

```
chance(19)
   send(mbox1, msg)
```

The language has also support for message passing through the primitives *send* and *recv*. Both can be associated with timeouts. Moreover, binary semaphores can be specified in ART-ML through *semtake* and *semgive*. Semtake can be used in combination with a timeout as well.

In Section 5.7 a more exhaustive example is presented where ART-ML is used for modeling of the example system that was initially introduced in chapter 2.7.1. The example also illustrates the method associated with this approach. The complete grammar for ART-ML is presented in appendix B.

5.6.2 The probabilistic property language

The impact of altering a component, or adding components due to new features can be analyzed based on the simulation results. Basically, we compare the result from simulating the extended system with simulations performed without the extension. The differences constitute the impact. For real-time systems there exists an overarching criteria somewhat parallel with the impact analysis. The utilization of available resources must not exceed the upper limit and the temporal requirements must not be

violated. Moreover, particular component may have temporal requirements associated with their execution that must be conformed to. Typically examples are deadlines and jitter, i.e. variations in periodicity. The temporal behavior can also affect other requirements. In the case study performed at ABB Robotics, the correctness of the system was partly dependent on the non-emptiness of a particular message queue. The temporal behavior of components in the system had influence on this requirement.

The result of an impact analysis is in the form of the probability of violating a requirement due to the modeled change of the system. If the system is in the class of hard real-time systems, i.e. all temporal requirements must always be fulfilled. Thus, the probability of complying with a requirement must be 1.

Even if all temporal requirements are fulfilled after changing the system, there still is an impact. For instance, the response times of a component may increase or decrease. The decrease and increase in response times corresponds to the differences in response time distribution obtained by simulating before and after changes in the analytical model.

The requirements are specified in the simulation approach with a *probabilistic requirement property language*, (PPL). PPL can specify probabilistic properties on tasks that control the execution of components and on message queues over which components communicate. Given the number of times a requirement property has been violated the probability of violating it can be calculated.

For every requirement property there must be a property theory which is used for evaluating the simulation. As the property theory for simulation is based on observations from simulating the system, the property gets proportionately simple compared to the correspondence in the analytical approach [HMSW02]. For instance, checking the deadline property of a task is done by comparing every observed response time, i.e. the response time distribution, with the required deadline. If the response time is greater than the deadline, the requirement is violated. Given a response time distribution we can calculate the probability of violating the deadline. As an example, consider the response time distribution displayed in Figure 5.7. The probability of violating a deadline requirement of 24 ms is equal to 0.1.

The requirement property language supports the definition of properties as well as theories for calculating the property, i.e. defining the property theory in terms of the variables available after a simulation with a probabilistic property language. The probabilistic property language

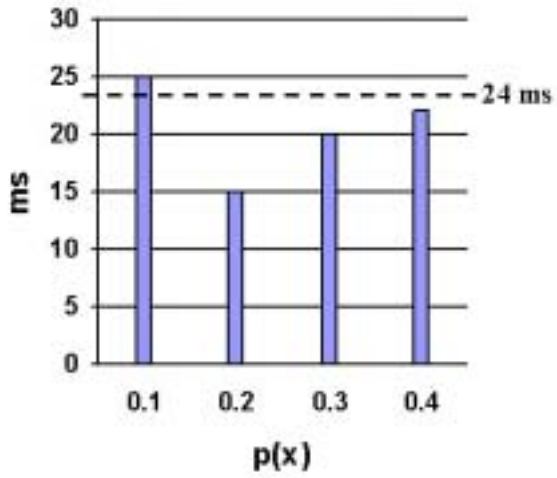

Figure 5.7: The response time distribution of a task

specifies properties based on the knowledge generated by the simulator, and includes relation operators, $=, >, \geq, <, \leq, \neq$, the logical operators $\wedge, \vee$, the arithmetic operators +, -, the functions *max* and *min*, and an *instance operator*. The instance operator binds a task to instances of execution and enables specifying properties where the relative mutual relation among instances of tasks is of importance.

The output generated by the simulator determines the properties available for every task and message queue. Currently the simulator generates the following temporal data about tasks and message queues in a system:

- Size of message queues at task switches, $q.size$
- Time when a task starts an execution, $\tau.start$
- Time when the execution of a task was interrupted
- Time when the execution of a task was restarted after an interrupt
- Time when a task has finished its execution, $\tau.end$

The response time for a task, $\tau.response$, is not generated as such but can be calculated as $\tau(i).end - \tau(i).start$.

Properties are specified as probabilistic statements. Specifying an invariant property, i.e. a property that should always be true, corresponds to a probability equal to 1. A property that verifies that all instances i of task τ, $\tau(i)$, always meet a deadline of 10 time units is expressed as follows:

$$P(\tau(i).response < 10) = 1 \ (hard\ deadline)$$

If it is not critical that every instance of a task meet its deadline, we say that the deadline is firm. In our probabilistic property language we can express a firm deadline as:

$$P(\tau(i).response < 12) \geq 0.75 \ (firm\ deadline)$$

The instance operator is used for distinguish different instances of the same task from one another, or to specify properties over the same instance number of different tasks. Separation is a property that specifies the minimum distance in time between two consecutive instances of a task.

$$P(\tau(i+1).start - \tau(i).end \geq 10) = 1 \ (separation)$$

A precedence relation specifies the order in which two tasks should execute.

$$P(\tau(i).end <= \sigma(i).start) = 1 \ (precedence)$$

The probabilistic statements may contain an unbounded variable. For instance, the probability may be unbounded which gives as result the probability of the statement being true. A property that specifies the probability of meeting a deadline equal to 10 time units is:

$$P(\tau(i).response < 10) = X$$

We can also leave variables in the predicate unbounded. This could, for instance, be used for feeding back temporal constraints to control engineers, e.g. the feedback loop delay. The probabilistic property that answers with what deadline will be met with a probability of 0.9 is:

$$P(\tau(i).response < d) \geq 0.9$$

Specifying firm deadlines only in terms of the probability of missing them may not be sufficient since deadline misses can be irregularly distributed. For instance, we can miss many consecutive deadlines and still

fulfill the temporal requirement since sufficiently many deadlines are met in between bursts of deadline misses. In the probabilistic property below, we specify that two consecutive instances of task τ must not both miss their deadline.

$$P(\tau(i).response > 10 \wedge \tau(i+1).response > 10) = 0$$

Correctness criterion for real-time systems may not only be specified in terms of explicit temporal requirements. As discussed earlier in this paper, the correctness of a system may be defined in terms of non-empty message queues. Such an invariant requirement expressed in our probabilistic property language would be as follows:

$$P(queue.size > 0) = 1 \ (non - emptiness)$$

Calculating the properties of a system is done offline from a simulation point of view, i.e. it is done when the simulation has produced its output. Thus, it will not influence the simulation performance. Moreover, the output generated by the simulator is equivalent with the format of the data measured on the real system. This makes it possible to apply the probabilistic properties on the implementation as part of the verification. Consequently, confidence in that the implementation is a refinement of the model can be established.

Grammar and semantics for the probabilistic property language

In this section we present the semantics for our probabilistic property language (PPL). The grammar of PPL is presented in Backus Naur Form (BNF), in appendix C.

We present the semantics of PPL in terms of ordinary set theory. A property specified in PPL explores the executional information about tasks and message queues in a system. The set theory approach is applicable since we can represent temporal data about tasks as pairs of task instances and time stamps. Currently, every task is represented by three ordered sets, *start time*, *ending time*, and *response time*. The start time is when a task starts to execute, the end time is the time when a task completes its execution, and response time is the time it takes for a task to complete its execution.

Definition 17. $(\tau(i).start, <)$ *is a finite totally ordered set of start times of task* τ*.* □

Definition 18. *$(\tau(i).end, <)$ is a finite totally ordered set of end times of task τ.* □

Definition 19. *$(\tau(i).response, <)$ is a finite totally ordered set of response times of task τ.* □

The size of message queues are observed at every task switch and instances makes no sense when in comes to queues. Thus, a message queue is represented as a set of queue sizes.

Definition 20. *queue.size is the finite non-ordered set of observed number of messages simultaneously on queue* □

All data sets for tasks and message queues are created from the data generated by a simulation or the data measured on the implemented system.

The unbounded variable in an expression constitute the objective of the query. Unbounded variables are scalar and represents either a probability or an integer. PPL allows for only one unbounded variable in every probabilistic expression. Thus, either a probability of satisfying the expression or an value that satisfies the expression with a specified probability. Two unknown variables make the expression undecidable.

In the semantical rules given below the sets defined in definition 17, 18, and 17, are generalized meaning that all sets apply to the rules. We denote sets as capital letters and constants and unbounded scalars as lower-case letters with the suffixes c for constants and v for unbounded variables. Consequently, the unbounded variable x is denoted x_v, and the constant x is denoted x_c

The arithmetic operators in PPL operates on the elements in an ordered set. In order to ensure that the finite ordered sets on which the arithmetic is applied have the same cardinality we only allow two different construction: between an ordered set and a scalar, or between ordered sets belonging to the same task, e.g. $\tau(i).start + 10$ and $\tau(i).end - \tau(i).start$

$$X_i \; aritop \; X_j \Rightarrow \{x_i \; aritop \; x_j : x_i \in X_i \wedge x_j \in X_j\}$$

$$X_i \; aritop \; x_c \Rightarrow \{x_i \; aritop \; x_c : x_i \in X_i\}$$

The label *aritop* is an arithmetic operator.

The semantics of the probabilistic properties in which the arithmetic expressions may be a part of is presented below. As we must be able to

reason about the relative ordinal of individuals in the sets we define a relative order operator.

Definition 21. *$x <^n y$ is the relative order relation between x and y such that n-1 individuals are ordered in between x and y:*

$x <^1 y$ *iff* $\neg\exists z : x < z < y$

$x <^2 y$ *iff* $\exists z : x <^1 z <^1 y$

$x <^n y$ *iff* $\exists z_1 ... z_{n-1} : x <^1 z_{n-1} <^1 ... <^1 z_1 <^1 y$ □

Especially we say that $x \in X$ and $y \in Y$ have the same order in X and Y respectively if $x <^0 y$.

In the semantical rules below we denote sets as capital letters. The sets correspond to the sets defined in definition 17 through definition 19. Note that since we cannot reason about instances and order when it comes to queue sizes, as defined in definition 20, we specify a special semantic rule for that case. We distinguish unbounded variables from constants by subscripting unbounded variables with v, e.g. x_v, and constants with c, e.g. x_c. If the relation operation in the rules below is not explicitly specified we refer to them as a *relop*.

There are two different types of expressions: properties without unbounded variables, and properties including unbounded variables. They differ mainly in the nature of their solutions. A property without unbounded variables are evaluated as true or false, i.e. whether or not the property is satisfied by the simulation results. A property with unbounded variable on the other hand, binds the variable to a value.

We start by describing the rules for properties without any unbounded variables. There are three different kinds: comparing instances in a set with a constant, comparing instances in two different sets, and comparing a message queue with a constant.

$$P(X\ relop1\ x_c)\ relop2\ y_c\ \Rightarrow \begin{cases} true & if\ y_c\ relop2\ \frac{|\{x:x\in X \wedge x\ relop1\ x_c\}|}{|X|} \\ false & otherwise \end{cases} \tag{5.6}$$

$$P(X_1(i)\ relop1\ X_2(i+n))\ relop2\ x_c\ \Rightarrow \begin{cases} true & if\ x_c\ relop2\ \frac{|\{\langle x,y\rangle : x\in X_1 \wedge y\in X_2 \wedge x <^n y \wedge x relop1 y\}|}{|\{\langle a,b\rangle : a\in X_i \wedge b\in X_j \wedge a <^n b\}|} \\ false & otherwise \end{cases} \tag{5.7}$$

For message queues we can not reason about instances, hence, we cannot specify properties that compares two different message queues over time. We only allow comparison between a message queue and a constant. However, as we will see later, the probability can be unbounded in a message queue expression.

$$P(Xrelop1x_c)relop2y_c \Rightarrow \begin{cases} true & if\ y_c\ relop2\ \frac{|\{x:x\in X\wedge xrelop1x_c\}}{|X|} \\ false & otherwise \end{cases} \tag{5.8}$$

The properties that includes unbounded variables can also be divided into two different classes: properties with an unbounded probability and properties with an unbounded variable in the comparison. Properties with unbounded probabilities are rather simple to evaluate. Basically, we calculate the probability of the property being true.

$$P(X(i)\ relop1\ x_c)\ =\ x_v\ \Rightarrow x_v = \frac{\mid \{x : x \in X \wedge x\ relop1\ x_c\} \mid}{\mid X \mid} \tag{5.9}$$

$$P(X_1(i)\ relop1\ X_2(i)) = x_v \Rightarrow$$
$$x_v = \frac{\mid \{\langle x, y\rangle : x \in X_1 \wedge y \in X_2 \wedge x <^0 y \wedge xrelop1\ y\} \mid}{\mid \{\langle a, b\rangle : a \in X_1 \wedge b \in X_2 \wedge a <^0 b\} \mid} \tag{5.10}$$

$$P(X_1(i)\ relop1\ X_2(i+n)) = x_v \Rightarrow$$
$$x_v = \frac{\mid \{\langle x, y\rangle : x \in X_1 \wedge y \in X_2 \wedge x <^n y \wedge xrelop1\ y\} \mid}{\mid \{\langle a, b\rangle : a \in X_1 \wedge b \in X_2 \wedge a <^n b\} \mid} \tag{5.11}$$

Message queue properties with unbounded probability are equally straight-forward to calculate:

$$P(X\ relop1\ x_c)\ =\ x_v\ \Rightarrow x_v = \frac{\mid \{x : x \in X \wedge x\ relop1\ x_c\} \mid}{\mid X \mid} \tag{5.12}$$

Unbounded variables in PPL are bounded to intervals, not scalar values. The reason for this is that there can be many valid bindings for a variable that fulfill a property. However, there always exists a minimum

and/or a maximum valid assignment. For instance, the deadline requirement $P(X(i).response \leq x_v) \geq 0.75$, which is expressing the question: what deadline is fulfilled in 75 % of the executions have many possible solution. However, there is one lower bound, i.e. minimum assignment to x_v that fulfills the property. If the set $X.response = \{1, 2, 3, 4\}$, than a deadline of 3 is complied with in 75 % of the cases. However, a deadline of 4 or bigger is complied with in 100 % of the cases which is greater or equal to 75 %. Consequently, the solution is to bind x_v to the interval $[3, \infty)$. If we, on the other hand, change the property to $P(X(i).response \leq x_v) \leq 0.75$, i.e. what deadline is fulfilled at most 75 %, we get a different interval. Given the same set, X.response, we can see that a deadline equal to 1 give deadline misses in 75 % of the execution. However, a deadline equal to 0 is also a valid solution. Hence, the interval assigned to x_v is [0,1].

Moreover, we must restrict the way in which properties with an unbounded variable within the comparison is constructed. If not, we may end up with expressions that are undecidable. For instance, the property P(X(i) relop x_v) = 0.5 have no valid solution for x_v if the set X consists of three elements. Hence, expressions with unbounded variables and a constant probability with a strict equality is only allowed in two cases, i.e. equal to one, or equal to zero. We can always find bindings to the variable that satisfy none, or all of the elements in a set. Moreover, binding a variable to a value equal to elements in a set with a certain probability may be undecidable, e.g. P(X(i) = x_v) relop 0.5. It is not necessarily the case that the set X have enough elements with the same value in order to fulfill the property.

$$P(X\ relop\ x_v) = 1\ \Rightarrow x_v = [j, \infty)\ :\ \frac{\mid \{x : x \in X \wedge x\ relop\ j\} \mid}{\mid X \mid} = 1 \tag{5.13}$$

where $X \neq \emptyset$, and relop $\neq' ='$.

$$P(X\ relop\ x_v) = 0\ \Rightarrow x_v = [0, j]\ :\ \frac{\mid \{x : x \in X \wedge x\ relop\ j\} \mid}{\mid X \mid} = 0 \tag{5.14}$$

where $X \neq \emptyset$, and relop $\neq\ ' ='$.

There may exist several solutions for an unbounded variable as discussed above. Hence, we get as a result the set of possible solutions which

is a subset of the set that is being analyzed, $x_s \subseteq X$, from which we can derive the interval that corresponds to the solution. Depending on the relational operand, we construct the interval by taking *max*, or *min* of x_s. It depends on whether it is an assignment of the unbounded variable that give at most a certain probability ($\leq$), or if it give at least a certain probability ($\geq$).

$$P(X \ relop \ x_v) \leq x_c \ \Rightarrow x_v = [0, max(x_s)] \ : \\ x_s = \{x : x \in X \wedge \frac{\mid \{i : i \in X \wedge i \ relop \ x\} \mid}{\mid X \mid} \leq x_c\} \quad (5.15)$$

$$P(X \ relop \ x_v) \geq x_c \ \Rightarrow x_v = [min(x_s), \infty) \ : \\ x_s = \{x : X \in X \wedge \frac{\mid \{i : i \in X \wedge i \ relop \ x\} \mid}{\mid X \mid} \geq x_c\} \quad (5.16)$$

5.7 ART-ML: An example

In this section we will go through a small example where we apply the method proposed in this chapter. The control system example that we last visited in Section 2.7.1 has now been in use for quite a while. Maintenance as well as new functionality has increased the complexity of the system. Furthermore, the analytical models that initially was constructed has become obsolete. Hence, we must re-introduce analyzability to the system.

According to the proposed method we start by identifying the structure of the system. The system now consists of six tasks out of which the three described in Section 2.7.1, i.e. the sampling task, the control task, and the actuate task, and a new task which is part of the human-machine-interface are of particular interest. Hence, those tasks will be modeled in details while the other three will, if possible, be composed into one composed task. The possibility of composing tasks is dependent on their priority levels, i.e. non of the "important" three tasks may have a priority level within the range of the composed. In our example we are lucky, the task to compose all have lower priority than the other tree. Consequently, only one composed task is required. In Table 5.2 is the identified basic structure displayed. The trigger column indicates how the execution of a task is triggered. All tasks are periodic except the

Task	Trigger	Priority
Sampling task	1000	0
Control task	2000	1
Actuate task	2000	1
HMI task	QUEUE	3
Other1	3000	4
Other2	6000	5

Table 5.2: The structure of the example system described as tasks, triggering event, and priorities

HMI task which is triggered by the event that a message is put onto the message queue named QUEUE. The priorities are still assigned according to the rate-monotonic algorithm.

Note that no relations other than the priorities are found yet. However, we can see that the HMI task is dependent on QUEUE to which we assume that other tasks in the system write. In order to construct a valid model it may be required that more details about the relations between tasks are identified, e.g. communication among the tasks. However, while constructing models it is desirable to keep them on an as high level of abstraction as possible while still producing a valid model. We will, consequently, keep our model on this rather high level in the initial modeling phase. The need for more details will be evident in the model validation phase of our method.

Having identified the structure of the system we have to populate the model with details about execution times and communication behaviors. Execution times can be based on estimations, or if the system is implemented, the execution times can be measured. In this particular example we are constructing a model in a re-engineering fashion in order to introduce analyzability. Hence, we have a system which we can measure. Note that by estimating the execution times of non-implemented functions we enforce requirements on the implementation. If the result from analyzing the system is correct, the system is also correct as long as the assumptions made in the model holds also in the implementation.

