Mälardalen University Press Licentiate Theses
No.52

Modeling the Temporal Behavior of Complex Embedded Systems

A Reverse Engineering Approach

Johan Andersson

June 2005

Department of Computer Science and Engineering
Mälardalen University

ISSN 1651-9256
ISBN 91-88834-71-9
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Software systems embedded in complex products such as cars, telecom systems and industrial robots are typically very large, containing millions of lines of code, and have been developed by hundreds of engineers over many years. We refer to such software systems as complex embedded systems.

When maintaining such systems it is difficult to predict how changes may impact the system behavior, due to the complexity. This is especially true for the temporal properties of the system, e.g. response times, since the temporal behavior is dependent on many factors that are not visible in the implementation, such as execution time. The state-of-the-practice is therefore often the trial-and-error approach, i.e. implement and test. However, errors related to the temporal behavior are often hard to find while testing the system and may cause major economic losses if they occur post-release, since they typically result in system failures.

This thesis presents a method for predicting these types of errors in an early stage of development. The specific method proposed is called behavior impact analysis, which aims to predict if a specific change to the system may result in errors related to the temporal behavior. The method especially targets complex embedded systems and by using this analysis method in the software development process, the number of errors introduced when maintaining the system can be reduced. This results in an increased productivity in maintenance as well as an improvement in system reliability.

This thesis focuses on the construction and validation of the temporal behavior model necessary for performing a behavior impact analysis. The conclusion of the thesis is that a combination of dynamic analysis and reverse engineering is suitable for modeling the temporal behavior of complex embedded systems. Regarding validation of temporal behavior models, the thesis propose a process containing five increasingly demanding tests of model validity. Tools are presented that support the model construction and validation processes.

Till Birgitta

Preface

This work has been supported by ABB Robotics and ASTEC, the VINNOVA competence center on software technology, as well as ARTES and SAVE-IT.

I would like to thank my supervisors, Professor Christer Norström, Dr. Anders Wall and Professor Björn Lisper, for their enthusiasm and excellent support during these years! Thank you! Also Professor Hans Hansson and Joel Huselius has taken the time to read and comment this thesis, thank you!

Peter Eriksson has provided a lot of valuable input and comments on this research from ABB Robotics point of view. Our interesting discussions have given me a much better understanding of the industrial robotics domain and ABB's robot control system. Your enthusiasm and positive spirit have been very supportive! Thank you!

I would like to thank Professor Wang Yi, Professor Bengt Jonsson, Pavel Krcal, Leonid Mokrushin and Xiaochun Shi, Uppsala University, for many interesting discussions! I would also like to thank all my colleagues at the department, especially Joel Huselius, Jonas Neander, Johan Fredriksson, Dag Nyström, Thomas Nolte, Daniel Sundmark, Anders Pettersson, Joakim Fröberg and Jukka Mäki-Turja, for the many laughs and interesting discussions!

I want to direct many thanks to my friends Christian Hultman, Christian Andersson, Klas Andersson and Rickard Söderbäck, for a total of 40 years of friendship. Finally I want to express my gratitude to my fiancé Birgitta, my parents Lennart and Susanne, and my sisters Josefin and Jessica, for your love, support and interest in my work.

Thank you all!

Johan Andersson
Västerås, June, 2005

Publications

The author of this thesis has authored or co-authored the following publications:

Articles in collection

A Dependable Open Platform for Industrial Robotics - A Case Study, Goran Mustapic, Johan Andersson, Christer Norström, Anders Wall, Architecting Dependable Systems II, Lecture Notes in Computer Science (LNCS) 3069, Editors: Rogrio de Lemos, Cristina Gacek, Alexander Romanovsky, ISBN: 3-540-23168-4, 2004.

Conferences and workshops

Model Synthesis for Real-Time Systems, Joel G Huselius, Johan Andersson, In Proceedings of the 9:th European Conference on Software Maintenance and Reengineering (CSMR '05), p 52-60, Manchester, UK, March, 2005

Decreasing Maintenance Costs by Introducing Formal Analysis of Real-Time Behavior in Industrial Settings, Johan Andersson, Anders Wall, Christer Norström, In Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA '04), Paphos, Cyprus, October, 2004

Validating Temporal Behavior Models of Complex Real-Time Systems, Johan Andersson, Anders Wall, Christer Norström, In Proceedings of the 4th Conference on Software Engineering Research and Practice in Sweden (SERPS'04), Linköping, Sweden, September, 2004

Real World Influences on Software Architecture - Interviews with Industrial Experts, Goran Mustapic, Anders Wall, Christer Norström, Ivica Crnkovic, Kristian Sandström, Joakim Fröberg, Johan Andersson, In Proceedings of the 4th IEEE Working Conference on Software Architectures (WICSA '04), Oslo, Norway, June, 2004

Correctness Criteria for Models Validation A Philosophical Perspective, Ijeoma Sandra Irobi, Johan Andersson, Anders Wall, In Proceedings of the International Multiconferences in Computer Science and Computer Engineering (MSV '04), Las Vegas, June, 2004

Increasing Maintainability in Complex Industrial Real-Time Systems by Employing a Non-Intrusive Method, Christer Norström, Anders Wall, Johan Andersson, Kristian Sandström, In Proceedings of the Workshop on Migration and Evolvability of Long-life Software Systems (MELLS '03), Erfurt, Germany, September, 2003

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

A Dependable Real-Time Platform for Industrial Robotics, Goran Mustapic, Johan Andersson, Christer Norström, In Proceedings of the ICSE 2003 Workshop on Software Architectures for Dependable Systems (ICSE WADS), Portland, OR USA, May, 2003

Introducing Temporal Analyzability Late in the Lifecycle of Complex Real-Time Systems, Anders Wall, Johan Andersson, Jonas Neander, Christer Norström, Martin Lembke, In Proceedings of the 9th International ConferenceR on eal-Time and Embedded Computing Systems and Applications (RTCSA '03), Tainan, Taiwan, February, 2003

Technical reports

Influences between Software Architecture and its Environment in Industrial Systems a Case Study, Goran Mustapic, Anders Wall, Christer Norström, Ivica

Crnkovic, Kristian Sandström, Joakim Fröberg, Johan Andersson, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-164/2004-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, February, 2004

A Framework for Analysis of Timing and Resource Utilization Targeting Industrial Real-Time Systems, Johan Andersson, Anders Wall, Christer Norström, Technical Report, MRTC, August, 2004

Contents

Chapter 1

Introduction

As computers have become more powerful and less expensive they have become a common and natural part of our everyday life. However, most computers manufactured today are not desktop computers, but embedded in products such as mobile phones, microwave ovens, refrigerators, toys, cars, trains, airplanes and many different types of audio and video equipment. Product developing companies have replaced electrical and mechanical solutions in their products with embedded computers. Computer-based solutions are less expensive, require less space and power and also allows for more advanced functionality. Embedded computers come in all sizes, from very small and simple 8-bit single-chip computers, with a few kilobytes of memory, to gigahertz 32-bit computers with vast resources. This thesis focuses on embedded computer systems of the latter category, large software systems embedded in complex products such as industrial robot control systems, automotive systems and telecommunication systems. Systems of this type often contain millions of lines of code and have been developed by hundreds of engineers over many years. Such systems are too large and complex for any single person to understand in detail. In this thesis, we refer to such software systems as *complex embedded systems.*

Common characteristics of complex embedded systems are their safety critical and/or business-critical nature. Typically, systems of this class are in control of machinery and therefore have requirements on dependability, such as safety, reliability and availability. Moreover, the majority of systems of this class are *real-time systems*, i.e. systems that must respond to input from its environment in a timely manner. For non-real-time computers such as home

PC's, CAD-workstations or game consoles, the focus is on the average performance, while for real-time systems another property is much more important, the worst case response time, i.e. the maximum latency possible from an input to the system's corresponding reaction. Since a violation of a temporal requirement may cause a system failure, it is critical for the system reliability that the worst case response time for each system function is known.

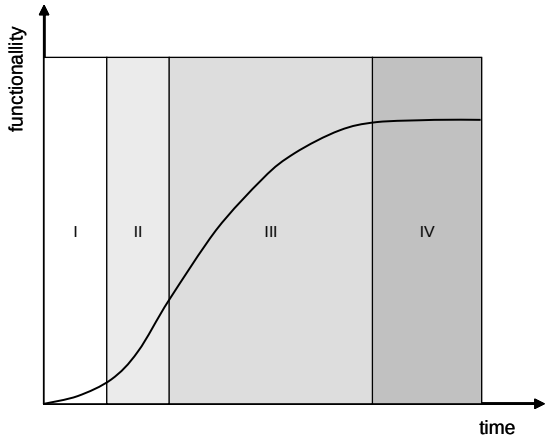

Figure 1.1: The life cycle of a complex embedded system

Another characteristic property of complex embedded systems is the long system life-cycle, measured in years, sometimes decades. Since the implementation of a complex embedded system represents a major investment for the company, many man years of development time, redesigning such a system from scratch is not an option unless it is absolutely necessary. Consequently, systems of this type are maintained for many years. The maintenance consists of *maintenance operations*, i.e. the implementation of changes to the software in order to correct errors, or to add new features in order to respond to new customer demands.

The life cycle can be divided into four different phases as depicted in Figure 1.1: (I) inception, (II) initial development, (III) maintenance and evolution, and (IV) end of life time. The curve in Figure 1.1 plots the functionality in the system over time. Hence, for a successful system it is desirable to stay in phase III as long as possible with a curve that has an inclination as steep as possible, because this implies a high degree of productivity in the software maintenance,

i.e. new features is implemeneted at a relativly low cost in terms of man hours.

However, due to the functionality increase during phase III, the system evolves from its original design. The system becomes more and more complex and thus harder to maintain, causing decreased productivity as depicted in Figure 1.1. The increased complexity is partially due to the increased size of the system, caused by new functionality, and partially due to the fact that the software architecture tends to degrade as changes are made over the years in a less than optimal manner due to e.g. time pressure or inconsistent/inadequate design documentation. Furthermore, due to the long life-cycle of the system, the personnel turnover is a major issue. Many engineers working with maintenance of complex embedded systems have limited experience of the system and may therefore not understand the implementation as well as more experienced developers do. Further, as they were not involved in the initial development of the code they are maintaining they may not be aware of the design rationale used in the initial development of the system.

In order to stay productive even though the system has a high and increasing complexity, i.e. to stay in phase III as long as possible, we must improve the way we develop and maintain software for complex embedded systems. Today, most companies that develop complex embedded systems rely heavily on code inspection and testing, which are necessary but, apparently, not sufficient. A significant effort is put into the testing of each new release of the system in order to capture as many errors as possible, but it is common that bugs are missed which may result in products being shipped with faulty software. According to a recent study [NIS02] by the National Institute of Standards and Technology (NIST) at the U.S. Department of Commerce, software bugs cost the U.S. economy an estimated $59.5 billion annually. The study concluded that more than a third of these costs could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects, i.e. finding an increased percentage of errors closer to the development stages in which they are introduced. According to [NIS02] over half of all errors are not found until ”downstream” in the development process or during post-sale software use.

When maintaining real-time systems it is important to verify that the system still complies with its temporal requirements, i.e. the requirements on worst case response time, after a change has been made to the system. The response time for a particular event is dependent on the time it takes to execute the software, which depends on the design of the software itself. Therefore, if the software is changed, it might cause the response time to exceed the specified limit, the deadline. In a worst case scenario, a maintenance operation will cause

a violation of the temporal requirements, but only in very rare situations. Such errors are easily missed during the testing of a system, but if they occur after the system has been delivered to customers, it may result in a system failure with severe consequences for the user of the system. For instance, a failing industrial robot could halt an entire production line in a factory for hours, causing a large monetary loss. Errors related to the timing of software systems can in most cases not be detected in unit testing as they only occur in the integrated system, when concurrent activities are interacting or interfering. Also, if errors related to timing and concurrency effects are discovered in full system testing, they are typically hard to reproduce. The problems associated with reproducing such errors have been discussed in e.g. [Sch91] and [MH89].

If the impact on the system's temporal behavior caused by a maintenance operation is predicted early, in the design-phase of the change, the risk of introducing errors related these aspects of the system behavior may be minimized. This way the productivity in system maintenance is improved and it is possible to stay longer in phase III of the life cycle depicted in Figure 1.1. Unfortunately, to predict the impact of a maintenance operation is often difficult due to the complexity and the evolving nature of these types of systems. A system expert can often make a qualified guess, but a more detailed analysis is often problematic and time consuming due to the size and complexity of the system. Furthermore, as all human beings make mistakes, it is dangerous to rely on someones subjective judgement. If it discovered that a performed maintenance operation, e.g. the implementation of a new feature, has introduced problems related to the temporal behavior, then large efforts have already been made on implementing a feature that may be too resource demanding for the current system and in need of modifications in order to function properly. The reliance of subjective judgement is far from an ideal solution, but unfortunately the prevalent method in industry today.

The alternative to a system expert's subjective judgment of a change is to introduce *analyzability* with respect to the important properties of the system behavior, i.e. suitable analysis methods that enable engineers to objectively analyze the impact of a change. There are basically two ways of introducing analyzability for complex embedded systems, either *intrusively* or *non-intrusively*. An intrusive approach changes the system in order to be more predicable and analyzable. The major problem with an intrusive approach is the large effort and risks involved. An intrusive approach implies rewriting code or completely redesigning the systems software architecture, an alternative which is costly and will most likely introduce new bugs in the system.

A non-intrusive approach focuses on enabling analysis of the existing sys-

tem. A common approach, such as the approach proposed in this thesis, describes the relevant aspects of the system in a model, which can then be analyzed using a well-defined method, either by hand or using tool support. The model is typically constructed through a process known as *reverse engineering*, i.e. extraction of the software architecture through analysis of the implementation. Interesting results within the area of reverse engineering are presented in Section 2.1.

A non-intrusive introduction of analyzability by modeling of the existing system is an attractive alternative to a major redesign of the system. Given that a sufficiently detailed model exists there exists a variety of formal methods and tools for analyzing properties of the model, i.e. model checkers [Hol97, SPI, Hol03, BLL+95, DY00, BDL04, UPP, BDM+98, DY95, KRO]. However, according to our experience such formal analysis methods are not widely used in industry, apart from in domains with extreme dependability requirements, such as aerospace systems, military systems or nuclear power plants, where system failure may have truly disastrous consequences and development costs are of less importance. Such systems have been designed to be analyzable and formal analysis methods have been used in the whole life cycle of the system. For companies that develop complex embedded systems in less extreme domains, formal analysis is often hard to apply for a number of reasons:

- Suitable models that allow analysis seldom exists in industry today since the need for analyzability has emerged after initial system design, as a result of the increased system complexity. To introduce analyzability in a non- intrusive manner thus requires the construction of a model from the system implementation, a significant reverse engineering effort.
- The systems may have a very complex behavior, too complex to analyze using rigorous analysis methods such as model checking without making many abstractions to reduce the complexity. To make the necessary abstraction is a non-trivial task that requires a deep understanding of the theory behind the analysis method.
- Not all available analysis methods may be applicable for a certain system since many analysis methods make assumptions on the software architecture. Most complex embedded systems have not been designed with analyzability in mind and their software architecture may therefore violate assumptions of the available analysis methods.
- In order to support evolving systems, the model needs to be kept up-to- date with the system implementation, in the same way as text based

documentation. If the model becomes obsolete, a substantial effort may be required to update the model to reflect the current implementation. This effort may cause the system developers to stop using the analysis method, if model maintenance is neglected for some time.

Despite the problems associated with non-intrusive introduction of analyzability, the potential benefits motivate the application of this approach. If errors related to the temporal behavior of the system could be predicted at design time, rather than discovered in system testing or by end users, it could cut costs and development time for the company. However, the non-intrusive approach requires a model of the system and specifications of the properties to be analyzed. Therefore, in order to enable analysis of complex embedded systems's temporal behavior, there are several questions that need to be answered.

- What modeling languages and analysis methods are suitable?
- How do we specify the properties of interest for analysis?
- How can a model be constructed based on an implementation of an existing system?
- How can we assure that a model is valid with respect to the properties of interest?

The first two questions, about modeling language, analysis method and property specification, have been addressed in earlier work [Wal03, AWN04a, WAN03b, WAN$^+$03a, AN02]. This thesis is primarily targeting the two latter questions, how to construct and validate a model of a complex embedded system.

1.1 Our Approach

In earlier work [Wal03, AWN04a, WAN03b, WAN$^+$03a, AN02] an approach has been proposed for introducing analyzability with respect to the temporal behavior of complex embedded systems. Analyzability is introduced by constructing a model describing the timing and behavior of the system. The model is constructed through reverse engineering of the existing implementation and measurements of the timing and behavior of the running system. This model can be used for *behavior impact analysis*, i.e. to predict the impact caused

by a maintenance operation on the temporal behavior of the system. A prototype of the change is implemented on the model and the resulting behavior can thereafter be analyzed and compared with an analysis of the original model.

We refer to the general method as the *ART Framework*. The current implementation of this framework is based on the ART-ML modeling language [Wal03, AWN04a, WAN+03a] and the framework is also named after the modeling language. An ART-ML model describes a system as a set of *tasks*, (semi-) parallel processes that share a single CPU. Each task in a model has a set of attributes, such as scheduling priority, and a behavior description. ART-ML is intended for modeling the temporal behavior of tasks, i.e. how tasks execute over time, how frequently and how long. However, as it is possible to specify behavior for each task in the model, it is also possible to include functional behavior and dependencies between tasks. This allows for very detailed models, which accurately capture the behavior of complex systems.

An ART-ML model is analyzed by executing the model in a simulator, which results in an *execution trace*, a log describing which tasks have been executed, when and for how long. The execution trace is analyzed with respect to a set of properties, that are specified in Probabilistic Property Language [Wal03, AWN04a, WAN03b], using a PPL analysis tool. The properties of interrest for analysis are typically response times and the utilization of logical resources, i.e. properties dependent on the temporal behavior.

1.2 Research Questions

This thesis has a single main research question, Q, which is broken down in two subquestions, Q1 and Q2. By answering the two subquestions, we have answered the main question Q. The context for these questions is the proposed approach for behavior impact analysis with respect to the temporal behavior of complex embedded systems.

Q: *How can models be developed that accurately describe the temporal behavior of complex embedded systems?*

Q1: *What methods are suitable for extracting the information necessary for a temporal behavior model from a complex embedded system implementation containing millions of lines of code?*

Q2: *What methods are suitable for validating models describing the temporal behavior of complex embedded systems?*

No hypotheses are formulated here due to the nature of the questions; instead chapters 3, 4 and 5 propose solutions answering Q1 and Q2. Each of the three chapters is concluded with a discussion that relates the contribution of the chapter to the research questions. Finally, Chapter 6 concludes the thesis by revisiting the research questions and briefly summarizing the proposed solutions.

1.3 Research Approach

The research behind this thesis has been conducted in collaboration with ABB Robotics, a large manufacturer of industrial robots and robot control systems. The author has worked at ABB Robotics with software development for an industrial robot control system, which is a typical example of a complex embedded system. Therefore, the author has a good understanding of the problems associated with complex embedded system development.

The problem described in the introduction was initially identified by ABB Robotics. An on-site study was conducted on the subject in the form of a master's thesis [AN02]. This initial work outlined the approach presented in [Wal03] and further discussed in this thesis.

In order to get feedback on the problem formulation and the approach proposed in this thesis, seminars have been arranged on a regular basis, with system experts from ABB Robotics as well as researchers from other universities. Further, several publications on this subject have been presented on relevant international scientific conferenses. The strong industrial connection enables the research to be focused on problems relevant for industry. In order to verify the scientific relevance and uniqueness, the literature in several related research areas has been studied. The results can be found in Chapter 2.

The work presented in this thesis is primarily focused on the robot control system developed by ABB Robotics. This system is, however, representative for many complex embedded systems, which can be concluded from a study that has been made on several companies in Sweden developing complex embedded systems [MWN+04]. Therefore, an approach suitable for the ABB Robotics will most likely be suitable for many other complex embedded systems. In future work, the solutions proposed in this thesis is to be evaluated in an industrial case study, at first at ABB Robotics and, depending on the result, on other companies developing complex embedded systems as well. This case study is described in Section 6.1.

1.4 Contribution

The approach for behavior impact analysis described in this thesis originates from Anders Wall's Ph.D. thesis [Wal03]. The contribution of this thesis compared to [Wal03] is as follows:

Modeling for behavior impact analysis The thesis presents an approach for how to construct a model for behavior impact analysis, based on an existing system, by extracting information from both source code and from execution traces recorded from the system at runtime.

Model validation In order for a model to be useful, it must be assured that the model is valid, i.e. an accurate description of the intended system at the appropriate level of abstraction. The thesis presents an approach for validating a model intended for behavior impact analysis, based on a set of existing model validation techniques.

Regression analysis An alternative application of the approach presented in this thesis allows a company to study their system's temporal behavior and

compare with previous versions of the system in order to identify unintended effects caused by recent maintenance operations. This approach has been developed in collaboration with ABB Robotics, which recently has begun introduing the proposed analysis method in their software development process.

Tools A set of tools and languages have been developed to enable the three above stated contributions of this thesis:

- The modeling language ART-ML and a discrete-event simulator for ART-ML models.
- The Tracealyzer, a tool for visualization of execution traces. The tool is highly useful in the modeling of complex embedded systems, as it visualizes system's behavior.
- The Property Evaluation Tool, a tool for PPL analysis and comparison of execution traces. The tool is applicable in behavior impact analysis, regression analysis, as well as in model validation.
- A behavior recorder for the RTOS VxWorks has been developed and integrated in a commercial complex embedded system. The thesis presents the design and the performance of the implemented recorder.

1.5 Thesis Outline

This thesis is organized in six chapters. Chapter 2 presents a state-of-the-art report on related research in the areas of reverse engineering, model validation, real-time systems and finally model checking. Chapter 3 presents the use of dynamic analysis for modeling and analysis of complex embedded systems: what information is of interest and the costs of recording this information. The chapter also presents a set of tools that has been developed for analysis and visualization of recordings and an additional use of the developed tools, regression analysis. Chapter 4 presents an approach for modeling the temporal behavior of complex embedded systems. The approach consists of a model framework dividing a model into four components and a process for construction of the model components using dynamic analysis and reverse engineering of the system's implementation.

Chapter 5 discusses the concepts of model validity, model robustness and the threats against model validity. Further, the chapter presents a five-step process for validation of temporal behavior models. The steps in the validation

process may utilize the tools presented in Chapter 3. Finally, Chapter 6 concludes the thesis and outlines future work.

Chapter 2

Temporal Behavior Modeling and Analysis

This chapter is a literature study investigating existing works related to the research questions of this thesis, i.e. how to model the temporal behavior of existing complex embedded systems. This study is broad, as there are several areas of interest. Four research areas have been identified as the most closely related and are described in this chapter:

- Reverse Engineering
- Model Validation
- Real-Time Systems
- Model Checking

The first two sections describe areas related to the construction of models for behavior impact analysis. The two latter areas are related to analysis methods suitable for behavior impact analysis of real-time systems. Reverse engineering is the process of extracting logic, designs and other information from an implementation. This area is highly relevant, as the construction of a model from an existing system is a reverse engineering activity. Section 2.1 presents the area by explaining basic terminology and interesting results. The section also includes results from the software verification community, where model extraction tools are used to extract verification models from source code. This is in essence a reverse engineering process.

Model validation is the process of assuring that a model describes the intended system correctly and with enough accuracy for the analysis in mind. This area has primarily been addressed by the simulation community. Section 2.2 describes results related to model validation, both subjective methods and methods based on statistics.

Real-time systems are systems with requirements on timeliness. This is a huge research area; different aspects of real-time systems have been studied extensivly since the early 1970's. The type of systems considered in this study, complex embedded systems, are typically real-time systems so including real-time systems research in this study comes naturally. Section 2.3 describes the basic concepts and terminology, scheduling algorithms and analytical methods for response-time analysis, simulators for analysis of real time systems and finally worst case execution time (WCET) analysis.

Model checking is a method for verification of models describing e.g. software systems. The method may be used to verify different properties of a software system, e.g. absence of deadlocks and safety properties, but some model checking tools also allow checking of timeliness properties. Model checking may thus be an alternative to the simulation-based approach of this thesis. Section 2.4 therefore presents the general principles, including modeling languages and temporal logics, and three common tools for model checking.

Finally, the chapter is concluded with Section 2.5, discussing on how the approach proposed in this thesis relates to the existing works presented in this chapter.

2.1 Reverse Engineering

The process of extracting information from an implementation (i.e. source code) is commonly referred to as *reverse engineering*. A related term is *reengineering*, which according to the "horseshoe model of reengineering" [CI90] is the process of first reverse engineering an implementation into a higher level of abstraction, *restructuring* the result of the reverse engineering, and finally *forward engineering* in order to introduce new functionallity. An extensive annotated bibliography is presented by van den Brant [vdBKV97] describing around 100 works in the area of Reengineering and Reverse Engineering.

Available tool support for reverse engineering is closely related to this thesis, as the usage of such tools is likely to facilitate the understanding and modeling of complex systems. There are many tools available that can analyze and present different views of a system's static structure, such as UML class diagrams.

2.1.1 Tools for Structural Analysis

Bellay and Gall [BG97] performed a study in 1997, where they presented and compared four Reverse Engineering tools: *Refine/C*, *Imagix 4D*, *Rigi* and *Sniff+*. The comparison was made by applying each of the tools to a commercial embedded system implemented in C. They compared 45 properties of these tools in the four categories: *analysis*, *representation*, *editing/browsing* and *general capabilities*. Examples of properties in the analysis-category are what source languages that are supported and the fault-tolerance of the parser. In the representation-category, properties such as support for filtering and grouping of information can be found. The editing/browsing category contains information about how the tool presents the program text, e.g. syntax highlighting, search support and hypertext capabilities. Finally, in general capabilities we find information about e.g. supported platforms, multiuser support and extensibility.

According to [BG97], Refine/C is an extensible, interactive workbench for reverse engineering of C programs. However, no further information about Refine/C could be found, apart from references in rather old research papers. Refine/C is a product of the company Reasoning Systems, Inc., which no longer supports this tool.

Imagix 4D is a tool for understanding C and C++ programs. It is today available as a commercial product. It can present UML class diagrams, file diagrams and can also perform control flow analysis. It can identify unused variables, present metrics of the individual routines in the code, such as line

count, McCabe complexity, fan in etc.

The third tool studied in [BG97] is Rigi, a public domain tool developed over the last decade by the Rigi Research Project at the University of Victoria, Canada. The Rigi tool can present the dependencies between functions, variables and data types and has a lot of features for filtering and grouping of functions into subsystems. Rigi is also highly customizable. In order to use Rigi, the code that is to be analyzed first has to be parsed into a graph. This is done using a separate program.

The last tool presented in this study is Sniff+. It is not a "pure" Reverse Engineering tool in the traditional sense. Sniff+ is a commercial advanced development environment from WindRiver, for development of large embedded solutions. Sniff+ also supports reverse engineering activities.

A more recent study is the one by Kollmann et al, from 2002. Their study [KSS+02], compares four tools for UML based static reverse engineering: *Together*, *Rational Rose*, *Fujaba* and *Idea*. The first two are commercial products and the latter ones are research prototypes. The tools are compared by using them for analyzing a Java implementation consisting of about 450 classes. Nine properties of the generated information are compared: the number of classes reported, the number of associations reported, types of associations used, handling of interfaces, handling of Java collection classes, recognition of multiplicities, use of role names, handling of inner classes and "class compartment details", i.e. the level of details used in resolving method signatures.

Other Reverse Engineering tools of a more lightweight nature are *Revealer* [PFGJ02] and *Semantic Grep* [BTMG02]. Revealer is a tool for architectural recovery, based on syntactical analysis. It allows searching for complex patterns in source code, corresponding to "hotspots" of a specific architectural view. For instance, the tool can be instructed to extract the hotspots, i.e. the relevant program statements, of socket communication. Revealer does not parse the source code like most of the heavyweight tools do, e.g. Rigi, instead it searches for patterns. It is therefore very error tolerant, allowing analysis of code containing errors or references to missing files. This error tolerance is very useful for e.g. a researcher analyzing a part of a commercial system off site, when the full source code is not available.