We measure the system by executing the system and monitoring execution times as well as events such as sending and receiving messages on queues. For the reason of simplicity in this example we assume that we collect only a small set of measurements from our system. The result

Task	Execution times μs	QUEUE
Sampling task	50, 55, 49, 51	0→0, 0→1, 0→0, 0→1
Control task	200, 201, 199	
Actuate task	50, 51, 54, 55, 49	
HMI task	60, 61	1→0, 1→0
Other1	31, 29, 31	
Other2	70, 70, 71, 70	

Table 5.3: The measurements sampled in the example system

from the measuring phase is a set of execution times for every task together with information about the queues that has been accessed by each task instance. In Table 5.3, the collected measurements are shown. The information about QUEUE in Table 5.3 is compiled from continuously monitoring the size of the queue for every instance of the tasks in the system. An increased queue size indicates a task putting messages on the queue, while a decrease indicates popping messages from a queue. The notation $x \rightarrow y$ in the QUEUE column indicates that the size of QUEUE was changed from x to y when a task instance was executed. From our measurements we can see that the sampling task writes to QUEUE with the probability of 0.5, and that the HMI task reads from QUEUE with the probability of 1.

We start by calculating the IECs for all tasks, i.e. creating equivalence classes for the execution times. This will filter small variations in execution times which are caused by, e.g. caches. Consequently we apply the algorithm presented in equation 5.1 to every task. The threshold is set to 1 μs. Starting with the sampling task we have the following sorted set of execution times: {49, 50, 51, 55}. Basically, the algorithm says that we shall start with the initial IEC containing the shortest execution time only, and if the distance to the second shortest is smaller than, or equal to, the threshold, it belongs to the same IEC. Starting with an initial IEC containing the execution time 49 only, the next value is 50 which is within the tolerance of the threshold. Consequently, they belong to the same IEC. The distance between 50 and 51 is also less or equal to the threshold. However, the distance between 51 and 55 is greater than 1 which results in a new IEC containing the execution time 55. We have found two IECs: (49, 50, 51) and (55). According to Definition 16 each IEC is represented by its average execution time and its probability of

Task	IEC
Sampling task	(50, 0.75), (55, 0.25)
Control task	(200, 1)
Actuate task	(50, 60), (54.5, 40)
HMI task	(60.5, 1)
Other1	(29, 0.33), (31, 0.67)
Other2	(70.25, 1)

Table 5.4: The calculated IECs for the example system

occurrence. The average of (49, 50, 51) is equal to 50 and its probability of occurrence is equal to the number of instances in the IEC divided by the total number of instances, i.e. 3/4. The second IEC only contains one instance which give an average of 55 with the probability of 1/4.

Repeating the same calculations for every task we end up with the IECs presented in Table 5.4,

Finally, before populating the model we have to calculate the temporal properties of the task composed of Other1 and Other2. Equation 5.2 suggests that the composed task shall be modeled with a period time equal to the minimum period time of the composed tasks. In Table 5.2 we can see that Other1 have a period time equal to 3000 μs, and Other2 has a period time equal to 6000 μs. Consequently, the composed task will be modeled as having a period time equal to 3000 μs. Moreover, the priority of the composed task shall, according to equation 5.5, be the highest among the composed tasks which in this case is 4.

In order to preserve the processor utilization we must normalize the IEC's for the tasks in the composed task that initially had a different period time. In this example we must normalize the IECs of the task Other2. Applying equation 5.3 gives us the new IEC for Other2 as $(\frac{3000}{6000}70.25,\ 1)$, which is equal to (35.125, 1).

The last step in order to arrive with a set of IECs for the composed tasks we must consider every possible scenario of execution of the composed task. Other1 and Other2 may execute according to two different scenarios:

1. Other1 executes for 29 μs and Other2 executes for 35.125 μs
2. Other1 executes for 31 μs and Other2 executes for 35.125 μs

Those two scenarios will result in two different IEC's for the composed

Task	IEC
Sampling task	(50, 0.75), (55, 0.25)
Control task	(200, 1)
Actuate task	(50, 60), (54.5, 40)
HMI task	(60.5, 1)
Composed	(64.125, 0.33), (66.125, 0.67)

Table 5.5: The structure of the example system described as tasks and priorities

task which are calculated with equation 5.4. $(29 + 35.125,\ 0.33 \times 1)$, and $(29 + 35.125,\ 0.33 \times 1)$. The final set of IECs for our model is shown in Table 5.5.

Below the model is shown in the ART-ML modeling language. The IECs constitute the arguments for the execute-statement in all tasks. Moreover, we have used the chance-statement to model that the sampling task is sending a message on QUEUE with the probability of 0.5. The HMITask reads the QUEUE and put the value in the local variable named *incoming* whenever a value is put in QUEUE.

```
system processor MicroController
variable input;
mailbox QUEUE 100;

  task SamplingTask
     trigger period 1000
     priority 0
  behaviour{
     execute((75,50),(25,55));
     chance(50){
       send(QUEUE, input);
     }
  }

  task ControlTask
     trigger period 2000
     priority 2
  behaviour{
     execute ((100,200));
```

```
  }

  task ActuateTask
     trigger period 2000
     priority 2
  behaviour{
     execute ((60,50),(40,54.5));
  }

  task HMITask
     trigger mailbox QUEUE
     priority 3
  behaviour{
     variable incoming;

     recv(incoming, QUEUE)
     execute ((1,60.5));
  }

  task COMPOSED
     trigger period 3000
     priority 4
  behaviour{
     execute ((33,64.125), (67,66.125));
  }

task IDLE
trigger startup
priority 255
  behaviour{
     while(1){
       execute((100,100000));
     }

  endproc
endsys
```

For now we are satisfied with a model on this high level of abstraction. The model validation activity will unveil the necessity of adding more details. Note also that the only resource we are analyzing is the CPU.

Running the model in the simulator results in response time distributions for the tasks, blocking times for the tasks, and execution times for all tasks in the model. Moreover, the size of message queues over the simulation time are generated. In Figure 5.8 through 5.11 is the response times for the tasks plotted.

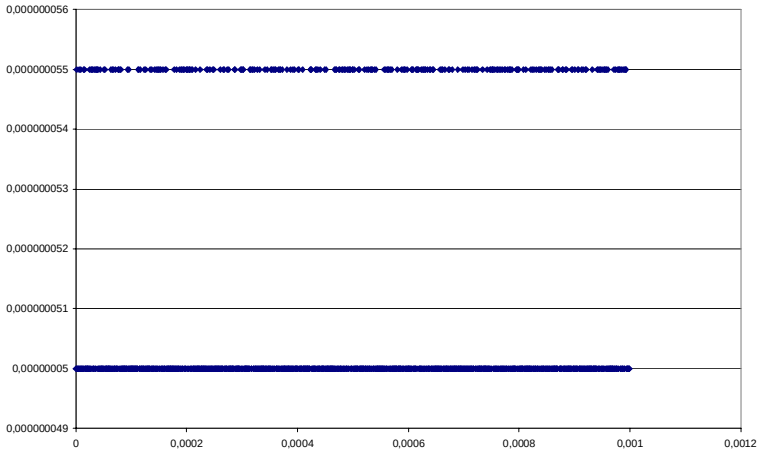

Figure 5.8: The response time distribution for the sampling task.

The size of QUEUE over time is displayed in Figure 5.12.

By comparing the output from the simulator, e.g. response times for the tasks, with response times measured in the system we can validate the correctness of our model with respect to the current system. However, as discussed in chapter 5.5, we do not know how well the model will behave when adding new functionality or maintaining existing functionality. In order to increase our confidence in the model we have to derive a set of scenarios that reflect likely changes. In this small example we assume the only likely change is adding functionality to the control task, i.e. an increase in the execution times. This is easily verified by injecting nonsense code that simulate the increased execution time. Again we can compare the response times measured in the "changed" system with the result from simulating the changed model. If they show reasonable equivalence in their behavior, we can assume that the model is reasonable correct.

Now that we have constructed a model that we believe is correct, we can start using it for analyzing its behavior, e.g. verifying the system's

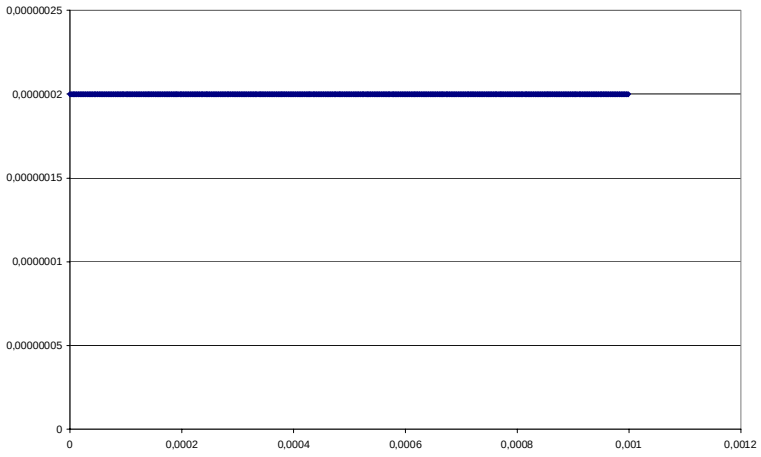

Figure 5.9: The response time distribution for the control task.

temporal requirements. In order to verify temporal requirements based on the simulation result we formalize the requirements into PPL queries. For instance, in Table 2.4 we see that the sampling task has a deadline requirement specified as 60 μs. In PPL such a requirements would be: P(SamplingTask(i).response $\leq$ 60) = 1. Such a PPL query is true if every instance of the sampling task has a response time less or equal to 60 μs. Moreover, we would like to verify the size of QUEUE never exceeds one. In PPL such a requirement is formalized as: P(QUEUE.size $\leq$ 1) = 1. Hence, the probability of the size being less or equal to one is one.

5.8 A robotic control system

The method described in this chapter was a result from studying the possibility of introducing analyzability in a large and complex real-time system. The system we have investigated is a robotic control system at ABB Robotics initially designed in the beginning of the nineties. In essence, the controller is object-oriented and consists of approximately 2 500 000 LOC divided on 400-500 classes organized in 15 subsystems. The system contains three nodes that are tightly connected, a main node that in essence generates the path to follow, the axis node, which controls each axis of the robot, and finally the I/O node, which interacts with external sensors and actuators. In this work we have studied a critical

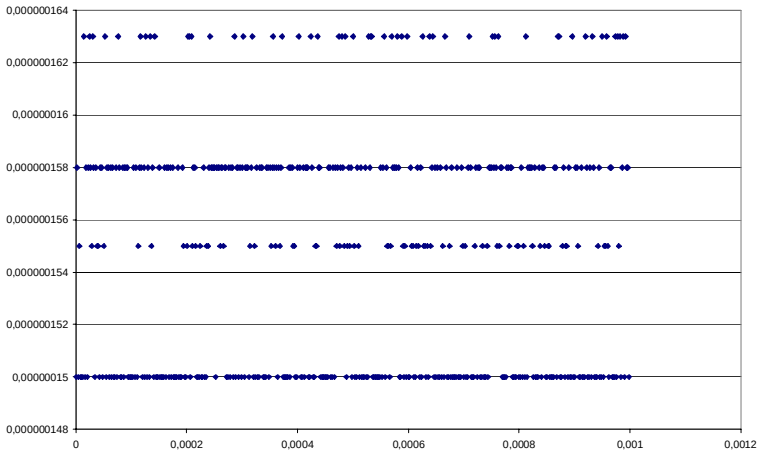

Figure 5.10: The response time distribution for the actuate task.

part with respect to control in the main node. The controller runs under the preemptive multitasking real-time operating system WxWorks.

Maintaining such a complex system requires careful analyses to be carried prior to adding new functions or redesigning parts of the system not to introduce unnecessary complexity and thereby increasing both the development and maintenance cost.

5.8.1 The model

We have modeled some critical tasks for the concrete robot system in the main node (see Figure 5.13). The main node generates the motor references and brake signals required by the axis computer. The axis node sends requests to the main node every 4'th millisecond and expects a reply in the form of motor references. This depends on three tasks: A, B, and C. B and C have high priority, are periodic, and runs frequently. A executes mostly in the beginning of each robot movement and has lower priority. The final processing of the motor references is performed by C. C sends the references to the axis node. Moreover, C is dependent on data produced by B. If the queue between them becomes empty, C cannot deliver any references to the axis node. This state is considered as a critical system state and the robot halts. A sends data to B when a movement of the robot is requested. If the queue between A and B

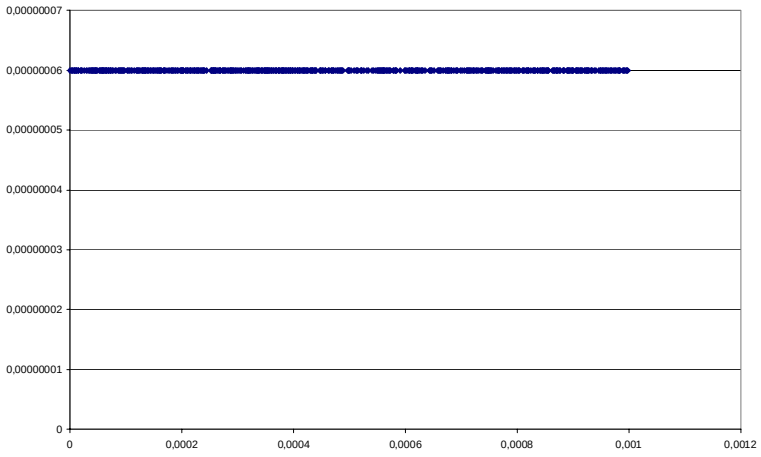

Figure 5.11: The response time distribution for the HMI task.

gets empty, the robot movement stops. In this state, B sends default references to C. The complete case study is presented in [AN02]. All comments have been removed and variable names have been changed for business secrecy reasons. The model is not complete with respect to all components in the system. All tasks, other than A, B and C, have been grouped into two dummy tasks according to the rules described in Section 5.4.2. One of the two dummy tasks has higher priority than A, and the other has lower priority than A. This is one way in which we can utilize different level of abstractions in our model.

In appendix D, the the complete model for the robot controller is presented.

5.8.2 The results

The model we made is quite an abstraction of the existing system. There were approximately 60 tasks in the system which was reduced to six in the model. This level of abstraction was selected since there were three tasks of particular interest which was modeled in detail. The rest of the tasks were modeled as two composed task. Finally, an extern subsystem was modeled as a task. The 2 500 000 LOC in the existing implementation was reduced to 200 LOC in the model.

A more detailed model would not only represent a more accurate

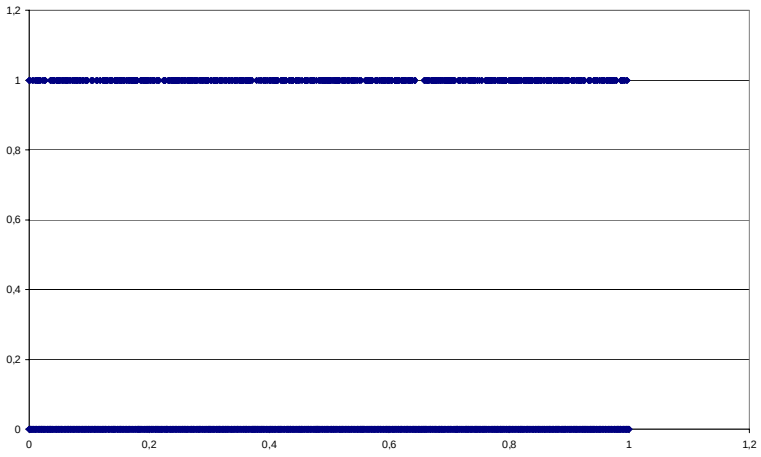

Figure 5.12: The size of QUEUE over simulation time.

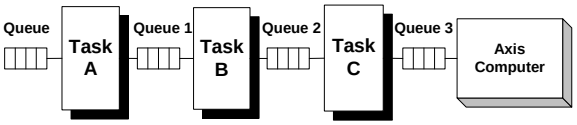

Figure 5.13: The task structure of the critical control part of the system

view of the system, it will also prune the state-space which the simulator has to consider. For instance, removing impossible system states by introducing functional dependencies among tasks will reduce the states that the simulator must explore. Thus, the simulation time is reduced.

Despite our course-grained model, the result when comparing response times produced by the simulator and the response times measured on the system is quite good. In Figure 5.14 and Figure 5.15, the response times from the simulation and the real system are plotted. The resemblance is obvious. However, methods for formally analyzing the correctness of a model should be developed as a continuation of this work.

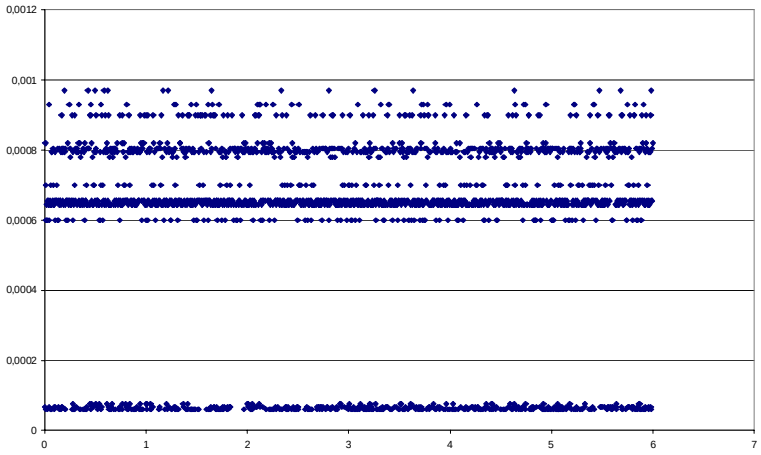

Figure 5.14: The simulated response time distribution

5.8.3 Validation results

The results from the case study indicates that we have made one valid model out of many which may be valid for the system in its current state. However, we can not assume the model to be completely correct. In order to validate the model and establish confidence in the model we must develop a set of change scenarios as described in chapter 5.5.2. Our initial list of scenarios was:

- add/remove tasks to the system
- add/remove functional behavior in an existing task
- change the behavior of existing functionality, i.e. changing execution times
- change the priority of existing tasks
- change message queue sizes
- add shared resources
- change the period time of a task
- change the triggering condition of a task

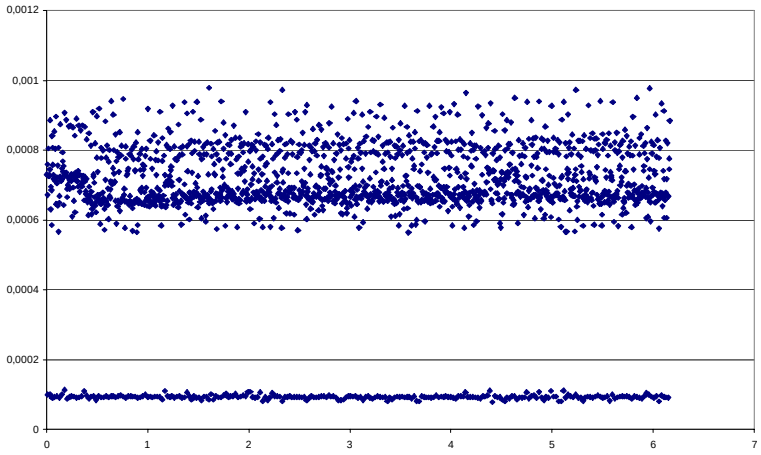

Figure 5.15: The measured response time distribution

After interviewing engineers at ABB Robotics, the list was reduced to only include the most likely changes: add/remove tasks to the system, add/remove functional behavior in an existing task, change the behavior of existing functionality, and change the priority of existing tasks.

We developed four different cases out of the scenarios:

- Case 0: No change at all
- Case 1: Add a new task called dummy with a short, oscillating execution time and low priority
- Case 2: Raise the priority of the dummy task drastically
- Case 3: Increase the period time for the dummy task and extend its execution time

We model changes in a task's functional behavior by changing its execution time.

In Figure 5.16 and Figure 5.17 we see an example of the result from the validation. We denote execution time by et and rt denotes response time. The simulated behavior in Figure 5.16 correspond quite well with the behavior measured at runtime which is shown in Figure 5.17.

In general, by observing the results we see that our model indeed capture the temporal behavior of the system quite well. The simulations

follow the measured system over the change scenarios. However, there are small differences in execution times between the model and the system. Consequently, we need to tune the execution time distributions in our model as it is too coarse grained. Moreover, we had to model yet another composed task since the priority of the dummy task is within the range of the low priority composed task.

However, the validation results is only informally reviewed. We need to develop formal mathematical methods that defines the equivalence between a simulation result an the real system behavior. This is further discussed in Section 8.2 where possible and required future work is presented.

The complete result from the validation is provided in appendix E

5.9 Comparing ART-FW with related work

None of the simulation-based approaches to analysis of a system's temporal behavior that was described in chapter 2.5.2, i.e. STRESS and DRTSS, conform completely to our assumptions and requirements. STRESS has neither support for modeling distributions of execution times or message queues nor a requirement language. DRTSS has no language where the behavior of component can be specified which is necessary for modeling executional dependencies.

Moreover, none of the above mentioned simulation-based solutions provide a property language that we have proposed for specifying probabilistic requirements.

The computational complexity of analytical methods that support probabilistic models is very high in comparison with a simulation approach. However, the analytical methods cover, as discussed in Section 5.5, the complete system behavior which is not necessarily the case with a simulation approach.

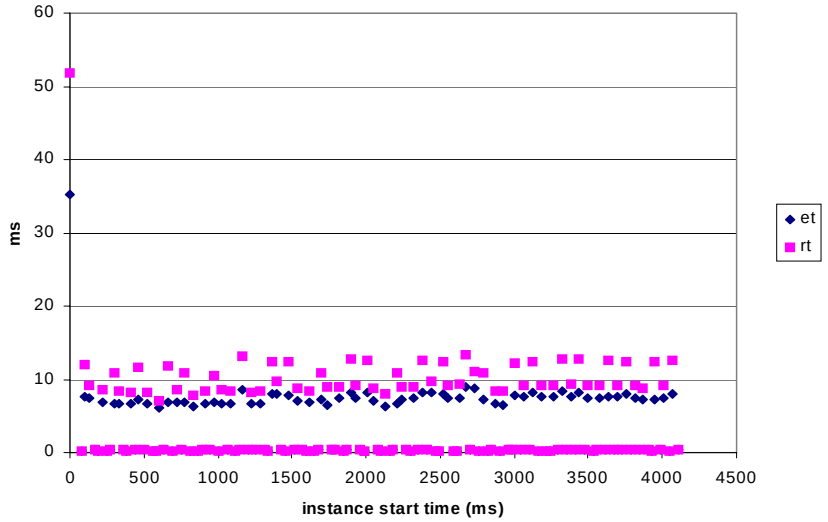

Figure 5.16: Measured execution times and response times for task A

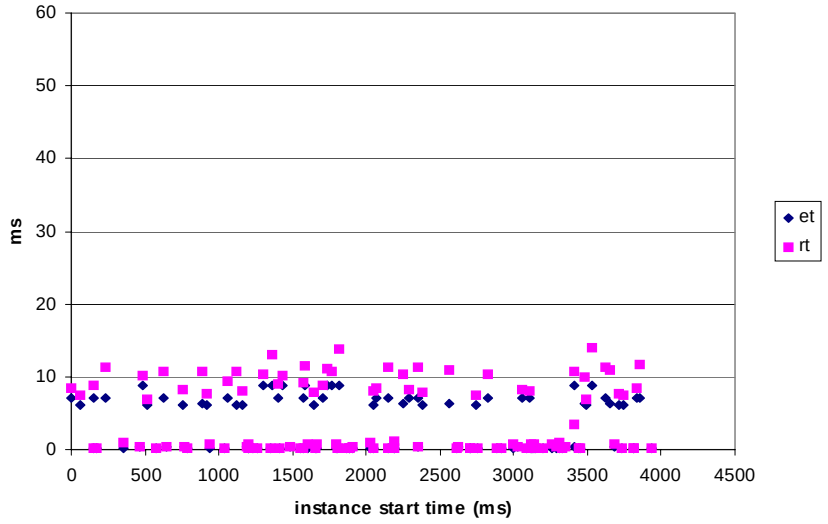

Figure 5.17: Simulated execution times and response times for task A

Chapter 6

Timed automata with tasks

In this chapter, we extend the classic model of timed automata with a notion of real time tasks. The main idea is to associate each discrete transition in a timed automaton with a task. Intuitively, a discrete transition in an extended timed automaton denotes an event releasing a task and the guard on the transition specifies all the possible arriving times of the event (instead of the so–called minimal inter-arrival time). This yields a general model for hard real-time systems in which tasks may be periodic and non-periodic.

We show that the schedulability problem for the extended model can be transformed to a reachability problem for standard timed automata and thus it is decidable. This allows us to apply model-checking tools for timed automata to schedulability analysis for event-driven systems. In addition, based on the same model of a system, we may use the tools to verify other properties (e.g. safety and functionality) of the system. This unifies schedulability analysis and formal verification in one framework. We present an example where the model–checker UPPAAL is applied to check the schedulability and safety properties of a control program in automotive applications.

6.1 Introduction

In Chapter 5 we presented a probabilistic modeling language which was analyzed using simulations. Furthermore, we have discussed the confidence that we can have in the analysis results when we use simulation contra analytical methods given that we have a valid model (see Figure 5.5). We know that the existing analytical methods for analyzing the temporal behavior of a real-time system provide a safe result given

that the model is correct. However, the models in existing analytical methods are often too simple in order to capture a systems temporal behavior. This was the motivation for developing the ART-ML modeling language. However, we can never be sure that we capture the worst-case scenario by simulation. Thus, the ideal framework would include a rich modeling language that allow us to specify execution times as intervals and in which we can model dependencies among tasks, and still allow us to mathematically analyze the models. Our solution is to use timed automata [AD94].

The traditional approach to the development of hard real-time system is often based on scheduling theory. There are various methods [But97, LL73, Der74] e.g. rate monotonic scheduling, which have been very successful for the analysis of time-driven systems as tasks are *periodic*. To deal with *non-periodic* tasks in event–driven systems, the standard method is to consider non-periodic tasks as periodic using the *minimal* inter-arrival times as *task periods*. Clearly, the analysis result based on such a task model would be pessimistic in many cases, e.g. a task set which is schedulable may be considered as non-schedulable as the inter-arrival times of the tasks may vary over time, that are not necessary minimal.

In recent years, in the area of formal methods, there have been several advances in formal modeling and analysis of real time systems based the theory of timed automata due to the pioneering work of Alur and Dill [AD90]. Notably, a number of verification tools have been developed (e.g. KRONOS and UPPAAL [DY95, BLL+96]) in the framework of timed automata, that have been successfully applied in industrial case studies (e.g. [BGK+96, LPY97b, LPY98]). Timed automata have proved expressive enough for many real-life examples, in particular, for event-driven systems. The advantage with timed automata is that one may specify very relaxed timing constraints on events (i.e. discrete transitions) than the traditional approach in which events are often considered to be periodic. However, it is not clear how the model of timed automata can be used for schedulability analysis. In this chapter, we present an extended version of timed automata with real-time tasks to provide a model for event-driven systems. We show that the extended model can be used for both schedulability analysis and verification of other properties, e.g. safety and liveness properties of timed systems. This unifies schedulability analysis and formal verification in one framework.

The main idea is to associate each discrete transition in a timed automaton with a task (or several tasks in the general case). A task is

assumed to be an executable process with two given parameters: its worst execution time and deadline. Intuitively, a discrete transition in an extended timed automaton denotes an event releasing a task and the guard (clock constraints) on the transition specifies all the possible arrival times of the associated task. Whenever a task is released, it will be put in the scheduling queue for execution. We assume that the tasks will be executed according to a given scheduling strategy e.g. earliest deadline first. Then a delay transition of the timed automaton corresponds to the execution of the task with earliest deadline and idling for the other waiting tasks.

Thus, the sequences of discrete transitions of an extended timed automaton will correspond to the sequences of *arrivals* of non-periodic tasks. We say that such a sequence of tasks is schedulable if all the tasks can be executed within their deadlines. Naturally an automaton is *schedulable* if all the task sequences are schedulable. We shall show that under the assumption that the tasks are non-preemptive, the schedulability problem can be transformed to a reachability problem for ordinary timed automata and thus it is decidable. This allows us to apply model-checking tools for timed automata to schedulability analysis for event-driven systems. We present an example where the model–checker UPPAAL is applied to check the schedulability and safety properties of a control program in control applications.

6.2 Related work

Related work is provided by Fredette and Cleaveland which have defined a timed process algebra called Real-Time Specification Language (RTSL) [FC93]. RTSL is based on the syntax and semantics of CCS [Mil89], and ACP. Furthermore, they use reachability analysis to verify whether the system is schedulable or not. In RTSL there are predicates which raise exceptions if, for instance a process misses its deadline. By verifying that no exception states are reachable, they can decide if a system of processes is schedulable.

6.3 Timed automata with real-time tasks

The theory of timed automata was first introduced in [AD90] and has since then established as a standard model for real time systems. We first give an brief review to fix the terminology and notation and then

present an extended version of the model with tasks.

6.3.1 Timed automata

A timed automaton is a standard finite-state automaton extended with a finite collection of real-valued clocks. The transitions of a timed automaton are labeled with a *guard* (a condition on clocks), an *action*, and a *clock reset* (a subset of clocks to be reset). Intuitively, a timed automaton starts execution with all clocks set to zero. Clocks increase uniformly with time while the automaton is within a node. A transition can be taken if the clocks fulfill the guard. By taking the transition, all clocks in the clock reset will be set to zero, while the remaining keep their values. Thus transitions occur instantaneously. Semantically, a state of an automaton is a pair of a control node and a *clock assignment*, i.e. the current setting of the clocks. Transitions in the semantic interpretation are either labeled with an action (if it is an instantaneous switch from the current node to another) or a positive real number i.e. a time delay (if the automaton stays within a node letting time pass).

For the formal definition, we assume a finite set of alphabets Act for actions and a finite set of real-valued variables C for clocks. We use a, b etc to range over Act and X_1, X_2 etc. to range over C. We use $\mathcal{B}(C)$ ranged over by g and later by ϕ etc, denote the set of conjunctive formulas of atomic constraints in the form: $X_i \sim m$ or $X_i - X_j \sim n$ where $X_i, X_j \in C$ are clocks, $\sim \in \{\leq, <, \geq, >\}$, and m, n are natural numbers. The elements of $\mathcal{B}(C)$ are called *clock constraints*.

Definition 22. *A timed automaton over actions Act and clocks C is a tuple* $\langle N, l_0, E\rangle$ *where*

- N *is a finite set of nodes,*
- $l_0 \in N$ *is the initial node, and*
- $E \subseteq N \times \mathcal{B}(C) \times Act \times 2^C \times N$ *is the set of edges.*

When $\langle l, g, a, r, l'\rangle \in E$, *we write* $l \xrightarrow{g,a,r} l'$. □

Formally, we represent the values of clocks as functions (called clock assignments) from C to the non–negative reals $\mathbb{R}_{\geq 0}$. We denote by $\mathcal{V}$ the set of clock assignments for C. A semantical *state* of an automaton is now a pair (l, u), where l is a node of the automaton and u is a clock assignment and the semantics of the automaton is given by a transition system with

the following two types of transitions (corresponding to delay–transitions and action–transitions):

- $(l,u) \xrightarrow{d} (l,u+d)$
- $(l,u) \xrightarrow{a} (l',u')$ if $l \xrightarrow{g,a,r} l'$, $u \in g$ and $u' = [r \mapsto 0]u$

where for $d \in \mathbb{R}_{\geq 0}$, $u+d$ denotes the clock assignment which maps each clock X in C to the value $u(X)+d$, and for $r \subseteq C$, $[r \mapsto 0]u$ denotes the assignment for C which maps each clock in r to the value 0 and agrees with u over $C \backslash r$. By $u \in g$ we denote that the clock assignment u satisfies the constraint g.

6.3.2 Extended timed automata with tasks

We shall view a timed automaton as an abstract model of a running process. The model describes the possible events (alphabets accepted by the automaton) that may occur during the execution of the process and the occurrence of the events must follow the timing constraints (given by the clock constraints). But the model gives no information on how these events should be handled. In many cases, for example in a control system, when an external event occurs, some computation must be performed to handle the event. A more concrete example is an interrupt handling system. Whenever an interrupt signal occurs, the associated interrupt handling program will be executed.

Now, assume that each action symbol in a timed automaton is associated with a program called *task*. Let P ranged over by p etc, denote the set of tasks. We further assume that the *worst case execution time* and *hard deadline* of the tasks in P are known. We shall use clock constraints to specify the arrival times of the tasks. Thus, each task p in P is characterized as a pair (c,d) of natural numbers with $c \leq d$ where c is the execution time of p and d is the relative deadline for p.

The deadline d is a relative deadline meaning that when task p is released, it should finish within d time units.

Definition 23. *An extended timed automaton with tasks (TAT), over actions Act, clocks C and tasks P is a tuple* $\langle N, l_0, E, T \rangle$ *where*

- $\langle N, l_0, E, T \rangle$ *is a standard timed automaton,*
- $T : Act \hookrightarrow P$ *is a partial function assigning tasks to actions.*

□

Semantically, an extended automaton may perform two types of transitions just as an ordinary timed automaton. In addition, an action transition will release a new instance of the task associated with the action. Assume that there is a queue holding all the task instances generated by action transitions and ready to run. The queue corresponds to the ready queue in an operating systems. A semantic state of an extended automaton is a triple consisting of a *node* (the current control node), a *clock assignment* (the current setting of the clocks) and a *task queue* (the current status of the ready queue).

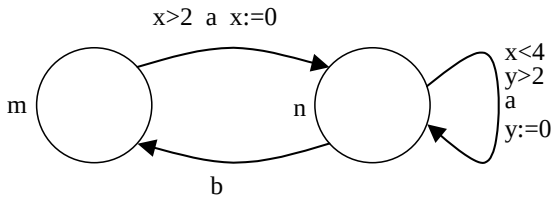

Figure 6.1: An example timed automaton with tasks

Consider the automaton of Figure 6.1. Let $p1$ and $p2$ be tasks handling the interrupt signals a and b respectively. Assume that the initial state is $(m, [x = 0, y = 0], [])$ where the clocks are 0 and the task queue is empty. Then the automaton may demonstrate the following sequence of transitions:

$$\begin{aligned}
(m, [x = 0, x = 0], []) &\xrightarrow{3} (m, [x = 3, y = 3], []) \\
&\xrightarrow{a} (n, [x = 0, y = 3], [p1]) \\
&\xrightarrow{a} (n, [x = 0, y = 0], [p1, p1]) \\
&\xrightarrow{3} (n, [x = 3, y = 3], [p1, p1]) \\
&\xrightarrow{a} (n, [x = 3, y = 0], [p1, p1, p1]) \\
&\xrightarrow{1} (n, [x = 4, y = 1], [p1, p1, p1]) \\
&\xrightarrow{b} (m, [x = 4, y = 1], [p1, p1, p1, p2]) \\
&\ldots
\end{aligned}$$

Note that several instances of the same task may be released. However, the number of copies may be bounded by the clock constraints. For

example, in state $(n, [x = 4, y = 1], [p1, p1, p1])$, no more instance of $p1$ will be released because the clock values will not satisfy the constraint $x < 4$ and $y > 2$, but an instance of $p2$ may be released by the b-transition (which has no timing constraint).

In the above example, we have only shown that the task queue is growing due to action transitions. Now we discuss the effect of delay transitions on task queue. We shall see that the queue will be shrinking due to delay transitions. Let $p1 = p2 = (2, 8)$ i.e. the computation time of both $p1$ and $p2$ is 2 and the deadline is 8. We assume that there is a processor running the task instances according to a certain scheduling strategy. A delay transition with t time units is to execute the tasks in the queue with t time units. After the transition, a task will be removed from the queue (shrinking) if its computation time becomes 0 and the deadlines of all tasks in the queue will be decreased by t (since time has progressed by t). Now we have a precise description on the state changes for the above transition sequence:

$$
\begin{aligned}
(m, [x = 0, x = 0], []) &\xrightarrow{3} (m, [x = 3, y = 3], []) \\
&\xrightarrow{a} (n, [x = 0, y = 3], [(2, 8)]) \\
&\xrightarrow{a} (n, [x = 0, y = 0], [(2, 8), (2, 8)]) \\
&\xrightarrow{3} (n, [x = 3, y = 3], [(1, 5)]) \\
&\xrightarrow{a} (n, [x = 3, y = 0], [(1, 5), (2, 8)]) \\
&\xrightarrow{1} (n, [x = 4, y = 1], [(2, 7)]) \\
&\xrightarrow{b} (n, [x = 4, y = 1], [(2, 7), (2, 8)]) \\
&\ldots
\end{aligned}
$$

More precisely we have the following assumptions on the underlining execution model:

1. A ready queue holding the task instances released and waiting for execution. A task instance will be removed from the queue when its computation time becomes 0.

2. An on-line scheduler Sch sorting the queue according to a given scheduling strategy. It will report $\perp$ if the queue becomes non-schedulable when a new task instance is added.

3. A single processor executing the tasks according to the ordering of the queue. It will always execute the task in the first position. The tasks are executed non-preemtive.

Further we use $\mathsf{Run}(q,t)$ to denote the resulted task queue after t time units of execution. The meaning of $\mathsf{Run}(q,t)$ should be obvious. For example, let $q = [(2,7),(2,8)]$ and $t = 3$ then $\mathsf{Run}(q,t) = [(1,5)]$ in which the first task is finished and the second has been executed for 1 time unit. Now we are ready to present the transitional rules for extended timed automata.

Definition 24. *The semantics of an extended automaton is a transition system defined by the following transition rules (corresponding to release of new task and execution of existing tasks):*

- $(l,u,q) \xrightarrow{a} (l',u',\mathsf{Sch}(q'))$ *if* $l \xrightarrow{g,a,r} l'$, $u \in g$, $u' = u[r \mapsto 0]$, *and* $q' = q :: T(a)$
- $(l,u,q) \xrightarrow{t} (l,u+t,\mathsf{Run}(q,t))$

We shall write $(l,u,q) \longrightarrow (l',u',q')$ *if* $(l,u,q) \xrightarrow{a} (l',u',q')$ *for an action* a *or* $(l,u,q) \xrightarrow{d} (l',u',q')$ *for a delay* d. □

Finally, to handle concurrency and synchronization, parallel composition of extended timed automata may be introduced in the same way as for ordinary timed automata (e.g. see [LPY95]) using the notion of synchronization function [HL89]. For example, consider the parallel composition $A||B$ of A and B over the same set of actions Act. The set of nodes of $A||B$ is simply the product of A's and B's nodes, the set of clocks is the (disjoint) union of A's and B's clocks, the edges are based on synchronizable A's and B's edges with enabling conditions conjuncted and reset-sets unioned. Note that due to the notion of synchronization function [HL89], the action set of the parallel composition will be Act and thus the task assignment function for $A||B$ is the same as for A and B.

6.4 Schedulability analysis as reachability analysis

Traditionally, the temporal attributes for a real-time computer systems are derived from their environment, e.g. period times, etc. These attributes are used for constructing a model of the system in terms of its

temporal behavior. Such a temporal model is often called a task model, which is used to verify whether the system is schedulable or not, but other properties such as functional and safety properties can not be verified based on such a model. In our approach, we may construct a model for the whole system including the environment and tasks in the control system. The parallel composition of these models give us the possibility of not only verifying temporal constraints, but also its other aspects such as synchronization between tasks and simple computations within tasks etc.