Semantic Grep, described in [BTMG02], allows queries on the source code, for instance "*show all functions in parser.c*" or a more advanced "*show all function calls from parser.c to scanner.c*" The tool is based on the established tools grok and grep. It transforms its queries into commands for these tools. This tool is however an academic prototype and does not seem to be available for downloading or purchase.

2.1.2 Tools for Behavior Analysis

Structural analysis tools are of great help for the understanding of complex systems, but do not constitue an adequate solution for understanding a system's behavior. However, there are many works focusing on analyzing the behavior of software, using model extraction tools. These works are highly relevant to this thesis. It is possible that an existing model extraction tool may be used directly or adapted to fullfill the reverse engineering needs of the approach proposed in this thesis.

There are basically two main types of tools that analyze the behavior of software systems; those who analyze the source code (static analysis), and those who analyze traces from the running system (dynamic analysis).

Static Analysis There are many works related to reverse engineering in the area of model checking. Many model checkers for software can analyze implementations in general purpose languages such as C or Java. Some of these tools translate the program into a modeling language, such as Promela, and perform abstractions by removing details irrelevant for the properties that are to be analyzed. This is the approach of the tools SLAM [BR01], BLAST [HJMS03], FeaVer/Modex [HS99] and Bandera [CDH+00].

SLAM is a toolkit developed by Microsoft Research, for checking safety properties of system software. In [BR01] a case study is presented where the toolkit has been used to verify Windows NT device drivers. The SLAM toolkit contains three tools. First, the tool *C2BP* is used to generate an abstraction of the C program, called a *boolean program*. Such programs are basically C programs, but contain only Boolean variables and may also contain nondeterministic selection. The abstraction is made with respect to the properties of interest for analysis, specified as state machines in the specification language *SLIC*. The Boolean program is analyzed using the *BEBOP* model checker in order to find a path through the program that violates any of the specified safety properties. If such a path is found, the tool *NEWTON* is used to verify that the path is possible in the real program.

BLAST, the Berkeley Lazy Abstraction Software verification Tool [HJMS03], is another solution for checking safety properties of C programs. To specify a safety property to check, a special *error location* is added to the program. If the code corresponding to the error location is executed, it represents a violation of the property. The tool transforms a C program into an abstract model, based on the property to check. The model of the program is internally represented using *control flow automata*, CFA.

Model checking is then used in order to search all possible locations of the model to determine if the error location is reachable or not. If the error location is not reachable in the model, BLAST reports that the program is safe and also provides a proof of this. If there is a path to the error location in the model, it is verified that the path is possible in the real program by using symbolic execution. If the path is possible, it is reported to the user; otherwise the model is refined by changing the abstraction process.

BLAST has been used in case studies, refered in [HJMS03], to verify safety properties of e.g. Windows and Linux device drivers. In some cases, bugs have been found and in other cases BLAST proved that the drivers correctly implemented a specification.

An interesting result is the tool FeaVer/Modex/AX [HS99, Hol00], from Bell Labs. There is a name confusion regarding this tool. FeaVer is the user interface for this toolkit while Modex is an acronym of Model Extractor, a tool for extracting verification models from ANSI C. Modex was previously known as AX (Automata Extractor). The output format of Modex is Promela, the input language of the software model checker SPIN. Modex first parses the C code and generate a parse tree. Thereafter it processes all basic actions and conditions of the program with respect to a set of rules, resulting in a Promela model.

This approach effectively moves the manual effort from constructing the model to defining the table of rules. The rules specify what statements that should be translated into Promela (and how) and what to ignore. There is a large set of default rules that can be used, but the user may add their own rules to improve the quality of the resulting model. Modex is available for download, and it seems very possible to customize Modex for other purposes than the generation of Promela models, due to the customizable rule table and open source code.

Bandera [CDH$^+$00] is an integrated collection of program analysis and transformation tools for automatic extraction of finite-state models from Java code. The models can be used for verifying correctness properties using existing model checking tools. No model checker is included; instead Bandera is designed to interoperate with existing, widely used model checkers such as SPIN and SMV. The authors of [CDH$^+$00] argue that the single most important method for extracting analyzable models of software is abstraction. Their goal is to provide automated support for the abstractions used by experienced model designers. Bandera uses techniques from the areas of program slicing [Tip95, Wei81] and abstract interpretation in order to eliminate irrelevant program components and to support data abstraction. They argue that specialized

models should be used for checking specific properties rather than developing a general model describing many aspects of a program. That way, the model can be optimized for analysis of that single property and thereby smaller and less complex. This is relevant as a major problem with model checking techniques is the state space explosion problem. Developing property specific models is rarely done when modeling systems by hand, due to the effort required, but if models are automatically generated, it is an option.

A different approach is the one used in VeriSoft [CGP02], also from Bell Labs. It is a model checker for software systems from. It is not a traditional model checking tool, in the sense that no model is required. VeriSoft uses the source code itself as the "model" to check. Verifying the behavior of a concurrent system using VeriSoft is similar to traditional testing, the difference is that it executes under the control of VeriSoft, which systematically explores the behaviors of the system. This requires that the system that is to be verified can be compiled and executed on a platform supported by VeriSoft, which today are limited to SunOS and Linux. Most embedded systems use other platforms such as VxWorks [WRW] or OSE [OSE]. The system to be verified could probably be ported to one of the supported platforms, e.g. Linux, but that is often associated with a major effort.

Dynamic Analysis The use of dynamic analysis techniques for the modeling of complex embedded systems is very interesting, as the resulting models may contain realistic timing information. This is for instance the case in research by Jensen [Jen98, Jen01], described later in this section. This kind of information is not possible to obtain using static analysis only. Unfortunatly, there is not much existing work dealing with reverse engineering of real-time systems.

One interesting study is the one presented by Marburger and Westfechtel in [MW03]. They report on a set of reverse engineering tools, developed in cooperation with Ericsson Eurolab Deutschland, including support for both structural analysis and behavioral analysis. The behavioral analysis includes state machine extraction from PLEX source code (a proprietary asynchronous real-time language). Traces recorded from a system emulator can be used to animate the state machines in order to illustrate the system behavior. This is basically low-speed simulation, using pre-recorded data to stimulate the model. The extraction of state machines from source code is highly related to construction of models for behavior impact analysis, unfortunately this study focuses on telecom system and the Ericsson-specific PLEX language.

An interesting study related to [MW03] is that by Systä and Koskimies [SK98] where state diagrams are synthesized from traces. The source code

of the system in focus is instrumented in order to generate a trace. The trace is then fed into the SCED tool, which generates a (minimal) state diagram corresponding to the observed behavior. The work does however not address real-time systems, no timing information is recorded.

A system called DiscoTech is presented in [YGS+04]. Based on runtime observations, an architectural view of the system is constructed. If the general design pattern used in the system is known, mappings can be made that transforms low level system events into high level architectural operations. With this information an architectural description of the system can be constructed. The system presented is designed for Java based systems. The types of operations that are monitored are typically object creation, method invocation and instance variable assignments. Note that the resulting model describes only the architectural structure of the system and does not include any behavioral descriptions.

Relevant research addressing real-time systems is the approach of Jensen [Jen98, Jen01], for automatic generation (synthesis) of behavioral models from recordings of a real-time systems behavior. The resulting model is expressed as timed automata for the UppAal model checking tool [BLL+95, DY00, BDL04, UPP].

The aim of the tool is testing properties such as response time of an implemented system, against implementation requirements using model checking. For the verification, it is assumed that the requirements are available in the form of timed automata which are then parallel composed with the synthesized model by the UppAal-tool to allow model checking. Jensen's thesis includes a schedulability test that (instead of WCET) uses a measure called Reliable Worst Case execution time (RWC). RWC is a statistical measure that is introduced in the thesis. As a proof of concept, Jensen includes a one shot experiment of the model synthesis.

The work by Jensen assumes that the system conforms to a generic architecture as follows: a system has a set of abstract *tasks* that each are implemented as a sequence of *subtasks* distributed over several servers. The allocation of subtasks to servers is derived from requirements such as periodicity, deadline, etc. Thus, each *job* of a task is a sequence of interactions with *subjobs* on several servers.

Jensen imposes restrictions on how selections are used in the model – no selections are allowed within the subtasks, they can only occur at the start of the job or after a message from another subtask has been received. Another restriction is an assumption of normal distributed subtask execution times – in real implementations, services (subtasks) often have complex distributions

consisting of several “peaks”, corresponding to selections between different behaviors.

2.2 Model Validation

When constructing a model of the behavior of a software system, model validation is necessary in order to assure that the model accurately describes the system at an appropriate level of abstraction. By validating the model, the analyst and system experts gain enough confidence in the model in order to trust its predictions.

However, the validation of a model is far from trivial, since a model is an abstraction of the real system. The validity of models have been studied in the simulation community. In [LM01], model validation is defined as ”*the process of determining whether a simulation model is an accurate representation of the system, for the particular objectives of the study*”. Their paper targets validation of models in general, e.g. describing a physical process. One of the authors of [LM01] has authored a book on simulation studies, “Simulation, Modeling and Analysis” [LK93], where one chapter covers model validation. The book presents two statistical methods for comparing a model with the corresponding real system:

- Inspection approach: to compute one or more statistics from the real world observation and the corresponding statistics from the model output data, and then compare the two sets of statistics without the use of a formal statistical procedure.
- Confidence-interval approach: a more reliable but also more demanding method. Several independent observations are made of the real system as well as of the corresponding model. From each observation the average value is calculated for the property that is to be compared. This result in two sets of average values where each value represents an observation, one set of values from the model and one set of values from the real system. These two sets of average values are compared and a confidence interval can be constructed using statistical methods. This confidence interval reveals if the difference is statistically significant, and also gives an indication of how close the model is to the system, in this particular aspect.

In [Bal90] guidelines are provided for conducting successful simulation

studies. The paper presents a life cycle for a simulation study, containing 10 processes (phases):

- problem formulation,
- investigation of solution techniques,
- system investigation,
- model formulation,
- model representation,
- programming,
- design of experiments,
- experimentation,
- redefinition, and finally,
- presentation of simulation results.

Associated with these processes are 13 credibility assessment stages, including model validation. According to [Bal90] there are basically two main techniques for model validation: *subjective validation techniques* and *statistical validation techniques*. The paper presents a summary of common subjective validation techniques, of which the most interesting are:

- Face Validation: This is a useful preliminary approach. System experts are allowed to study the model and subjectively compare the model with their knowledge of the system.
- Graphical Comparison: A subjective, but according to [Bal90] and the authors experiences also a practical method, especially useful as a preliminary approach. By presenting data based on the model and data from the real system, graphically, patterns can easily be identified and compared.
- Predictive Validation: The model is driven with past (real) system input data and its predictions are compared with the corresponding past system output data. Obviously, this requires that there are measurements made of the real systems input and corresponding output.

- Sensitivity Analysis: To systematically change values of model input variables and observing the effect on model behavior. Unexpected effects may reveal flaws in the model. This is discussed in Chapter 5.
- Turing tests: System experts are shown two anonymous outputs, one from the model and one from the real system, generated from identical inputs. The experts are asked to identify which is which. If they succeed, they are asked how they did it, and their feedback is used to improve the model.

The paper [Bal90] also lists 22 statistical techniques which have been proposed for use in model validation, but the techniques are not described further. Model validity from a general simulation point of view is also discussed in [Sar99]. Different processes for validation of models are described in the paper; one process is *Independent Verification and Validation*, IV&V. It states that a third party reviewer should be used to increase the confidence in the model. A scoring model is also described, where various aspects are weighted and a total score can be calculated as a measure of validity for the model. This is, as pointed out in the paper, dangerous since it appears more objective than it really is and may result in over-confidence in the model validity. The author describes a simplified version of the modeling process described in [Bal90], consisting of the *Problem Entity* (the system), a *Conceptual Model* (the understanding of the system), and a *Computerized Model* (the implementation of the Conceptual Model). Furthermore, Conceptual Model validity is defined as the relationship between the Problem Entity and the Conceptual Model, i.e. if the person constructing the model had a correct understanding of the system. Operational Validity is the relationship between the Computerized Model and the Problem Entity, i.e. if the Computerized model was correctly implemented.

In [LM01] many aspects of the validity of models in general are discussed and a seven-step approach for conducting a successful simulation study is described. This approach requires a high level of abstraction and can be applied on any domain. The steps are:

- problem formulation,
- collecting data and construction of the conceptual model,
- validation of the conceptual model,
- programming the model,
- validation of programmed model,

- experiments and analysis, and
- presentation of results.

The paper stresses the importance of a definite problem formulation, comparisons between the model and the system, and the use of sensitivity analysis.

2.3 Real-Time Systems

A *real-time system* is a system where correct behavior is not only dependent on what results that are delivered, but also when they are delivered, i.e. a computer system that has demands on timeliness. Real-time systems are often connected to machinery, i.e. sensors and actuators, controlling a physical process. The demands on the timeliness, the temporal constraints, on such systems are defined by the process that is controlled. The main problem in real-time system research is to guarantee the timeliness.

Real-time systems are often composed of *tasks*, processes, usually communicating with each other. The *response time* of a task in a real-time system is the latency from stimuli (input) to reaction (output). A task's response time is effected by both the *execution time* of the task, i.e. the CPU time required to process the code of the task, as well as interference from other tasks in the system with higher priority and blocking semaphores. If a task is allowed to execute without disturbances, the response time of the task will be equal to its execution time.

A real-time system has deadlines, specifying the maximum response time allowed. If a real-time system is unable to finish a task before its deadline, it is a *deadline miss*. The deadline miss might be caused by a global *overload situation*, i.e. that the currently active tasks in the system together require more CPU-time than available in order to finish before their corresponding deadlines, i.e. the CPU utilization is above 100 %. The handling of overload situations is a major area within real-time research. A deadline miss may however occur in other situations, e.g. if a deadlock situation occurs in a task with a deadline, the task can not be completed, even though the CPU may be idle.

Real-time systems are often divided into two categories based on the severity of the consequences of a deadline-miss. A *soft real-time system* allows some occasional deadline-misses. An example is a telecom system. The systems temporal requirements do not need to be guaranteed at all times. It is not a disaster if a phone call is disconnected in rare circumstances, as long as it does not happen recurrently. Another example of a soft real-time system is

DVD player software on a PC, which must decompress a certain number of frames every second. The temporal requirements are in this case more focused on quality of service rather than 100% reliability. A software DVD-player can tolerate small transient delays in the video processing; this does not result in a failure, only a minor disturbance in a reduced quality of the result, which the user (viewer) might not even notice.

In a *hard real-time system* a single missed deadline is considered a failure. If the system is *safety-critical* it might result in injuries or catastrophic damage. An example is modern all computer controlled "fly-by-wire" airplanes, such as the Swedish fighter-jet JAS 39 "Gripen" or the Boeing 777. Another example in a different domain is railway signaling system. For such safety-critical real-time systems, there is a need to guarantee that the system will never violate its temporal requirements.

A large area within real-time research is *scheduling theory*, i.e. algorithms for selecting the next task to execute in a multitasking system. The scheduling algorithms can be divided into offline and online scheduling. When using online scheduling, the scheduling decisions are taken during runtime. An offline-scheduled system makes no decisions regarding the execution order of the tasks during runtime, as a pre-calculated schedule is used. However, in such systems it is not possible to create new tasks in runtime since adding of new tasks to the system requires reconstructing the schedule. A more flexible scheduling policy is online scheduling. In this case, no schedule exists, but the operating system makes all the scheduling decisions during runtime.

A very common algorithm for online scheduling is FPS (Fixed Priority Scheduling). Each task has a priority, which is used by the operating system to select the next task to execute if there is more than one task ready. Many commercial real-time operating systems, such as VxWorks from WindRiver [WRW], uses *preemptive* fixed priority scheduling, i.e. the executing task may be preempted by other tasks with higher priority, at any time.

The EDF algorithm, Earliest Deadline First, is another common online scheduling algorithm. EDF always selects the task with least time left until deadline, i.e. the task with earliest deadline. EDF guarantees that all deadlines are met if the CPU-utilization (U) is less than 100 %. In an overload situation ($U > 100\%$) it is not possible to finish all tasks before their corresponding deadlines. EDF is not a good algorithm in overload-situations. Since it does not do anything to lower the CPU-utilization, i.e. reject tasks, it tends to let every task miss their deadline. EDF can however be combined with other scheduling algorithms, such as overload handling or aperiodic server algorithms such as Total Bandwidth Server [SB94] or Constant Bandwidth Server [Abe98].

2.3.1 Analytical Response-Time Analysis

There are a variety of analytical methods for schedulability analysis, i.e. to determine if a real-time system is schedulable with respect to the deadlines of its tasks. In this section, we present the seminal results within scheduling theory and the analytical response-time analysis methods commonly known as RTA. One of the most well known results in the real-time community is the one by Liu and Layland from 1973 [LL73], where they introduced fixed priority scheduling which is widely used today in many real-time operating systems. They showed that a system with strictly periodic and independent tasks that is scheduled using fixed priority scheduling is always *schedulable*, i.e. will meet its deadlines, if the total CPU utilization (U) is below a certain value, the *Liu-Layland bound*, and the tasks have been assigned priorities according to the *rate monotonic* policy. Rate monotonic is a policy for assigning priorities to the tasks based on their rate, i.e. period time, where the task with highest rate receives the highest priority; the task with second highest rate received the second highest priority, and so on.

The value of the Liu-Layland bound is dependent on the number of tasks in the system, but for a large number of tasks, the value is approximately 69 %. For systems containing only tasks with harmonic periods, the bound is 100 %.

Another important result is the *Exact Analysis* [MJ86] presented by Joseph and Pandya in 1986. It is a method for calculating the worst case response-times of periodic independent tasks with deadlines less or equal to the periods, scheduled using fixed priority scheduling. It is an iterative method that from a set of tasks calculates the worst case response time for each task, i.e. the response time of the tasks in the situation when all tasks are ready to execute at the same time, the crititical instant, and executes with their worst-case execution time. The method has later been extended to handle e.g. semaphores [But97], deadlines longer than the periods [Leh90], variations in the task periodicity (release jitter) [Tin92, ABRT93] and distributed systems [TC94]. This family of methods for response time analysis is commonly known as RTA.

2.3.2 Simulation based Analysis

Another method for analysis of response times of software systems, but also of other properties, is the use of a simulation framework. Using simulation, rich modeling languages can be used to construct very realistic models. Often ordinary programming languages, such as C, are used in combination with a special simulation library. This is the case for both the DRTSS [SL96] and Vir-

tualTime [RSW] simulation frameworks, described below. The rich modeling languages allow modeling of the semantic dependencies between tasks in the system, e.g. communication, synchronization and shared state variables. This makes the model more accurate and also easier to analyze, since the dependencies reduces the number of possible execution scenarios. Simulation models may also be non-deterministic, for instance using probability distributions. A simulation model of a real-time system may use probability distributions to describe e.g. execution times of tasks with high realism.

A large problem with simulation is the lower confidence in the result, in comparison to other analysis methods. An analysis of a model based on (random) simulation is not exhaustive; instead a simulator randomly executes the model and only explores a minor and random subset of the possible execution scenarios. Even though it is possible to perform a large amount of simulations of a certain scenario in a short time, the number of possible execution scenarios, i.e. the state space, is often too large for an exhaustive analysis, especially if the model uses probability distributions or other sources of non-determinism. On the other hand, simulation allows for an analysis, even though not exhaustive, in situations where other analysis methods fail.

STRESS A tool-suite called STRESS is presented in [ABRW94]. The STRESS environment is a collection of tools for "analyzing and simulating behavior of hard real-time safety-critical applications". STRESS contains a special-purpose 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 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.

DRTSS The DRTSS simulation framework, presented in [SL96], allows its users to easily construct discrete-event simulators of complex, multi-paradigm, distributed real-time systems. Preliminary, high-level system designs can be entered into DRTSS to gain initial insight into the timing feasibility of the system. Later, detailed hierarchical designs can be evaluated and more detailed analysis can be undertaken. DRTSS is a member of the PERTS family of timing-oriented prototyping and verification tools. It complements the PERTS schedulability analyzer tool by dealing with complex real-time systems for which analytical schedulability analysis is difficult or impossible.

VirtualTime A very recent commercial simulation framework is VirtualTime [RSW]. It is suitable for analysis of the temporal behavior of complex systems, typically soft real-time systems. The simulation framework allows detailed models including process interactions, scheduling, message passing, queue behavior and dynamic priority changes. According to the company behind VirtualTime, Rapita Systems ltd, there are few limitations to the models that can be produced using VirtualTime. However, this solution is primarily targeting telecom systems and as far as we know only available for the systems based on the OSE operating system [OSE], from ENEA [ENE]. Rapita Systems is a spin-off company from the Real-Time Systems Research Group at the University of York, UK.

2.3.3 Execution Time Analysis

When modeling a real-time system for analysis of timing related properties, the model needs to contain timing information, e.g. execution times. A common method in industry, and the approach of this thesis, is to obtain timing information by performing measurements of the real system as it is executed under realistic conditions. The major problem with this approach is that we are unable to determine if the worst case execution time (WCET) has been observed. If the model is populated with execution time data from measurements, we risk a too optimistic model, as the real system sometimes might have longer execution times than our model specifies.

Measuring is however not the only approach to obtain execution times. WCET analysis is a well studied area in program analysis and real-time systems research. Static WCET analysis tools compute a safe, but tight, upper bound for the execution time of a program on a specific hardware. On hardware platforms with rather simple CPUs, such as 8 bit microcontrollers, the WCET can be accurately calculated, but on more complex hardware architectures, with cache memory, pipelines, branch prediction tables and out-of-order execution, estimating a tight but safe WCET is very difficult, due to the complex behavior of the hardware. The WCET analysis tool can not predict every possible behavior of the hardware and is therefore forced to make some worst case assumptions in order to report a safe WCET estimate. Due to these assumptions the estimated WCET becomes pessimistic. Also, static WCET analysis is dependent on a timing model of the hardware, which is a threat to model validity as the real hardware might, in some situations, have a different temporal behavior than the timing model specifies.

An interesting approach is that of Bernat et al [BCP03, BCP02]. Their

solution, probabilistic WCET, combines the strengths of static WCET analysis and measuring of the real system. The pWCET approach basically measures the execution times of the individual basic blocks in the program and use the worst cases observed locally in a static analysis, based upon the object code.

This approach is not dependent on a model of the hardware, as in the case with static WCET analysis; instead the approach relies on the execution time measurements. The dependence on a hardware timing model is a major criticism against the static approach, as it is an abstraction of the real hardware behavior and might not describe all effects of the real hardware. On the other hand, this is a probabilistic approach, based on measurements, and may therefore be optimistic in some cases by reporting too low worst case execution time.

2.4 Model checking

Model checking is a method for verifying that a (model of a) system meets its requirements, and has been proposed as a method for software verification, including verification of timeliness properties for real-time systems. The method is commonly used to verify hardware designs, communication protocols etc. In recent years model checkers for software have been developed and proposed as complementary method to testing, code inspections etc. This section will describe the basic concepts of model checking and temporal logics, a commonly used model checker as well as two model checkers especially targeting real-time systems.

2.4.1 Basic Concepts

By describing the behavior of a system in a model, where all constructs have formally defined semantics, it is possible to automatically verify properties of the system using a model checking tool. The model is described in a modeling language, the input language of the tool, often a variant of finite-state automata. A system is often modeled using a network of automata, where the automatons are connected by synchronization channels. When the model checking tool is to analyze the model, it performs a *parallel-composition*, resulting in a single, much larger automaton, describing the behavior of the complete system.

The properties that are to be checked are usually specified in a temporal logic, such as CTL [CE82] or LTL [Pnu77]. Temporal logics allows specification of safety properties, i.e. ”something (bad) will never happen”, and liveness

properties, i.e. ”something (good) must eventually happen”. An example of a safety property in CTL is as follows:

```
AG not (A and B)
```

The formula states that A and B may never be true at the same time, using the temporal operator AG (“always”). CTL contains several temporal operators, apart from AG, and is presented further in Section 2.4.3.

The benefits and problems of model checking have been discussed extensively in several works, e.g. [Kat98]. Model checking is a general approach, as it can be applied to many domains such as hardware verification, software engineering, communication protocols and embedded systems. Model checking has been shown to be usable in industrial settings for finding subtle errors that are hard to find using other methods and according to [Kat98], case studies have shown that the use of model checking does not delay the design process more than using simulation and testing. Also, model checking is based on a sound mathematical foundation, including e.g. modeling, semantics, concurrency theory, logic and automata theory.

There are also problems associated with model checking. One of the most well-known problems is commonly known as the state-space explosion problem. When modeling a non-trivial system, the number of states in the complete behavior of the system, the state space, easily becomes very large. This as the state space grows exponentially with the number of parallel processes. This is a serious problem, as model checking tools needs to search the state space exhaustively in order to verify or falsify the property to check. If the state space becomes too large, it is not possible to perform this search, due to memory or run time constraints. Model checking is appropriate for control-intensive applications, such as communication protocols, but it is less suited for data-intensive applications, as the treatment of data usually leads to infinite state spaces. Another problem when using model checking, or any model based method, is model validity. As the tool do not verify the system, but a model of the system, it is very important that this model is an accurate description of the system, otherwise the analysis results are not trustworthy.

2.4.2 The model checker SPIN

SPIN is a well established tool for model checking and simulation of software [Hol97]. SPIN supports simulation (random, guided and interactive) and model checking of formulas in the temporal logic LTL [Pnu77]. According to [SPI]

SPIN is designed to scale well and can perform exhaustive verification of very large state-space models. The modeling language of SPIN is called Promela, "PROcess MEta Language". Promela is a guarded command language with a syntax similar to the programming language C. SPIN is open-source and available for most platforms, including Linux, Windows and Mac. For further information about SPIN, there is a book [Hol03] by Holzmann containing tutorials on using SPIN and Promela, as well as reference material.

Promela A Promela model consists roughly of a set of sequential processes, local and global variables and communication channels. Each process is a sequence of statements, where each statement may be enabled or disabled. A disabled statement blocks the execution of the process until the statement becomes enabled.

Promela support non-deterministic selection. The if-statement allows several alternative behaviors to be specified. Each behavior may be associated with a *guard*, a condition, just like in e.g. C, but if several guards are enabled, i.e. true, only one is selected, in a non-deterministic way. As an example, consider the following:

```
if :: (a $>$ 10) -$>$ smtA;
   :: (true) -$>$ smtB;
   :: (true) -$>$ smtC;
fi;
```

The two last statements are always enabled (true) and may therefore be executed, but the first has a guard allowing execution only when "a" is more then 10. Promela also supports loops, using the do-statement; the syntax is similar to if.

```
i = 1;
do :: i <= 10 -> looping;
   :: i > 10 -> break;
od;
```

Promela processes may communicate using communication channels. A channel is a fixed size FIFO buffer. The size of the buffer may be 0; in such a case it is a synchronization operation, requiring that the send and receive operation occurs simultaneously. If the buffer size is 1 or more, the communication becomes asynchronous, as a send operation may occur even though the receiver is not ready to receive. To declare and use channels is very straight-forward.

A send-operation is expressed using a "!" together with the channel name and data. A receive-operation is similar, using "?": The following example demonstrates how to declare a channel and use it for communication.

```
chan chn = [4] of byte; /* four slots */
...
chn ! 42 /* send data ''42'' to chn */
...
chn ? foo /* receive from chn */
```

A process may be instantiated and invoked dynamically and processes may be executed in parallel. For instance, consider the following example, a simple but complete Promela model:

```
proctype prc(byte ident)
{
    printf("%d\n",ident);
}

init{
    atomic{
        run prc(1);
        run prc(2);
    }
}
```

The init-section specifies the entry point, similar to "main" in most programming languages. The atomic-statement allows the two processes start at the same time. The observant reader might notice that the printf-statements have the same syntax as in C.

LTL To specify the properties of the model to check, linear temporal logic (LTL) is used. LTL is classic propositional logic, extended with temporal operators. Using LTL for program verification was first proposed in [Pnu77]. The LTL operators that are supported by SPIN are:

```
[]  - always
<>  - eventually
!   - logical negation
U   - strong until
```

```
&&  - logical and
||  - logical or
->  - implication
<-> - equivalence
```

As an example, the following LTL formula specifies that the logical proposition p should remain true at least until q becomes true:

```
[](p U q)
```

2.4.3 Model checking Real-time Systems

Model checkers such as SPIN do not have a notion of time, and can therefore not analyze requirements on timeliness, e.g. "if X, then Y must occur within 10 ms". There are however tools for model checking of real-time systems. The most well-known are UppAal, first proposed in [BLL+95] and further described in e.g. [DY00, BDL04, UPP] and KRONOS [BDM+98, DY95, KRO], both described later in this section. These tools analyze models described in *timed automata* using variants of the temporal logic CTL.

Timed Automata Timed automata were first proposed by Alur and Dill in [AD94]. They basically extended regular finite automata with real-valued clocks. A timed automaton may contain an arbitrary number of clocks, which run at the same rate. There are extensions of timed automata where clocks can have different rates [DY95]. The clocks may be reset to zero, independently of each other, and used in conditions on state transitions and state invariants. A simple yet illustrative example is presented in Figure 2.1, which was generated in UppAal.

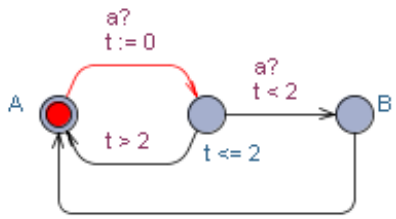

Figure 2.1: A small example of (UppAal) timed automata

The modeled system in Figure 2.1 changes state from A to B if event "a" occurs twice within 2 time units. There is a clock, "t", which is reset after an initial occurrence of event "a". If the clock reaches 2 time units before any additional event "a" arrives, the invariant on the middle state forces a state transition back to the initial state A.

CTL Both the UppAal and KRONOS model checkers uses variants of CTL, Computation Tree Logic [CE82]. CTL is a branching-time temporal logic, meaning that in each moment there may be several possible futures, in contrast to LTL. Therefore, CTL allows for expressing possibility properties such as "*in the future, X may be true*", which is not possible in LTL. On the other hand, CTL can not express fairness properties, such as "if A is scheduled to run, it will eventually run". Neither of these logics fully includes the other, but there are extensions of CTL, such as CTL* [EH84], which subsume both LTL and CTL.

A CTL formula consists of a state formula and a path formula. The state formulae describe properties of individual states, whereas path formulae quantify over paths, i.e. potential executions of the model. The path formulae may be nested, allowing more complex expressions. Apart from ordinary propositional logic, CTL contains four temporal operators:

```
EX - for some time next
E  - for some path
A  - for all paths
U  - until
```

Based on the four temporal operators and the propositional logic, it is possible to derive an additional five very usable, temporal operators:

```
EF  - possible
AF  - inevitable
EG  - potentially always
AG  - always
AX  - next
```

UppAal The tool UppAal [BLL+95, DY00, BDL04, UPP] is based on Timed Automata and a subset of CTL. UppAal is an integrated tool environment for the modeling, simulation and verification of real-time systems. This tool has been developed jointly by Basic Research in Computer Science at Aalborg

University, Denmark, and the Department of Computer Systems at Uppsala University in Sweden.

UppAal is described as "appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables." In practice, typical application areas include real-time controllers and communication protocols where timing aspects are critical. The tool was first proposed in the mid 90's and after almost ten years of development it has now reached version 3.4. The tool is available for many platforms including Windows and Linux, and can be downloaded without charge from the UppAal website [UPP].

UppAal extends Timed Automata with support for e.g. automaton templates, bounded integer variables, arrays, and different variants of restricted synchronization channels and locations. The query language used is a simplified version of CTL, where nested path formulae are not allowed. The subset of CTL allows reachability properties, safety properties and liveness properties. Timeliness properties are expressed as conditions on clocks and state in the state formula part of the CTL formulae.

KRONOS Another well-known model checker for real-time system is Kronos [BDM+98, DY95, KRO] which has been developed at Verimag in France. Like UppAal it is based on Timed Automata but uses a more powerful query language, Timed Computation Tree Logic (TCTL). Timed Computation Tree Logic was proposed in [ACD93], where they extended CTL with quantitative time for the purpose of specifying timeliness properties, i.e. liveness properties with a deadline. Kronos also allows for checking safety properties as well as both forward and backward reachability. Further, Kronos can also check models and properties expressed in other, less common formalisms. The tool is available for several platforms, including Windows and Linux, and can be downloaded without charge at the Kronos website [KRO].

2.5 Discussion

This section discusses the works presented in this chapter with respect to the approach of this thesis, outlined in Section 1.1. This discussion is divided into Section 2.5.1, that discuss the works related to modeling and model validation, and Section 2.5.2 that discuss the different methods available for analysis of temporal behavior models.

2.5.1 Modeling

Existing work related to modeling the behavior of software systems can be divided into two categories, approaches using dynamic analysis [MW03, SK98, YGS+04, Jen98] and static analysis approaches [BR01, HJMS03, CDH+00, HS99]. The works based on static analysis are in general more mature than the dynamic approaches that have been found, but they do not represent a sufficient solution for real-time systems as they do not model the temporal behavior, e.g. execution times. Information about execution times is necessary in order to model and analyze the timing of a system. Static analysis techniques can contribute when constructing models for behavior impact analysis, but they can probably not replace dynamic analysis.

Much research has been done in the estimation of a programs worst case execution time (WCET), but the author has not found any works combining static analysis model extraction with WCET analysis. Further, the approach of this thesis, simulation-based analysis, also requires realistic execution-time distributions representing typical execution of the system. However, WCET estimations may be used in a temporal behavior model to complement the execution time distributions that are obtained from measurements.

Dynamic analysis can provide the realistic execution time distributions required, but unfortunately not without problems. Dynamic analysis implies to observe and record the behavior of a software system, which requires adding monitoring functionality to the system, which may reduce the system performance. Also, a dynamic analysis only models the behavior that has been observed. Therefore, the confidence of the resulting model is highly dependent on the test-cases executed when monitoring the system. The problems with dynamic analysis are discussed in depth in Chapter 3.

To assure that a model correctly describes the intended system, the model needs to be validated. Works exist regarding validation of models, but mainly in the simulation community, while the model checking community seems to take model validity for granted. In many cases, model checkers are used to

verify a specification of a system that has not yet been implemented. In such a case, this assumption might be valid; the question is in that case if the implementation conforms to the specifications. However, if the model describes an existing system and is the result of a major reverse-engineering effort, the model validity can not be assumed.

The results found in the simulation community include two main classes of model validation techniques, subjective techniques (i.e. inspection) and those based on statistics. Both can be used to validate models in this approach. Chapter 5 describes a process for model validation including statistical techniques adapted for the temporal behavior models of our apporach.

2.5.2 Analysis

There exist many analytical methods in research literature for response times analysis, i.e. RTA [ABD+95, LL73, MJ86]. However, these analytical models used by RTA are not expressive enough in order to capture the behavior of large and complex systems. RTA does not consider the behavior of the tasks, only their individual worst-case execution time. The result of such an analysis may therefore be very pessimistic, as tasks may have large variations in the execution time and the theoretical worst case situation, i.e. that all tasks wish to execute at the same time, and all with the individual worst-case execution time, may not even be possible in the real system due to dependencies between the temporal behaviors of the tasks. The worst case execution times of two tasks may be mutually exclusive, e.g. if they are associated to different states of the same shared state variable.

Moreover, RTA targets timeliness properties only, i.e. whether or not any deadlines are violated. In many real systems the temporal requirements are not specified in terms of deadlines, but may be specified as invariants on the functional behavior. In some situations, it may be possibe to derive task deadlines from such requirements, but in other cases that is hard. A typical example is a FIFO data buffer, shared between two tasks, one “consumer” and one “producer”. The invariant is that the buffer must never be empty when the consumer attempts to read. This requirement is formulated in terms of the functional behavior but highly dependent on the temporal behavior of the two tasks involved. Such requirements can not easily be verified by using the existing methods for response-time analysis. Even though fixed priority scheduling is a common scheduling algorithm in complex embedded systems, RTA may be problematic to apply since many systems are not designed to allow analyzability. They might contain aperiodic tasks scheduled with a fixed priority, or tasks that alter

their priority.

On the other extreme are model-checking methods with rich modeling languages such as timed automata [AD94, BDL04]. Timed automata allows for the modeling of temporal behavior as well as functional behavior. By using synchronization channels we can model dependencies between tasks in a system. However, model-checking does not scale properly to larger systems due to the state-space explosion which makes such an approach hard to apply on complex embedded systems. Simulation is better from that point of view. Using simulation, rich modeling languages can be used to construct very realistic models, using e.g. realistic distributions of execution times. A disadvantage of the simulation approach is that we can not be confident in finding the worst possible temporal behavior through simulation, since the state-space is only partially explored.

In the work preceeding this thesis we have chosen to focus on simulation, as we basically have one major trade-off to consider: being able to predict something at the cost of precision. Even though a simulation is not an exhaustive analysis and thus might fail to analyze the worst case situation, it may still point out potential problems and assist the developers in making the right decisions, while analytical methods are often not applicable in practice, either due to a too simple model or to the state space explosion problem.

Even if an exhaustive, “safe”, analysis would be applicable, the analysis results are not necessarily ”safe”, as the trustworthiness of the analysis results depends on the model used for the analysis. Thus, regardless of analysis method, there are always uncertainties due to the problems associated with modeling and validation of models, especially when considering modeling of large complex software systems. This thesis is therefore focused on modeling and model validation rather than analysis methods.

Chapter 3

Dynamic Analysis

This chapter presents how dynamic analysis can be used to improve analyzability of complex embedded systems. The term dynamic analysis captures a broad spectrum of program analyses that deal with data produced by programs during runtime. This chapter presents how dynamic analysis can be used in order to analyze and visualize the temporal behavior of complex embedded systems.

Our approach to modeling the temporal behavior of a system is dependent on the use of dynamic analysis, as the construction of the model requires quantitative information on the temporal behavior of the system, such as task inter-arrival times, task execution times and task response times. This information may be obtained through dynamic analysis, i.e. by recording the relevant information from the system at runtime, and analyzing the recorded data offline. The result of the recording is an *execution trace* which is a list of time-stamped events that describe the system behavior during a period of time. Typical events that are registered are task-switches, inter-process communication (IPC), and changes of important state variables.

Execution traces are also useful for purposes other than modeling. By visualizing the content of an execution trace, the system behavior becomes directly observable and tangible, which facilitates debugging and overall system understanding. Recorded execution traces may also be used to compare the temporal behavior of the latest release of the system with previous releases, which allows system developers to study how recent maintenance may have effected the system with respect to its temporal behavior, e.g. response times and the use of limited logical resources. This use of dynamic analysis is referred to as

regression analysis, analogous to regression testing. Regression analysis can identify recent negative effects on specific properties of the temporal behavior, which may be due to e.g. sub-optimal implementation of new features. Moreover, regression analysis can also be used to identify trends as well as potential problems, such as when the response time of a task is approaching a deadline, or when the utilization of a logical resource is approaching the maximum allowed utilization. By identifying such problems at an early stage, developers may take preventive actions in order to minimize the risk of serious problems in future versions of the system.

The remainder of this chapter is divided into four sections as follows: Section 3.1 presents three ways in which dynamic analysis can be used in this context, system understanding, modeling and regression analysis. Section 3.2 discusses the recording complex embedded system's of temporal behavior, the problems associated with recording, what properties that is of interest as well as the effects on system performance associated with recording of these properties. Moreover, the section presents the design and performance of an implemented behavior recorder that has been integrated in a complex embedded system. Section 3.3 discusses analysis and comparison of recorded execution traces and presents the implementation of the probabilistic property language proposed in earlier work [WAN03b]. Moreover, a set of tools developed for analysis and visualization of recorded execution traces is presented. Finally, Section 3.4 concludes this chapter with a discussion on how these contributions relate to the thesis research questions stated in Section 1.2.

3.1 Uses of Dynamic Analysis

Dynamic analysis is an established general technique which may be used to study many different aspects of a system by performing recordings during runtime, e.g. dynamic memory allocation, function calls, interrupts, cache behavior, etc. In this thesis, dynamic analysis is used to study the temporal behavior of tasks, e.g. task execution times, task response times and task inter-arrival times, and properties dependent on the temporal behavior, such as the usage of limited logical resources. The rest of this section presents three different uses of dynamic analysis in the context of complex embedded system's temporal behavior.

3.1.1 System Understanding

One factor contributing to the complexity of software development and maintenance for complex embedded system, is that the temporal behavior of the system is not tangible. The temporal behavior, i.e. the global ordering and timing of events in the system, can not be understood by only studying the implementation, as the temporal behavior is dependent on the execution time of the code in the system's tasks and of the stimuli from the system's environment. The execution times depends on the hardware used, and may vary from time to time due to the many factors involved, e.g. data dependencies (parameter, state) and complex, seemingly non-deterministic, hardware such as cache memories. By presenting a recording of the system graphically, the temporal behavior is visualized over a time-line, which allows for better understanding for software developers of the complex behavior of their system. This view of the system behavior is highly useful while debugging, designing new features or changing the software architecture in other ways. It is also highly useful for educational purposes when new developers are introduced to the system.

3.1.2 Modeling System Behavior

The approach for behavior impact analysis discussed in Section 1.1 depends on a model containing quantitative information regarding the system's temporal behavior, i.e. execution times and inter-arrival times of tasks, and probabilities of events that are modeled in a probabilistic manner. To obtain this information from sources other than dynamic analysis is difficult. Even though execution time analysis of the tasks is possible, the required tools are in general not mature. Furthermore, such tools typically focus on identifying the worst case execution time, while the model necessary for behavioral impact analysis requires the description of the typical distributions of e.g. execution times for a task. The behavior impact analysis is described in depth in Section 4.1.

3.1.3 Regression Analysis

Dynamic analysis can be integrated in the software development process in order to monitor the effects of the software evolution on important properties of the system's temporal behavior, such as response times. On a regular basis, an analysis is performed on the latest version of the system and the results are compared with previous results, i.e. analysis results from previous versions of the system. This allows system developers to study how recent changes have

effected the system with respect to a set of important properties of the system's temporal behavior, such as response times and resource usage. We refer to this use of dynamic analysis as regression analysis.

Regression analysis can point out sub-optimal implementations that are reflected in the system properties under observation. If such impacts can be captured automatically, future problems related to the temporal behavior can be avoided. Moreover, regression analysis can identify potentially dangerous trends in system properties, e.g. if the utilization of a newly introduced logical resource gradually increases as more and more components of the system begin using it. Such trends may cause problems in future releases, but if discovered, the appropriate actions may be taken early, before errors occur, e.g. by increasing the amount of resources available (if possible) or by decreasing the use of the resource.

In order to introduce regression analysis for a complex embedded system, there is an initial effort of specifying the properties of interest and implementing the necessary code instrumentation. Moreover, the system setup and test cases executed when measuring the system should be specified in a document, in order to allow system measurements to be reliably reproduced. After this initial work, performing a regression analysis is straightforward and can be performed as one out of many test-cases by a system tester without requiring deeper system understanding or programming knowledge. Measurements are made according to the documented test cases. This results in execution traces, which are analyzed and compared with earlier releases using a highly automated analysis tool. Based on the comparison rules, the tool decides if there are alarming differences and informs the user of the outcome.

We are working in cooperation with system architects and developers at ABB Robotics with the introduction of regression analysis in their development process. The recorder has been integrated into their robot control system and is activated by default which allows engineers at the company to perform the recordings necessary for regression analysis with a minimal effort. Currently, a small group of experienced engineers are gradually introducing regression analysis in one of the subsystems.

3.2 Recording – What, How and Costs

When using dynamic analysis for recording the temporal behavior of tasks in a complex embedded system, it is important to note that each task in a complex embedded system typically consists of a collection of *services*. These services

may exhibit very different temporal behaviors. Below is an example of a C implementation of a task with several services. The main routine of the task contains an infinite loop that receives messages from other tasks and depending on the content of the message, different services are then executed. Tasks may also have a behavior which is executed when the task has been waiting for messages for a certain amount of time. The timeout behavior can be considered a time-triggered service.

```
void taskX()
{
    int status;
    int timeout;
    MSG msg;

    init();

    while( forever )
    {
        status = ipc_receive(&msg, timeout);

        if (status == TIMEOUT)
        {
            timeout_behavior();
        }
        else
        {
            switch( msg.command )
            {
                case COMMAND1:  service1(...);
                                break;
                case COMMAND2:  service2(...);
                                break;
                case COMMAND3:  service3(...);
                                break;
                ...
            }

            if (msg.answer_requested == TRUE)
            {
                ipc_send(msg.sender, answer);
            }
        }
    }
}
```

When recording the temporal behavior of a system where the tasks consist of multiple services, it is not sufficient to only record which tasks that are executed. In order to allow analysis of the temporal behavior of individual services, it is necessary to record the service that is executed in each task execution.

It is also important to note that temporal requirements of complex embedded systems are not always expressed as limits on task response time, they

may also be expressed as limits on *end-to-end response times*. An end-to-end response time is a more general term which represents the time between two defined events in the system, a *start-event* and a *stop-event*. Typically, an end-to-end response time represents the sequential execution of a set of related tasks.

Another interesting property is the utilization of limited logical resources, e.g. a fixed size FIFO queue. The utilization of such logical resources varies over time as a result of the execution of certain tasks and is therefore dependent on the tasks temporal behavior, e.g. inter-arrival times. A common property of interest is if the utilization of a specific logical resource is above or below a certain limit, i.e. starvation. Based on these properties of complex embedded systems, the properties of interest can be grouped into three categories:

- *Execution and Response times*: This category includes properties related to the execution time and response time of tasks and specific services, as well as properties related to end-to-end response time. This category also includes properties describing how execution and response times are effected by important state variables.

- *Resource usage*: Properties in this category describe the system's usage of specific limited resources, logical or physical (e.g. CPU time), and how the utilization varies over time. Typical properties are maximum, minimum, and average utilization, but the category also includes properties describing the relationship between resource usage and the execution of tasks or specific services, e.g. the resource allocation of specific services. Resource usage also includes the CPU utilization of individual tasks as well as the complete system. This information is often used as a rough measure of the temporal behavior of a system.

- *Inter-arrival times and Patterns*: This category includes inter- arrival time distributions of specific events, such as activation of specific tasks or services, or other important events such as state changes. This category also includes properties related to patterns in the activation of tasks, services, or occurrences of other events.

When recording for the purpose of modeling a system's temporal behavior, properties from all three categories are relevant, but for different purposes. Execution times and inter-arrival times are necessary as parameters of the model, while response times, resource usage and patterns are predicted by an analysis of the model.

However, data regarding the properties predicted by the model need to be recorded from the real system as well, in order to allow validation of the constructed model. By comparing the predictions from the model with the observations of the real system, it is possible to determine if the model is valid. Model validation is discussed in depth in Chapter 5.

This may cause confusion, why bother to construct a model and perform an analysis when the properties can be measured directly from the existing system? The answer is in the purpose of the model, i.e. allowing for prototyping of new features, as discussed in Chapter 1. When using the model for prototyping, there is no implementation that allows measurements to be made. However, by updating the model with an abstract prototype of the planned new feature, it is possible to make predictions on the effects on the system's temporal behavior caused by an implementation of this feature. This is further discussed in Chapter 4.

3.2.1 The Probe Effect

A problem with dynamic analysis of multi-tasking systems is that the code instrumentation necessary for recording, the *probes*, may alter the temporal behavior of the system, if the order of important events is changed. This is commonly known as the *probe effect* [Sch91, MH89]. The probing proposed in this chapter would increase both the task-switch overhead as well as the execution time of the tasks. Another impact is that by increasing the time it takes to perform a task-switch, the interrupt latency is increased as well, as interrupts are disabled during this time.

If an analysis is performed based on the runtime behavior of a system that has been instrumented with software probes, the analysis will only be valid as long as the probes remain in the system. If the probes are removed before releasing the system, the released system will be different from the system that was analyzed. The differences are in most cases small and often negligible, but there is always a risk of a probe effect. Even the smallest change in timing, caused by e.g. by software probes, can potentially change the order of important events, which may lead to erroneous system behaviors.

A common way of avoiding the probe effect is to let the probes remain in the released version of the system. This way, the released system will be identical to the system that has been studied in the analysis. The types of systems considered in this thesis usually have sufficient resources (CPU-time, memory) to allow for a reasonable amount of probing, which makes this solution attractive.

3.2.2 Relevant code instrumentation

In order to record information regarding the properties described in Section 3.2, software probes of different types are necessary. In this section these probes will be described, including their purpose and their impact on CPU and memory.

Monitoring task execution By adding a single probe to the system that is executed by the task-switch, a major part of the required information can be derived, e.g. what tasks have been executed and when, all preemptions points, the execution times and response times of the tasks. Thus, the value of adding this single probe is significant, but the cost of adding the probe, i.e. CPU-time and memory usage, might also be significant depending on the average rate of the task-switch event. This probe is referred to as the *task-switch probe*.

In the system that has been studied in this work, the robot control system from ABB Robotics, task-switches occur rather frequently, with an average rate of about 3 KHz. This is a rather high rate in comparison to other events of interest in the system. The task-switch probe is therefore expensive, but also very valuable.

Monitoring activation of services For tasks that are of interest for more detailed analysis, it is desired to know not only record the task execution, i.e. what tasks that are executed and when, but also what service, i.e. specific behavior, that is executed at each task instance. To record this information it is necessary to add a probe to the dispatcher of the task, i.e. where the task receives incoming messages and selects what service to execute. This probe is referred to as the *dispatcher probe*. Each time the task is activated, typically by an incoming message, the dispatcher probe stores a code that identifies the service. Since not all tasks need to have this detailed instrumentation, adding dispatcher probes to tasks is rather inexpensive compared to the task-switch probe. Even if all tasks in the system are equipped with a dispatcher probe, the cost of this instrumentation is less than the task-switch probe. Task activations occur less frequently than task-switches, as task switches occur not only on task activations, but also when a task terminates and a pre-empted task is resumed.

Monitoring end-to-end response times In order to study an end-to-end response time, i.e. the time between two arbitrary events (typically a request and a corresponding response/reaction), it is necessary to insert two probes, a *start-probe* and a *stop-probe*. The difference in time between the execution of

a start-probe and the next stop-probe (with the same ID) corresponds to the end-to-end response time.

Monitoring inter-process communication End-to-end response time monitoring can be used to study latencies in the inter-process communication, i.e. the time from "send" to "receive" of a particular message. This information is interesting for the identification of performance bottlenecks and for model validation purposes. If there are interface routines which encapsulate the sending of messages to a specific task, this instrumentation can easily be performed by inserting interface probes, acting as start-probes for an end-to-end response time measurement. If a dispatcher probe exists in the receiving task, it can be used as the corresponding stop-probe.

When a task receives an IPC message, the sender of the message can be identified as the task that executes at the execution of the send-probe. It is however possible to register the sender of each message without using send-probes, if the dispatcher-probe is extended to also register which task that sent the received IPC message. Naturally, this requires that the IPC messages contain this information. The purpose of recording the senders of IPC messages is to document the dependencies between different tasks in the system, i.e. which tasks use the different services of each task. The dependencies between tasks is valuable information for system understanding, maintenance and modeling.

An IPC message sent to a task under observation results in one or two probes being executed, depending on if the IPC latency is measured (two probes) or not (one probe). The impact of these probes is dependent on the intensity of the IPC, which varies drastically between different tasks. Some tasks may have a low average rate of incoming messages, but if the messages arrive in short intensive bursts instead of evenly distributed over time, the dispatcher probe is executed frequently during the duration of the burst. Even though the probing may have little impact on total CPU utilization, the quick bursts may cause the probing to have a significant impact locally. Thus, in order to estimate if the impact of the dispatcher probe, not only average message frequency is interesting but it is also interesting to study the minimum inter-arrival time between messages and how often several "short" inter-arrival times occur in sequence.

Monitoring all IPC traffic in the system is rather expensive if the IPC latency is to be measured, as an additional probe is necessary apart from the dispatcher probe, for each IPC message sent. It is however not necessary to record all IPC traffic in the system. It is often the case that only a single, or a few, tasks are in focus and the IPC monitoring can be limited to those, which significantly reduces the cost of this probe.

Monitoring usage of logical resources In order to record the usage of a specific logical resource, such as a data buffer, it is required to add probes to the interface routines of the resource. Interface routines are e.g. the *put* and *get* routines of a data buffer. Probes in interface routines are referred to as *interface probes.* By encapsulating probes in the interface routines, the probing is invisible in the application code.

Monitoring semaphores Semaphores are logical resources and may be instrumented using interface probes. However, if the semaphores do not have interface routines specific for each semaphore, this would require instrumenting the Operating System (OS) routines, which are typically used for semaphore operations by all tasks in the system. This would result in all semaphore operations being recorded, thus consuming a lot of resources (CPU time and memory) as semaphore operations are common in multitasking systems. In most cases only a single or a few semaphores are of interest. The semaphore interface routines may, however, be probed if a filter is implemented, that only executes the probe if the semaphore operation is of interest, i.e. involves a specific task or semaphore. Such a filter is implemented in the OS code corresponding to the semaphore interface routines or preferably in an OS isolation layer, when existing. The filter would in principle check the current semaphore operation (task ID and semaphore ID) against a list of semaphore operations of interest. It is important to implement such a filter in an efficient and deterministic way, in order to minimize the execution time overhead and jitter.

Monitoring state variables It may also be interesting to monitor specific state variables, as these may effect the temporal behavior of the system and thus be of interest for modeling. If the state variables are accessed through interface routines such probing can be performed easily using interface probes. Probes monitoring state variables will not be executed as frequently as e.g. the task-switch probe and would therefore be relatively inexpensive.

If the state variable is an ordinary global variable, accessed directly from many locations in the code, it is not convenient to add probes to each location that change the variable. Instead, the state variable can be sampled using *state sampling probes*. The state variables of interest are sampled in the immediate beginning of the execution of a task/service that is dependent on the state variable. If there are a large number of state variables that need to be sampled, this may require a lot of resources, as the sampling may be rather frequent.

3.2.3 Resource Consumption

When planning the introduction of code instrumentation in a complex embedded system, it is important to estimate the resource consumption (CPU time and memory) required by the different types of probes. This is dependent on the properties of the system, for instance the average task-switch rate. Some types of probes may be too expensive for a particular system, while others may be inexpensive but they still contribute a significant value for an analysis. The CPU utilization caused by a specific probe, p, is given by

$$U_P = f_P * C_{PROBE}$$

where f_P is the expected rate (in Hz) of the probe p and C_{PROBE} the probe execution time, in seconds. The amount of recording bandwidth required, i.e. the amount of data produced by the probe every second, is given by

$$B_P = f_P * N_{PROBE}$$

where N_{PROBE} is the probe size, the amount of memory required to store the data from a single probe execution. Thus, there are three parameters that need to be determined in order to estimate the impact on CPU utilization and memory usage: f_P, C_{PROBE} and N_{PROBE}. To provide reference values for comparison of the different probe types, $C_{PROBE}$is assumed to be 5 μs. This is realistic for modern systems and supported by the probe execution time measurements presented later, in Section 3.2.4. A reference value for N_{PROBE} is found in our implementation of the software behavior recorder, also presented later, in Section 3.2.4. Each probe execution requires 4 bytes of memory. The expected rate is dependent on the type of probe and the general characteristics of the system. To provide values for comparison, typical rates of the different types of probes have been estimated based on our experience and studies of a representative system in earlier work [WAN+03a, AN02]. Table 3.1 presents typical rates of the different probe types and the resulting CPU utilization and memory usage.