Normally, a system is said to be schedulable if all tasks can always be executed within their deadlines, i.e. no deadlines are violated. The objective of the schedulability analysis is to verify that there are no violation of deadlines in all situations where the system may evolve to. Now we formalize the notion of schedulability for extended timed automata.

Definition 25. *An extended timed automaton A is non-schedulable if it may reach a non-schedulable state, that is:* $(l_0, u_0, q_0) \longrightarrow^* (l, u, \perp)$ *where* (l_0, u_0, q_0) *is the initial state of A, and* $\longrightarrow^*$ *is the transitive closure of* $\longrightarrow$. *We say that A is schedulable if and only if all its reachable states are schedulable.* □

Thus, the schedulability of extended automata can be checked by reachability analysis, to prove that $(l, u, \perp)$ is not reachable in the automaton. However, it is not obvious that the reachability problem for extended automata is decidable. In fact, the decidability of this problem is closely related to the preemptiveness of the tasks P.

Theorem 1. *The problem of checking schedulability for extended timed automata over non-preemptive tasks P is decidable.*

Proof idea: It is based on the fact that the problem of schedulability checking for extended timed automata can be transformed to the reachability problem for standard timed automata, which is known to be decidable [Alu93]. See Section 6.4.1 for details on the transformation. □

6.4.1 Transformation from TAT to ordinary timed automata

The idea is to construct a timed automaton simulating a ready queue and a scheduler that code all possible scenarios of the system described by a TAT, including the tasks in the queue and schedules. For example, consider the temporal attributes of the two tasks τ_a and task τ_b, where

τ_a had a worst-case-execution time (WCET), of 4 time units (tu), and a deadline (d), of 7 tu. The second task τ_b has a WCET of 3 tu and a deadline of 5 tu.

Intuitively for a system to be schedulable, the ready queue can contain only a finite number of task instances. More precisely, there can only be MNT_i instances of task i, where MNT_i is given by:

$$MNT_i = \left\lfloor \frac{d_i}{c_i} \right\rfloor \tag{6.1}$$

where d_i denotes the deadline for task i and c_i denotes the computation time.

By calculating the maximum length of the ready queue, we know that to be schedulable, the queue in our example can only contain one instance of τ_a and one instance of τ_b. If at any time point, there are more than one instances of a particular task in the ready queue waiting for execution, we know for sure that the system is non-schedulable and the error state should be reached. This ensures a finite number of states in our model of the scheduler and the ready queue. Now, we use the above example to present the algorithm for constructing the scheduler and queue automaton, which can be generalized easily to the general case.

1. Create three different nodes, one node in which the ready queue is empty, one for which there exists task instances in the ready queue and, finally an error node.

2. Create transitions from the empty node to the running node, one for every action associated with a task. Furthermore, tasks can arrive while in the run node, consequently we need one transition from run back to run for every possible task instance as well. In order to keep track of every new task instance, an unique semaphore for every instance is introduced (denoted as taska and taskb in Figure 6.2). We also need an unique deadline clock for every instance in order to know which task to execute and to detect deadline violations.

3. According to EDF, the task having least time left until its deadline should be executed. For all possible task instances, create a transition from run to run which compares its relative deadline to all the other ready tasks. In our example τ_a should be executing

if $7 - d_a < 5 - d_b$, and τ_b if $5 - d_b < 7 - d_a$ where d_a and d_b are the deadline clocks. In order to keep track of execution time of the running task, a clock is reseted on every release of a task. In our example, this clock is denoted as c. Furthermore, as we consider the non-preemptive case, no task can start to execute while another task already is executing. Thus we need a semaphore to know whether the processor is idle or not (denoted r in Figure 6.2).

4. Introduce one transition from run to run for every possible instance which terminates the task whenever c becomes equal to its specified execution time and its deadline clock is less or equal to its specified deadline. Termination is modeled by resetting the instance semaphore.

5. If ready queue gets empty, i.e. no tasks instances are present in the queue a transition to the empty node should be taken.

6. For each possible task instance we introduce a transition from run to error if:

   - An action A occurs, making the number of instances of A exceeding MNT_a
   - The executing task has overrun its deadline
   - A task pending for execution in the ready queue has exceeded its deadline

Figure 6.2 shows the result from transforming our example system shown. This is an ordinary timed automata for which decidability has been proven in [Alu93].

For the general case, the scheduler and queue automata is illustrated in Figure 6.3 where q denotes a queue, r is the executing task, c measures how long time the executing task has been running and d(i) is a vector keeping track of the time elapsed since the tasks entered the ready queue. C(i) is a vector holding the worst case execution time of all tasks. Both are vectors are finite as been discussed above. Moreover, the function sch() returns the instance among all tasks residing in the queue having least time left until its deadline. Task i is returned by sch() if the predicate $\bigwedge_{\forall m \in q: m \neq i} d(i) - d(m) \leq D_i - D_m$ is true, where D_i denotes the relative deadline specified for task i.

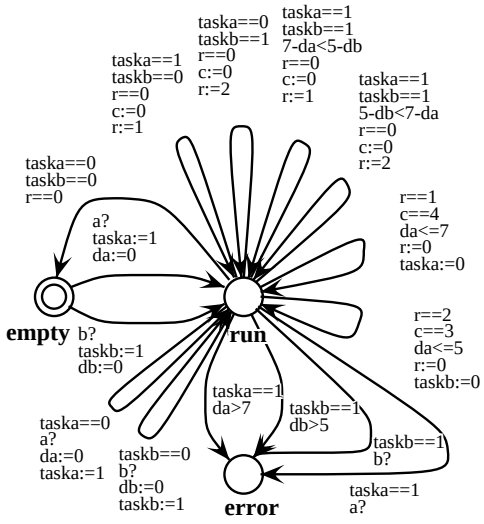

Figure 6.2: A model of the ready queue and the scheduler using ordinary timed automata

6.5 A case study with Uppaal

Uppaal is a model-checker for timed automata [LPY97a]. As shown in the previous section, the scheduler and ready queue can be modeled as an ordinary timed automaton. In this section, we present an example showing how to use Uppaal for schedulability checking.

Our example system is an event-driven application controlling the speed of the shaft in a turning lathe. The objectives of the formal verification is to verify that the system is schedulable and the safety requirement that the engine is not turned on by the control task while the emergency stop is active. An event reports the current speed of the shaft and a control task is checking that the speed is within the speed limits (in our example speed=3). If the speed is to high (over 3), the engine is turned off and if the speed is to low (below 3), the engine is turned on. There is also an emergency stop function which is implemented in software. The setup is shown in Figure 6.4.

As shown in Figure 6.4, the parts belonging to the systems environment are the shaft having an optical sensor generating an event on every complete revolution, the emergency stop button having two states: up or down and the engine, being either on or off. Consequently, we have to

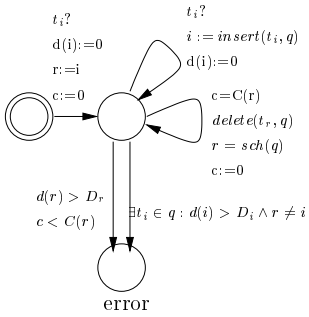

Figure 6.3: A general model of the scheduler using ordinary timed automata

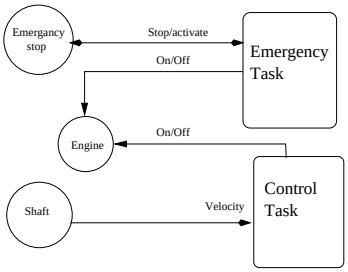

Figure 6.4: The setup for our example system

model all these parts as a network of TATs. Moreover, we have two software tasks, the control task and the emergency stop handler. These parts also have to be modeled in TATs belonging to the network constituting the complete system.

6.5.1 Modeling the system

We start by modeling the environment, i.e. the shaft, the emergency stop button and the engine. This can for instance be done as shown in Figure 6.5, 6.6.

If the engine is on, the shaft makes a complete revolution in between 4-8 time units, and an event is generated every time the optical sensor detects a complete revolution.

Next to model is the emergency stop handler and the control task. The control task has a calculated WCET of 2 tu and a hard deadline of

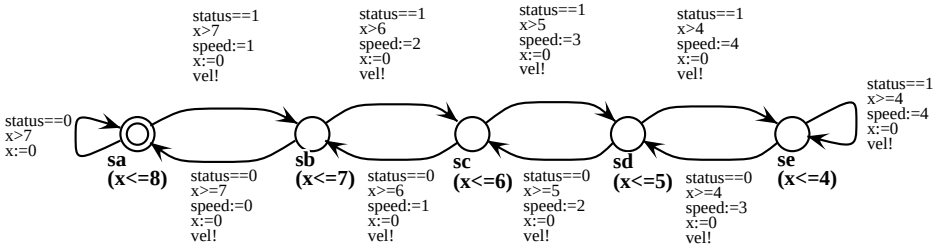

Figure 6.5: A model of the shaft in timed automata

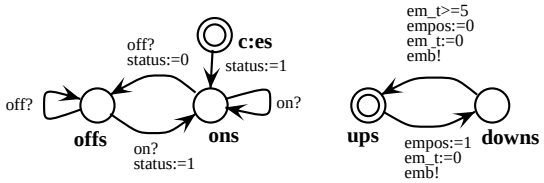

Figure 6.6: A model of the engine and the emergency stop button

3 tu (Figure 6.8).

As for the control task, a deadline and a WCET must be specified for the emergency stop handler. According to our imagined requirement specification, it must respond within 2 tu, i.e. it has a deadline at 2 tu. The WCET estimation result in a WCET of 1 tu (see Figure 6.7). Furthermore, two subsequent activations/deactivation of the emergency stop can not be less than 5 tu in between. This gives us a minimum inter-arrival time for the emergency stop handler of 5 tu.

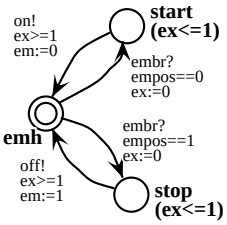

Figure 6.7: A model of the emergency handler in timed automata

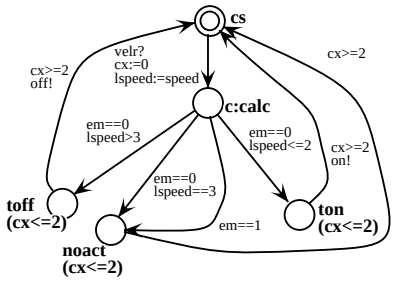

Figure 6.8: A model of the control task in timed automata

The model of the scheduler is omitted. However, this process will be generated automatically by UPPAAL according to the algorithm given in Section 6.4.1 and will be invisible for the designer.

6.5.2 Verifying schedulability and safety

We use model checking and reachability analysis on our network of TAT for this purpose. UPPAAL uses a timed CTL language for specifying properties to verify. To verify that the system is schedulable, we must show that the error state is never reachable. We will use the *always* predicate in our example as *always* not α is equivalent to *never*. This property is specified as shown in the formula below, *scheduler.error* means the state error in the process named scheduler:

$$\forall\Box not\ scheduler.error \tag{6.2}$$

For the safety property we need to verify that the system never reach a state where the control task is in position to turn the engine on while the emergency stop has been activated. For our model, such an expression looks like the formula given below:

$$\forall\Box not(control\_task.ton\ and\ em = 1) \tag{6.3}$$

First we will verify the schedulability property. As a result UPPAAL tells us that the property is not satisfied by giving a counter example. Consequently, the system is not schedulable. In order to obtain a schedulable system, the temporal constraints on the tasks have to be modified.

The counter example given by UPPAAL, shows that the emergency handler task misses its deadline if this event happens just after the control task has been invoked. By changing the deadlines for the control task and the emergency stop handler to 4 tu, the system becomes schedulable. This is verified by the same property, but with an updated scheduler model. The model of the scheduler must be updated since now there can exist two instances of the control task and four instances of the emergency handler simultaneously in the ready queue.

Next to verify is our safety property, i.e. the control task should not be able to turn the engine on as long as the emergency stop is activated. In this case UPPAAL reports that the property is satisfied and consequently, the safety requirement is fulfilled.

It is of course possible to verify other functional properties. For instance, we can verify that the shaft eventually will rotate with the set value. In our model, the set value is the speed of 3, i.e. *the speed is eventually equal to 3*. The corresponding formula given in UPPAAL logic is:

$$\exists \Diamond speed = 3 \tag{6.4}$$

6.6 Comparing TAT and RTSL

The major difference between RTSL and our approach is the modeling language. Instead of using process algebra, we use timed automata which we believe is easier to use when designing complex systems.

6.7 Discussion

The results presented in this chapter has been adopted in Times which is a tool for modeling, analysis and implementation of embedded systems [AFM+02].

At a first glimpse, TAT seems like the perfect solution. We have a rich modeling language that allow us to model, e.g. dependencies among tasks, and still the analysis is mathematically precise and correct. Moreover, TAT makes it possible to analyze other properties mathematically, e.g. safety properties. However, in the context of large, complex systems this approach does not scale very well. The scalability problem is not related to the modeling language and the reachability analysis as

such, but rather a problem of state-space explosion, i.e. representation of state-space, and algorithms for performing automatic verification, i.e. exploring the state-space. Nevertheless, computation power is likely to increase which is in favor for such an approach.

Chapter 7

Conclusions

Architectural modeling and analysis of large and complex real-time systems is necessary in order to reduce the cost for maintenance and, hence, extending the lifetime of an implementation. Especially important is the long lifetime in a software reuse effort such as a software product line. The payoff of time and money invested in developing reusable software and architectures are dependent on the number of instances in which it is reused. In this thesis we present methods that aim at increasing maintainability by providing analytical models of a system. Such models provide the possibility of analyzing the effects that certain maintenance activities may have on different quality properties of the system. In this particular case we are focusing on timing properties.

Besides providing general discussions about modeling and analysis of software architectures and software product-lines we have presented three different specific methods for modeling and analysis of complex real-time systems: a component model with analytical interfaces (ReFlex), a probabilistic approach to modeling and analysis (ART-FW), and a formal approach where ordinary timed automata is extended with a notion of real-time tasks (TAT).

The results presented in this thesis is the result of studying two different large and complex real-time systems: a control system for construction vehicles at Volvo Conctruction Equipment (VCE), and a control system for industrial robots (ABB Robotics). The construction vehicles system and the robot system are both currently based on a product line approach.

The purpose of studying the two product lines, i.e. the VCE and ABB Robotics, was to investigate how to analyze the impact of alter-

ing a system in the temporal domain, e.g. add new features or altering the behavior of existing features. We refer to this as maintenance. A necessity for being able to analyze a system is that there exist analyzable models of the system, i.e. analytical models. Based on the case studies we have found two different approaches to the construction and maintenance of the analytical models. We refer the two different approaches to constructing an analytical model as by-construction and by re-engineering.

If a system is constructed and maintained on the basis of the analytical model, i.e. models are part of the implementation, we say that the model is constructed by construction. In the constructive approach, the model is a product of the implementation activity. On the other hand, if the models are constructed and populated by measuring an already existing implementation, it is constructed by re-engineering. The re-engineering approach is a continuous activity. To use this approach successfully the analytical model must be maintained during the life cycle of the system.

There are several possible conclusions that can be drawn from a re-engineering initiative, i.e. identifying the existing system's architecture and measuring its behavior in order to introduce/re-introduce analyzability. We can re-design the complete system and, for instance, use a component model that facilitate analytical interfaces as the ReFlex component model proposed in this thesis. Such a solution is costly and associated with many risks. The risks are typically of economical nature, e.g. investing man-hours in projects that do not increase the functionality. Nevertheless, it can be feasible and economically justifiable to take this approach. By re-designing the system's architecture only we can reduce the risk as a considerable part of the system's implementation already exists. Furthermore, such an approach will not introduce new technologies which always constitutes a risk. These two approaches are both *intrusive* meaning that it will change the implementation of the system.

The modeling frameworks ART-FW and TAT offer a non-intrusive solution which eliminates the risks discussed above. The implementation is untouched but still we have the possibility to analyze and foresee the impact of altering the system. However, the risk associated with a non-intrusive approach is that the model is not kept consistent with the implementation as the system evolves which makes the models obsolete.

Generally, we can not say that any approach, i.e. intrusive or non-intrusive, is to prefer. It must be investigated from case to case , i.e.

from system to system, what makes the best solution in order to handle system complexity.

We think that the contributions presented in this thesis are relevant and provides realistic and applicable solutions to the problem of analyzing the effects of maintaining large and complex real-time systems. The ability to analyze the software is provided by having relevant and correct models of the software architecture as discussed in Chapter 2 and Chapter 3. The contributions in this thesis focuses on modeling and analysis of systems' structure and temporal behavior. Hence, architectural descriptions with temporal aspects has been developed.

The ReFlex component model provides the mechanisms necessary for building and maintaining a component-based product line architecture. No component model in the related work that we have studied offers the same flexibility. Moreover, we show how to apply the concept of PECT in order to reason about the impact of maintaining components in a product line perspective. Moreover, ReFlex is designed with resource constrained systems that have requirements on predictability with respect to temporal behavior in mind. Consequently, large and complex component technologies such as .NET and CORBA can not be used.

ART-FW provides a framework in which we can measure a system, make a model of it, simulate the model, and analyze the simulation results with respect to temporal requirements. Compared to the existing simulators we offer the possibility of making probabilistic models that enables us to construct more realistic models of complex real-time systems. Moreover, non of the existing simulators have a requirements language such as PPL. Compared to the existing analytical methods that facilitate probabilistic models we think that a simulation based approach is more manageable in terms of computational complexity. However, a very important area which we only touched upon in this thesis is model validation, i.e. how can we be sure that a model indeed is a correct model of the software system. We gave some suggestions on how to increase the confidence in a model but there are many open questions that are subjects for future work. Moreover, we would like to have a modeling language that can model complete product lines, i.e. all products and their components. Such a language would enable us to analyze the effects of changing one product with respect to the complete product line.

Chapter 8

Future work

8.1 ReFlex

As future work we will implement ReFlex and integrate it into a framework for designing software product-line architectures. The actual syntax has not yet been decided. However, we will investigate the possibility to use the industrial de facto standard UML [PR99]. Being a language for specification of embedded real-time systems, we must look into the problem of specifying temporal and resource constraints, e.g. memory consumption, CPU consumption. Preferably, we would like to assign budgets to components that all parts of a component that is part of its execution in a particular product instance must adhere to.

Moreover, more property theories can be defined. As a base for this work we will provide a tool for specifying and analyzing systems based in the component model. Such a tool should support the framework ideas. Thus, it must provide means for extending the component's analytical model with new analytical properties and to define new property theories on assemblies.

Another application of ReFlex with its analytical models is a strategy for handling dynamically configurable systems. Consumer-products such as cellular phones may be configured/customized by the consumer himself, e.g. by downloading a new feature to the phone. Thus, the end-customer assembles products based on a product line architecture. By distributing the analytical model together with the constructive software, the system itself can predict the impact the new feature will have on the system. Based on such an analysis the system can decide whether to accept the new product as valid or not.

8.2 ART-FW

Both ART-ML and PPL are still prototypes, thus many improvements of the method, the language, and the tools are possible. Currently we are expanding ART-ML to also support the modeling and analysis of multi-processor systems and modeling of complete product lines.

8.2.1 ART-ML and product lines architectures

Currently the ART-ML framework can only model and analyze single products. We would like to extend the modeling language with constructions that allow us to describe complete product lines. Typically, we would like to analyze the impact of adding new features or altering the behavior of a component with respect to all products in a product line.