Probe type	**f$_P$(in Hz)**	**U$_P$**	**B$_P$ (in B/s)**
Task-switch	3000	0,015	12000
Service activation	1500	0,0075	6000
Usage of logical resources	50	0,00025	200
End-to-end response times	100	0,0005	400
State variables	50	0,00025	200
Semaphore usage	400	0,002	1600
IPC Send	360	0,0018	1440
Total	**5460**	**0,0273**	**21840**

Table 3.1: Typical rates of different types of probes and resulting resource usage

The rates used as reference in Table 3.1 are motivated as follows:

- Task-switch rate - One event per task-switch, average task-switch rate was observed on reference system
- Service activation - One event per task activation, average task activation rate was observed on reference system (for all tasks)
- Usage of logical resources - 10 logical resources instrumented, each with an average change rate of 5 Hz
- End-to-end response times - 10 response time monitored, each with an average rate of 5 Hz, two probes are executed each time (10*5*2 Hz)
- State variables - 10 state variables monitored, each with an average change rate of 5 Hz
- Semaphore usage - 10 semaphores monitored, each semaphore is used by two different tasks. Each task uses the semaphore (lock and release) with an average rate of 10 Hz. (10*2*2*10 Hz)
- IPC Send - 10 tasks monitored, one event per task activation, observed an average task activation rate of 36 Hz on reference system

Some of the probing described in this section should be permanent, present in the system at all times, e.g. the task-switch probe and the dispatcher probe, as they generate a large amount of important information with a low impact

on the source code as well as an impact on CPU and memory that in absolute terms is rather small. Probes on logical resources and state variables could also be of a permanent nature, since they are executed sparsely but maintain significant value.

Probes that generate information that in the typical case is of less interest could be disabled by default and activated when the system is to be studied in detail, e.g. for modeling or fault localization purposes. Examples of such probes are semaphore probes, state sampling probes and "IPC send" probes. As discussed earlier, to activate probes on demand implies a risk of causing a probe effect, but this risk can be minimized if the "inactive" probes are allowed to execute and thus consume CPU time as usual, but not allowed to store the data. This is accomplished by writing the data from all inactive probes to a single memory location. This way, the probes that are of little interest are inactive in the sense that they do not use any memory, which allows for longer execution traces to be recorded.

3.2.4 Implementation and Evaluation of a Behavior Recorder

In order to be able to record the behavior of a running system, the system must have the appropriate recording functionality. We have developed a software behavior recorder suitable for complex embedded systems with a single CPU and the real-time operating system VxWorks, from WindRiver [WRW]. The recorder is manually integrated in the system to be analyzed by adding the recorder module to the system's base platform. The recorder uses a feature in VxWorks to associate a specific routine with the task-switch (context-switch), which executes the associated routine each time the operating system performs a task-switch. In our case, the task-switch routine contains a probe that registers each task-switch event in the system. Each execution of the task-switch probe stores a timestamp together with an ID of the next task execute and the scheduling status code of the previously executing task. The scheduling status code contains the reason behind the task-switch, e.g. preemption by a task with higher priority ("READY"), blocking by a semaphore or a message receive ("PEND") or a waiting for a specified time ("DELAY"). From the collected task-switch information, it is possible to generate an execution trace, which accurately describes how the tasks have executed over time. It is also possible to extract execution times, response times, preemptions and inter-arrival times. Moreover, the recorder also supports *generic probes*, i.e. explicit probes inserted in the application code. Such probes may be used for any type of probing proposed in Section 3.2.2, e.g. as dispatcher probe, start/stop probes,

interface probe, state sampling probes etc.

The data from the task-switch probe and generic probes are stored in a ring-buffer, i.e. the oldest data is overwritten when the buffer is full. Since the amount of memory available for recording is limited, the alternative to a ring-buffer is to stop recording when the buffer is full, which is not desirable. Writing directly to a permanent storage device, such as a hard-drive, is not an option due to the associated increase in probe execution time.

A ring-buffer always contains the most recent data. This is necessary as it might be required to have the recorder active for long periods before interesting behavior occurs. When the behavior of interest has been observed, the recorder can be stopped and the content of the buffer is written to a file. The file can then be transferred to a PC for analysis.

In the implementation of the software behavior recorder, the size of the ring-buffer is by default set to 100.000 events. As each event requires 4 bytes of memory for storage, this corresponds to a memory allocation of 400 KB. The motivation for this large buffer size is the nature of the system that has been studied, a robot control system [Wal03, WAN+03a, MWN+04]. Industrial robots are typically used to perform repetitive tasks, where the robot arm follows a cyclic path. As the temporal behavior of the control system depends on the position of the robot arm, it is desired to record at least one cycle. Assuming a probe execution rate of 3000 Hz, i.e. only task switches are recorded, the chosen buffer size allows recordings exceeding 30 seconds, which is sufficient for recording whole cycles in most test cases. Assuming a probe execution rate of 5500 Hz as presented in Section 3.2.3, this buffer size allows only for 18 seconds. However, it is not a problem to use buffer sizes larger than 100.000 events; we have successfully performed recordings using buffer sizes up to 300.000 events (using 1200 KB of memory). This buffer size allows for recordings exceeding 90 seconds at a probe execution rate of 3000 Hz (task switches only). At the probe execution rate of 5500 Hz proposed in Section 3.2.3, this buffer size allows for recording 54 seconds of execution. The only limitation on the buffer size is the amount of memory available.

The file that is the result of the recording uses a publicly available file-format, called TRC, (named using the consonants from the word “trace”). The TRC format was developed together with the recorder and is very simple. A TRC file contains an execution trace, i.e. a list of time-stamped events generated by the code instrumentation. The time-stamps are read from a clock with microsecond resolution. A detailed description of TRC file format can be found in the tool documentation.

The effect on the system’s temporal behavior caused by a specific software

probe is dependent on two factors, the rate of which the probes are executed in the system and the time it takes to execute a probe. The execution rate of a probe depends on the context in which it is used, as discussed in Section 3.2.2 and Section 3.2.3. We have made measurements on a representative system in order to estimate the execution time of a probe. The system consisted of a 200 MHz Intel Pentium system running the real-time operating system VxWorks, from WindRiver [WRW]. The measurement of the probe execution time was accomplished by configurating the system to execute only a single task containing two immediately adjacent probes, in a loop. The difference in time-stamps between two adjacent probes provides an estimate on the probe execution time, as depicted by Figure 3.1.

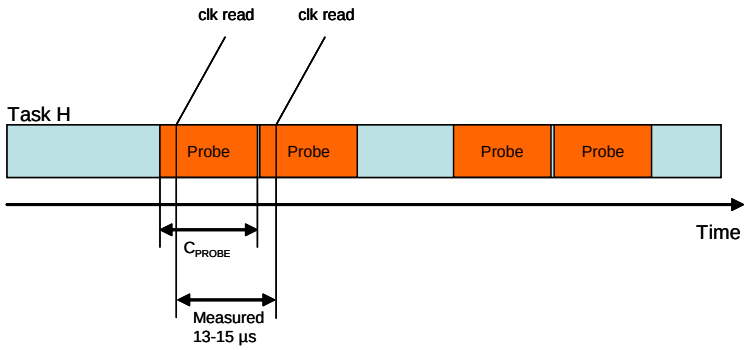

Figure 3.1: The measurement of probe execution time (C_{PROBE})

The results from the performed probe execution time measurements are presented in Figure 3.1. The measured probe execution times are unfortunately truncated since the resolution of the clock used, 1 μs, is a considerable fraction of the measured times. The measurements indicate that a probe takes on average 14 μs. All but one of the observed probes took between 13 μs to 15 μs to execute, but there is a single peak on 23 μs. We were unable to find any obvious cause of this peak, but a plausible explanation is effects from hardware, such as cache memory misses. We also measured the execution time of the task-switch, in order to estimate the relative increase in task-switch overhead caused by a task-switch probe. To measure this execution time the system was configured to execute two tasks, one with high priority and one with low priority. The high priority task executed an infinite loop consisting of two operations; a generic probe followed by a delay-operation caused a task-switch to the lower priority

task. The lower priority task contained a second generic probe that executed immediately when the task was activated, i.e. after the task-switch, as depicted by Figure 3.2.

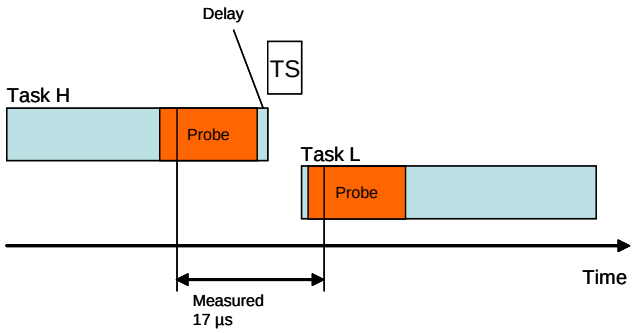

Figure 3.2: The measurement of task-switch execution time

The observed difference in time between the two generic probes was approximately 17 μs, but this also includes execution time from parts of the two probes used for the measurement, corresponding to the execution time of a single probe. Thus, if compensating for the 14 μs probe execution time, the separation in time of the two probes, i.e. the task-switch execution time, is only around 3 μs in this system.

The impact of a task-switch probe is thus significant with respect to the task switch overhead, an increase with over 500 %. As discussed previously, the interrupt latency is effected by this impact as well. The execution of interrupt service routines may be delayed by approximately 17 μs if the interrupt occurs at the immediate start of a task-switch, in comparison to 3 μs without the task-switch probe.

It is however important to note that the hardware used for these measurements was quite old, a 200 MHz Intel Pentium system. This CPU was released in the mid 1990's, i.e. over 10 years ago, and has performance equivalent to a modern mid-range PDA. If measurements were to be made using a modern system, the probe execution times would be significantly lower as complex embedded systems today often contain CPUs that are many times faster than the one used in these measurements. It is also important to note that in the system that has been studied many tasks have execution times measured in milliseconds and large execution time variations. For systems of this kind, adding a

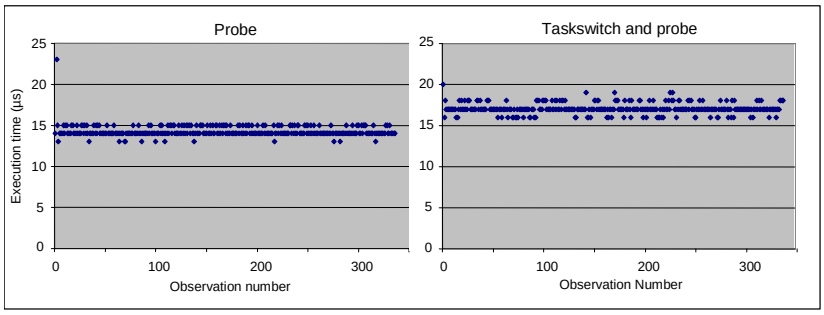

Figure 3.3: Results from measurement of probe and task-switch execution time.

few microseconds to the execution time of the tasks will, in most cases, have a negligible impact. Changes of similar magnitude are made frequently during the system maintenance (bug fixes etc.), in most cases without introducing any problems related to the temporal behavior.

3.3 Analysis and Comparison of Execution Traces

In this section we will discuss the analysis and comparison of data from recorded execution traces and present tools supporting analysis, comparison and visualization. Further, we will present an implementation of the *Probabilistic Property Language*, PPL, initially proposed in [WAN03b].

Together with the information available, the analysis method decides which properties that can be analyzed and also effects the confidence assessment of the result. Dynamic analysis techniques are based on a recording of the system behavior during a limited period of time. This gives a realistic picture of the system behavior, but the analysis result is not necessarily *safe*, i.e. it is not certain that the “worst case” has been captured in the recording, as the recording is merely a sample from a large set of possible scenarios and corresponding system behaviors.

A problem with dynamic analysis is how to determine the confidence of the results. For instance, we want to compare the average response time of a particular task in two different versions of the system. If there are large variations in response times between executions of the task, how do we know if an observed difference is actually an effect of the difference between the systems and not

due to random fluctuation? This has been addressed in literature, e.g. in the book by Law & Kelton [LK93]. They present two approaches for comparing real-world observations with simulation results, the *inspection approach* and the *confidence interval approach*. These methods can also be used to compare two real-world observations of potentially different systems or two different simulations of models. These methods are necessary for behavior impact analysis (discussed further in Chapter 4), regression analysis (presented in Section 3.1.3) and model validation (discussed in Chapter 5).

The inspection approach is probably the most common method used among simulation practitioners. Basically, this method computes statistical measures, e.g. average values, from the two observations and compares these values. This is depicted in Figure 3.4.

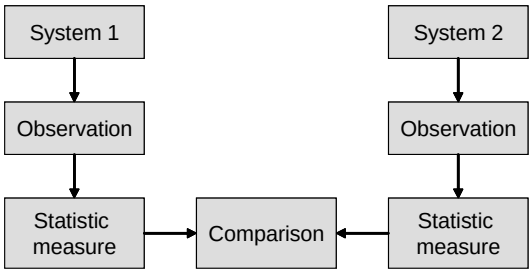

Figure 3.4: Inspection approach

According to [LK93] a problem with this method is that the statistics that are compared only represent a single sample of an underlying population. As the statistical values are computed based on stochastic variables, e.g. response times, also the statistical measures will be stochastic variables. Thus, the calculated statistics may have varying values from time to time. Applied to our approach this would mean that, e.g. the average execution time calculated from a single execution trace could vary from time to time. This is probably true, even though the variation may be very small if sufficiently long execution traces are recorded and the system has the same configuration in both observations.

The confidence interval approach, depicted in Figure 3.5, requires large amounts of data to be collected for analysis. Instead of comparing two values calculated from two sets of data, many sets of data are collected for each of the two systems to be compared. From each set of data from the two systems, the statistic measure to compare is computed, e.g. the average value, resulting in two sets of data points where each data point represents the behavior of the

system during a particular observation. A confidence interval is constructed that describe the deviation between the two sets of data, e.g. “a 99 % confidence interval of the deviation is 2.42 ± 1.15”. The 99 % confidence expresses the ratio of independently constructed confidence intervals that will cover the expected value, i.e. the mean difference between the two data sets.

If the confidence interval does not cover zero (0), the difference between the systems compared is statistically significant, as the expected deviation is more significant than the variation. This indicates “real” differences between the two systems. In the other case, when the confidence interval covers zero (0), the observed differences may be due to random fluctuations.

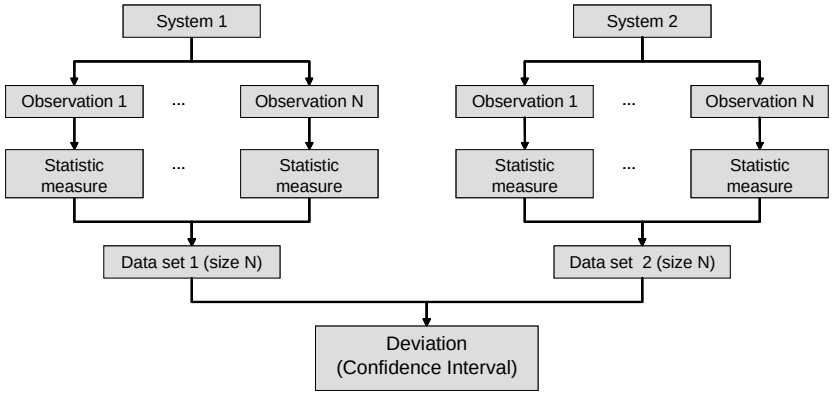

Figure 3.5: Confidence interval approach

The confidence interval approach is preferable, as it allows for several observations of the systems used which results in a higher confidence in the comparison. However, to make a large set of observations of a real system is typically a substantial effort.

3.3.1 The Probabilistic Property Language

The purpose of the Probabilistic Property Language, PPL, is to allow formulation of queries on properties related to the temporal behavior a system, such as response times and usage of logical resources. PPL can describe all properties discussed in Section 3.2.2 and more. PPL allows formulation of probabilistic properties, e.g. soft deadlines such as “at least 99 % of task X should be completed within 1000 time units”.

The temporal logics discussed in Section 2.4 can not express probabilities or quantitative time, but there are other temporal logics which can, e.g. TPCTL [HJ94]. Compared to such temporal logics, PPL is used for similar purposes, i.e. specifying properties to check, but PPL is more of a database query language than a mathematical logic, where the "database" that is queried by PPL is an execution trace. It might be possible to use a temporal logic rather than PPL. However, PPL is specially designed for expressing probabilistic properties of the temporal behavior of tasks, which makes PPL queries more intuitive for software developers without previous experience of formal methods.

In this section an implemented version of PPL is presented using a set of examples. This version differs a bit from the original definition of PPL found in [WAN03b] and [Wal03]. A detailed specification of the implemented version of PPL and the differences compared to the original version can be found in Appendix B.

The PPL query A typical use of PPL is to check a deadline property of a task. Example 1 presents a PPL query that checks if all instances i of task A meets a deadline of 1000 time units with a probability of 1. A task instance is a particular execution of the task. A task instance is represented in PPL using it's task identity, start time, finishing time and execution time.

Example 1:

```
P(A(i), A(i).response < 1000) = 1
```

The first parameter to the P operator is a quantifier specifying that the condition in the second argument should be checked for all instances i of the task A. This is different from the original version of PPL [Wal03, WAN03b], where the P-function did not accept any quantifier argument. It was discovered during the implementation of the PPL analysis tool that the original definition of PPL had ambiguous semantics when multiple tasks are referred in a query. The quantifier parameter is necessary in those cases to solve the ambiguity. The second parameter is the condition to check. In the example the condition specifies that the response time of the task A should be below 1000. The P operator returns the ratio of the instances in the execution trace for which the condition holds. If the P operator has a return value of 1, it means that the condition holds for all observed instances of the task, i.e. a probability of 1.

PPL allows checking probabilistic properties such as a soft deadline. For instance, a soft deadline requirement could be that at least 90% of the task instances should meet the deadline. An example of a soft deadline is presented in Example 2.

Example 2:

```
P(A(i), A(i).response < 1000) > 0.9
```

PPL has a data model allowing a query to refer to task instances in the execution trace using a combination of task name, instance index and property name, on the form "*task(i).property*". The data model provides five properties for each task instances in the execution trace, which can be used in PPL queries. These are specified in Table 3.2.

Property	**Description**
start	The time when the instance started
end	The time when the instance finished
exec	The execution time of the instance
response	The response time of the instance
probeN	The value of probe N when the instance started

Table 3.2: The data model of PPL

The "probeN" property of a task instance corresponds to the value of the generic probe "probeN" at the time the task instance is started. A generic probe may monitor any quantifiable property, but typically generic probes are used to monitor logical resources of different kinds, such as the current utilization of a buffer. Further, PPL contains a set of operators and function that allow conditions to be formulated on the data model. These operators are described in Table 3.3.

Relational operators	=, <, <=, >=, >	value op value -> bool
Logical connectives	and, or, not	bool op bool -> bool
Arithmetic operators	+, -, *, /, abs	value op value -> value
Statistical functions	max, min, avg, median	op(list) -> value
Index operator	X(i)	op(list, index) -> instance
Following operator	X(following(Y(i)))	op(list, list, index)-> instance

Table 3.3: The operators of PPL

PPL queries using the instance operator The index operator is used to differentiate instances of the same task. One property that can be checked using

the index operator is temporal separation, i.e. a property that specifies the minimum distance in time between two consecutive instances of a task. This is demonstrated by Example 3.

Example 3:

```
P(A(i), A(i+1).start - A(i).end >= 1000) = 1}
```

Another use of the instance operator is demonstrated in Example 4, which specifies that two consecutive instances must not violate the deadline of 1000 time units.

Example 4:

```
P(A(i), A(i).response > 1000 and
        A(i+1).response > 1000) = 0
```

Expressing a requirement that e.g. 5 consecutive instances must not miss their deadline would result in a very large expression if the presented from the previous example is used. To simplify such queries it is possible to specify intervals rather then single integers in the index operator. Example 5 specifies that there must never be 5 consecutive task instances that violate the deadline of 1000 time units.

Example 5:

```
P(A(i), A(i +[1..4]).response > 1000) = 0
```

Queries using functions and unbounded variables In order to relate adjacent instances of different tasks, the *following* function can be used. Example 6 shows a query checking if there are any situations where an instance of A and the following instance of B have execution times above 1100 time units and 1700 time units respectively.

Example 6:

```
P(A(i), A(i).exec > 1100 and
        B(following(A(i))).exec > 1700) > 0
```

Moreover, PPL queries may contain an unbounded variable. For instance, by specifying the probability as an unbounded variable, the result of the query is the minimum/maximum value for which the condition holds. A query using an unbounded variable to evaluate the probability of meeting a deadline of 2000 time units is presented in Example 7.

Example 7:

```
P(A(i), A(i).response < 2000) = X
```

It is also possible to use unbounded variables inside the second parameter of the P-operator. A query evaluating the shortest deadline D that is met with a probability of at least 0.9 is presented in Example 8.

Example 8:

```
P(A(i), A(i).response < D) >= 0.9
```

Statistical functions As presented in Table 3.3, there is also a set of statistical functions that may be used to extract simple statistical measures of the different tasks. The statistical functions can be used as stand-alone queries as in Example 9.

Example 9:

```
avg(A.response)

median(A.exec)

max(A.exec)
```

The statistical functions can also be used instead of constant values inside the second parameter of the P-operator, as in Example 10.

Example 10:

```
P(A(i), A(i).resp > avg(A.resp)*2 ) = X
```

The above described PPL query returns the probability (X) of task A having a response time above a certain limit, which is specified as "two times the average response time".

Queries on logical resources PPL also allows queries on data from generic probes. A generic probe may monitor any quantifiable property of the system, but are typically used to monitor logical resources, such as the usage of a data buffer. In the current implementation, the generic probes are identified using a number. If the number of messages in a certain message queue is monitored using generic probe number 21, it is possible to formulate a PPL query checking that the message queue is never empty when taskX is activated as presented in Example 11.

Example 11:

```
P(taskX(i), taskX(i).probe21 > 0) = 1
```

It is also possible to specify conditions on a probe that are independent of what tasks that are to be executed, by replacing the name of the task with a wildcard character. This is demonstrated by Example 12. For such queries the probabilities are calculated differently, by summing the lengths of the time intervals where the condition holds and divide that with the length of the recording. The resulting value is thus the fraction of the total time in the recording where the condition holds. This is an approximation of the probability that the condition holds at an arbitrary point in time.

Example 12:

```
P(*, *.probe21 > 0) = 1
```

Tool Support The PPL language is supported by two tools available as a part of the ART Framework: the *Property Evaluation Tool* and the *Tracealyzer*. The Property Evaluation Tool is a dedicated front-end application for PPL which analyzes batches of PPL queries on two different execution traces and presents the results side-by-side. The tool is presented in Section 3.3.2. The Tracealyzer tool contains among other features a PPL terminal, where it is possible to formulate and run single PPL queries with respect to an execution trace. This is the preferred tool for experimenting with PPL. The Tracealyzer is presented in Section 3.3.3.

3.3.2 The Property Evaluation Tool

The Property Evaluation Tool (PET) is a tool for analyzing and comparing execution traces with respect to different system properties formulated in PPL. The application has uses in all three processes described in this thesis:

- Regression analysis, presented in Section 3.1.3.
- Behavior impact analysis, presented in Chapter 4.
- Model validation, discussed in Chapter 5.

The user interface of the tool is depicted by Figure 3.6. Performing an analysis is generally swift but depends on the size of the execution traces and the amount and calculation complexity of the PPL queries. The queries in

Figure 3.6 take less than 3 seconds on a modern computer (Intel Pentium 4, 2.4 GHz). The two execution traces in the example are medium-sized (containing 10.000 and 60.000 events respectively). However, some exotic PPL queries where more than one task is involved may take considerable time to compute, minutes rather than seconds for larger execution traces, i.e. 100.000 events.

The tool reads one or two execution traces as well as a *comparison file* containing a set of PPL queries. If two execution traces have been specified, the results are presented side-by-side in a table, consisting of two columns corresponding to the execution traces and one row for each PPL query. In the user interface depicted in Figure 3.6, the actual PPL queries are however not visible. For better tool understandability, instead of presenting a cryptic PPL expression, the tool presents a descriptive name of the property that is queried. The PPL queries can however be inspected and edited in this tool by clicking on the property.

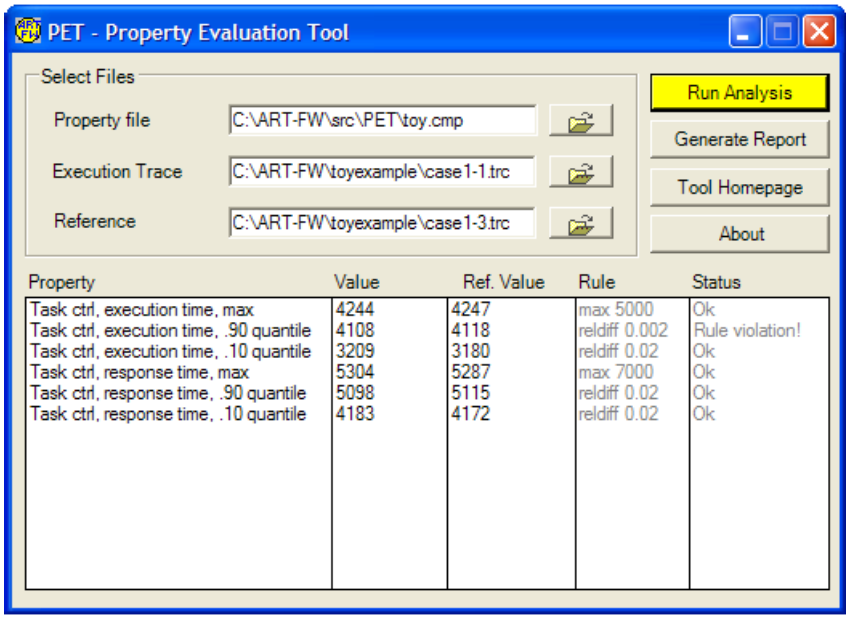

Figure 3.6: The Property Evaluation Tool

The tool can automatically inspect the analysis results using *comparison rules*. These comparison rules may be specified for each property in the com-

parison file and are intended to define which analysis results are acceptable and which are not. When PET has evaluated the PPL queries, it inspects the results of each query and applies any specified comparison rule. Any results that violate a comparison rule is then pointed out to the user. The comparison rules are defined in a simple language consisting of four functions, where each function has one parameter.

- absdiff(n) – Absolute difference at most n
- reldiff(n) – Relative difference at most n %
- max(n) – The value should be below n
- min(n) – The value should be above n

A comparison rule consists of one of these four functions and a specified value, which may be a limit (max, min) or a tolerance (absdiff, reldiff). The two latter comparison rules require two execution traces, as they compare two analysis results, while the first two (min, max) compares a single analysis result with a constant value.

3.3.3 The Tracealyzer

The Tracealyzer has two main features, visualization of an execution trace and a PPL terminal, i.e. a front-end for the PPL analysis tool. The execution trace is presented graphically. The task execution and the values of generic probes are presented in parallel, allowing e.g. resource usage and the executed services to be presented next to the task execution. Moreover, it is possible to navigate in the trace by using the mouse and also to zoom in and out and to search for task instances based on name and (optional) a minimum or maximum execution time, response time or fragmentation (number of preemptions).

The user can select a task instance (execution) by clicking in the graphics. The selected task instance is highlighted, which visualize what fragments correspond to the task instance and information about the task instance is presented, such as the execution time and response time of the instance and the average execution and response times for the task. If more task statistics are desired, it is possible to generate a report, containing a variety of information about all tasks.

The window contains a list labeled *probe visibility*, presenting a list of the generic probes that have been recorded in the current execution trace. Selecting one of the generic probes in the probe list will display its value over time,

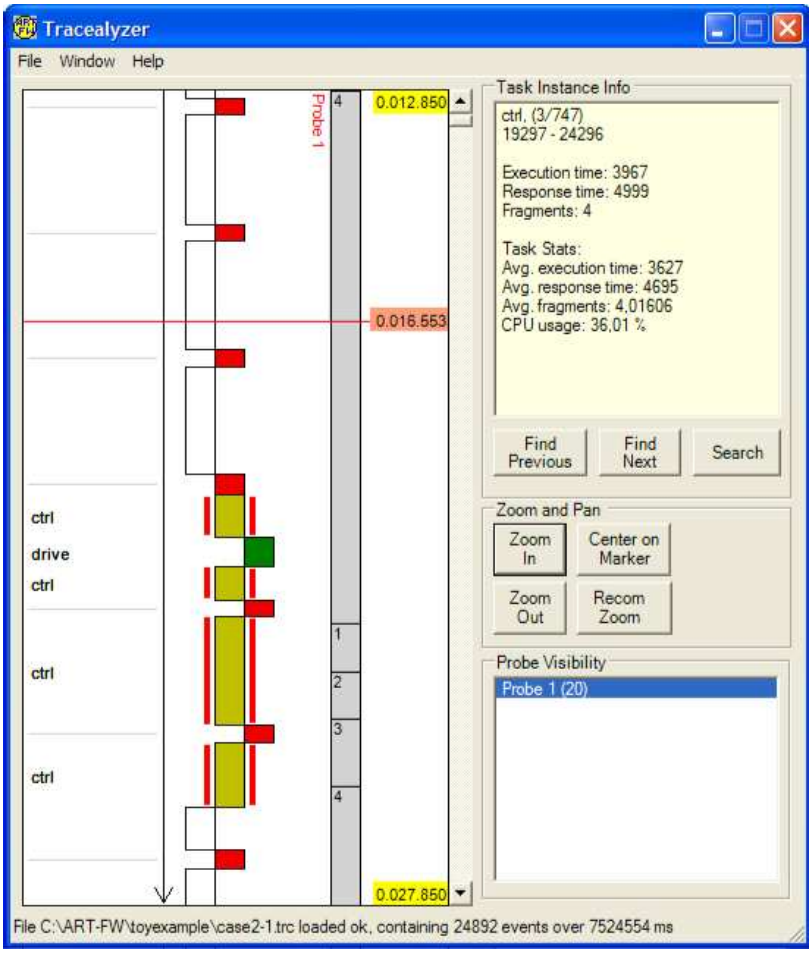

Figure 3.7: The Tracealyzer tool

next to the task execution. Since a probe can monitor logical resources of any type, the meaning of a certain probes is defined by the context of the associated probe. The probe visualized in Figure 3.7 of the Tracealyzer monitors the number of messages in a particular message queue. The value “20” next to the probe name is the numerical identifier of the probe. This information is important as the PPL language is limited to identifying probes by their numerical identifier. It is also possible to save a list of the task instances to a text file. This way, the data can be imported into other applications and visualized in other ways than the ones provided by Tracealyzer.

Apart from visualizing the data in an execution trace, the Tracealyzer also contains a PPL terminal. It is basically a front-end for the PPL analysis engine. The terminal contains two fields, one input where PPL queries can be typed and one output where the result are presented.

The graphical visualization of execution traces, provided by the Tracealyzer tool is, according to our experiences, an effective way of increasing the understandability of the system. The tool has been introduced at a collaborating company, ABB Robotics, with good results. When developers at the company first visualized execution traces from the latest version of their system, there were immediate reactions on details and suspicious behaviors in the execution trace. We provided them with a new view of the system behavior, which increased the system understandability and facilitated debugging activities.

3.4 Discussion

In this chapter we have presented how dynamic analysis may be used for improving the analyzability of complex embedded systems. We have presented what properties are of interest for analysis and why. Moreover, the necessary code instrumentation and the resulting impact on resources such as CPU and memory were discussed. The impacts on such resources are negligible for large systems as the one considered in this thesis. However, there is a risk of experiencing problems with the probe effect if code instrumentation is removed after the analysis. In this work, we assume that we can leave probes in the system, partially due to the fact that they have little impact, and partially due to the fact that they can be used for other purposes than modeling for behavior impact analysis; they allow regression analysis and may also be used for debugging purposes.

In relation to the research questions stated in Section 1.2, this chapter has partially answered Q1, the first of two sub-questions.

Q1: What methods are suitable for extracting the information necessary for a temporal behavior model from a complex embedded system implementation containing millions of lines of code?

Dynamic analysis is a suitable method for extracting information from very large implementations and may be used for modeling of complex embedded systems. Dynamic analysis is especially suitable for collecting quantitative information regarding the dynamic aspects of the system behavior. However, since only a limited amount of software probes may be used in order to keep the resource usage on a reasonable level, additional sources of information is necessary, especially for the construction of detailed models also including semantic dependencies between the temporal behaviors of tasks.

Dynamic analysis does not only enable the construction of models for behavior impact analysis, but also enables regression analysis and improves system understandability. Two tools have been presented which utilize dynamic analysis, for different but related purposes.

The main purpose of the Tracealyzer is to visualize an execution trace. The task execution and the values of generic probes are presented graphically, in parallel. Visualizing the task execution is far from unique; there are several commercial tools that have similar functionality, e.g. the system WindView from WindRiver [WRW].

The second tool, the Property Evaluation Tool, allows comparison of exe-

cution traces with respect to a set of PPL queries. This tool enables regression analysis, presented in Section 3.1.3, as well as behavior impact analysis based on temporal behavior models (Chapter 4) and validation of temporal behavior models (Chapter 5). PPL is unique as far as we know and the most important part of this framework, as it provides the possibility to formulate and systematically evaluate system properties with respect to recordings.

Developers at ABB Robotics are successfully using the Tracealyzer today as a debugging tool. Further, regression analysis and the Property Evaluation Tool have been introduced to a small group of experienced system developers. They quickly understood how they could benefit from the new possibility of analyzing the system and the method will gradually be introduced into their software development process. The software behavior recorder that is required by both the presented tools has been integrated in their robot control system and is active in both debug- and release-versions of the system.

The Property Evaluation Tool basically uses what [LK93] refers to as the inspection approach, extended with comparison rules allowing a tolerance to be specified for each property to be compared. This is not an ideal solution, since it is based on a comparison of a single execution trace from the two system versions. There is a risk that at least one of these execution traces is not representative for the system version as it contains extreme values.

The confidence interval approach, presented in [LK93], is highly relevant for the comparison of execution traces, e.g. in a regression analysis. A future implementation of this method could utilize PPL in order to extract the statistics of interest from each recording, e.g. average response times of tasks etc. However, additional functionality in the form of a program or a script of some sort is necessary in order to construct the confidence interval.

Chapter 4

Modeling Temporal Behavior

This chapter presents an approach for the development of models describing the temporal behavior of existing complex embedded systems, i.e. large industrial software systems typically with cost-, dependability- and real-time requirements.

Temporal behavior models allow for analyses of important properties of the system behavior related to timing and resource usage, e.g. response times, CPU utilization and utilization of limited logical resources. Such properties may be of vital importance for the correct operation of the system and also effect softer issues such as user-perceived system performance. As discussed in Chapter 1, introducing analyzability with respect to these properties may improve productivity in maintenance of such systems, since potential problems associated with changes may be predicted early, before implementation, and thereby avoided.

Without such models it is often hard to predict how changes to the system may effect the temporal behavior since the temporal behavior is dependent on many factors, such as varying execution times, the system environment, the task scheduling causing preemptions, and communication/synchronization between tasks.

Today, most companies developing complex embedded systems have no suitable models that allow analysis of the temporal behavior of their systems. The exceptions include systems that have been designed with analyzability in mind, e.g. the automotive systems produced by Volvo Construction Equipment [MWN+04]. For systems that have not been initially designed with analyzability in mind, introduce analyzability without redesigning the system requires the

development of an analyzable behavior model of the system, an abstraction focusing on a particular aspect of the system behavior, in this case the temporal behavior.

A temporal behavior model, together with suitable analysis methods, provides analyzability which can be used to predict the effect on the system behavior caused by a proposed change to the system, e.g. adding a new feature. The original model of the system is extended with a prototype of the change. Due to the high level of abstraction in the temporal behavior model, changes can be prototyped with little effort.

The impact on the system behavior caused by the proposed change is predicted by comparing analyses of the original model with analyses of the modified model, the one containing the prototype change. If the comparison reveals unacceptable impacts of a proposed change, the change may be re-designed and future problems associated with this impact are thereby avoided.

The ability to predict the temporal behavior of a future system can also be used to predict how existing parts of the system will effect new functionality. If an analysis reveals that the new functionality is significantly affected by other parts of the system, i.e. higher priority tasks, the new functionality may have to be re-designed in order to function as intended. An example is to estimate the response time distribution of a new task. For simple systems response times can be calculated using the analytical methods discussed in Section 2.3.1, i.e. [LL73, MJ86], but as discussed in Section 2.5, such methods can typically not be used for complex embedded systems.

As an analysis of the temporal behavior of a future version of a system can be performed in an early phase, it is possible to avoid problems that otherwise are discovered late, in integration testing or post-release, and thereby costly. This improves productivity during system maintenance and as potential problems are identified and avoided in an early phase, the quality and reliability of the system can be improved as well. The use of this analysis allows companies developing complex embedded systems to better handle a high and increasing complexity, i.e. to stay longer in life-cycle phase III as depicted by Figure 1.1 presented in the introduction.

In previous works [Wal03, AWN04a, WAN03b, WAN$^+$03a, AN02] we have used the term “impact analysis” referring to the analysis of the impact on a system’s temporal behavior caused by a maintenance operation. However, this term is general and there is at least one other definition of the term “impact analysis” in Software Engineering research; analysis of the static dependencies between components in an implementation which needs to be considered when implementing a change. Works in this area are e.g. [AB93] where the

authors define a framework for comparison of different approaches for impact analysis and [CFV99, QVWM94] which presents two interesting case studies. In order to avoid confusion we will hereafter use the more specific term *behavior impact analysis* to denote impact analysis with respect to system behavior. Another terminology issue is the relationship between the terms “change” and “maintenance operation”. The term “maintenance operation” refers to the activity of adding or changing functionality in the system, while “change” refers to the resulting alteration of the implementation. Thus, a behavior impact analysis predicts the impact of a change, which in turn is the result of a maintenance operation.

The construction of a temporal behavior model of a complex embedded system, detailed enough to allow behavior impact analysis, requires information about the system design and behavior from several sources:

System Implementation The most accurate and reliable documentation available is the systems implementation, i.e. the source code. However, due to the size of complex embedded systems, the construction of a behavior model from a systems implementation requires a significant effort, which may require tool support. It is therefore important to focus the modeling effort on the relevant parts of the system. However, information about the temporal behavior can not be obtained from the implementation alone, as e.g. execution time is not visible in the implementation.

Run-time system As the systems considered in this thesis are real-time systems, the model needs to include information about timing, e.g. execution times. Moreover, systems of this type interact with an environment, physical processes or other computers. This environmental interaction effects the system behavior and thus needs to be modeled as well. This information can be collected using dynamic analysis, discussed in Chapter 3, i.e. recordings on the run-time system in a realistic environment.

System Documentation Various documentation typically exists that can facilitate the understanding of the system implementation, system architecture, and the requirements of the system. Other important documentation is realistic test-cases, which are necessary for dynamic analysis activities.

System and Domain Experts An important information source is the interviews and discussions with system experts and developers. This facilitates the

understanding of e.g. the application domain, the requirements of the system, the software architecture and what properties that are of interest for analysis.

Given that a model has been developed, the main problem associated with performing a behavior impact analysis is the *model validity*, i.e. if the model sufficiently and accurately describes the system with respect to the properties of interest. One way of increasing the probability that the model is valid is to follow a well-defined process when developing the model; we therefore propose a modeling process suitable for complex embedded systems later in this chapter. However, in order to assure that the model is a valid description of the intended system, a model validation activity is necessary. A model validation typically compares the results from analyzing the model with observations from the corresponding system. Model validity and techniques for model validation are discussed in Chapter 5.

The remainder of this chapter is divided into four sections where Section 4.1 discusses the primary use of temporal behavior models, the behavior impact analysis, in greater depth. Section 4.2 presents an approach for modeling of a complex embedded systems behavior and timing, while Section 4.3 focuses on modeling the environment of complex embedded systems. Finally, Section 4.4 concludes the chapter and discusses how this contribution relates to the research questions of this thesis, stated in Section 1.2.

4.1 Behavior Impact Analysis

A behavior impact analysis predicts how a change will impact the behavior of the system with respect to specific properties of the system behavior, the properties of interest. Using this analysis, a designer of a new feature can try alternative designs on a model, evaluate their impact on the system and thereby avoid designs that negatively impact the system behavior, causing e.g. reduced performance or potential timing errors. Such problems only manifest themselves in full system testing, i.e. after implementation, unit testing and integration, and often occur in rare situations only making them difficult to reproduce [Sch91] and thereby time-consuming and costly to detect and correct.

Consequently, as the risk of introducing such errors is reduced, the productivity during maintenance is increased and the development time becomes more predictable. This may also improve system reliability as the quality of the system is improved through better design.

In a behavior impact analysis, two behavior models are compared, the *orig-*

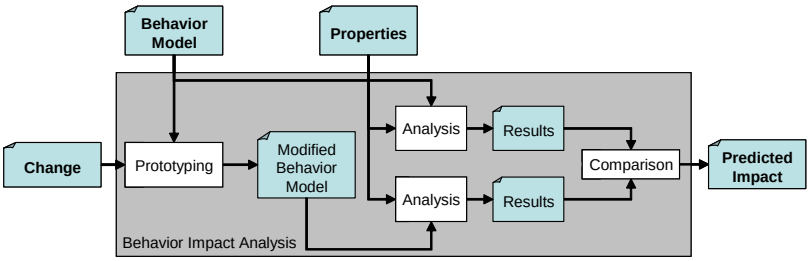

Figure 4.1: The behavior impact analysis

inal model corresponding to the current version of the system, and a *prototype* corresponding to the future version of the system, which represent the results of implementing the change. The comparison is done by analyzing both models with respect to the properties of interest, resulting in two sets of analyses results, one set for each model. These sets of analyses results are compared property by property. Any statistically significant differences are impacts of the change. The steps of the behavior impact analysis are depicted in Figure 4.1.

Changes are prototyped on a behavior model by comparing the existing model with the design of the proposed change. Thereby, the parts of the model that may be effected by the change can be identified and modified to represent the change. As the model represents the system using different levels of abstraction for the different tasks and services, depending on their relevance for the model, a prototyped change may in some cases correspond only to a small increase in the execution time of a service. In other cases, where a change effects the tasks that are described in detail, the change may be prototyped in a detailed manner, e.g. similar to a full implementation but with focus on the control flow and timing.

Figure 4.2 depicts the process of using behavior impact analysis in the maintenance of complex embedded systems. The output of the analysis, the predicted impact, typically needs to be inspected by a system expert in order to decide whether or not the predicted impact is acceptable. However, if a set of rules are defined, that for each property specify what results are acceptable, this activity can easily be automated.

If the predicted impact is unacceptable, the designers need to change their design in order to consume fewer resources, i.e. CPU time or logical resources, or if possible, change the system in order to provide the resources required by the new feature, by e.g. switching to a faster CPU, and repeat the anal-

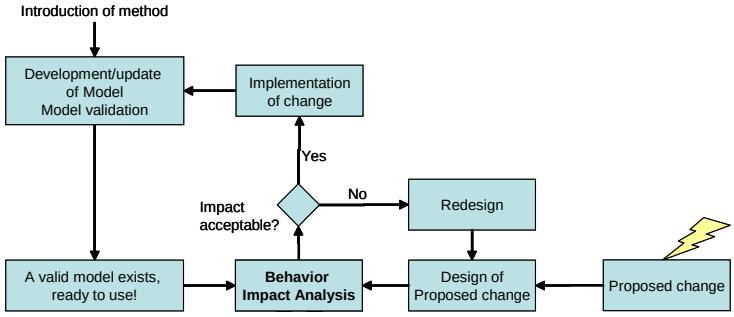

Figure 4.2: Using behavior impact analysis in maintenance

ysis. When the predicted impact is acceptable, the proposed change may be implemented, but the implementation of the change must not consume more resources than the analyzed prototype. This means in practice, to impose a resource budget on the implementer, specifying e.g. a maximum allowed execution time of the functionality. This implies that the developer needs to be aware of the resource usage when developing a feature. When the change has been implemented, the behavior model needs to be updated in order to reflect the final implementation of the change, since it may be different compared to the prototype used in the early analysis. If a major difference is discovered between the prototype and the final implementation when updating the model, the analysis should be repeated in order to check if the previous analysis result remains valid. This is especially important if the model used for the early analysis turns out to be optimistic.

A behavior impact analysis is motivated for major changes to the system, such as:

- changes effecting the activation of services in other tasks,
- changes effecting the use of shared state variables and other logical resources such as message queues and semaphores,
- changes that are likely to have a significant impact on the execution time of a service (especially if the task has a high scheduling priority),
- the addition of new tasks to the system.

It may not be motivated to use behavior impact analysis if the changes are very small and unlikely to have any side effects, for instance a small change

in a calculation of output data. Even though small changes may have major undesirable side effects in the worst case due to the discontinuous nature of software systems, the probability of this is low with regard to our experiences; otherwise industry would use formal analysis tools much more extensively than today.

4.2 Modeling System Behavior

This section presents a process for developing models of the temporal behavior of complex embedded systems. Further, the section proposes a structure dividing the model into two components; where each component is further divided into two subcomponents. Thus, in total, the complete analyzable model, the *system model*, consists of four components:

- A *behavior model* – describing the functionality of a complex embedded system, including timing. The behavior model consists of the following components:
  - *Functional model* – describes the behavior of the individual tasks and services in the system, with a focus on control flow.
  - *Model parameters* – contains quantitative information on systems timing and probabilities that is used by the Functional model.
- An *environment model* – describing stimuli from the environment that effects the behavior of the system. The Environment model consists of the following components:
  - *Specific stimuli model* - describing stimuli specific for a certain test case.
  - *Common stimuli model* - describing stimuli that are always present.

The rest of this section presents an approach for the development of the behavior model, while the environment model and its components are separately discussed in Section 4.3.

4.2.1 The Modeling Process

The construction of a behavior model of a complex embedded system consists of three main activities:

- Development of the *model specification*
- Construction of the functional model
- Constructing the model parameters

Figure 4.3 depicts these activities in construction of a behavior model and the resources on which the activities depend. When constructing the behavior model, the first activity is the construction of a model specification, a document describing the system and the properties of interest for analysis. The purpose of this document is to gather information from several sources in order to describe the many factors that effects the temporal behavior of the system. This is especially important when the modeler has limited experience of the system, but also important if the modeler is an experienced system developer or architect, as the document serves as a specification of the modeling effort and may be communicated with other system experts.

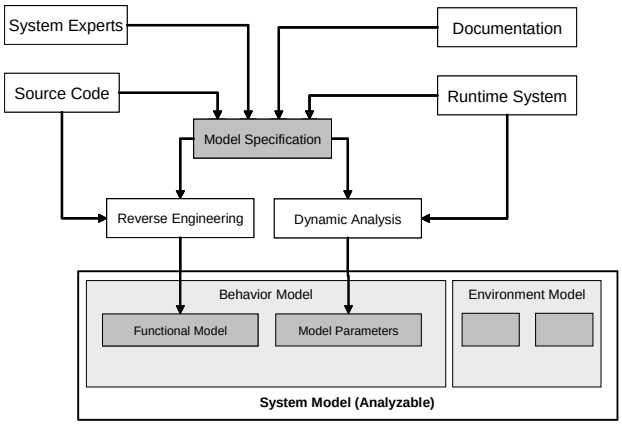

Figure 4.3: The process of constructing a behavior model

The next activity is to construct the functional model, based on the model specifications and reverse engineering of the system implementation. Based on the information in the model specification the relevant tasks and services are

modeled with respect to the externally visible events. During the reverse engineering it is appropriate to prepare for the construction of the model parameters by performing the code instrumentation necessary for the dynamic analysis, as discussed in Chapter 3. This code instrumentation needs to specify identifiers for the different model parameters resulting from the dynamic analysis. These identifiers are necessary in the functional model in order to refer to specific model parameters.

The third activity in the modeling process is to build the instrumented code into an executable system and perform recordings using appropriate test cases. From the recorded execution traces the model parameters are finally extracted.

4.2.2 The Model Specification

When constructing a behavior model, it is important that the modeler understands the "big picture", e.g. how the system is used by the customers, the requirements on the system and the various configurations that may exist. However, the modeler does not need to be a system expert if regular meetings can be arranged with system experts or other experienced developers in order to develop the necessary system understanding. The understanding of the system requires documentation in a model specification, developed by the modeler in cooperation with the system experts.

The model specification is a document describing the system architecture and runtime behavior at a high level of abstraction, as well as the properties of interest for analysis. It is important that the model specification is carefully reviewed by other system experts in order to avoid misunderstandings and allows for system experts to contribute with additional information and comments relevant for the modeling.

In order to assure the validity of the model specification, it should be based on interviews/discussions with several system experts and developers. Moreover, the modeler needs to study the code, runtime behavior (through dynamic analysis tools) and available documentation in order to better understand the details of system and verify the model specification.

To provide the reader with an example of a small model specification as well as a brief description of the system that has been studied in earlier work [Wal03, WAN+03a, AN02], we present a model specification describing the system in Appendix C. Names are omitted due to business secrecy reasons.

According to Figure 4.3, the model specification is based not only on code studies, documentation and discussions with system experts, but also on studies of the runtime system, i.e. the information in recorded execution traces. When

trying to understand the behavior of a complex embedded system, visualization of execution traces is a powerful method complementary to studying code and documentation. This is especially true when studying the temporal behavior of the system, as it is seldom documented and not visible in the implementation.

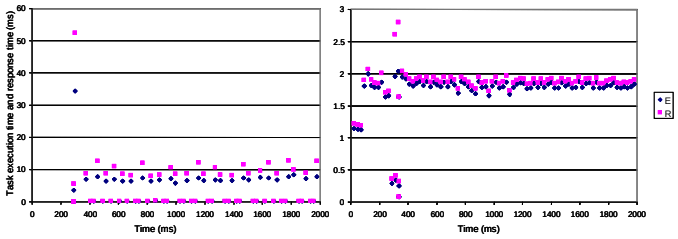

Figure 4.4: Visualization of observed execution and response times of two tasks

An execution trace can be graphically visualized in different ways in order to increase the understandability of the systems behavior. In Figure 4.4, two graphs are depicted, showing the temporal behavior of two tasks in a real system. These tasks are complex and time-critical, communicating with many parts of the system.

The two graphs correspond to two particular tasks during the same period of time. Each execution of a task results in two dots, where the Y-value describe the execution time (E) or response time (R) and the X-value correspond to the start time of the task instance. In both these graphs, we can observe a discrepancy around time 300. This is most likely due to a dependency between these tasks, e.g. they might both react to changes in the same global state variable. Such dependencies between task's temporal behavior may be difficult to detect based on the other sources of information, since they are rarely documented.

4.2.3 The Functional Model

Constructing the functional model of a system is the task of documenting the behavior of the system in a notation suitable for analysis, at an appropriate level of abstraction. The resulting model describes the behavior of the tasks in the system, the attributes of the tasks such as scheduling priority. In many cases, a task consists of a set of services, separate behaviors, which are executed on request from other tasks in the system. Each service in a task may be regarded

as a set of sequences of *externally visible events*, i.e. events that effects other tasks or the environment. An externally visible event may be e.g. the sending an IPC message to another task, the writing to a shared state variable or the locking of a semaphore. By modeling the implementation on this level of abstraction, the interactions between tasks are modeled while still keeping the model sufficiently abstract. This level of abstraction is commonly used in formal verification tools for software e.g. VeriSoft [CGP02] and SPIN/Modex [Hol97, SPI, Hol03]. In comparison with the real implementation, models on this level of abstraction do not describe the data flow, i.e. calculations and assignments of data variables, but the focus is on the events that may effect the task scheduling.

In order to find such events in the source code, reverse engineering tools may be utilized, such as Rigi [BG97], Imagix 4D [BG97], Fujaba [KSS+02] and Rational Rose [KSS+02]. These tools parse the code and visualize the relationship between classes and files, for instance as class diagrams or call graphs. A problem with such tools is that they are general, i.e. not adapted for the specific system in focus, and the resulting views are therefore suboptimal. Most embedded systems are implemented in imperative languages, not object oriented languages, and therefore naming conventions are commonly used to group related functionality. Also, if tasks consist of multiple services, the reverse engineering tool needs to be aware of this in order to identify the individual services. This is very important, as the reverse engineering is to be performed service-by-service, and it is therefore desired to isolate the behavior of individual services in the task. Thus, knowledge of naming conventions and other system-specific properties of the implementation can facilitate the reverse engineering significantly. There are existing reverse engineering tools that are highly adaptable, e.g. Rigi. Such a tool could possibly be adapted in order to better present implementations containing naming conventions and services.

When constructing the functional model, it is important to focus the modeling effort on the areas of the system that are relevant to the properties of interest. Moreover, as a task may consist of a large set of services, the relevant services need to be identified as well. Only tasks and services that are considered relevant need to be described in detail, while less relevant tasks/services can be described on a higher level of abstraction. In a temporal behavior model, tasks can be modeled in a high level of abstraction by describing the temporal behavior only, i.e. inter-arrival time, scheduling priority and execution time. In the same way, individual services may be modeled solely by their execution times. The tasks and services in focus of the modeling effort should be specified in the model specification.

A problem when modeling is that most services have more than one possible path of execution, as there are selections (including loops). To model all possible paths through a service in detail may not always be possible, as the selections may depend on variables not included in the model due to the necessary level of abstraction. When such selections are discovered the modeler may either extend the model to include the necessary variables, or model the selection in a probabilistic manner, by identifying the probabilities of different behavior using dynamic analysis and specifying these probabilities in the model. Probabilistic models are often accurate in the average case, which is suitable for e.g. performance estimation, but as the underlying mechanisms of selection is not modeled there is a risk of missing semantic dependencies between the selection and other behavior in the system. Such dependencies may make the model overly optimistic or pessimistic, depending upon the situation. Consider a case where a service contains two selections, which are both modeled in a probabilistic manner. The code executed by these selections corresponds to a major part of the total execution time of the service. The code and the corresponding probabilistic behavior model are presented in Figure 4.5. The behavior model is divided into a functional model and model parameters as discussed in Section 4.2.

Implementation	Behavior model	

if (foo > 100)
{
  CalculateA(foo);
}
if (bar > 200)
{
  CalculateB(bar);
}
	Functional model
	chance (P1)
	{
	  execute(A);
	}
	chance (P2)
	{
	  execute(B);
	}
		Model parameters
		P1: 0.1
		A: 1000-2000
		P2: 0.05
		B: 1100-2300

Figure 4.5: Probabilistic modeling of selections

The functional model is expressed using the modeling language ART-ML proposed in earlier work [Wal03, AWN04a, WAN+03a]. This thesis presets a new version of ART-ML, version 2.0, which can use temporal data stored separately, in the model parameters. This was not possible in the previously proposed version, version 1.0. By separating the information in the model parameters from the functional model, a more modular model is obtained, where

different model parameters can be used for the same functional model to represent different hardware platforms.

In ART-ML, the *chance* statement expresses probabilistic selection, while the *execute* statement expresses the consumption of CPU time. Consequently, in Example 1, the calculations in the system implementation (CalculateA and CalculateB) are only modeled from a temporal perspective. According to the behavior model, the probability of both execute-statements being executed is 0.5 %, as the probabilities P1 and P2 are assumed to be independent. However, in the real system, this may not be the case. If the variables foo and bar are correlated, the probability of executing both calculations may be very different. It may be the case that CalculateA and CalculateB are mutually exclusive due to a dependency between the variables foo and bar. Probabilistic modeling are thus not suitable methods for sequences of selections that are likely to be dependent, but may in other cases be a powerful way of modeling an observed behavior for which the underlying mechanisms are unknown.. Probabilistic models are however non-deterministic and thus harder to analyze compared to a deterministic model, as the number of possible scenarios become very large. The modeling should therefore strive to minimize the number of probabilistic selection in the functional model.

4.2.4 The Model Parameters

When modeling real-time systems, the timing of the system needs to be modeled, e.g. the execution times of different services in tasks and the inter-arrival times of different events or task activations. When modeling systems in a probabilistic manner, the probabilities of different events need to be modeled as well. Furthermore, to improve the accuracy of the behavior model it is necessary to also model state variables that are likely to have a large impact on the execution times of the system.