As introduced in Section 3.3.1, we divide a product model into three different levels of abstraction: the feature view, the component view, and the implementation view. Each of the different abstraction levels provides means for analyses. Analyses are performed on an intra-view basis even though information on a higher level of abstraction may be constructed based on information from a lower level (see Figure 8.1). As the different views provide information regarding the system with different level of abstraction, the analyses performed based on information from the feature view are more coarse-grained than those on the component level of abstraction. Nevertheless, analysis based on information from the feature view is not inaccurate and can be made earlier in the development phase.

Information flows through the abstraction levels both from the feature view and down, and from the implementation view and up depending on the current phase of the development procedure. The different views are depicted in Figure 8.1 together with a work-flow and brief descriptions of the activities performed in the procedure. Note that the evolutionary development may add new resources and new features may share components with other already existing features.

On the feature level of abstraction the analytical model specifies a real-time system in terms of resource utilization. Resources are CPU capacity, communication bandwidth and memory. Initially, the utilization is specified based on estimates and constraints given as non-functional requirements. The feature utilization may not be evident until the first feedback iteration is completed. Thus, when designing a system from scratch, the utilization part of the analytical model of the feature view

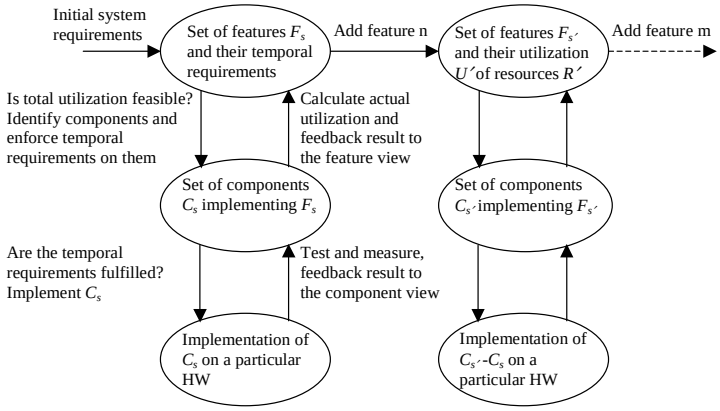

Figure 8.1: The different levels of abstraction in the analytical model

may not be populated until the first iteration through the component view and the implementation view is completed. However, the systems we have been studying are large and complex, they have long service times. Consequently, the cost of the initial development may be only a small fraction of the total cost. Maintenance is by far the major part of the cost as we include in maintenance such activities as error corrections, improving existing features, as well as implementing completely new features. Thus, the model we proposed will be mostly utilized when evidence of the models validity already exists, i.e. the models are consistent with the implementation through feedback from testing and measuring the system.

Our view of a system is depicted in Figure 8.2. Basically, we say that a system consists of a set of interconnected nodes. Communication busses connects the different nodes. A node is described by its system architecture, i.e. its hardware resources and the software features allocated to the node:

- cpu is the computational resource of the node,
- m_s is the static non-volatile memory,
- m_d is the dynamic memory,
- m_p is the persistent memory,

- F is the set of features whose components C(F) are partially or completely allocated to the node n. Moreover, feature $f \in F$ is a function as experienced from a system users point of view and it collects the functional-, and non-functional requirements.
- I/O is a set, possible empty, consisting of I/O-units.

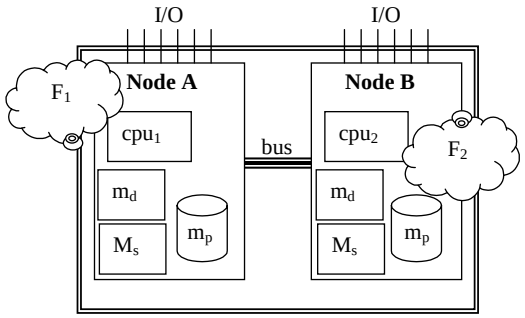

Figure 8.2: Our view of a system

The component view identifies the components in the system and their analytical model. We assume a component to be an encapsulation of a service implemented in software as described in Chapter 4. The temporal attributes e.g. period times are an essential part of the analytical model of a component in real-time systems. We will refer to temporal attributes in the component view as the temporal analytical model. This level of abstraction is suitable for analytical system analyses and for simulations of the system. Thus, a system's temporal requirements, which were initially partitioned into features, are implemented and verified in the component view. The appearance of the temporal analytical model corresponds with that of the real-time operating system (RTOS) and communication mechanisms in the system infrastructure. As the functional requirements are implemented in the chosen programming language, and non-functional requirements are implemented in the architecture, the temporal requirements are implemented by assigning temporal attribute in the task model provided by the RTOS. An example of such temporal attributes is period time and priorities. Consequently, the temporal attributes provided by the infrastructure are also part of the implementation view. The implementation view consists of the actual implementation. Thus, it provides necessary information for testing

the system, as well as measuring execution times and response times.

The implementation view is represented by the functional behavior of the components as well as their temporal behavior. Temporal requirements are implemented through the task model provided by the RTOS. Thus, we consider it a part of the implementation view.

When a real-time system is implemented on a particular hardware architecture, the model becomes valid for that particular instance only. The reason for this is that the temporal behavior of the components is affected when the hardware architecture is changed. Typically, the execution times will be affected. This phenomenon is typical for real-time systems and is a problem in large and complex systems with a long service life where software usually survives several generations of the hardware architecture. Ideally, we could treat changes in the hardware architecture as a point of variation in the product line architecture. However, changing the CPU requires a complete re-measure of the execution time distributions for the system.

As future work we propose an extended version of ART-ML that specifies product lines in a hierarchical manner with the product line at the top level. The hierarchy reflects the level of abstractions discussed earlier in this chapter. The top level defines all products within the scope of a company's product line. Products are specified in terms of their hardware system and features which they may share, or not. Note that the modeling language proposed here is presented in a pseudo manner. For instance, the memory statement is a collection of the different memory types available in a node, i.e. static non-volatile memory, dynamic memory, and persistent memory. On the system level of abstraction, memory specifies the memory available on a node, whereas on the feature level and on the component level we specify the resource consumption, i.e. the amount of memory required. Below is an example of how a product line with two products, Product A and Product B, could be described on the highest level of abstraction with a modeling language that have support for product lines:

```
ProductLine RealTimePLA
{
  Product A
  {
    system
    {
      node_1
```

```
      node_2
      bus
      memory
    }
    feature X
    feature Y
    feature Z
  }

  Product B
  {
    system
    {
      node_1
      node_3
      bus
      memory
    }
    feature V
    feature X
    feature Y
  }
}
```

In real-time systems the same feature, supplying the same functionality, may exhibit different temporal behavior. Variations are due to differences in the hardware architecture. The allocation of a feature may be distributed and is specified in the system part of the feature as illustrated below:

```
feature V
{
  System
  {
    node_1
    bus
    memory
  }
  component A
  component B
```

```
}

feature X
{
  System
  {
    node_1
    node_3
    bus
    memory
  }
  component A
  component C
  component D
}
```

Components constitute the entity that implements a feature. We do not consider components that migrate at runtime. Consequently, deployment of components is statically tied to one node. However, the same component may be deployed at several nodes, either as part of different features, or as part of a distributed feature. The variations in temporal behavior of features are due to different component deployment and the hardware on which components execute. The temporal attributes that describe the temporal behavior of a component are specified as a task. Thus, tasks together with a component's execution time forms the temporal analytical model of the system. The execution time of a component is specified as a distribution that reflects a realistic behavior. The distribution is described as pairs of execution times and their probability of occurrence. The most abstract specification of the execution time corresponds to worst-case execution time with the probability of 100%. ART-ML can also specify the functional behavior. Arithmetical operators, selections, and iterations are supported in the modeling language.

In order to capture the phenomenon that components exhibit variations in their temporal behavior as well as functional behavior, depending on the product in which they are deployed we use parameterization. The model is parameterized using the products, i.e. the product that currently is analyzed. The variations can be due to differences in hardware between products which affect the execution times. Furthermore, the tasks that control the execution of components can differ due to, e.g. differences in period times as shown in Component B below:

```
Component A
{
  System
  {
    node_1
    bus
    memory
  }

  task T1
  if (product == ProductA)
    execute(Node_1, (19,10),(81,56))
  if (product == ProductB)
    execute(ProductB, Node_1, (19,13),(75,43), (6,53))
  behavior
  {
    // Do something
  }
}
Component B
{
  System
  {
    node_3
    bus
    memory
  }

  if (product == ProductA)
    task T2
  if (product == ProductB)
    task T3
  if (product == ProductA)
    execute(ProductA Node_2, (100,30))
  if (product == ProductB)
    execute(ProductB Node_2, (10,30), (90,45))
  behavior
  {
    // Do something
  }
```

```
}
```

In the domain of real-time systems product lines is the temporal behavior also subject for variations. That point of variation is provided explicitly by the tasks. That is the reason way we have isolated the specification of a task from the component. Moreover, as the task model provided by the system's infrastructure, i.e. the real-time operating system (RTOS), may vary among different RTOS manufacturers, the language could easily adopt to new infrastructures if the tasks are separated from the components. The current implementation of ART-ML was developed for a system built on the RTOS VxWorks which schedules tasks according to fixed priorities. Consequently, tasks in our examples reflect that.

```
task T1
{
   trigger period 3000
   priority 0
}
```

The analysis of a single system temporal behavior differs a bit compared to analysis of a complete product line. Altering the behavior of a component affects all products that use it. Thus the analysis must cover the complete product line. ART-ML is developed as a modeling language for a discrete-event simulator. From the simulators point of view the difference from simulating one single system is relatively small. Instead of simulating only one system, the simulator now have to run the models for all systems in a model. Consequently, the impact of adding a feature, a component, or changing the implementation with respect to the complete product line can be analyzed.

8.2.2 Model validity

One of the most important and interesting area which is subject for further research is model validity. We need to develop and verify methods for ensuring model validity. Moreover, we would like to develop mathematical methods for verifying equivalence between simulation results and the results measured in the system. This is important in order to establish confidence in the models and the simulation results.

Appendix A

Terminology

ADL Architectural Description Language, Language for describing software architectures

Analytical interface Provides means for reasoning about properties of a set of assembled (composed), components

Analytical model Provides the information necessary for analyzing a software construction. The contents of the analytical model depends on what information the analyses require

Architectural style Standard types of architectures identified with names and patterns

Architectural view Provide the architecture description with information needed when analyzing it. The components and their interconnections are shown in the structural view

Architectural transformation Changing the architecture in order to obtain required functionality and quality

Assembly A configuration of a set of components that defines the components interconnections

Availability The probability of a system functioning correctly at any given time

Checklist based questions Domain specific questions used when evaluating a software architecture

Cost The cost for performing any action such as development, evolution and verification

COTS Commercial Off The Shelf components

Design patterns Named object oriented solutions in the object oriented community

Design space A N-dimensional space where every axis represents a design parameter, scaled with the different design options possible for that particular parameter

Direct scenario A scenario that is directly supported by the architecture

Fault-tolerance The ability of software to detect and tolerate errors in the design and/or from its environment

Feature A feature is the highest level of functional decomposition of a software system, typically a function as being grasped by a user of a system

Feature graph The set of feature in a product line and their interdependencies

Feature scope The set of feature in a product line

Framework An architectural pattern for a particular domain, widely used in the object oriented community

Functional quality property Quality properties concerned with the run-time behavior of the software system

Impact analysis Analyzing the consequences of changing a system

Indirect scenario A scenario that requires an architectural transformation to be supported by the architecture

Maintainability The aptitude of a system to undergo repair and evolution

Modifiability How sensible the architecture is to changes in one or several components

MTBF Mean-Time-Between-Failure

MTTR Mean-Time-To-Repair

Nonfunctional quality property Quality properties concerned with the software itself

Performance How fast or slow the system performs its functions measured in time or the systems capacity measured in event-throughput

PLA Product line architecture

Platform A subsystem that is used in several products in a product line which does not necessarily provide a particular architecture as a product line architecture does

Portability How easy it is to move the software system to a different hardware- and/or software platform

Product scope The set of products in a product line

Reference style Architectural styles widely used in particular application domains, e.g. the pipe-and-filter Architecture used in compilers

Reliability The probability of a system functioning correctly over a given period of time

Reusability The extent to which the architecture can be reused

Safety The property of the system that it will not endanger human life or the environment

Scenario based questions Application specific questions used when evaluating a software architecture

Scenario execution Method for analyzing an architecture by asking "what if" questions

Security The ability of a software system to resist malicious intended actions

Temporal attributes Temporal attributes implement the temporal behavior, e.g. frequencies, priorities

Temporal requirements Requirements such as deadlines, jitter, response time, worst case execution times (WCET), etc

Temporal correct A system is temporally correct if it complies with all its temporal requirements

Testability How easy it is to prove correctness of the system by testing

Tradeoff A relation between two or more quality attributes where an increased level of on property results in a decrease of another property

Variability The ability to vary the behavior of a software construction

Variation points Describes the way in which a software construction can be varied

Questionnaire based evaluation Questions used when evaluating project logistic properties of software architectures

Appendix B

The grammar of ART-ML in BNF

```
<system> ::= SYSTEM <header> <processor> <processorlist> ENDSYS

<processor> ::= PROCESSOR ID <header> <task> <tasklist> ENDPROC

<processorlist> ::=
                        | <processor> <processorlist>

<header> ::=
                | VARIABLE  ID ; <header>
                | MAILBOX   ID NUM ; <header>
                | SEMAPHORE ID NUM ; <header>
                | CONST     ID NUM ; <header>

<tasklist> ::=
                | <task> <tasklist>

<task> ::= TASK ID <taskheader> BEHAVIOUR {<header> <stmntlist>}

<taskheader> ::= <trigger> <prio> <dl> <mem>

<prio> ::= PRIORITY NUM

<dl>:
        | DEADLINE NUM
```

```
<mem> ::=
        | MEMORY NUM

<trigger> ::= TRIGGER <whattrigger>

<whattrigger> ::=  STARTUP
                        | MAILBOX ID
                        | PROBABILITY NUM
                        | PERIOD NUM

<stmntlist> ::=
                | <stmnt> <stmntlist>

<stmnt> ::=    <bool_expr> ;
                | <send> ;
                | <recv> ;
                | <semtake> ;
                | <semgive> ;
                | <sleep> ;
                | <consume> ;
                | <assignment> ;
                | <block>
                | <for>
                | <ifelse>
                | <while>
                | <chance>

<semtake> ::= SEMTAKE ( ID ) <semtake_timeout>

<semtake_timeout> ::=
                | TIMEOUT NUM

<semgive> ::= SEMGIVE ( ID )

<sleep> ::= SLEEP ( <bool_expr> )

<ifelse> ::= IF ( <bool_expr> ) <stmnt> <ifelse2>

<ifelse2> ::=
```

```
                | ELSE <stmnt>

<for> ::= FOR ( ID , <bool_expr> , <bool_expr> ) <stmnt>

<while> ::= WHILE ( <bool_expr> ) <stmnt>

<assignment> ::= ID = <bool_expr>

<block> ::= { <stmntlist> }

<chance> ::= CHANCE ( <bool_expr> ) <stmnt> <chance2>

<chance2> ::=
                | ELSE <stmnt>

<consume> ::= CONSUME ( <exec_pair> <exec_list> )

<exec_list> ::=
                | , <exec_pair> <exec_list>

<exec_pair> ::= ( NUM , NUM )

<send> ::= SEND ( ID , <bool_expr> ) <send_timeout>

<recv> ::= RECV ( ID , ID ) <recv_timeout>

<send_timeout> ::=
                | TIMEOUT NUM

<recv_timeout> ::=
                | TIMEOUT NUM

<bool_expr> ::= <rel_expr>
        |  <bool_expr> AND <bool_expr>
        |  <bool_expr> OR <bool_expr>

<rel_expr> ::= <neg_expr>
        |  <rel_expr> > <rel_expr>
        |  <rel_expr> < <rel_expr>
        |  <rel_expr> GTEQ <rel_expr>
```

```
        |  <rel_expr> LTEQ <rel_expr>
        |  <rel_expr> EQ <rel_expr>
        |  <rel_expr> NEQ <rel_expr>

<neg_expr> ::= <ari_expr>
               | ! <bool_expr>

<ari_expr> ::= <term>
               | <term> + <ari_expr>
               | <term> - <ari_expr>

<term> ::= <factor>
        | <factor> * <term>
        | <factor> / <term>
        | - <term>

<factor> ::= <atom>
               |  ( <bool_expr> )

<atom> ::= NUM
        | ID
```

Appendix C

The grammar of PPL in BNF

```
<property> ::= ''P'' ''('' <expr> '')'' <relop> <probability>
<expr> ::= <exp> <logop> <expr> | <exp>
<exp> ::= <taskExpr> <relop> id |
  <taskExpr> <relop> <taskExpr> |
  <taskExpr> <relop> NAT |
  <taskExpr> <relop> REAL |
  <queue> <relop> NAT

<taskExpr> ::= <task> |
                <task> <aritop> <task> |
                <task> <aritop> NAT

<task> ::= id''(''<instance>'')'' ''.'' start |
    id''(''<instance>'')'' ''.'' end |
    id''(''<instance>'')'' ''.'' response

<queue> ::=    id''.''size

<instance> ::= id | id <aritop> NAT
<aritop> ::= + | -
<logop> ::= and | or
<relop> ::= < | <= | > | >= | =
<probability> ::= id | REAL
```

Appendix D

The robot model

```
system
processor MC
variable sys_online;
const ref_request 66;

mailbox QUEUE1 100;
mailbox QUEUE2 6;
mailbox QUEUE3 2;
mailbox DUMMY_MAILBOX 1;

task AXISCOMPUTER_DUMMY
trigger period 4000
priority 0
deadline 4000
behaviour{
variable incoming;

if (sys_online == 1){
  send(C_MBOX, ref_request);
  execute((100,10));
  recv(incoming,QUEUE3);
}
}
task C
trigger mailbox C_MBOX
priority 2
```

```
behaviour{
variable incoming;
incoming = 0;
recv(incoming, C_MBOX) timeout 100;
if ( incoming == ref_request){
      recv(incoming, QUEUE2) timeout 10000;
  execute ((60,600),(30,670),(10,750));
  send(QUEUE3,1);
}else{
  execute ((60,60),(30,67),(10,75));
}
}
task B
trigger startup
priority 3
behaviour{
variable x,z;
const y 678;
variable i;
const x_init_value 789;

while (! sys_online ){
  sleep(150);
}
while( sys_online ){
  y = 678;
  x = x_init_value;
  i = QUEUE1.size;
  while(i>0){
    recv(z, QUEUE1) timeout 0;
    i = i - 1;
  }
  recv(i,DUMMY_MAILBOX) timeout 1;
  if ( x != x_init_value ){

  }
  for (i,0,6){
       send(QUEUE2,y);
    execute ((25,15),(25,23),(25,30),(25,50));
  }
```

```
  chance(50){
  execute ((25,150),(25,200),(25,250),(25,300));
  }else{
  execute ((25,350),(25,1200),(25,1800),(25,1900));
  }
}
}
task A
trigger probability 3
priority 90
memory 350000
behaviour{
chance(60){
  execute ((25,300),(25,400),(25,500),(25,600));
  send(QUEUE1,1);
  send(QUEUE1,1);
}else{
  execute ((33,4000),(34,5500),(33,7000));
  chance(70){
    send(QUEUE1,2);
    send(QUEUE1,2);
  }else{
    send(QUEUE1,3);
    send(QUEUE1,3);
    send(QUEUE1,3);
    send(QUEUE1,3);
    }
}
}
task others_high
trigger period 880
priority 10
behaviour{

if (sys_online == 0){
  send(QUEUE1,1);
  sys_online = 1;
}
chance(90){
  execute((25,29),(25,39),(25,69),(25,89));
```

```
}else{
  chance(95){
    execute((25,109),(25,149),(25,299),(25,499));
  }else{
    execute((50,699),(50,799));
}
}
execute((25,1),(25,3),(25,6),(25,7));
}
task others_low
trigger period 32200
priority 100
behaviour{
execute((45,15000),(35,22000),(20,25000));
}
task IDLE
trigger startup
priority 255
behaviour{
while(1){
  execute((100,100000));
}
}
endproc
endsys
```

Appendix E

The validation results

In the graphs presented in Section E.1 trough Section E.4 *et* denotes execution time and *rt* denotes response time.