This information should be separated from the functional model, as the temporal behavior is dependent on the systems hardware platform (the CPU), which often changes. Companies manufacturing complex embedded systems often switch to faster, more recent CPU's, as the previously used CPU becomes less available and more expensive, while new versions offer better performance for similar or lower price. In a well-designed system, a CPU switch should not require code changes, apart from in device drivers and other HW-specific code, and is therefore easily performed. However, as the execution times of the system are decreased by the faster CPU, the temporal behavior is changed.

By having the timing information stored separately, as model parameters,

the updating of the timing information can be done without involving the functional model. Model parameters can be automatically generated from execution traces with ease.

Moreover, if the system has several different hardware and/or software configurations, there may be one set of model parameters for each hardware configuration, while a single functional model can be used for all configurations, thereby avoiding problems with inconsistencies between models of different configurations.

The functional model contains references to the model parameters, where each reference corresponds to a particular value or a probability density distribution, representing the probabilities of different execution times, inter-arrival times of tasks, or outcomes of probabilistic selections.

An execution time distribution may describe the execution times of a whole task, of a service or of a minor part of a service, depending upon the level of abstraction required. An inter-arrival time distribution describes the inter-arrival times of tasks, which may be periodic (fixed inter-arrival time) or sporadic (probabilistic inter-arrival time). To describe a probabilistic selection requires only one value, the probability of the condition being true.

The model parameters are acquired through dynamic analysis, as discussed in Chapter 3. Recordings are made of the real system in different realistic situations, using existing test cases. Given that the system contains the necessary code instrumentation, the data can be extracted from the resulting execution traces.

It is important that several different test cases are used, in order to identify dependencies between the test cases used and the distributions. For instance, in a specific test case the execution times of a specific service may be significantly higher, as the service is dependent on a state variable effected by the test case. Such dependencies may be included in the behavior model to improve accuracy. An example is presented in Figure 4.6.

Implementation	Behavior model	

Service A:
PROBE(state7);
result = calculateX(...);
ipc_send(result);
	Functional model
	Service A:
	if (state7 == X)
	execute(A1);
	else
	execute(A2);
	ipc_send(emptymsg);
		Model parameters
		A1: 1000-2000
		A2: 200-300

Figure 4.6: Modeling a dependency between state and temporal behavior

Figure 4.6 shows the implementation of a service and the corresponding functional model and model parameters. The state variable *state7* is assumed to effect the execution time of the routine calculateX. A software probe has therefore been inserted in the service to record the state each time the service is executed and recordings are made using this probe and the general probes proposed in Chapter 3, e.g. task-switch probe. If the resulting data shows a correlation between the state and the execution time, the behavior model needs to be updated to take this state into account, i.e. to specify different distributions for the execution time of the service depending on the value of the state variable.

To update the functional model, it is necessary to model this state variable, which implies the construction of a finite state machine describing the different states and possible state transitions. The finite state machine will be integrated in the functional model, but may also be stored separately as a UML state diagram, which contributes to the documentation of the system.

The next step in the process of modeling this dependency is to update the functional model of the service to choose between the available distributions based on the value of the state variable. This selection could be implemented as in Figure 4.6, where the functional model explicitly makes a selection between two distributions. Another possible solution is to express the mapping between states and distributions as expressed in the model parameters. The state would be used as an argument to the execute statement, which selects the proper distribution based on the model parameters. This would improve the readability of the functional model and is therefore preferable. An example of a behavior model using this approach is found in Figure 4.7.

In order to ensure the validity of the behavior model, the model parameters must be constructed in a statistically sound manner that accurately represents the real system. Preferably the model parameters should be based on a large

Implementation	Behavior model	

Service A:
PROBE(state);
result = calculateX(...);
ipc_send(result);
	Functional model
	Service A:
	execute(state,A);
	ipc_send(emptymsg);
		Model parameters
		A(state):
		1: 1000-2000
		2: 200-300

Figure 4.7: Expressing the mapping between state and temporal behavior in the model parameters

set of measurements, collected from the system in various situations and configurations. Ad-hoc estimations based on a few recordings can at most be used for a rough model of a single configuration of the system.

Given that sufficient amounts of data exist, another important issue is how to transform execution traces into model parameters. Data on observed execution times, inter-arrival times and selection outcomes can easily be extracted from the execution trace. There are however several methods of representing this data in the model parameters. In some situations it may be possible to represent the data using limits only, i.e. minimum and maximum values, and to allow for an analysis to randomly choose values in the specified range. This corresponds to representing the data with a uniform distribution.

However, using uniform distributions is often a major simplification and may result in models that behave significantly different compared to the real system. In order to accurately model a system containing queues and other limited logical resources it then becomes necessary to describe the probability distribution of the recorded data, i.e. the probabilities of different execution times, inter-arrival times or selection outcomes. One way of describing the recorded data is to find a suitable theoretical distribution, such as the Normal distribution, the Exponential distribution or the Weibull distribution and use standard techniques of statistical inference to calculate the parameters that fits a theoretical distribution to the recorded data. This is discussed in [LK93]. However, the observed data often has a complex distribution, which makes the theoretical probability distributions unsuitable. As an example, Figure 4.8 shows an execution time probability distribution measured from a real system. The observed execution times have been grouped with a granularity of 3 μs and the dots in the graph indicate the number of service executions observed (y-value) for each group of execution times (x-value).

The depicted probability distribution shows two major peaks, which im-

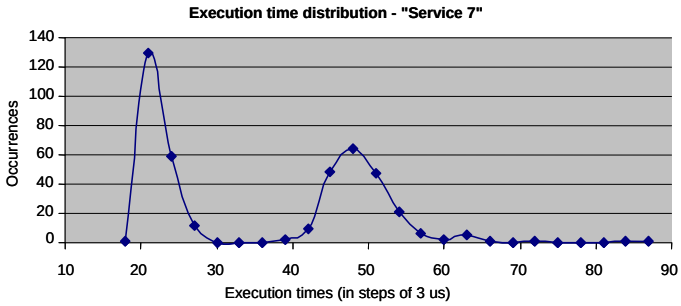

Figure 4.8: An example of a complex execution time distribution

plies that there are at least two paths through this service. This has not been taken into account when preparing for the recording and as a result all paths through the service are described using a single complex probability distribution. To model this data using one of the standard theoretical distributions would result in a poor fit. In order to approximate a theoretical distribution to such complex probability distributions as the one depicted by Figure 4.8, more advanced distributions must be used, such as Bézier distributions [LK93, WW96]. A Bézier distribution functions with sufficiently high degree can approximate a distribution of any shape. It is not easy to find the parameters that produce best fit using a manual trial-and-error approach, but there is software available that does this with accuracy [WW96]. Another problem that is more serious with Bézier distributions is that there are few or no analysis tools (simulation frameworks) that support the use of these distributions, according to [LK93].

Another solution to the problem of describing complex probability distributions is to divide the data into a set of simpler distributions, which may be modeled using the standard theoretical distributions with good result. This is especially suitable if the distribution resembles the one in Figure 4.8, i.e. several “peaks” with “empty space” in between. For such distributions, the data may be divided based upon their values. For instance, if we consider the data in Figure 4.8, we can define three regions, one from 0 μs to 35 μs, one from 36 μs to 60 μs and finally one region from 61 μs and up. Each recorded executions of the service are assigned to one of the three regions depending upon its execution time. This result in three sets of data which can be individually expressed using a theoretical probability distribution with good result. The functional model selects between the three distributions in a probabilistic man-

ner, based upon the number of service executions in each region. Figure 4.9 shows a behavior model describing the service as depicted by Figure 4.8. The probabilities P1 and P2 express the probabilities of selecting one of the distributions A, B or C. The probability of selecting distribution C is 1-P1-P2. Note the inner chance statement with the probability P2/(1-P1). This construction is necessary as the chance statement only allow binary selection. In order to better support modeling of complex execution time distributions ART-ML can be extended with a probabilistic version of the C statement "switch". Figure 4.10 shows the same model using such a statement, referred to as *pswitch*.

Functional model	Model parameters

Service A:
chance(p1){
execute(A);
}else{
chance(P2/(1-P1)) {
execute(B);
}else{
execute(C);
}
}
	A: Norm(...);
	B: Norm(...);
	C: Uniform(...);
	P1: 20%
	P2: 10%

Figure 4.9: An ART-ML model using three distributions and probabilistic selection to model a complex execution time distribution

Functional model	Model parameters

Service A:
pswitch{
p(P1){ execute(A); }
p(P2){ execute(B); }
default{ execute(C); }
}
	A: Norm(...);
	B: Norm(...);
	C: Uniform(...);
	P1: 20%
	P2: 10%

Figure 4.10: An ART-ML model using the proposed pswitch-statement

Note that the pswitch statement has quite different syntax compared to an ordinary "switch". This and other extensions of ART-ML proposed in this thesis can be found in Appendix A.

Another method to split the data is to identify the cause of the different peaks, i.e. the state variable(s) responsible for the different temporal behaviors, and add a software probe to record this state as discussed in Chapter 3. The execution time can then be modeled with respect to the state variable, as discussed earlier in this section. This is especially suitable if it is not intuitive how to split the data into regions, for instance if there is no "empty space" between the peaks.

However, in many cases it is not desired or not possible to investigate and record the cause of the different temporal behavior, but it is still desired to accurately describe the probability distribution. An example is when modeling the inter-arrival times of external events, which e.g. triggers a task. In such situations, an empirical distribution can be used, i.e. a list of observed values, from which values are either randomly sampled or used in the same order as they where recorded. That way the probability distribution is accurately described. When using the latter approach in the context of simulation, this is referred to as *trace-driven* simulation [LK93]. There are however drawbacks with this approach. One problem is that it only provides the exact values that have been observed. There may be "gaps" in range of values observed, which may not be representative to the "true" underlying distribution but due to random fluctuations in the system during the recording. Another problem is that extreme values, that seldom occur, may not be included in the model. Such values are often very interesting for analysis. These problems can be reduced by making more recordings in order to observe a greater number of values.

To summarize, four general methods of modeling the recorded data have been identified:

- Simple distributions may be modeled using a standard theoretical distribution, e.g. a uniform distribution, Normal distribution, Exponential distribution or Weibull distribution.
- Complex distributions may be divided into several simple distributions based on values or extra recorded information, e.g. outcome of selections. The subsets of data are more suitable for modeling using a standard theoretical distribution.
- Complex distributions may be described by using more advanced theoretical distributions, such as Bézier distributions.
- The data is used directly, in the form of an empirical distribution, i.e. a list of observed values. The analysis method samples values from this list.

Different methods may be used in the same set of model parameters for modeling different set of data. For tasks which are described on a high level of abstraction, advanced theoretical distributions may be used to accurately model e.g. complex execution time distributions even without describing the behavior of the task in the functional model, while prototypes of changes may use uniform distributions to model execution times (or a fixed worst-case value) since little is known before implementation.

4.2.5 Identification of Dependencies

When constructing a behavior model of a complex embedded system there are at least two types of dependencies that are of interest. Dependencies between temporal behavior and state variables are highly relevant for the model parameters, as discussed in Section 4.2.2 and Section 4.2.3. Another type of dependency is interaction between tasks, i.e. externally visible events such as IPC messages. Dependencies of both types may be identified manually, by inspecting the source code of each service, but this approach is tedious and error prone, automation is therefore desired.

A pragmatic approach to automatically identifying state variables that are likely to effect execution times is to search the source code for state variables that are used in specific contexts. If a state variable is read in the condition of a selection, the value of the state variable is likely to affect the outcome of the selection, i.e. which branch of the selection that is executed. If the difference in execution time between the branches is significant, then the state variable is of interest for modeling. Heuristic rules can be used in order to automatically identify selections (and thereby state variables) that are likely to effect the execution time. For instance, an if-statement containing a large number of statements in one branch and no statements at all in the other branch (no else-statement) is more likely to have a significant impact on execution time compared to a selection where both branches consists of a few assignments only. Since it is not required to determine the exact execution times of the branches, but sufficient to determine if one of the branches have significantly longer execution times than the other, such heuristic rules can be rather simple, for instance comparing the presence of loops, the number of routine calls or the number of statements in each branch. However, the accuracy of this method has not yet been investigated.

Static analysis may by used to identify explicit dependencies between tasks, i.e. communcation, with a high degree of automation by generating the function call graph (the reachable functions) of the service and searching it for calls

to OS routines. Each matching routine call will result in an entry in a list, containing the path of control leading to the externally visible event, in order to store the context. This is a highly interesting approach, as most reverse engineering tools can generate call-graphs and the only thing that needs to be known by an analysis tool is the call-graph and the names of the OS routines of interest. However, the call-graph should preferably contain not only routine calls, but also the selections, especially "switch" statements, as they are commonly used in dispatchers in multi-service tasks, where each "case" corresponds to a service. The call graph must therefore show what function calls are associated to what services in order to allow modeling of individual services.

Using these approaches for dependency identification the modeler can with little effort obtain lists of both important state variables as well as interactions between different tasks and services, which significantly facilitate the modeling of the system.

4.3 Modeling the Environment

The process presented in Section 4.2 targets modeling the behavior of complex embedded systems. However, in order to construct a complete analyzable model, the system model, and the environmental stimuli must be modeled as well, i.e. external events that effect the behavior of the system. Examples of environmental stimuli are:

- commands from a human operator,
- interaction with other computer systems,
- interaction with subsystem not included in model,
- interrupts caused by e.g. network traffic or I/O signals, or
- variations in input values from sensors.

By modeling the environment as well as the system behavior, a closed system is obtained from which an analysis can be performed using e.g. discrete event simulation.

Information on environmental stimuli is typically collected using dynamic analysis, i.e. by recording the events corresponding to environmental stimulus on the real system and extracting the necessary information from the recorded

execution traces. The information that is of interest when recording environmental stimuli is the inter-arrival time probability distribution of the different events that are recorded.

The environmental stimuli are modeled using *environment tasks*, which interact with the tasks in the behavior model. The environment tasks are included in an analysis of the model in the same way as "real" tasks are, but do not consume any CPU time and can therefore be safely excluded from the analysis output if desired. Environment tasks have higher scheduling priority than the behavior model tasks, allowing them to preempt all behavior model tasks at any given point in time, as the environment tasks corresponds to truly concurrent external events. The corresponding reaction to the stimuli is described in the behavior model and may thus be delayed due to the task scheduling, in the same way as in a real system.

If the modeled system is interacting with external computer systems for which the implementation is available, e.g. another system developed in-house, that system does not have to be modeled as an environment tasks, but can be modeled in detail, as described in Section 4.2, and integrated in the behavior model. This, however, requires that the modeling language and analysis tools support distributed systems, i.e. systems with more than one CPU, which is not always the case. This is especially valuable if the external system is complex and it is desirable in order to describe sequences of interaction between the two systems in detail.

4.3.1 Identification and Classification of stimuli

The environment affects the system through well-defined interfaces, which are identified based on documentation, code studies and discussions with system experts and documented in the model specification (discussed in Section 4.2.2). By introducing code instrumentation in the environmental interfaces, dynamic analysis can be used in the same way as when recording the systems behavior, as discussed in Chapter 3. The information of interest is the inter-arrival times (rate) of the different environmental events, i.e. minimum-, maximum- and average inter-arrival time but also the inter-arrival time probability distribution, in order to allow for probabilistic modeling.

Environmental stimuli can be classified into two categories, as follows:

- *Common stimuli* - stimuli occurring in any situation and configuration, e.g. interrupts caused by network traffic. The common stimuli are described, typically in a probabilistic manner, in the *common stimuli model*.
- *Specific stimuli* - stimuli specific for a particular situation, i.e. a test case. The specific stimuli is described in the *specific stimuli model*, typically in a more detailed manner compared to the common stimuli model, i.e. as a timed sequence of events.

The motivation for separating the two types of stimuli in different models is to make the complete analyzable model modular, allowing the use of a single common stimuli model together with several different specific stimuli models. The specific stimuli models can be automatically generated from execution traces recorded from different systems configurations and situations.

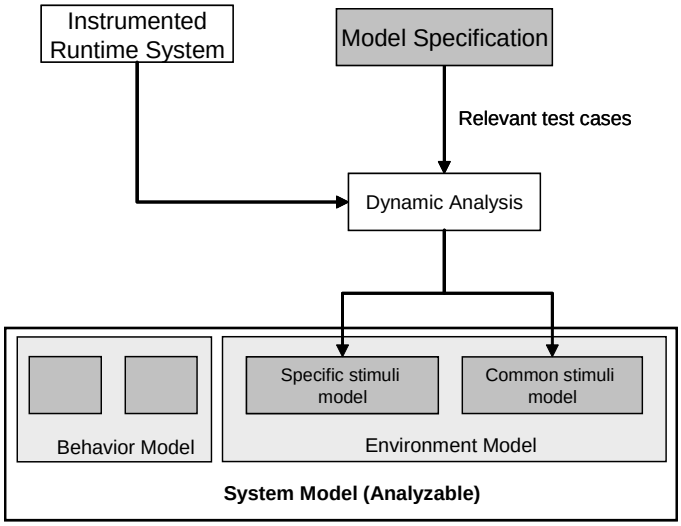

Figure 4.11: Modeling environmental stimuli

As depicted in Figure 4.11, both the common stimuli model and the specific stimuli model are based on dynamic analysis, i.e. recordings made on an instrumented system. The environment interfaces, found in the model specification, are instrumented in order to register a time-stamp and an identifier

for each time an environmental event occurs. It is also important to identify relevant test cases for the dynamic analysis, in order to execute realistic scenarios. Many different test cases can typically be found in system documentation (test documentation) but in order to facilitate the selection of relevant test cases, where the specific environment interfaces are frequently used, it is recommended to consult system experts.

Stimuli is classified based on the situations in which it occurs. By studying the recordings of environmental stimuli in different situations a certain stimuli can be classified either as common stimuli or specific stimuli, depending on if the stimuli is found in all recordings or only in subset of the recordings. If the stimuli occurs in all recordings, it should be included in the common stimuli model. Otherwise, the stimuli should be included in the specific stimuli models corresponding to the test cases executed when recording the stimuli.

present in all recordings is modeled as common stimuli, while stimuli that only occur in a subset of the test cases (recordings) should be modeled in the specific stimuli models corresponding to the test-case. The specific stimuli models should include information on what test-case the modeled stimuli represent. This way, an analysis tool can attach this information to the analysis output, thereby avoiding possible mix-ups of analysis results based on different test-cases.

4.3.2 Modeling Approaches for Environment Models

Given that recordings have been made providing the necessary information on inter-arrival times of different types of environmental stimuli, the information needs to be included in the environment models. One option is to use the same solution as for modeling execution times and probabilities in the behavior model, i.e. separate sets of model parameters for each environment model, which contains the inter-arrival time information.

However, since the behavior of the environment tasks are typically trivial, they may be automatically generated based on the recordings. It is therefore not motivated to have two sub-models for each environment model, the inter-arrival time data may be integrated in the environment tasks. An ART-ML implementation of a typical environment task is described in Figure 4.12. The example depicts an extension of ART-ML in the form of the keyword “EnvTask”, which declares an environmental task.
This environment task is activated every p1 time units and sends a message to a task in the behavior model, TaskX, containing the value “1”. After a delay of d1 time units, another message is sent to TaskX, containing the value “0”.

Specific Stimuli Model

```
EnvTask SignalX
   Trigger period p1
Behavior
{
   send(TaskX, 1);
   delay(d1);
   send(TaskX, 0);
}
```

Figure 4.12: An environment task in ART-ML

This may e.g. correspond to when an I/O interface of system receives a short pulse on a “digital in”. It is assumed that TaskX is modeled in adequate detail in order to receive and react to these the messages.

In the same way as e.g. execution times are modeled in the behavior model, inter-arrival times and delays such as p1 and d1 may be modeled in several different ways. Section 4.2.4 that discusses the model parameters, i.e. the quantitative information on the systems temporal behavior, mentions basically two methods for probabilistic modeling of the recorded information. Firstly, fitting the inter-arrival time data to a theoretical distribution or secondly, using values from a list of observed values, i.e. an empirical distribution. Either approach may also be used for environment models. For further information on the use of theoretical standard distributions the reader is referred to Section 4.2.4. Using *list-based* environment models allow a more detailed analysis, instead of using sampling in the analysis, the list of observed inter-arrival times are used in the same order as they where registered, in order to replay recorded stimuli exactly as it was observed. This is very suitable for modeling the specific stimuli. Constructing list-based environment models is straight-forward, it is only a matter of identifying the specific stimuli, as described earlier, and extract lists of the inter-arrival times of the different types of specific stimuli. It may also be of interest to use worst case environment models, where the minimum inter-arrival times are used all the times. Constructing worst-case environment models is similar, but instead of storing all observed inter-arrival times as lists in the environment models, only the shortest inter-arrival times are stored, as they typically correspond to the worst-case.

4.4 Discussion

This chapter has further motivated and explained the approach of this thesis by presenting how analyzable behavior models may be utilized in maintenance of complex embedded system. Together with Chapter 3 the chapter answers the sub-question Q1 stated in the introduction of the thesis:

Q1: What methods are suitable for extracting the information necessary for a temporal behavior model from a complex embedded system implementation containing millions of lines of code?

A modeling process has been presented targeting complex embedded systems, specifying how to obtain and structure the necessary information into a set of submodels, a functional model, which is typically constructed through reverse engineering of the system implementation, and three other components containing different kinds of quantitative information, the model parameters, describing the execution times, inter-arrival times and probabilities used in the behavior model, and two environmental models, describing inter-arrival times of common and situation-specific environmental stimuli. All components apart from the functional model may be automatically generated using dynamic analysis. To automatically generate also the functional model is important, challenging future work.

The modeling process emphasizes the development and use of a model specification serving as a specification for the modeling effort and focus. Each task is modeled using different levels of abstraction depending on the tasks importance for the properties of interest for analysis. For tasks that are to be modeled in detail, the important services and the state variables affecting these services are identified and modeled.

Models constructed using this modeling approach should be of high quality due to the well-defined modeling process adapted for the domain. However, to ensure that a constructed model is a valid description of the system, a model validation activity is necessary, where model validation techniques are used in order to increase the confidence of the constructed models. This is further discussed in Chapter 5, which focus on model validation and model validation techniques.

Chapter 5

Model Validity

Since a model is by definition an abstraction of a real system, a model can not precisely predict the behavior of a complex system in all situations. This is an inevitable consequence of the higher level of abstraction of models in comparison to the corresponding implementation. The higher level of abstraction is, however, a desired property of a model. It improves understandability and allows for easy prototyping and analysis. Even though a model is not a perfect description of a system, it can be sufficiently detailed and accurate in order to allow for accurate predictions to be made with a high degree of confidence.

How to determine if a model is sufficiently detailed and accurate, i.e. *valid*, is not trivial. A valid model is not "perfect" but an analysis of the model should nevertheless give predictions that are "good enough". The validity of a model is investigated in an activity known as *model validation*.

Model validation has been defined as "*substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model*" [SCG+79]. Thus, a model can not be shown valid in general, only for a specific use. According to the definition of model validation, a model can only be validated given that the following has been defined:

- **The domain of applicability** specifies the system that is described by the model. For a model of the temporal behavior of a complex embedded system, this includes versions and setup of software, as well as hardware.
- **The required accuracy** is dependent on the properties of interest and the situation. For instance, if the model is used for studying the response

times of software functions without hard real-time requirements but with requirements on user-perceived performance, i.e. typical response times, it may be sufficient with, e.g. 10 % accuracy in the predictions of the model, since the consequences of a minor error in the predictions are low and can be easily verified after implementation. In other situations, if the model is used to predict properties critical for correct system operation, such as a critical response time or the utilization of a limited logical resource, a much higher accuracy is required since the consequences of an error in the prediction may result in a system failure and it may be difficult to verify the analysis results after implementation. The worst case scenario predicted by the model may be hard to test on the real system, and it is difficult to verify if the the predicted worst case scenario is the actual worst case scenario.

- **The intended application of the model** considered is, as discussed in Chapter 4, primarily behavior impact analysis with respect to typical changes of the system. Another application of the model is to serve as documentation.

This chapter proposes a five-step process for validation of temporal behavior models of complex embedded systems. The model validation process utilizes the tools presented in Chapter 3 and the behavior impact analysis presented in Section 4.1. Section 5.1 provides a discussion of the potential threats against the validity of a model. Section 5.2 presents the proposed validation process, consisting of five tests of the model. Section 5.3 discusses the fourth test in the validation process in greater depth, the observable property equivalence test. Section 5.4 discusses model robustness and presents the fifth and final test in the validation process, the sensitivity analysis, which is a test of model robustness. Finally, Section 5.5 concludes the chapter and relates its contributions to the research questions of this thesis stated in Section 1.2.

5.1 Validity Threats

The need for model validation emerges from the risk of making decisions based on a model that contains errors or lacks information about important details of the system's behavior. The process of constructing a model of a software system consists of several different activities and errors could be introduced in any of them. There are at least five potential error sources:

- the understanding of the system,
- the understanding of modeling language and tools,
- the observations of the system,
- the probe effect, and
- the level of abstraction.

The understanding of the system and domain In order to develop a valid model, it is important that the person or persons that construct the model, the *modeler*, understands the system's software architecture and how it works in general, i.e. the roles, responsibilities, and dependencies between the subsystems and the external systems. It is also important that the modeler understands the typical use of the system, i.e. the domain in which it is used, and the resulting requirements. This is addressed by the modeling process proposed in Chapter 4 through the development and review of a model specification. However, missing or ambiguous information in the model specification may result in model errors.

The understanding of modeling language and tools The modelers must have adequate knowledge about the different tools used for modeling and analysis as well as the semantics of the modeling language. To avoid misunderstandings or misinterpretations, the tools and modeling language must be well documented and communicated. It is important to document both the grammar of the languages as well as the exact semantic meaning of the different primitives. Moreover, it is also important to document more trivial issues, such as the time unit used in the model (ms, μs or ns?). Such information may be obvious for an expert, but may be confusing for developers that less knowledge of the details of the system's temporal behavior.

The observations of the system When constructing a model based on the observations of a system's behavior, it is important that the observations are made in several different, but representative, situations in order to ensure that as much as possible of the behavior of the system is captured. For instance, it is likely that a system that is exposed to large amounts of stimuli from its environment behaves differently from a system that is in its idle mode. Thus, if a model is based on recordings from a single system environment solely, it may not be valid for other environments. The concepts of system environment and environment models are discussed in Section 4.3.

The probe effect If software probes are used to make recordings on which a model is constructed, and then the software probes are removed, the behavior of the system may be effected in such as way that the model is no longer valid. As discussed in Section 3.2.1, the impact of software probes is commonly referred to as the probe effect [Sch91]. In this thesis it is assumed that the probe effect can be avoided by allowing the probes to remain in the system. However, this may not be possible for some systems due to the cost of these probes, i.e. CPU and memory usage. Another solution to avoid the probe effect is to use specialized hardware monitors that non-intrusively observe the system without effecting the temporal behavior of the system [Sho02]. This is however not always an option, since custom hardware is required.

The level of abstraction If information about important details of the system's behavior is missing, the model will be less accurate, and the validity of the model may also be more sensitive to changes as fewer dependencies between tasks, state variables, and other system component are modeled. Missing dependencies may prevent that a change in one part of a system propagates to other parts of the model in the same way as it does in the real system. As a result, the updated model may behave differently compared to the corresponding updated system.

5.2 A Model Validation Process

This section presents a five-step process for validation of behavior models of complex embedded systems. The process utilizes the tools described in Section 3.3. Each step in the process is a test that either fails the model, or allows the model to pass to the next test in the validation process. The individual tests in this process have been previously proposed in research literature, e.g. [LM01, Sar99], but not in this specific context.

It is important to bear in mind that the purpose of model validation is not to show the validity of a model. This is is not possible in the same way as it is not possible to show the absence of errors in a software implementation through testing. Each test is only a single sample from a huge set of possible behavior. The purpose of the model validation is, in the same way as software testing, to attempt to show that the model is incorrect. The more tests performed that fail to show that the model is incorrect, the more confidence in the model.

The proposed model validation process starts with less powerful but also less time-consuming tests, which allows for quick discovery of any major errors in the model. The more powerful tests in the later steps of the process are only used when the model has passed the previous tests, thus avoiding the use of overly powerful and time-consuming tests early in the validation process.

1. **Trace comparison** An execution trace from an analysis of the model is visualized and subjectively compared with a corresponding visualization of an execution trace recorded from the real system. The visualization is accomplished by presenting the recorded execution traces graphically over time, using e.g. the Tracealyzer tool presented in Section 3.3.3. The purpose of this test is to determine if there are major errors in the model or if the execution traces from the model are reasonably similar to the execution traces from the real system. The trace comparison test is discussed in Section 5.2.1.

2. **Property comparison** The second test compares execution traces in a more detailed manner. Execution traces from a model analysis and from a real system recording are visualized with respect to a set of properties of the temporal behavior, the *comparison properties*, such as interarrival-time and response-time distributions. Each property results in two visualizations, one representing the model and the other representing the real system. The vizualisations are subjectively compared in order to identify major differences in specific properties. The property comparison test is discussed further in Section 5.2.2.

3. **Analysis variability** Since an analysis (simulation) of a probabilistic model corresponds to random samples from a very large set of possible behavior, the values predicted by the model will differ between analyses. In this test several independent replications are made of the model analysis in order to study the amount of variability in the analysis output, i.e. the predicted values of different properties. If the analysis variability is too high the model is failed. The analysis variability can be reduced by making the model "less probabilistic" or use longer simulation runs. Analysis variability is further discussed in Section 5.2.3.

4. **Observable property equivalence** The test of observable property equivalence is a detailed numerical comparison between the predictions from the model and the behavior observed on the real system, with respect to concrete statistical measures of the comparison properties used previously, in the property comparison test. Minor errors in the model that reflects in the comparison properties are identified by this test and fail the test if the differences exceed specified limits. This test identifies any minor errors in the model, which may not be apparent in visualizations. The test of observable property equivalence is discussed in detail in a dedicated Section 5.3.

5. **Sensitivity analysis** The accuracy indicated by step 4 is not a sufficient measure of model validity if the model is to be used for behavior impact analysis or in other ways used to prototype changes. The model need to be *robust* with respect to typical changes, meaning that if the model is exposed to a certain change, the impact of the change should correspond to the impact caused by the same change on the real system. The sensitivity analysis investigates whether or not the model is robust with respect to common types of changes to the system, *change scenarios*. This is accomplished by performing a set of behavior impact analyses of different change scenarios, where an expected result is known from experiments with the real system. Model robustness and the sensitivity analysis test is discussed in a dedicated Section 5.4.

Before the validation process can be initiated it is important to select at least one system environment on which the tests in the model validation process can be based, the *validation environment(s)*. An environment specifies e.g. what test cases that are used to stimulate the system and the amount of disturbances in the form of interrupts, caused by network traffic or I/O events, as discussed

in Section 4.3. A validation environment specifies the environment models as well as the corresponding setup of the real system.

Preferably, more than one validation environment should be used to better compare the system and the model, since a model that is valid in one environment may not be valid in other environments. Unfortunately, since the effort of performing the test is linearly proportional to the number of validation environments used, only a limited amount of validation environments can be used in order to keep the required effort on a realistic level. It is therefore important to select the validation environments with care.

The validation environments should stimulate the model in many different ways in order to compare as much as possible of the model behavior with the corresponding behavior of the real system. Since only a limited amount of validation environments can be used they should differ as much as possible from each other in order to compare the model with the real system in a variety of situations. At least one validation environment should correspond to the presumed worst-case system stimuli that may occur in any realistic situation, but it is also important to use validation environments corresponding to the normal use of the system, i.e. different common scenarios, including when the system is idle.

The selected validation environments are used in all steps of the process. Each test is performed once for each validation environment, and if a test fails for any of the validation environments, the model validation is terminated in order to debug the model. When the model has been adjusted, the validation process is restarted from step 1.

5.2.1 The Trace Comparison Test

The first step in the process is trace comparison, i.e. visualization and comparison of execution traces. This may be performed by using the Tracealyzer tool presented in Section 3.3.3. Two instances of the Tracealyzer is started, one displaying the execution trace from the model, and the other one displaying the execution trace from an analysis of the model.

When comparing the traces, it is important to note that the traces are samples of a very large set of possible behaviors. Even though the validation environment has been specified, the model is still an abstraction of the real system, modeled in a probabilistic manner. Hence, an exact match can not be expected. However, it should be possible to identify patterns in the task execution depicted by the two traces. If the execution pattern of a task that has been predicted by the model differs considerably from the observation, the model will

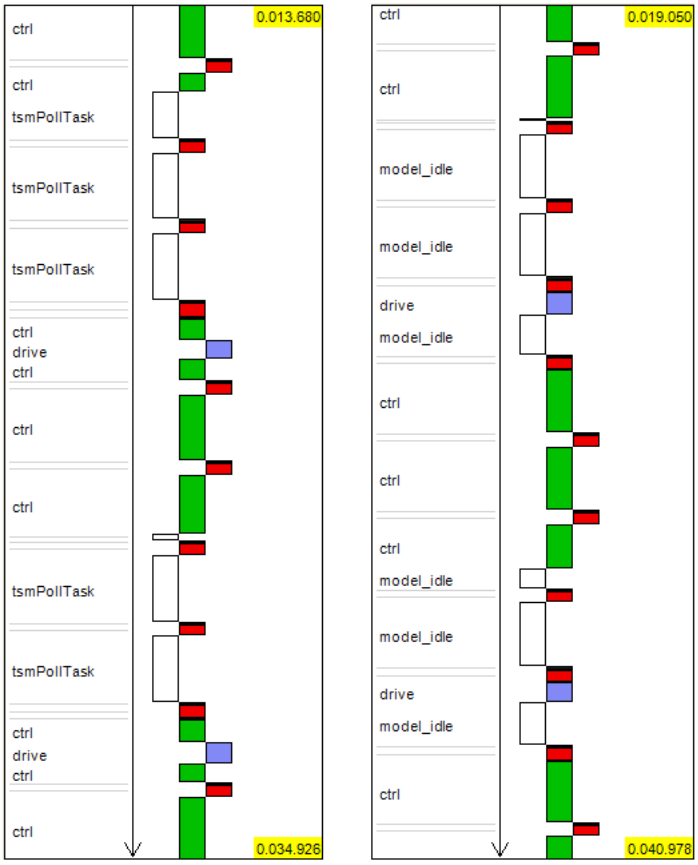

Figure 5.1: Trace Comparison using the Tracealyzer tool

fail the test.

An example is depicted in Figure 5.1, where two execution traces are compared side-by-side, one from an analysis of the model (on the right) and the other recorded on the corresponding real system. In the real system, the task *Drive* always preempts the task *Ctrl*, but in the model this is not the case. This is a typical example of an execution pattern which may also be used as a comparison property in later tests in the process, such as the observable property equivalence test which will be discussed in Section 5.3.

As depicted by Figure 5.1, the *Drive* task has a matching inter-arrival time (periodicity) and execution time, but it has the wrong offset compared to *Ctrl*; it is therefore executed too early.

In general, it is likely that the first versions of a model are failed by this test, but when the model becomes more refined, more demanding tests are required.

5.2.2 The Property Comparison Test

The second step in the validation process is the property comparison test. In this test specific properties of the observed system behavior and the corresponding predictions from the model are visualized and compared subjectively. This test has been discussed in e.g. [Sar99], where it was referred to as the operational graphics test.

This test is stronger than trace comparison, as it apart from a set of validation environments also requires selecting a set of properties to compare, the *comparison properties*. This is very similar to the behavior impact analysis presented in Chapter 4 and the regression analysis presented in Chapter 3. In model validation, two execution traces are compared with respect to a set of properties, in the same manner as behavior impact analysis and regression analysis. However, in these analyses the execution traces are of the same origin, i.e. both are recorded from a real system (regression analysis) or both are from model analyses (behavior impact analysis). In model validation, execution traces from the two different origins are compared, one from an analysis based on a model and one recorded from a real system.

The definition of comparison properties is a very important part of the validation process, since the comparison properties are used in all later steps of the process. For each validation environment, all comparison properties are to be visualized and compared. Suitable properties to compare in this test are response time distributions (an example is depicted in Figure 5.2), and utilization of logical resources over time (see Figure 5.3). These properties are affected by many different tasks and are, consequently, sensitive to a large set of possible

differences between the model and the real system. These properties may be presented in a scatter-plot, with the X-axis as a time-line and the Y-axis showing the corresponding value, i.e. response-time of each instance of the task or the utilization of the resource.

Since execution traces exists from the previous step, trace comparison, assuming that the generation of the visualizations are automated the main effort in this test is then the visual comparisons of each property for each validation environment. The amount of comparisons required may be significant since it is the product of the number of environments and the number of properties to compare. If 5 environments are used for the model validation and 20 properties are to be compared, a total of 200 visualizations are generated, resulting in 100 comparisons. However, if each comparison takes on average 1 minute, this takes less than 2 hours for a single person to perform

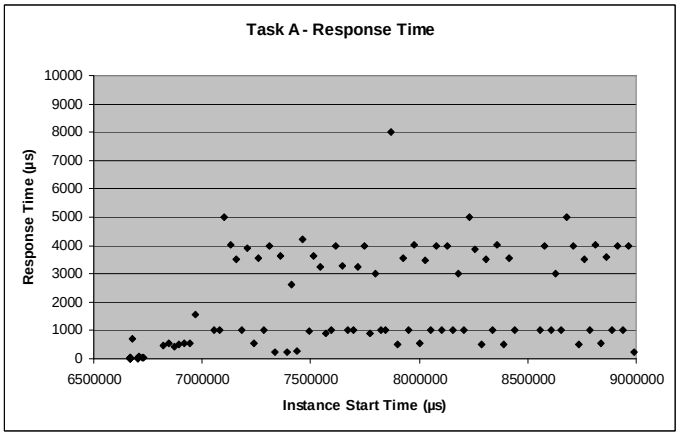

Figure 5.2: Visualization of the usage of a task response time

It is important to understand that the purpose of this test is to look for major differences only. In most cases there will be small differences even if the model is of good quality. However, to determine if these differences are small enough is done in a more systematic and objective way later in the validation process. Property comparison is a quick method of identifying the major errors at an early stage in the validation process, prior to more time-consuming testing.

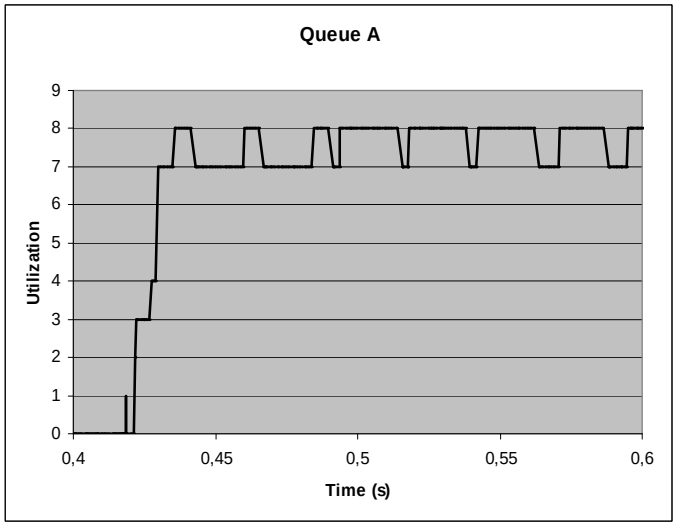

Figure 5.3: Visualization of the usage of a logical resource

5.2.3 The Analysis Variability Test

The third step in the validation process, the analysis variability test, is important when using probabilistic modeling since an analysis (simulation) of a probabilistic model may generate different results from time to time. The reason for this is that a probabilistic simulation corresponds to a random sample from a large set of possible behavior. If the amount of variability in the analysis results is large, this implies that the results from a single analysis of the model may not be representative for the system behavior. The results of such an analysis are not incorrect in the sense that the behavior predicted by the model may occur in the real system, but there are other situations in which the system may behave differently. Thus, such results are of low confidence. To increase the confidence level of the analysis results, the analysis variability can be reduced by basing the predictions on multiple or longer execution traces.

The analysis variability test does not rely on visualization. By making several independent replications of an analysis and calculating the statistic measures of the comparison properties, e.g. average response times, it is possible to use standard statistical methods when calculating the amount of variability for each statistic measure. A model passes the analysis variability test if the

amount of variability is low, e.g. below 1 %, for all comparison properties in all validation environments.

The statistical measures of the comparison properties can be formulated using the Probabilistic Property Language, PPL, and calculated using the Property Evaluation Tool, as presented in Section 3.3.2. The analysis variability test is mentioned in [Sar99] as the internal validity test.

5.3 Observable Property Equivalence

This section presents the fourth test of the model validation process proposed in Section 5.2, the observable property equivalence test. This test is a detailed numerical comparison of specific properties of the temporal behavior, with respect to two execution traces. This enables a model and the corresponding implemented system to be compared in a more objective and detailed manner, compared to the property comparison test, which relies on subjective comparison of visualizations.

The test of observable property equivalence is made with respect to the comparison properties defined in step 2 of the validation process, property comparison, and in all validation environments. This test is potentially very demanding for the model, if many comparison properties is used. Any minor errors in the model that reflects in the comparison properties are pointed out by this test, and depending on the desired tolerance, the model is either failed or passes the test.

The observable property equivalence test may, in the same way as the Analysis Variability test, utilize the Property Evaluation Tool and the previously defined PPL specification of the comparison properties. However, additional information, apart from the comparison properties and the set of validation environments, are required in order to perform the observable property equivalence test: the tolerance to use in the different comparisons, e.g. the prediction should be within 1 % of the observations. This tolerance is necessary in order to compensate for the analysis variability. A model passes this test if it is observable property equivalent to the real system, i.e. all predictions of the comparison properties are within the specified tolerances. A formal definition of observable property equivalence is given in Section 5.3.2.

5.3.1 Comparing Behavior

As presented in Section 5.2, the two first steps of the validation process rely on a subjective comparison of visualizations of the behavior of the model and the real system. In order to test the model validity in more detailed, accurate and objective way, a numerical comparison is necessary.

Since a model is per definition an abstraction of the system, an analysis of a model can not predict the behavior of the real system precisely in all situations. Hence, it makes little sense to compare the predicted behavior with the observed behavior directly, event by event, especially if the model is probabilistic. As an example consider Figure 5.4, which depicts the predicted and real response times of a task. Each dot represents the response time of an instance, or execution, of the task where the Y-axis is the response time and the X-axis is the time when the instance started.

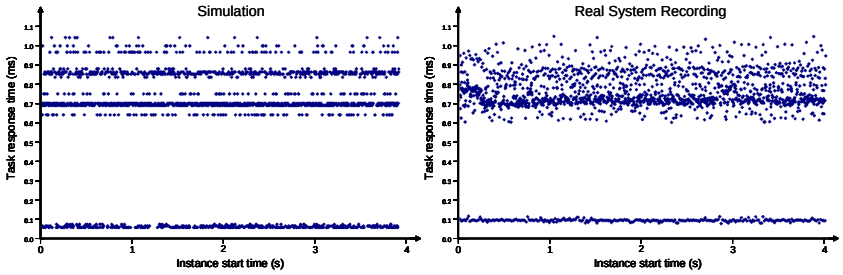

Figure 5.4: Response Time Distribution of task C - Simulation vs. Real System Recording

The temporal behavior predicted by the model resembles the behavior observed on the real system. Distinct classes of response times can be identified in the observed and the predicted behavior and these match very well. However, it is not possible to compare these two data sets task instance by task instance.

Obviously, an exact match of the execution traces is a too strict criterion of equivalence for probabilistic models. Instead, the execution traces have to be compared on a higher level of abstraction by using the comparison properties defined in step 2 of the validation process, e.g. task response times. The comparison properties used for visualizations in step 2 typically correspond to distributions, so in order to allow for simple comparison of two execution traces, it is necessary to use statistical measures describing these distributions,

e.g. mean value, the median, various quantiles etc., in the same way as in the analysis variability test. Each statistical measure corresponds to a single numerical value which may easily be compared to the corresponding value extracted from another execution trace.

5.3.2 Observable Property Equivalence – A Formal Definition

This section gives a formal definition of observable property equivalence. In previous works, e.g. [AWN04b], this was presented as an equivalence relation, but since the relation of observable property equivalence is not transitive, this is not true. The relation expresses similarity, but not equivalence.

A model and a corresponding system are observable property equivalent if they are equivalent with respect to a set of comparison properties, i.e. statistical measures of the observed temporal behavior. However, as discussed earlier, since the model is an abstraction of the system, it is necessary to allow a certain amount of tolerance in comparison.

In Definition 1 we formalize the observation of a system, x, that is either the real implemented system, executing on the real hardware, or a model of a system executed in a simulator. The resulting recording is a list of time-stamped events related to tasks-switches and operations on logical resources. The environment e specifies the configuration of the system and any external stimuli of the system, as discussed in Section 4.3.

Definition 1. *R = Rec(x, e, d)*
The function Rec returns a recording, R, of the execution of x, in the environment e, with the duration d time units. R is a list of events, where each event contains a time-stamp, an event type and generic data, where the semantics are specific for each event type. □

Definition 2 presents the function Eval, which evaluates a system property p with respect to the recording R.

Definition 2. *v = Eval(p, R)*
The function Eval evaluates the property p with respect to the recording R. The result, v, is a decimal value. If the property p is a boolean expression, v is either 1 (true) or 0 (false). □

Since a certain amount of tolerance is often necessary in the comparison, we introduce a function which expressing the tolerance allowed for a specific comparison property,

Definition 3. *t = Tol(p)*
The function Tol returns the maximum allowed difference between two evaluations of the same property p on two different recordings. The return value, t, is a decimal value. If the property p is of boolean type, the function returns a tolerance of 0. □

Definition 4 presents the definition of observable property equivalence. If evaluations of all comparison properties with respect to the model results in values sufficiently close to the values from the real system recording, the model and the system are observable property equivalent.

Definition 4. *Given that P is the set of comparison properties, M is a model of the system S, E_M is the environment model of M and E_S is the environment of the system S, iff*

$$\forall p \in P : Abs(Eval(p, Rec(M, E_M, d)) - Eval(p, Rec(S, E_S, d))) \leq Tol(p)$$

then S ≡ M, i.e. S and M are observable property equivalent with respect to P, in the specific environment.

□

Obviously, this relation of similarity relies heavily on the comparison properties and tolerances used. It is important to select a suitable set of comparison properties in order to compare as much as possible of the behavior of the model with the corresponding system. A discussion regarding the selection of comparison properties is therefore provided in Section 5.3.3.

5.3.3 Selecting Comparison Properties

The observable property equivalence test depends on a set of comparison properties. If sufficient comparison properties have been used and the comparison has been made with little tolerance, any model that passes the observable property equivalence test should be highly accurate. If too few (relevant) comparison properties is used, the comparison is of low value. Typically, as many comparison properties as possible should be used, in the same way as when defining test cases for software. However, in the same way as when defining test cases for software, it is only possible to use a limited amount of test cases. The selection of comparison properties is therefore crucial for this test as well as other tests in this process.

In order to allow for numerical comparison of two execution traces, the comparison properties used are statistical measures of the recorded data, in the same way as in the analysis variability test, i.e. the third step of the validation process. Each statistical measure corresponds to a single numerical value which may easily be compared with the corresponding value extracted from another execution trace. Examples of suitable statistical measures to use as comparison properties are:

- The maximum task response time
- The average task response time
- Different quantiles for task response times
- The average task interarrival time

Typically, for each comparison property used in step 2 of the validation process, the property comparison test, a set of statistical measures are formulated. These statistical measures correspond to concrete comparison properties which may be specified as PPL queries and evaluated using the Property Evaluation Tool.

The comparison properties typically includes explicitly defined system requirements and other system properties of interest for analysis but may also include system properties that are of less interest when analyzing the model, but required in order to increase the coverage of the comparison. We refer to these extra properties as *supporting properties*. These supporting properties are typically effected by many aspects of the system and characterize the temporal behavior. Typical supporting properties are average task interarrival times and response time properties that are not explicit requirements.

Selecting the appropriate system properties for the comparison is very important in order to achieve a valid comparison. As many system properties as practically possible should be included in the set of comparison properties in order to get a high confidence level in the comparison. However, the use of irrelevant comparison properties may result in the rejection of a valid model. In [Sar99] this is denoted a *Type I error*, or the *model builder's risk*. The opposite situation, i.e. an erroneous model is accepted as valid, may occur if too few relevant comparison properties are used or if the model has not been sufficiently analyzed in order to detect the erroneous behavior. In [Sar99] this is denoted a *Type II error*, or the *model user's risk*.

Even if a large set of system properties are used for a comparison there is a risk of accepting an invalid model, e.g. if they represent too few types

of system properties, For instance, imagine that only response-time properties are used as comparison properties. The rate of a task could in that case differ between the system and the model without being discovered in the comparison. If system properties related to patterns in the scheduling had been used as well, this would have been discovered. Thus, the selected system properties should not only be relevant, but also represent a variety of aspects of the temporal behavior.

We have identified three general types of comparison properties that are suitable for comparison of the temporal behavior of complex embedded systems:

- response-time properties,
- pattern properties, and
- resource utilization properties.

Response-time properties The response time of tasks can be used as a comparison property, since it is dependant on not only the execution time of the task, but it also depends on the temporal behavior of other tasks. The response time may be interesting in terms of worst case, since it might be a requirement (a deadline), but also the distribution of response times can be used as a supporting property, as it contains a significant amount of information about the temporal behavior of the system.

Pattern properties It is often possible to identify patterns in the scheduling of tasks and in the occurrence of different internal events. A system property of this type can, for instance, be that a certain fraction of the instances of task A are preempted by task B. The occurrence of a certain pattern in the execution time of a task is also a pattern property that can be used for comparison.

Resource utilization properties Properties in this category include those related to logical resources, such as the minimum or maximum utilization of message queues, how long a task waits for a message, or how often a task writes or reads messages from the buffer. Another example of such a property is the probability of a certain message buffer being empty (or full).

5.4 Model Robustness

A model is *robust* with respect to a change in the implementation of the system if the change, when applied to the model, effects the predictions based on the model in the same way as it effects the observed behavior of the system. If a model is robust, it implies that the relevant behaviors of the system are indeed captured by the model at an appropriate level of abstraction. In this section we propose a method for determining the robustness of a behavior model of a complex embedded system. This activity is referred to as a *sensitivity analysis*.

To exemplify the importance of model robustness, consider a system containing a binary semaphore protecting a shared resource. A time-out occurs if a task has been waiting for the semaphore for a certain predefined time. If the time-out occurs, the task is activated as usual, but executes longer than normal due to the necessary error handling. In all previous versions of the system, this time-out has never occurred. If the time-out is left out when constructing the timing model of the system the model still seems accurate since the time-out never occurs.

However, as a result from changing the system, e.g. increasing the execution time of another task, the time-out will in some cases occur. Since the time-out was not captured in the model the system's behavior will now differ from the predicted behavior.

Our approach to sensitivity analysis is influenced by *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 upon the system identification approach is somewhat related to testing. Typically, output signals are predicted by using the model which are then compared with the output signals of the physical process. Hence, the model is regarded as correct if the analysis and the physical processes generate approximately the same output, when fed with the same input.

Testing the model with different input signals and comparing the prediction with the signals produced by the actual system is acceptable given that 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 have a discontinuous nature, meaning that the behavior may change dramatically as a result of small changes in the system. A model of a software system can therefore quickly become invalid as the system evolves, if the model is not robust with respect to typical changes. By analyzing the

impact on the system caused by different changes, it is possible to determine if the model is sensitive to such changes, i.e. less robust.

5.4.1 Sensitivity Analysis

In this section, we will present how to analyze the robustness of a model using a sensitivity analysis. The basic idea is to test different alterations and verify that they effect the behavior predicted by the model in the same way as they effect the observed behavior of the system. First a set of *change scenarios* has to be selected. The change scenarios should be representative for the probable changes that the system may undergo. Typical examples of change scenarios are:

- to change the execution times distribution of a task or service,
- to introduce or remove new services in existing tasks,
- to change the usage of logical resources.

The selection of change scenario requires experienced engineers that can perform educated guesses about relevant and probable changes. It is also valuable to study the documentation of previous changes to the system, i.e. change logs, in order to identify different types of common changes.

Given that a set of N changes scenarios have been defined, the next step is to construct a set of N systems variants $\{S_1, ..., S_N\}$ and a set of corresponding models $\{M_1, ..., M_N\}$ by applying the change scenarios on the original versions of the system and model.

Note that applying the change scenarios to the system does not require real implementations of new features, i.e. functional improvements of the system. The sole purpose of the necessary changes is to reflect the impact on the temporal behavior caused by the change scenarios, for instance by adding an empty loop that increases the execution time of a specific task. These changes are therefore easy to implement. The model variants are constructed in a similar way, by applying the N change scenarios to the original model.

Each model variant is then compared to its corresponding system variant by investigating if they are observable property equivalent as defined in Definition 3, Section 5.3. If all variants are equivalent, including the original model and system, the model is robust with respect to the change scenarios. Formally we define the robustness test, sensitivity analysis, as follows:

Definition 5. *A model M is robust with respect to a system implementation S and a set of change scenarios C, iff:*

$$\forall c \in C : S_c \equiv M_c$$

where S_c and M_c corresponds to the implementation of the change scenario c on the system S and model M respectivly. □

As an example, consider a sensitivity analysis consisting of a single validation environment and a single change scenario: an overall increase in the execution time of task Y by 100 μs. The increase in execution time is implemented in the real system by e.g. an empty loop tuned to execute for 100 μs. A corresponding ART-ML model is changed by adding an execute-statement to the task, specifying 100 μs additional execution-time consumption.

The next step is to perform recordings of the modified system version in the selected validation environment (test case etc) and an analysis of the modified model using the appropriate environment model. The recording of the real system is compared to the analysis output with respect to the comparison properties, which, in this case, should include at a minimum the average response times of task Y. If the model is robust with respect to this change scenario there should not be any statistically significant discrepancies in this comparison, assuming that the model was sufficiently accurate prior to the sensitivity analysis. The general sensitivity analysis process is illustrated by Figure 5.5. This process is performed for each validation environment.

A sensitivity analysis can be regarded as a behavior impact analysis, where the expected result is known from recordings of the prototype implementations. Since change scenarios are rather abstract descriptions of changes, they are representative for a large set of concrete changes of the specified type. For instance, the change scenario "*increases the execution time of task X with 100 μs in all executions*" is representative for a large set of changes to internal computations in the task which results in a similar increase in average execution time.

It is therefore not necessary to perform the sensitivity analysis every time the model has been updated and is to be validated. It is sufficient if a sensitivity analysis is performed on the initial model of system, after major changes of the model, or if new change scenarios are identified. A sensitivity analysis is also necessary if details are removed from the model, i.e. the level of abstraction is increased. Thus, a sensitivity analysis is valid as long as changes are made to the model that can be considered equivalent to one of the change scenarios used in the sensitivity analysis.

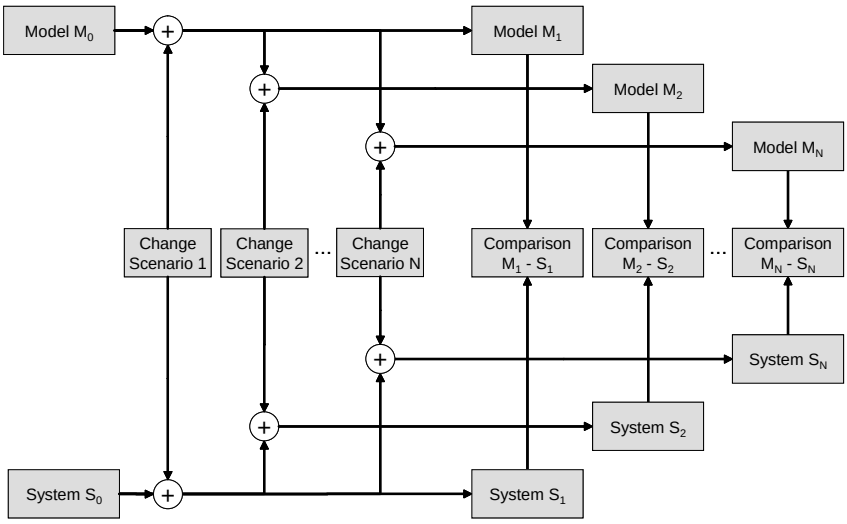

Figure 5.5: The sensitivity analysis

A sensitivity analysis typically represents a significant effort. If e is the number of validation environments, c is the number of change scenarios, and p is the number of concrete comparison properties, the number of numerical comparisons required in a sensitivity analysis is $e \times c \times p$.