E.1 Case 0

In figures E.1 through E.6 we report the result from measuring and simulating the system without any changes.

E.2 Case 1

In figures E.7 through E.14 we report the result from measuring and simulating the system with the dummy task having a short oscillating execution time and low priority.

E.3 Case 2

In figures E.15 through E.22 we report the result from measuring and simulating the system with the dummy task having a short oscillating execution time and high priority.

E.4 Case 3

In figures E.23 through E.30 we report the result from smeasuring and simulating the system with the dummy task having longer oscillating execution time and low priority and a longer period time.

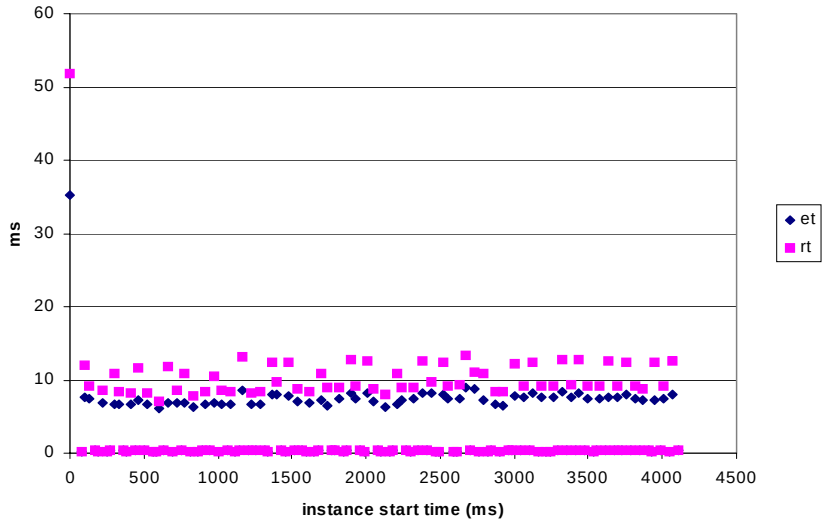

Figure E.1: Measured execution times and response times for task A in case 0

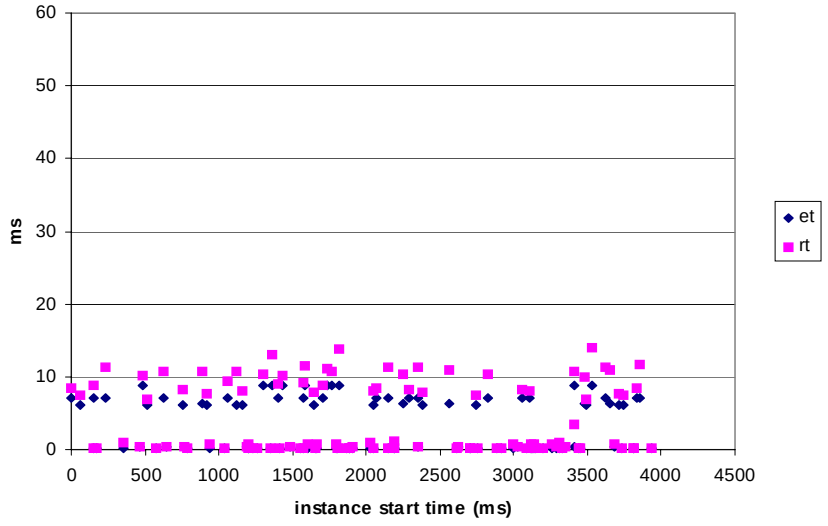

Figure E.2: Simulated execution times and response times for task A in case 0

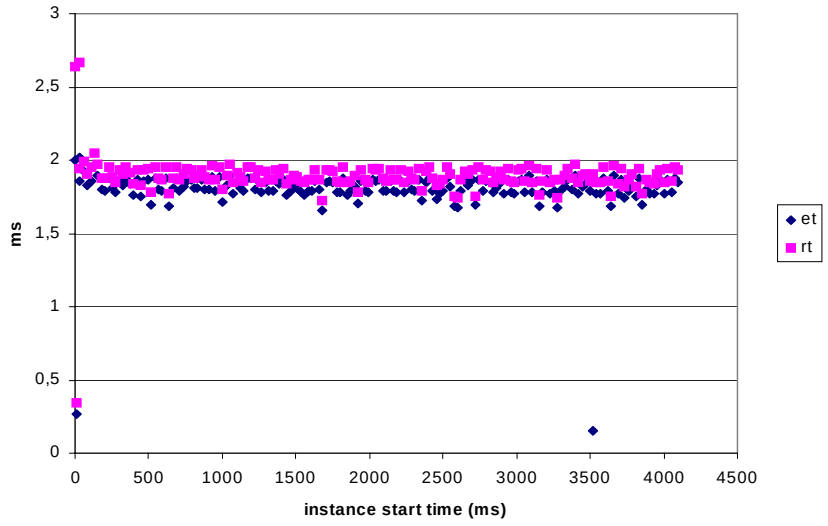

Figure E.3: Measured execution times and response times for task B in case 0

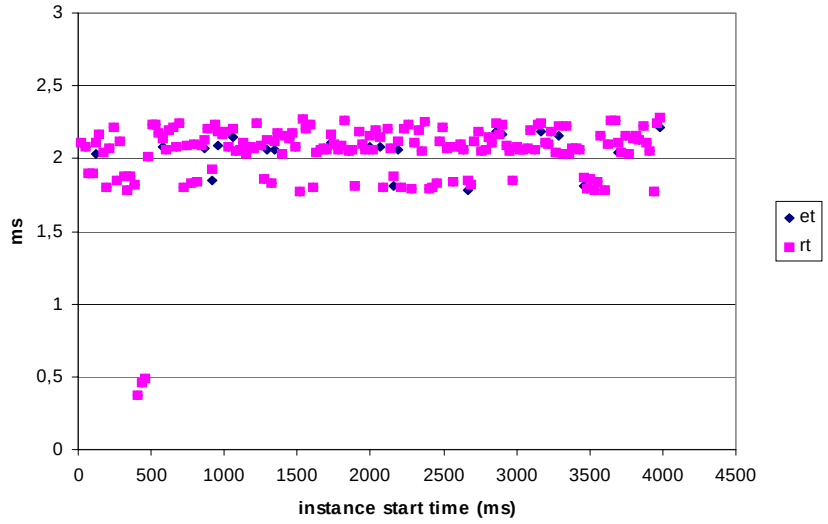

Figure E.4: Simulated execution times and response times for task B in case 0

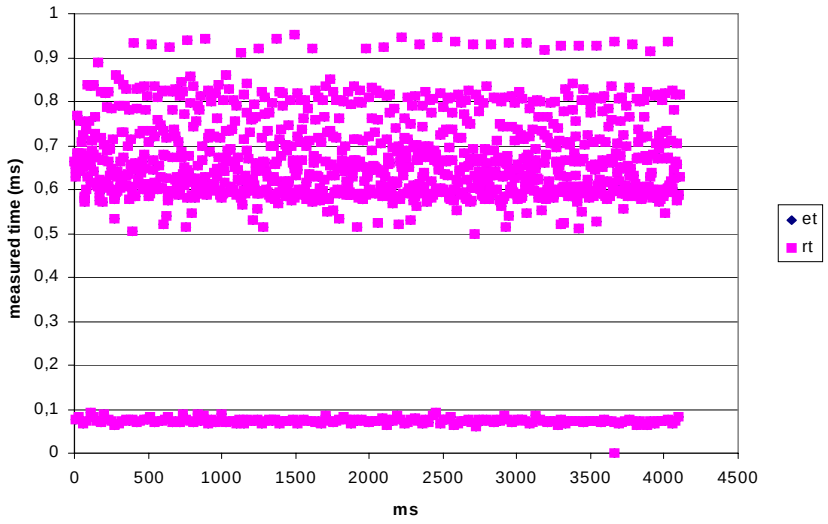

Figure E.5: Measured execution times and response times for task C in case 0

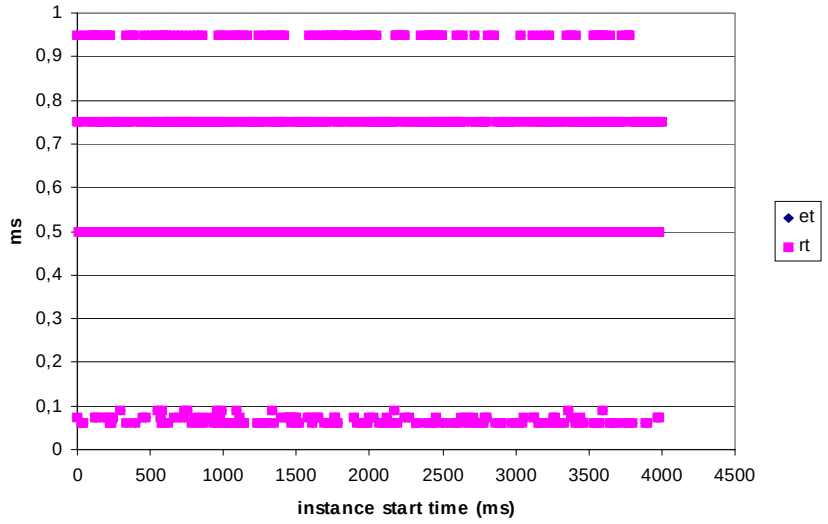

Figure E.6: Simulated execution times and response times for task C in case 0

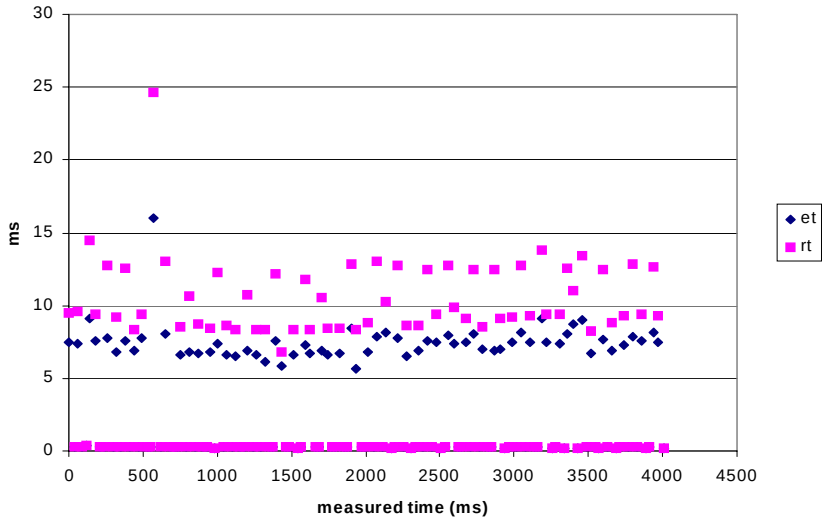

Figure E.7: Measured execution times and response times for task A in case 1

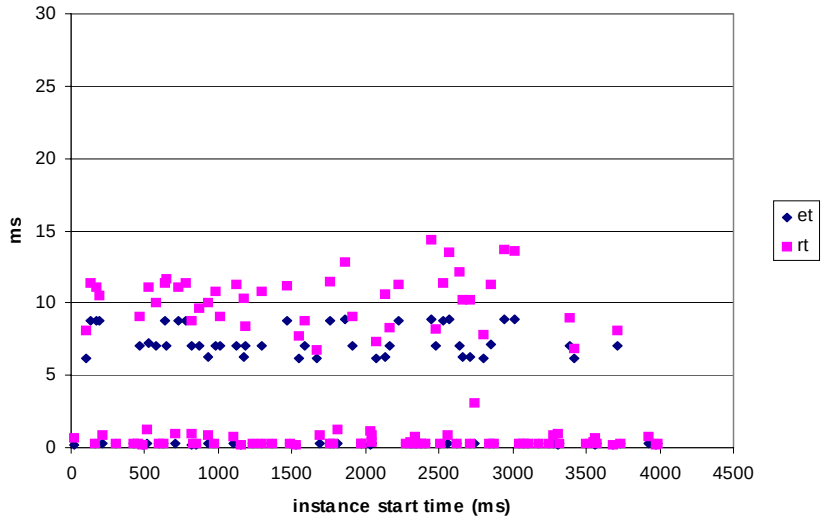

Figure E.8: Simulated execution times and response times for task A in case 1

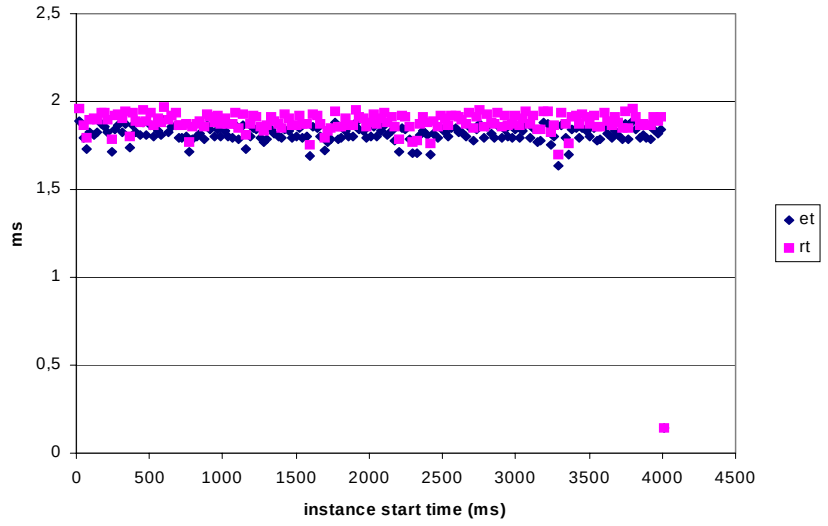

Figure E.9: Measured execution times and response times for task B in case 1

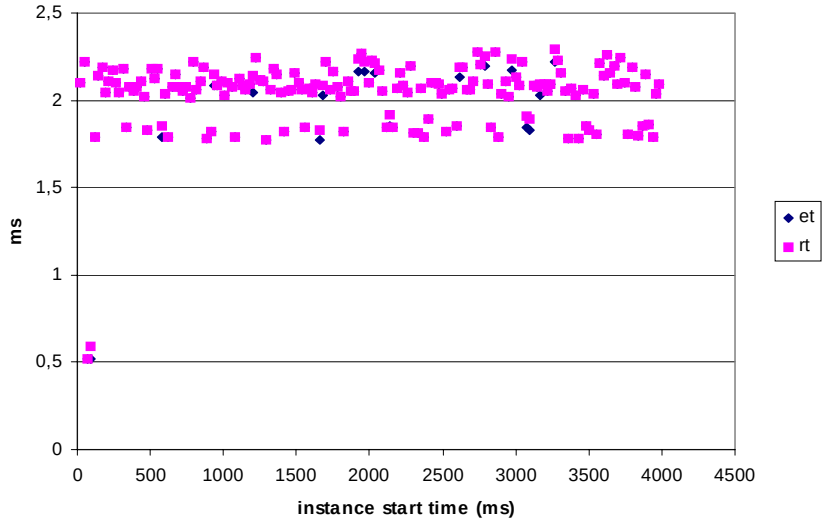

Figure E.10: Simulated execution times and response times for task B in case 1

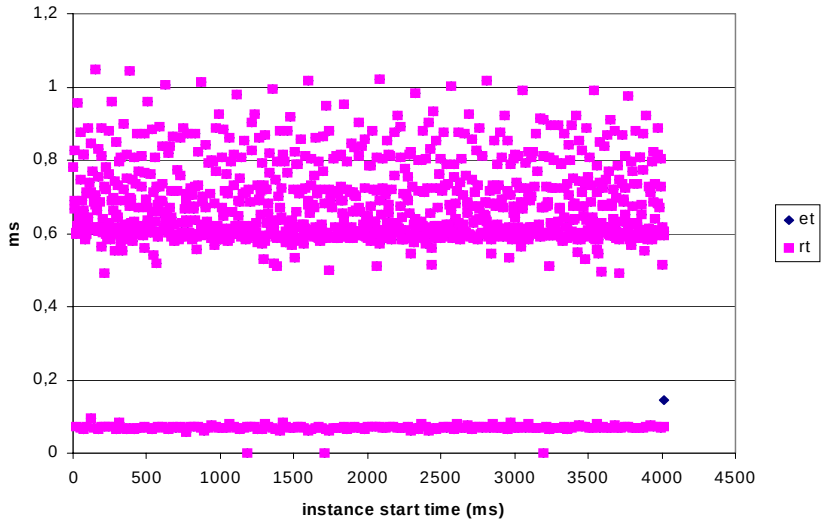

Figure E.11: Measured execution times and response times for task C in case 1

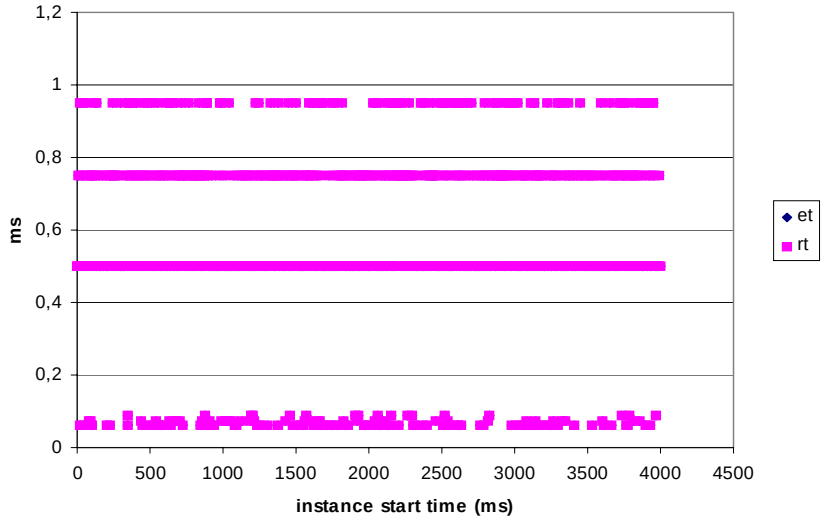

Figure E.12: Simulated execution times and response times for task C in case 1

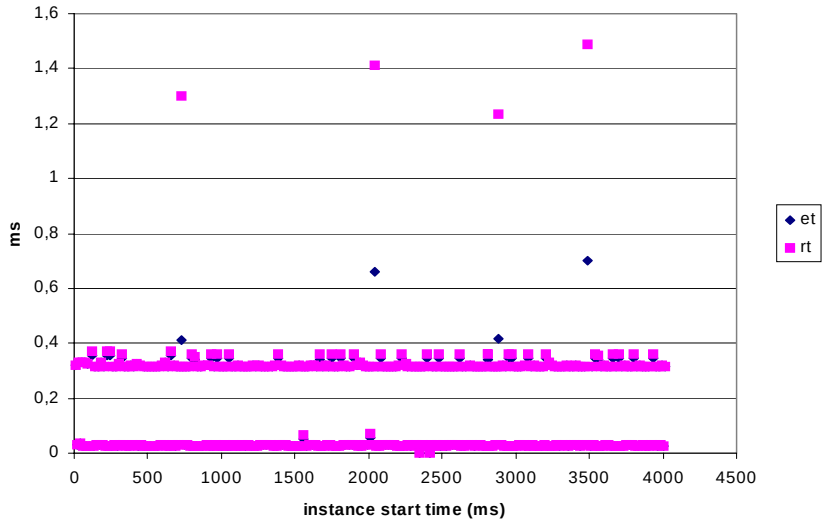

Figure E.13: Measured execution times and response times for dummy task in case 1

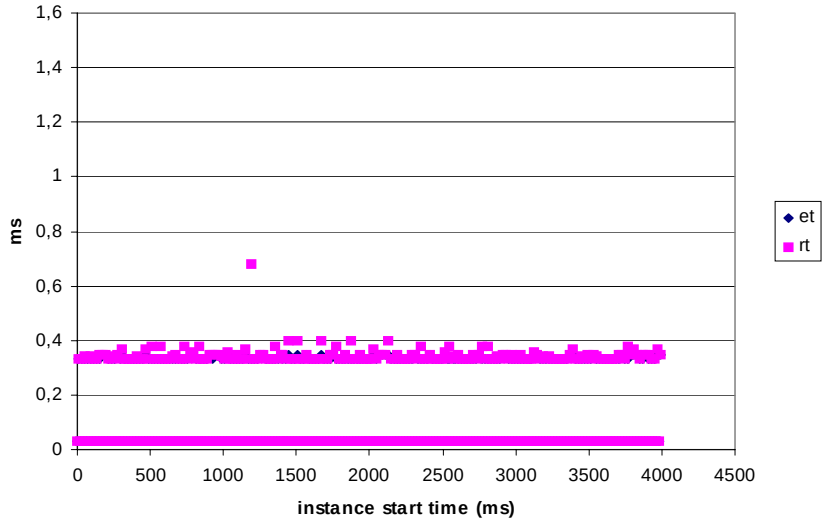

Figure E.14: Simulated execution times and response times for dummy task in case 1

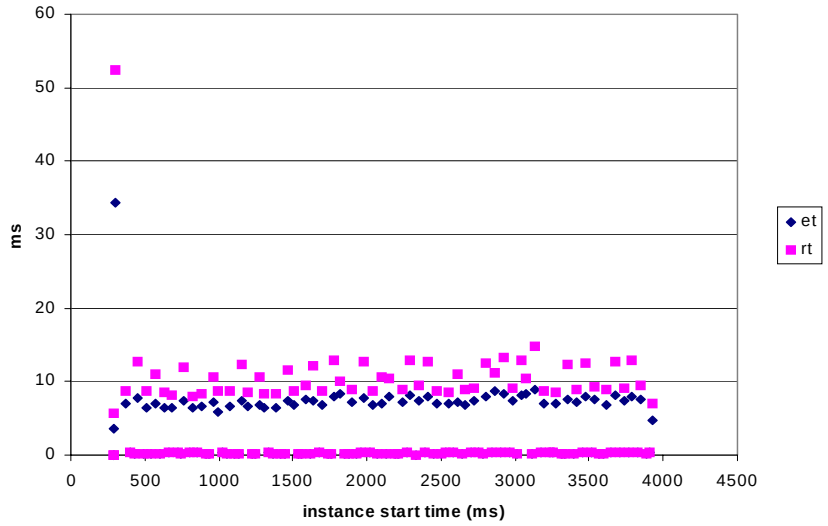

Figure E.15: Measured execution times and response times for task A in case 2

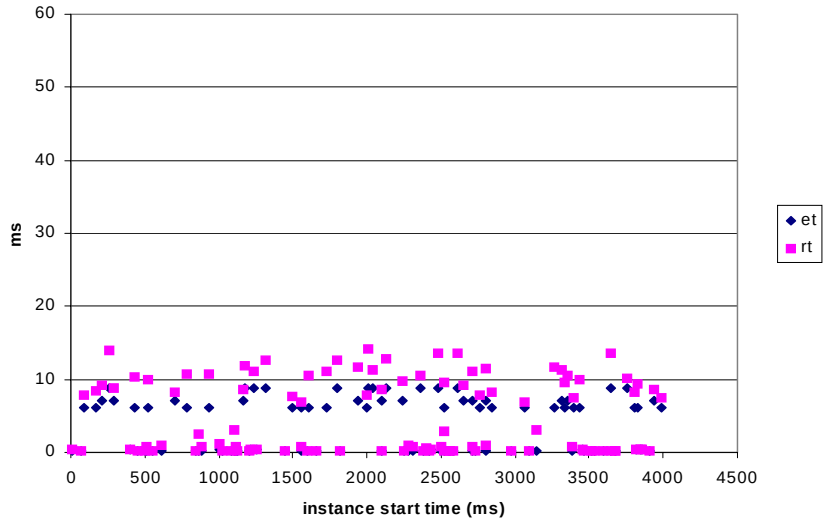

Figure E.16: Simulated execution times and response times for task A in case 2

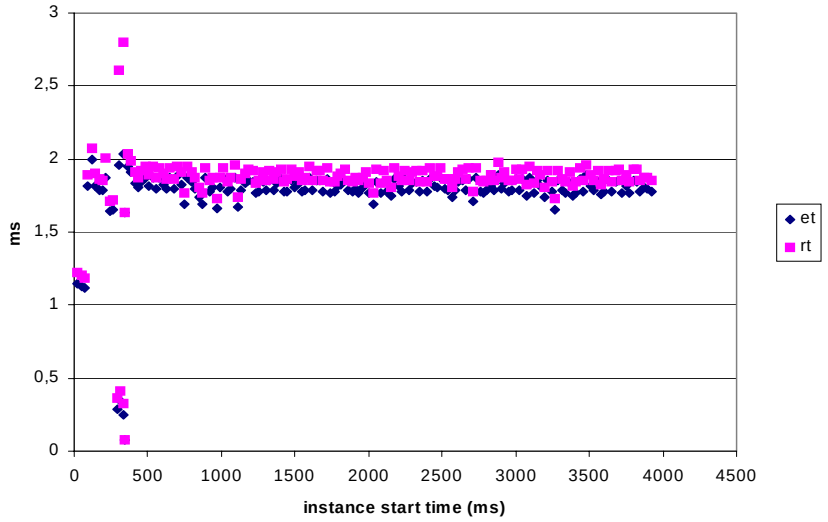

Figure E.17: Measured execution times and response times for task B in case 2

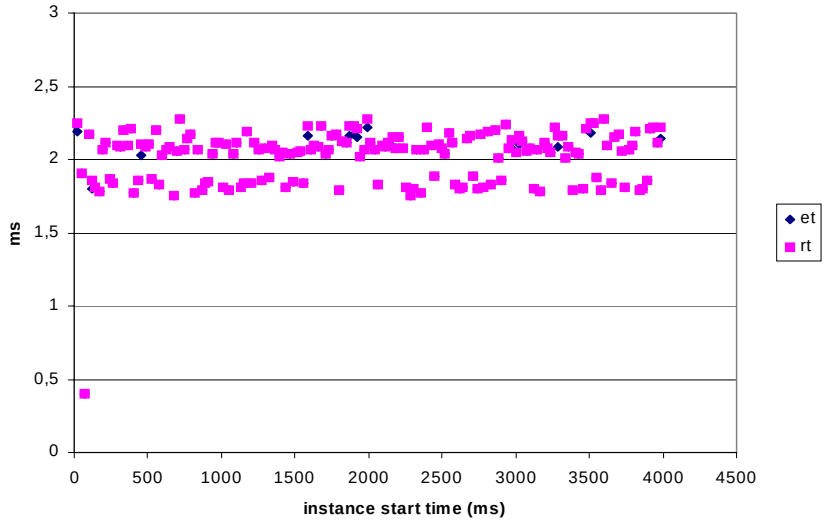

Figure E.18: Simulated execution times and response times for task B in case 2

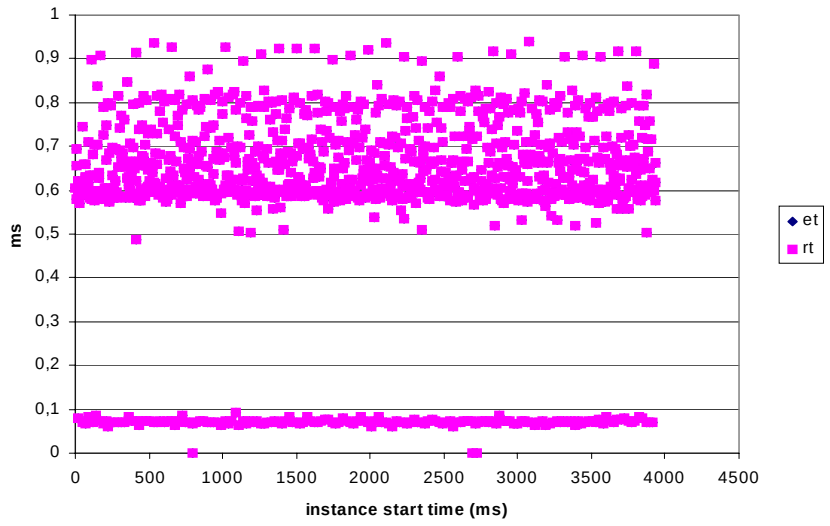

Figure E.19: Measured execution times and response times for task C in case 2

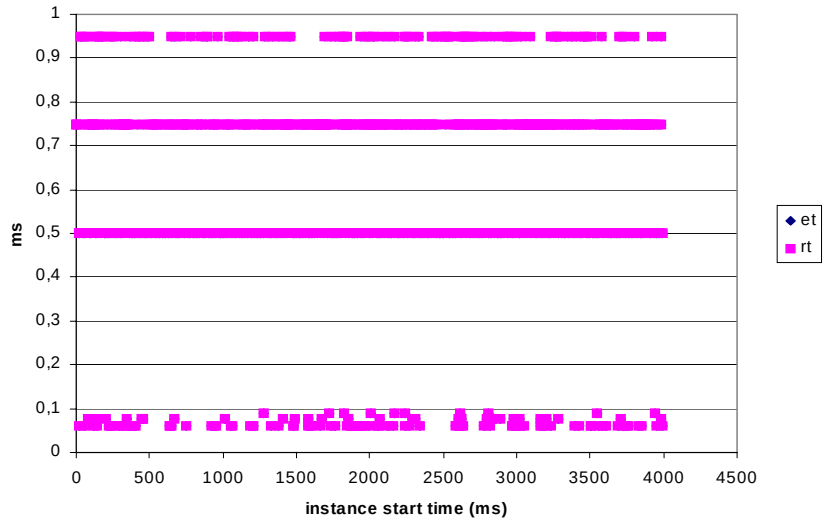

Figure E.20: Simulated execution times and response times for task C in case 2

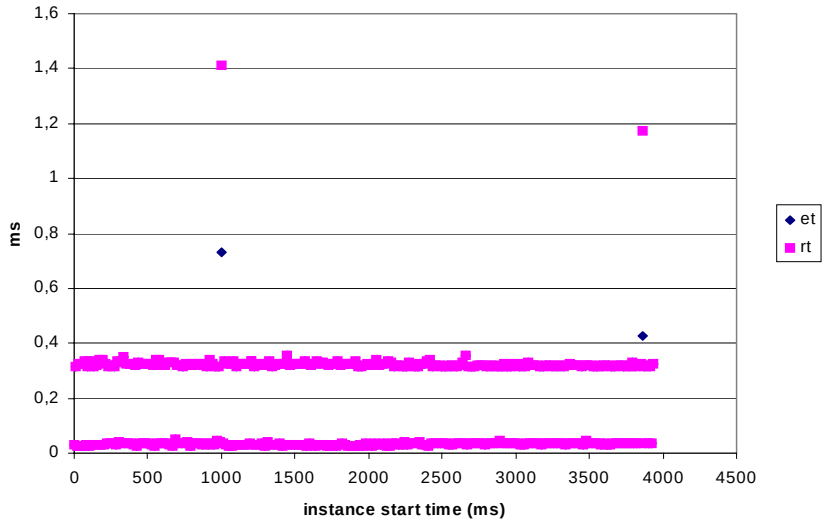

Figure E.21: Measured execution times and response times for dummy task in case 2

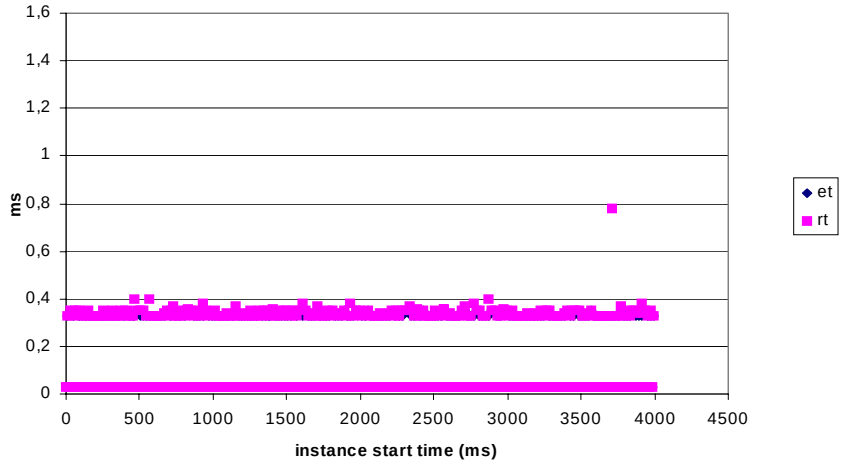

Figure E.22: Simulated execution times and response times for dummy task in case 2

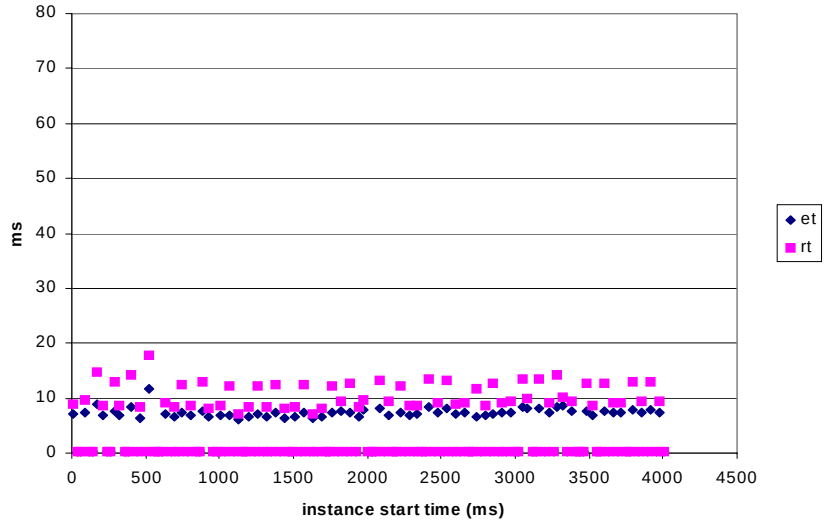

Figure E.23: Measured execution times and response times for task A in case 3

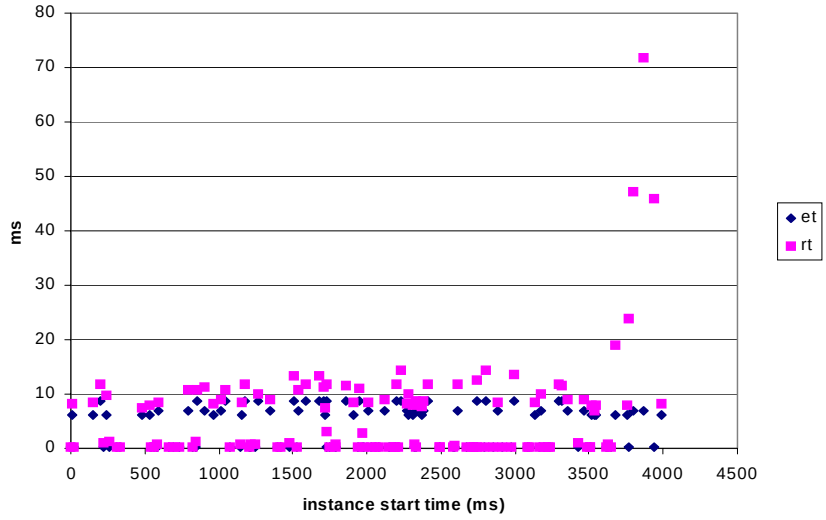

Figure E.24: Simulated execution times and response times for task A in case 3

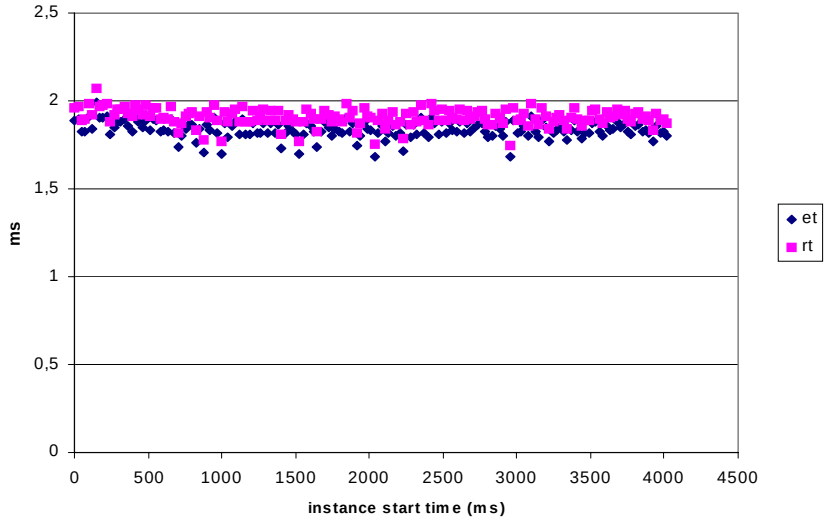

Figure E.25: Measured execution times and response times for task B in case 3

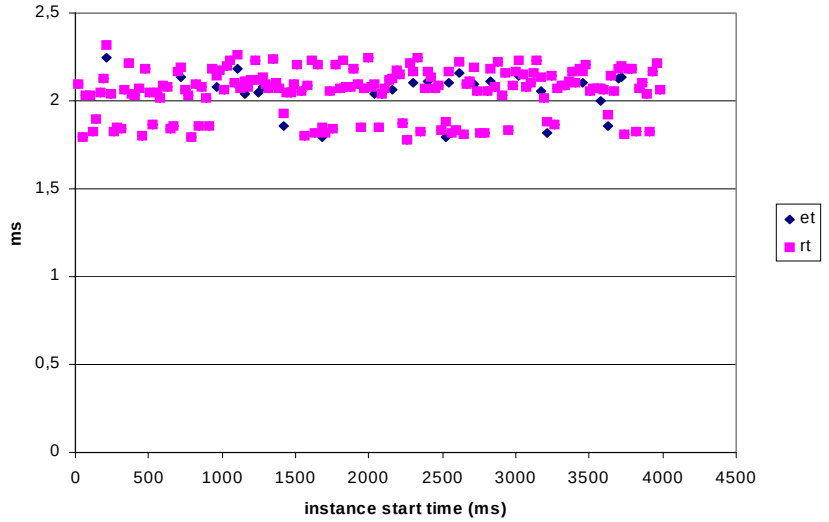

Figure E.26: Simulated execution times and response times for B task in case 3

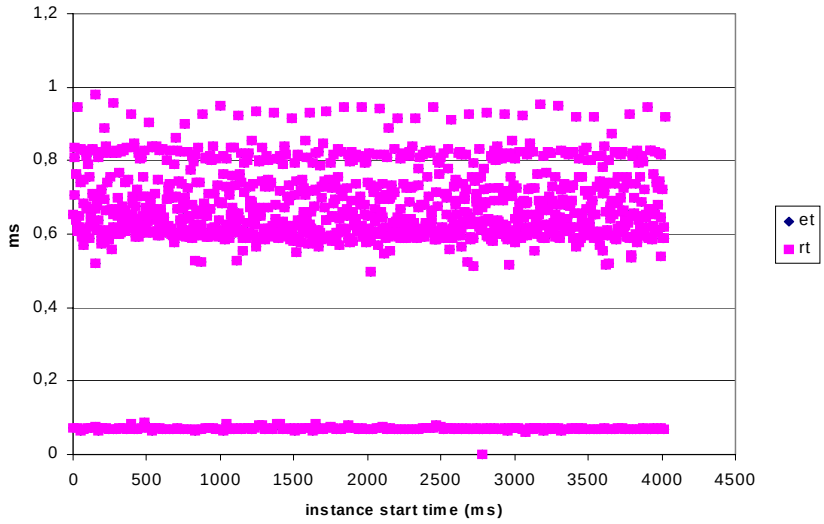

Figure E.27: Measured execution times and response times for task C in case 3

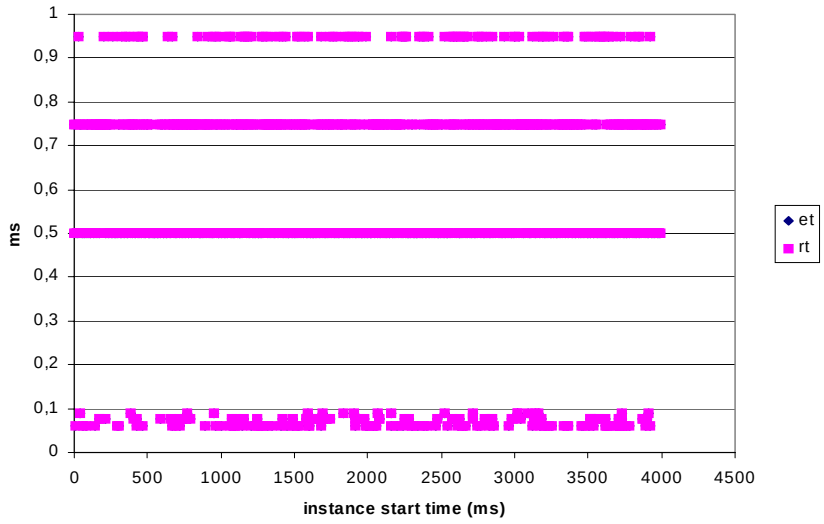

Figure E.28: Simulated execution times and response times for task C in case 3

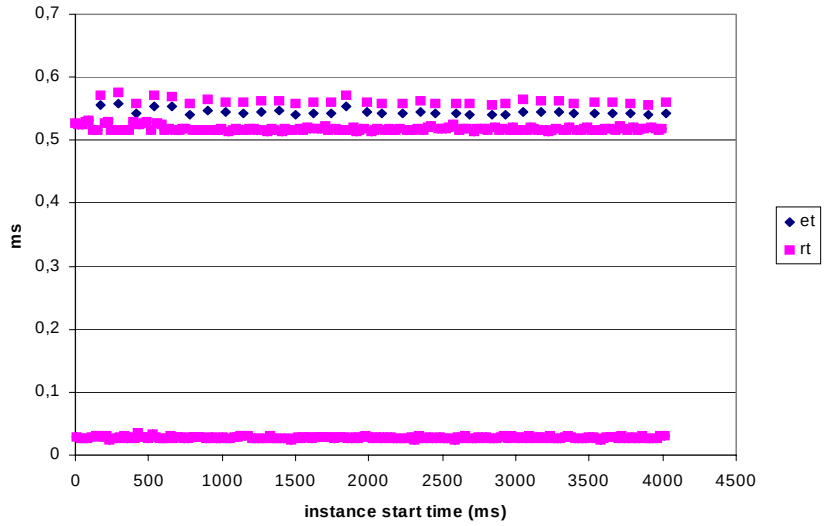

Figure E.29: Measured execution times and response times for dummy task in case 3

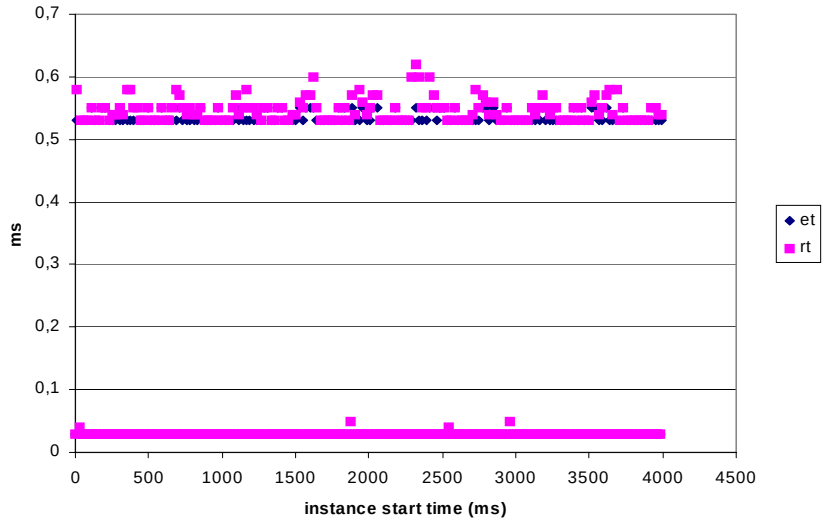

Figure E.30: Simulated execution times and response times for dummy task in case 3

Bibliography

[ABD+95] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, , and A. J. Wellings. Fixed priority pre-emptive scheduling: An historical perspective. *Real-Time Systems Journal*, 8(2/3):173–198, 1995.

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

[ABRW94] N. C. Audsly, A. Burns, M. F. Richardson, and A. J. Wellings. Stress: A simulator for hard real-time systems. *Software-Practice and Experience*, 24(6):543–564, June 1994.

[AD90] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proceedings of ICALP'90*, volume 443 of *Lecture Notes in Computer Science*. Springer, 1990.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2), 1994.

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - A Tool for Modelling and Implementation of Embedded Systems. In *Proceedings of 8th International Conference on Theory and Practice of Software*, 2002.

[Alu93] R. Alur. Model-Checking in Dense Real-Time. *Information and computing*, 1993.

[AN02] J. Andersson and J. Neander. Timing Analysis of a Robot Controller, 2002.

[BCK97] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. ISBN 0-201-19930-0. Addison-Wesley, 1997.

[Ben02] P. O. Bengtsson. *Architectural-Level Modifiability Analysis*. PhD thesis, Blekinge Institute of Technology, 2002.

[BGK+96] J. Bengtsson, Griffioen, K. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and Y Wang. Verification of an Audio Protocol with Bus collision Using UPPAAL. In *Proceedings of CAV'96*, volume 1102, 1996.

[Bin94] R. V. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*, 37(9):87–101, 1994.

[BLL+96] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. UPPAAL in 1995. In *Proceedings of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer Verlag, Mars 1996.

[BMR+99] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. ISBN 0-471-95869-7. Wiley, 1999.

[Bos99] J. Bosch. Product-Line Architectures in Industry: A Case Study. In *Proceedings of the international conference on Software Engineering*, pages 544–554, 1999.

[Bos00] J. Bosch. *Design and Use of Software Architectures*. ISBN 0-201-67494-7. Addison-Wesley, 2000.

[BRJ98a] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. ISBN 0-201-57168-4. Addison-Wesley, 1998.

[BRJ98b] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. ISBN 0-201-57168-4. Addison Wesley, 1998.

[Bur93] A. Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. Technical Report Technical Report YCS 214, University of York, 1993.

[But97] G. C. Buttazzo. *Hard Real-Time Computing Systems: PredictableScheduling Algorithms and Applications*. ISBN 0-7923-9994-3. Kluwer Academic Publisher, 1997.

[BW94] A. Burns and A. Wellings. HRT-HOOD, a Structured Design Method for Hard Real-Time Systems. Technical report, University of York, 1994.

[CA78] L. Chen and A. Avizienis. N-version programming: a Fault Tolerant Approach to Reliability of Software Operation. In *In proceedings of 8th Annual International Conference on Fault Tolerant Computing*, pages 3–9, 1978.

[CKK02] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures - Methods and Case studies*. ISBN 0-201-70482-X. Addison-Wesley, 2002.

[Cle96] P. C. Clements. Comming Attractions in Software Architecture. Technical Report CMU/SEI-96-TR-008, Software Engineering Institute, 1996.

[CN96] P. C. Clements and L. M. Northrop. Software Architecture: An Executive Overview. Technical Report CMU/SEI-96-TR-003, Software Engineering Institute, 1996.

[CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. ISBN 0-201-70332-7. Addison-Wesley, 2001.

[Coo91] J. E. Cooling. *Software Design for Real-time Systems*. ISBN 1-85032-279-1. International Thomson Computer Press, 1991.

[Der74] M. L. Dertouzos. Control robotics: The Procedural Control of Physical Processes. *Information Processing*, 1974.

[Dij67] E. W. Dijkstra. The Structure of "THE"-Multiprogramming System. *ACM on Operating System Principle*, 1967.

[DKO+96] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying Software Product-Line Architecture. *IEEE Software*, 30(8):49–55, August 1996.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16*th *IEEE Real-Time Systems Symposium*, pages 66–75, December 1995.

[EHL+94] S. Edwards, W Heym, T. Long, M. Sitarman, and B. Weide. Specifying Components in RESOLVE. *Software Engineering Notes*, 19(4), 1994.

[EMTP+96] C. Eriksson, J. Mäki-Turja, K. Post, M. Gustafsson, K. Sandström, and E. Brorson. An Overview of RTT: A design Framework for Real-Time Systems. *Journal of Parallel and Distributed Computing*, 36(10):66–80, October 1996.

[Faf94] D. Fafchamps. Organizational Factors and Reuse. *IEEE Software*, 11(5):31–41, September 1994.

[Fav90] J. Favaro. What Price Reusability?: A Case Study. In *Proceedings ACM First Symposium on Environments and tools for Ada*, pages 115–124, 1990.

[FC93] A.N. Fredette and R. Cleaveland. RTSL: A Language for Real-Time Schedulability Analysis. In *Proceedings of Real-Time Systems symposium*, pages 274–283, 1993.

[FEHC02] A. V. Fioukov, E. M. Eskenazi, D. K. Hammer, and M. R. V. Chaudron. Evaluation of Static Properties for Component-Based Architectures. In *Proceedings 28th Euromicro Conferance*, pages 33–39, 2002.

[FP96] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A rigorous & practical approach.* ISBN 0-534-95600-9. International Thomson Computer Press, 1996.

[FP97] N.E. Fenton and S. Lawrence Pfleeger. *Software Metrics - A Rigorous and Practical Approach.* UK ISBN 1-85032-275-9. International Thomson Computer Press, 1997.

[GB98] H. Grahn and J. Bosch. A Simulation Approach to Predict and Evaluate the Performance of Software Architectures. Technical report, University of Karlskrona/Ronneby, 1998.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. ISBN 0-201-63361-2. Addison-Wesley, 1994.

[HL89] Hans Hüttel and Kim G. Larsen. The use of static constructs in a modal process logic. In *Logic at Botik'89*, number 363, pages 163–180. Springer–Verlag, 1989.

[HMSW01] S.A. Hissam, G. A. Moreno, J. Stafford, and K. C. Wallnau. Packaging Predictable Assembly with Prediction-Enabled Component Technology. Technical Report CMU/SEI-2001-TR-024 ESC-TR-2001-024, Software Engineering Institute, 2001.

[HMSW02] S.A Hissam, G. A. Moreno, J. A. Stafford, and K. C. Wallnau. Packaging Predictable Assembly. In *Proceedings of the 1st IFIP/ACM Working Conferance on Component Deployment*, 2002.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 1996.

[HNS99] C. Hofmeister, R. L. Nord, and D. Soni. Describing Software Architecture with UML. In *Proceedings 1st Working IFIP Conference on Software Architecture*, pages 145–159, 1999.

[IEC] International standard 812 analysis techniques for system reliability: Procedures for failure mode and effect analysis. Geneva: International Electrotechnical Commission.

[IEC95] Application and Implementation of IEC 1131-3, May 1995. Standard provided by the International Electrotechnical Commision.

[IEE00] IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[JM72] Z. Jelinsky and B.P. Moranda. *Software Reliability Research, Statistical Computer Performance Evaluation.* Academic Press, 1972.

[Joh93] R. Johansson. *System Modeling Identification.* ISBN 0-13-482308-7. Prentice-Hall, 1993.

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

[KABC96] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-Based Analysis of Software Architecture. *IEEE Software*, 1996.

[KBAW94] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Engineering. In *Proceedings of IEEE International Conferance on Software Engineering*, pages 81–90, 1994.

[KBK+99] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, and S. G. Woods. The Architecture Tradeoff Analysis Method. In *Proceedings of ICSE99*, pages 54–63, 1999.

[Kle95] T. Kletz. Computer Control and Human Error. Technical report, Rugby: Institute of Chemical Engineers, 1995.

[Kru95] P. Kruchten. The 4+1 View of Architecture. *IEEE Software*, 1995.

[Kru99] P. Kruchten. *The Rational Unified Process: an introduction.* ISBN 0201604590. Addison-Wesley, 1999.

[Lap92] J.C. Laprie. Dependability: Basic Concepts and Associated Terminology. *Dependable Computing and Fault-Tolerant Systems*, 1992.

[LBJ+95] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, july 1995.

[Lev95] N.G. Leveson. *Safeware - System Safety and Computers.* ISBN 0-201-11972-2. Addison Wesley, 1995.

[LKA+93] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. Technical report, Stanford University technical report, 1993.

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

[LN03] A. Leulseged and N. Nissanke. Probabilistic Analysis of Multi-processor Scheduling of Tasks with Uncertain Parameters. In *Proceedings 9th Conferance on Real-Time and Embedded Computing Systems and Applications*, pages 317–336, 2003.

[LNP00] M. Lindgren, C. Norström, and S. Punnekkat. Deriving Reliability Estimates of Distributed Real-Time Systems. In *Proceedings International Conferance on Real-Time Computing Systems and Applications*, 2000.

[LPY95] K. G. Larsen, P. Petterson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[LPY97a] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2), 1997.

[LPY97b] H. Lönn, P. Pettersson, and Wang Yi. Formal Verification of a TDMA Protocol Start-Up Mechanism. In *Proceedings of 1997 IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, 1997.

[LPY98] M. Lindahl, P. Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. *Lecture Notes in Computer Science*, 1384:281–297, 1998.

[Mas] The official handbook of MASCOT. Version 3.1, Issue 1, 1987.

[MEP01] S. Manolache, P. Eles, and Z. Peng. Memory and Time-efficient Schedulability Analysis of Task Sets with Stochastic Execution Time. In *Proceedings of the 13nd Euromicro Conference on Real-Time Systems*. Department of Computer and Information Science, Linköping University, Sweden, 2001.

[Mil89] R. Milner. *Communication and Concurrency*. ISBN 0-13-114984-9. Prentice Hall, 1989.

[Mrv97] M. Mrva. Reuse Factors in Embedded Systems Design. Technical report, High-Level Design Techniques Dept. at Siemens AG, Munich, Germany, 1997.

[N.G00] Leveson N.G. Intent specifications: an approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(1), 2000.

[NSG+00] C. Norström, K. Sandström, M. Gustafsson, J. Mäki-Turja, and N. Bånkestad. Findings from Introducing State-of-the-art Real-Time Techniques in Vehicle Industry. In *Proceedings* 4^{th} *In industrial session of the* 12^{th} *Euromicro Conferance on Real-Time Systems*, 2000.

[NSG+01] Christer Norström, Kristian Sandström, Mikael Gustafsson, Jukka Mäki-Turja, and Nils-Erik Bånkestad. Experiences from Introducing State-of-the-art Real-Time Techniques in the Automotive Industry. In *Proceedings of 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2001.

[NWY99] Christer Norström, Anders Wall, and Wang Yi. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of RTCSA'99*, 1999.

[Pau94] F. Paulisch. Software architecture and reuse - an inherent conflict? In *Proceedings of 3rd International Conference on Software Reuse*, 1994.

[PDB97] Sasikumar Punnekkat, Rob Davis, and Alan Burns. Sensitivity Analysis of Real-Time Task Sets. In *Proceedings of Asian Computing Science Conference*, 1997.

[PR99] Stevens P. and Pooley R. *Using UML -Software Engineering with Objects and Components*. Addison-Wesley, 1999.

[PSW99] D. Plakosh, D. Smith, and K. Wallnau. Builders Guide for WaterBeans Components. Technical Report CMU/SEI-99-TR-024 ADA373154, Software Engineering Institute, 1999.

[SEG97] K. Sandström, C. Eriksson, and M. Gustafsson. RealTimeTalk - a Design Framework for Real Time Systems - a Case Study. In *Proceedings of SNART'97*, 1997.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Dicipline*. ISBN 0-13-182957-2. Prentice-Hall, 1996.

[Sha96] M. Shaw. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. In *Proceedings of 8th International Workshop on Software Specification and Design*, 1996.

[Sho02] Mohammed El Shobaki. On-chip monitoring of single- and multiprocessor hardware real-time operating systems. In *8th International Conference on Real-Time Computing Systems and Applications*. IEEE, March 2002.

[SL96] M.F. Storch and J.W.−S. Liu. DRTSS: a simulation framework for complex real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium (RTAS '96)*. Dept. of Comput. Sci., Illinois Univ., Urbana, IL, USA, 1996.

[Ste00] D. B. Stewart. Software Components for Real Time. *Embedded Systems Programming*, 13(13), December 2000.

[Sto96] N. Storey. *Safety-Critical Computer Systems*. ISBN 0-201-42787-7. Addison-Wesley, 1996.

[SVK97] D. B. Stewart, R.A. Volpe, and P.K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. on Software Engineering*, 23(12), December 1997.

[Szy02] C. Szyperski. *Component Software: Beyond Object-oriented Programming*. ISBN 0201745720. Pearson Professional Education, 2002.

[T̈98] M. Törngren. Fundamentals of Implementing Real-Time Control Applications in Distributed Computer Systems. *Journal of Real-Time Systems*, (14):219–250, 1998.

[TH99] H. Thane and H. Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[TH01] H. Thane and H. Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems*, 24(9):463–471, 2001.

[TL01] T. Thai and H. Lam. *.NET Framework*. O'Reilly, 2001.

[Tra95] C. Trammell. Quantifying the reliability of software: statistical testing based on a usage model. In *Proceedings of IEEE 2nd International Software Engineering Standards Symposium*, pages 208–218, 1995.

[Tri] Rapid sim: High-performance simulation of real-time systems. `http://www.tripac.com`.

[Ves94] S. Vestal. Mode changes in a real-time architecture description language. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, 1994.

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

[WAN+03] A. Wall, J. Andersson, J. Neander, C. Norström, and M. Lembke. Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems. In *Proceedings International Conferance on Real-Time Computing Systems and Applications*, 2003.

[WN01] A. Wall and C. Norström. A Component Model for Embedded Real-Time Software Product Lines. In *Proceed-*

ings 4^{th} *International Conferance on Fieldbus Systems and their Applications*, 2001.

[XP00] J. Xu and D. L. Parnas. Priority scheduling versus pre-run-time scheduling. *Real-Time Systems Journal*, 18(1):159–176, January 2000.