If using PPL, described in Chapter 3, for specifying the concrete comparison properties, the Property Evaluation Tool, PET, can be used to evaluate and compare a batch of PPL queries in a single job. However, in the current implementation this is possible only with respect to two execution traces at a time. Therefore, $e \times c$ runs of PET are necessary, where each run compares two execution traces with respect to p comparison properties, one execution trace from a model variant and one execution trace from a system variant. Thus, the comparison of execution traces is relatively simple. The time-consuming part is the generation of the execution traces for comparison. To run simulations of the model is relatively swift, to make a small change of the model and run an extensive simulation typically takes 1-2 minutes according to our experience.

However, to record the behavior of the real system variants is a more time consuming activity. Complex embedded systems often takes considerable time to compile and start up, so each execution trace often takes 20 minutes to generate. Since $e \times c$ sets of system variants and corresponding models have been

defined, $e \times c$ recordings are therefore necessary. Thus, if three validation environments and five change scenarios have been defined, recording the execution traces takes at least 5 hours ($3 \times 5 \times 20$ minutes), while the simulations are approximately 10 times quicker.

5.5 Discussion

In this chapter we have proposed an approach for validation of models describing the temporal behavior of complex embedded systems. This approach consists of a five-step model validation process targeting temporal behavior models of complex embedded systems.

The proposed validation process answers the second sub-question (Q2) of the research questions stated in Section 1.2.

Q2: *What methods are suitable for validating models describing the temporal behavior of complex embedded systems?*

The process consists of five steps, increasingly demanding tests of model validity. These tests are performed with respect to a set of validation environments and a set of comparison properties. Different types of suitable comparison properties have been described.

The proposed validation process does not only consider the similarity between the model and the current version of the system, but also contains a sensitivity analysis, a method for evaluation of the robustness of a model based upon a set of change scenarios.

The individual tests have been previously proposed in research literature, but in other contexts. Even though there are other methods available for model validation, these five methods are especially suitable for validation of temporal behavior models.

Chapter 6

Conclusions

Early work preceding this thesis [WAN+03a, AN02] presented the modeling language ART-ML, a simulator for ART-ML models and an industrial case study which showed the viability of the approach. Later work [WAN03b] presented PPL, the Probabilistic Property Language, allowing the specification of probabilistic properties of interest for analysis. This thesis further contributes by presenting a solution to a fundamental problem of this approach:

***Q**: How can models be developed that accurately describe the temporal behavior of complex embedded systems?*

The main research question, Q, was broken down into two subquestions, Q1 and Q2. By providing answers for these subquestions, the main question Q is thereby answered.

***Q1**: What methods are suitable for extracting the information necessary for a temporal behavior model from a complex embedded system implementation containing millions of lines of code?*

The thesis has proposed a framework that divides a temporal behavior model into four components:

- The functional model, a set of behavior descriptions (imperative programs) describing the behavior of the tasks and services in the system.
- The model parameters, a set of parameters of the functional model nec-

essary for describing a systems temporal behavior, e.g. execution times.

- The specific stimuli model, describing environmental stimuli associated to certain scenarios, e.g. a particular test case.
- The common stimuli model, describing environmental stimuli not specific to a particular scenario, but commonly occurring.

This thesis proposes a modeling process where the functional model is obtained through reverse engineering of the systems implementation and the other three components, containing quantitative information of the systems temporal behavior, are obtained through dynamic analysis, i.e. analysis of recordings of the systems temporal behavior. This modeling process, which consists of both of dynamic analysis and reverse engineering, is proposed as an answer for Q1.

Given that a temporal behavior model has been developed, it is necessary to assure that the model is valid, i.e. accurately describes the systems temporal behavior at an appropriate level of abstraction. The matter of how to assure the validity of temporal behavior models is the second sub-question Q2:

***Q2**: What methods are suitable for validating models describing the temporal behavior of complex embedded systems?*

This thesis proposes an answer for Q2 in the form of a process for model validation consisting of a series of increasingly demanding tests of the model: trace comparison, property comparison, analysis variability, observable property comparison and, finally, sensitivity analysis. These tests have been previously proposed in research literature, but not in the context of validation of temporal behavior models. Other tests of model validity have been proposed in the literature, but the five tests proposed in Chapter 5 have been found especially suitable for validation of temporal behavior models.

To support the solutions proposed, a set of three tools have been developed and are presented in the thesis:

- The Tracealyzer, a tool for visualization of execution traces. The tool graphically presents the task execution together with the values of generic probes over time. The tool supports the model construction process by allowing the modeler to better understand the temporal behavior of the system and also supports the first step of the proposed validation process, the trace comparison.

- The Property Evaluation Tool (PET), a tool for analysis and comparison of execution traces with respect to a set of PPL queries. PET allows for the behavior impact analysis presented in Chapter 4 and may also be used in the three latter steps of the model validation process presented in Chapter 5, analysis variability, observable property equivalence and sensitivity analysis. PET is the first implementation of the PPL language proposed in [WAN03b]. The implemented version of PPL has been extended in comparison to the original specification, as presented in Appendix B.

- A software behavior recorder for the commercial RTOS VxWorks, allowing for the recording of execution traces for the above mentioned tools. The overall design of the recorder is presented together with an evaluation of its impact on system performance.

These tools have been introduced in the software development at ABB Robotics, a world leading developer of industrial robots and robot control systems. The software behavior recorder has been integrated in their robot control system and is activate by default, also in the released versions of the system. This allows developers at ABB Robotics to use the Tracealyzer for debugging as well as general system understanding and PET for more systematic analysis of the systems temporal behavior. Another use of a dynamic analysis tool such as PET is what we refer to as regression analysis, i.e. to compare recordings from the latest version of the system with recordings of a previous system version. This way, it is possible to automatically identify undesired impacts on the temporal behavior caused by recent changes. Regression analysis was presented in Chapter 3. The regression analysis is being introduced gradually at ABB Robotics, in an initial phase for a single subsystem.

6.1 Future Work

This thesis is based on experiences from complex embedded systems development and on earlier case studies, but the solutions proposed in this thesis have not yet been validated in an industrial context. Plans for future studies therefore include a modeling case study, where the practical usability of the modeling and model validation processes proposed in this thesis are to be evaluated by applying them on a real complex embedded system. The goal of the study would be to perform realistic behavior impact analyses using the developed model and confirming the accuracy of the analysis by implementing the

analyzed change scenarios and to make recordings of the updated system in order to identify the real impact. Apart from the modeling case study required for validation of the solutions proposed in this thesis, many ideas for future research exist. The most relevant are:

- Automated modeling
- Alternative analysis methods
- Regression analysis case-study

We intend to look into all three areas, which are further described below.

6.1.1 Automated modeling

The effort of constructing a temporal behavior model is the weakest link of the behavior impact analysis approach. By automating the construction of models as much as possible, the effort required for constructing and validating the necessary model can be significantly reduced.

The model framework proposed in this thesis consists of four components, where three components, the model parameters and the two stimuli models, can be automatically generated by using dynamic analysis. The functional model can be extracted from the source code using a special purpose reverse engineering tool that we plan to implement. This tool would extract and analyze the function call-graph of each individual service in a task in order to identify activations of other tasks. This is accomplished by comparing the names in the function call-graph with a list of names corresponding to common OS services, such as IPC communication. For each match, the path of control is stored and used to construct a rough model, an abstraction of the real implementation focusing on selections and calls to OS routines. Only function calls that encapsulate a call to an OS routine are included in the model. Thus, if a service does not use any OS routines, the service will only be modeled with respect to its execution time. Even though the tool recognizes and models selections, it will not attempt to analyze the conditions of the selections, but instead model the selection in a probabilistic manner. The statistical information required for probabilistic modeling of selections is obtained through dynamic analysis, i.e. recording the outcomes of the selections during execution of the system. The tool automatically inserts the necessary code instrumentation while analyzing the source code.

In order to obtain more detailed models, it is possible to manually analyze the conditions of the modeled selections and improve the model manually.

However, a solution must be found to support evolving systems, where the automatic model synthesis needs to be repeated after each change of the system. If the model has been improved manually after synthesis, a new synthesis will require a migration of these improvements to the new model. In order to avoid the need for manual migration, an automatic solution is necessary. In a further perspective, program analysis techniques such as program slicing [Tip95] may be used to automatically model the conditions of selections, which effectively solves the problem with migration of manual improvements, as high quality models, with a minimum of probabilistic selections, could then be generated automatically.

6.1.2 Alternative Analysis Methods

This thesis has assumed the use of random simulation as analysis method. There are however other analysis methods based on simulation. Future work therefore includes investigating the use of a multi-stage simulation approach, in many ways similar to genetic algorithms. Compared to using random simulation only, this analysis method is expected to better identify the possible worst-case scenarios of a model. The first step in this multi-stage simulation process is to run a large series of random simulations, in order to identify a set of interesting states on which to focus further simulations. These interesting states correspond to situations with extreme values in a specified property of interest, e.g. a task's response time. By storing the simulation state preceding these situations, a large amount of random simulations can be performed, starting from this state. This way, the analysis is focused on investigating scenarios likely to be "close" to a worst case scenario. This process may be repeated several times in order to further focus the analysis, which effectively identifies a scenario resulting in an extreme value. There is however a risk that this value may only be a local maximum, and that there are other scenarios, completely different, that result in even higher values, e.g. response times.

6.1.3 Regression Analysis Case Study

Given that regression analysis (Section 3.1.3) is introduced at ABB Robotics, relevant research areas include the verification of the expected effect, i.e. reduced maintenance cost for the company. This can be investigated in the form of a case study. This would be a *holistic single case study* [Yin03], motivated by the belief that ABB Robotics represents a typical case, i.e. a typical example of a company developing a complex dependable software system.

Question and Propositions The question in focus of this case study is how the introduction of regression analysis will effect the quality of the software within the robot control system. The *unit of analysis* [Yin03] is the introduction of the analysis method. We believe that, as a result of the new analysis tool, more potential problems related to timing and resource usage will be reported by system developers during the 12 months after the introduction of regression analysis, compared to the 12 months preceding the introduction of this analysis method. Due to the increase in potential problems discovered, there should be a decrease in the number of errors reported from late system testing and end users.

Collecting the Data ABB Robotics has an extensive database containing suggestions for improvements and error reports from system developers, testers and end users. The reported errors are often well documented; for each error reported, there is a description of the error and what actions that have been proposed. The developer(s) that are assigned to the error report usually extends the error report further and add a description of how the problem was solved.

By searching this database, it is possible to adequately estimate the number of errors related to timing or resource usage. A single source of evidence is however not sufficient. Potential problems that are discovered are often not reported formally, and will therefore not be in the database. In order to accommodate for this weakness, the case study will also contain interviews with developers. This may also help to explain error reports found in the database as well as to give an estimate on how often the analysis tools are used, the developers' opinion on the usability of the tools and hopefully also examples on when the analysis tools have reported potential problems.

Appendix A

ART-ML 2.0

ART-ML is a modeling language developed for describing the temporal behavior of complex software systems with real-time requirements. An ART-ML model consists of a set of tasks communicating through common OS services such as message boxes (IPC) and semaphores. An ART-ML task consists of two parts, a set of attributes, such as scheduling priority, and a behavioral description which is an abstraction of the corresponding task in the real system, describing both temporal and functional behavior.

This appendix describes ART-ML version 2.0. Compared to the ART-ML version 1.0 [AN02, Wal03, WAN+03a] the differences are:

- It is based on ANSI C and therefore allows all constructions of ANSI C embedded in the model.
- A new statement "pswitch" for probabilistic selection, replacing "chance"
- The execute statement has been extended to accept references to distributions of any type, declared elsewhere in the model.

Models in ART-ML 2.0 are intended for analysis in the ART-ML 2.0 simulation environment, which is under development. An analysis of a model in ART-ML 2.0 consists of three stages: translating the model to a pure ANSI C program, compiling and linking with the ART-ML 2.0 C-library and finally executing the resulting executable file, which produces an output in the form of an execution trace. This process is fully automated. The resulting execution trace may be inspected using the Tracealyzer tool or analyzed using PET, both presented in Chapter 3 of this thesis.

Elements of ART-ML 2.0

The new version of ART-ML contains the following elements:

C-Block

A C-block is a piece of ANSI C code encapsulated in the ART-ML model. A c-block is declared using the keyword CBLOCK and ended by the keyword END. C-blocks are global and declarations inside a CBLOCK are visible from all tasks in the model.

Message box

A message box is a FIFO buffer storing messages between tasks. A message box is declared using MESSAGEBOX keyword, followed by name and maximum size of the FIFO buffer. An ART-ML task may put messages in the message box using the sendMessage library routine and fetch messages using the recvMessage library routine.

Semaphore

An ART-ML semaphore is a classic Djikstra binary semaphore, providing mutual exclusion between tasks. A semaphore is declared using the SEMAPHORE keyword, followed by the name of the semaphore. A semaphore is locked using the sem_wait library routine (corresponding to Djikstra's P) and released using sem_post routine (corresponding to Djikstra's V).

Task

ART-ML tasks define the behavior of the system. A task consists of three parts, its name, its attributes and its behavior. The attributes are scheduling priority and task activation strategy. The attributes can be one or more of the following among.

- TASK TYPE:
  - PERIODIC,
  - ONESHOT,
  - SPORADIC

- PERIOD: (value) – The periodicity of the task (if periodic)
- DISTR: (identifier) – A reference to a specific inter-arrival time distribution declared in the model. The inter-arrival time distributions are declared in the same way as execution time distributions.
- OFFSET: (value) Optional offset of periodic or one-shot tasks.
- PRIORITY: (value) The priority of the task (between 0-255) where 0 is the best priority in the system

The behavior is described using a special C-block, identified using the keyword BEHAVIOR, that immediately following the attributes. The BEHAVIOR C-block is ended using the keyword END, which also ends the task declaration. When transforming the ART-ML model into ANSI C, the BEHAVIOR C-blocks are transformed into an ANSI C functions, the code is therefore subject to the same rules as code in the body of an ANSI C function. Variables and types declared in a BEHAVIOR C-block are therefore not accessible outside the task.

Syntax of language elements

Message box - declaration

```
MESSAGEBOX name size;
```

Example:
MESSAGEBOX MBOX9 5;

Message box - sending

```
int sendMessage(MBOX mbox, int msg, int timeout);
```

Example:

```
result = sendMessage(MBOX9, REQUEST7, 1000);
if (result == TIMEOUT)
{
    /* time out*/
    ...
}
```

Message box - receiving

```
int recvMessage(MBOX mbox, int timeout);
```

Example:

```
result = recvMessage(MBOX9, FOREVER);
switch (result)
{
    case REQUEST1: ...
                   break;
    case REQUEST2: ...
                   break;
}
```

Semaphore - Declaring

```
SEMAPHORE name;
```

Example:

```
SEMAPHORE sem7;
```

Semaphore - Locking

```
int sem_wait(SEMAPHORE sem, int timeout);
```

Example:

```
result = sem_wait(sem7, 10000);

if (result == TIMEOUT)
{
    /* failed locking the semaphore */
    ...
}
```

Semaphore - Releasing

```
void sem_post(SEMAPHORE sem);
```

Example:

```
sem_post(sem7);
```

Inline C declarations

```
CBLOCK
    /* c-code */
END
```

Example:

```
CBLOCK
    int sys_online = 0;
    const int CODE_5_MSGS_AVAILABLE = 123;
    const int CODE_GENERIC_DATA = 125;
END
```

Declaring a task

```
TASK name
    Attribute0: value0
    ...
    AttributeN: valueN
BEHAVIOR
    /* c-code*/
    ...
END
```

Example:

```
TASK SYSTEM
    TASK_TYPE: ONESHOT
    PRIORITY: 0
BEHAVIOR
    sleep(40000);
    sys_online = 1;
END
```

An ART-ML 2.0 model

```
CBLOCK
    #include "modelparameters.h"
    /* where execution time distributions */
    /* are defined (S1, S2, C1, C2...)    */

    int sys_online = 0;
    const int CODE_5_MSGS_AVAILABLE = 123;
    const int CODE_NET_COMMAND = 124;
    const int CODE_GENERIC_DATA = 125;
END

/* messagebox for sensor data */
MBOX CTRLDATAQ 5;
MBOX CTRLCMDQ 4;

/* This task produces data by reading a hardware sensor */
TASK SENSOR
    TASK_TYPE: PERIODIC
    PERIOD: 2000
    PRIORITY: 1
BEHAVIOR
    static int msg_counter = 0;

    execute(S1);

    if(sendMessage(CTRLDATAQ, CODE_GENERIC_DATA, 0) > -1)
    {
        msg_counter = msg_counter + 1;
        if (msg_counter == 5)
        {
            execute(S2);
            sendMessage(CTRLCMDQ,
             CODE_5_MSGS_AVAILABLE,
             FOREVER);
            msg_counter = 0;
        }
    }
END
```

```
TASK CTRL
    TASK_TYPE: ONE_SHOT
    OFFSET: 10000
    PRIORITY: 1
BEHAVIOR

    execute(C1);

    while( forever )
    {
        if(recvMessage(CTRLCMDQ, forever) > -1)
        {
            int i;
            for(i = 0; i < 5; i++)
            {
                recvMessage(CTRLDATAQ, forever) > -1)
                execute(C2);
            }
            execute(C3);

        }else{

            //error
            execute(C4);
        }
    }
END
```

ART-ML Library routines

The following ART-ML specific routines are available in an ART-ML task:

int *sendMessage*(MBOX mbox, int msg, int timeout)
The message msg is sent to the messagebox mbox. If mbox is full, the sending waits for an empty slot for the duration specified in timeout. If timeout is FOREVER, it waits forever, if it is specified to 0, it immediately aborts if there is no empty slot in the messagebox. If a timeout occurs, the return code is TIMEOUT, otherwise OK. The message is a single 32-bit integer value. Negative values are not allowed, they are used for error codes.

int *recvMessage*(MBOX mbox, int timeout)
A message is received from the messagebox mbox. If mbox is empty, the task is blocked until a message arrives or the timeout occurs. If timeout is specified as FOREVER (-1), no timeout will occur. If timeout is specified to 0, the task is not blocked by an empty mbox, but immediately timeouts if no message is available.

int *sem_wait*(SEMAPHORE sem, int timeout)
This routine attempts to lock a semaphore for a specified amount of time. If the semaphore is already locked by another task, the task is blocked until it is allowed to lock the semaphore or the timeout occur. If the semaphore was locked, the return code is OK, otherwise, if a timeout occurs, the return code is TIMEOUT. If timeout is specified as 0, the timeout will immediately occur if the semaphore was already locked. If the timeout is specified to FOREVER, sem_wait will never timeout.

int *sem_post*(SEMAPHORE sem)
A previously locked semaphore is unlocked. If other tasks are waiting to lock the semaphore, they will be made ready to execute. There are two return codes: if the semaphore is already unlocked or is locked by another task, the return code is ERROR, otherwise OK.

void *delay*(int time)
A call to this routine puts the task to sleep for *time* time units. After time time units the task is resumed and put in the ready-state.

Modeling execution time

The execute statement is used to model the execution time of code, i.e. the consumption of CPU time. Depending on the selected level of abstraction when constructing the model, an execute statement can represent a whole task or a smaller section of code. The execute statement takes an execution time distribution as parameter from which it samples a value, the amount of CPU time to consume. The consumption of time corresponds to advancing the simulation clock, which drives the simulation forwards. During this the duration of an execute statement the task may be preempted by other tasks. In that case, the execute statement remembers the amount of execution time left to consume, and continues consuming the remaining CPU time when the task is again allowed to execute.

The discrete execution time distributions used in the previous version of ART-ML will still be supported by ART-ML 2.0, e.g.:

```
execute( (10, 1000), (90,1300) );
```

In the above presented example, the probability of selecting an execution time of 1000 time units in 10 % and the probability of 1300 is 90 %. ART-ML 2.0 will also allow specifying identifiers instead of immediate data, e.g.

```
execute( C1 );
```

where the identifier refers to a distribution of any type declared elsewhere in the model. Distributions that are planned to be supported include the Normal distribution, the Uniform distribution and the Weibull distribution. It will also be possible to use empiric distribution, i.e. raw data from measurements.

Probabilistic selection

ART-ML 2.0 allows for probabilistic selection through the "pswitch" statement – probabilistic switch. This statement replaces the chance-statement in the original version of ART-ML.

The syntax of pswitch is as follows:

```
pswitch{
  p(p1): statement;
         statement;
         statement;
         ...
         break;

  p(p2): ...
         break;

         ...

  p(pn): ...
         break;

  default:
         statement;
         ...
         break;
}
```

The pswitch is similar to the well-known "switch" statement in ANSI C, apart from that the selection is probabilistic. An arbitrary number of labels are allowed where each label has a specified probability in the form of a floating point value in the range $[0..1]$. The sum of the probabilities has to be equal to, or below, 1.0. If the sum of the n probabilities is below 1.0, an extra label may is required, "default", which receives the remaining probability of $1 - (p_1 + p_2 + \ldots + p_n)$.
Even though pswitch is not the only probabilistic element in ART-ML 2.0 (there is also sporadic tasks and the execute statement), it is the only way of specifying probabilistic selection between behaviors.

Appendix B

PPL Implementation

The Probabilistic Property Language, PPL, was first proposed in [WAN03b]. In comparison to the original specification, the implemented version of PPL contains many extensions and some differences. The most important are:

- A second argument has been added to the P operator, the quantifier.
- A new operator, following, has been added, which returns the next instance of the specified task, that follows the activation of the particular task instance.
- The statistical functions min, max, avg and median have been added, which returns statistical measures of a task property, e.g. response time.
- The function subset has been added, which allows the task instances matching a condition to be exported to a text file.
- Message queues are handled differently, as generic probes instead of queues. A generic probe may monitor any logical resource.

These extensions are described in Chapter 3 as well as in the tool documentation, available at the project website:

`http://www.idt.mdh.se/~jxn01/projects/remodel`

A tool for PPL analysis of execution traces, PET (Property Evaluation Tool), can also be found there, together with the Tracealyzer (for execution trace visualization). Currently, the tools are available for Microsoft Windows only.

The grammar of PPL

This section presents the grammar of the implemented version of the Probabilistic Property Language, PPL, in Bachus Naur Form (BNF).

```
<query>     ::= <property> ";" <query>
            |   <property>

<property>  ::= <value> <relop> <value>
            |   <function>
            |   subset "(" <arg> ")" ">" FILENAME

<value>     ::= "P" "(" ID "(" ID ")" "," <cond> ")"
            |   "P" "(" "*" "," <cond> ")"
            |   PROB
            |   <unbounded>

<cond>      ::= <expr> <moreexpr>
            |   <expr>

<moreexpr>  ::= <logop> <expr> <moreexpr>
            |   <logop> <expr>

<expr>      ::= <exp> <relop> <exp>
            |   <exp> <relop> <unbounded>
            |   NOT "(" <cond> ")"
            |   "(" <cond> ")"

<exp>       ::= <term> <moreterms>
            |   <term>

<moreterms> ::= + <term> <moreterms>
            |   - <term> <moreterms>
            |   + <term>
            |   - <term>

<term>      ::= <factor> <morefactors>
            |   <factor>

<morefactors> ::= * <factor> <morefactors>
              |   / <factor> <morefactors>
              |   * <factor>
              |   / <factor>

<factor>      ::= "(" <exp> ")"
              |   abs "(" <exp> ")"
              |   <function>
              |   CONST
              |   <task>
              |   "*" "." probe NUM
              |   - <factor>

<function>    ::= min "(" <arg> ")"
              |   max "(" <arg> ")"
              |   avg "(" <arg> ")"
              |   median "(" <arg> ")"
```

```
<arg>          ::= ID "." <data member>
               |   ID "(" ID ")" "." <data member>
               |   ID "(" ID ")" "." <data member> "," <expr>
               |   "*" "." probe NUM
               |   "*" "." probe NUM "," <expr>

<unbounded>    ::= ID

<task>         ::= ID "(" <instance> ")" "." <data member>
               |   ID "(" <following> ")" "." <data member>

<instance>     ::= ID
               |   ID + <num>
               |   ID - <num>

<num>          ::= "[" NUM ".." NUM "]"
               |   "[" - NUM ".." NUM "]"
               |   "[" - NUM ".." - NUM "]"
               |   NUM

<following>    ::= following "(" ID "(" <instance> ")" ")"
               |   following "(" ID "(" <instance> ")" ")" + <num>
               |   following "(" ID "(" <instance> ")" ")" - <num>

<data member>  ::= start
               | end
               | resp
               | exec
               | probe NUM

<relop>        ::= <
               |   >
               |   <=
               |   >=
               |   =

<logop>        ::= AND
               |   OR

PROB           ::=  x : x ∈ ℝ AND 0 ≤ x ≤ 1
CONST          ::=  x : x ∈ ℝ
NUM            ::=  x : x ∈ ℤ

ID             ::= LETTER(DIGIT|LETTER|'_')*
FILENAME       ::= '"'ID('.'ID)*'"'
```

Appendix C

An example model specification

The system in focus is a control system for industrial robots, developed by ABB Robotics. This system was initially designed in the beginning of the nineties and has been maintained and further developed over 10 years by a staff consisting of about 150 software developers. In essence, the robot controller has an object-oriented design, but implemented in C. It consists of approximately 2500 KLOC distributed in 400-500 classes, in turn organized in a set of subsystems. The controller uses the real-time operating system VxWorks, from WindRiver [WRW]. The hardware platform is an industrial PC using high-end Intel processors. The controller consists of three computers: the axis computer, a DSP which controls the motors of the robot, the I/O computer, and the main computer, the most complex part and the focus of the modeling study. The software system in the main computer consists of more than 60 tasks, which are scheduled using preemptive fixed priority scheduling. The tasks communicate through message queues and shared data areas. Many tasks consist of several services, sometimes over 200, which are activated by messages from other tasks or by a timer. A task typically spends most of the time blocked, waiting for incoming messages. When a message arrives, or a timer expires, the task executes the service corresponding to the event occurred. During the execution of a service, any incoming messages are buffered and later processed in a FIFO manner.

A critical part of the main computer is the motion control subsystem, which is responsible for generating the motor references and brake signals required

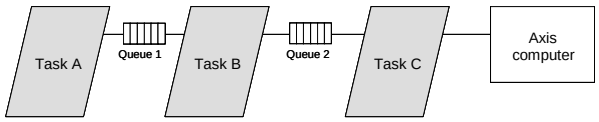

Figure C.1: The Motion Control Subsystem

by the axis computer. The axis node sends requests to the main computer with a fixed, high rate, over 200 Hz. The axis computer expects a reply in the form of motor references within a certain time. This depends on three tasks on the main computer, in the motion control subsystem. We refer to these tasks as A, B and C. The tasks B and C have high priority and run frequently with a fixed period. The task A executes mostly in the beginning of each robot movement and has lower priority, but produces data required by the B task. The B task processes the data and forwards it in smaller parts to the Ctask, which makes the final processing and sends motor references to the axis computer. The data passed between C and B, and between B and C are passed through data queues, as depicted in Figure 4. If any of the queues become empty while the robot is under control, the C task cannot deliver any references to the axis node. This state is considered as a system failure, and the robot halts. The queues may only be empty if the robot has applied its brakes and stopped controlling the motors. The interface of the subsystem in the task A, which receives orders to move the robot and other commands from a client, in a way determined by the application program which varies between different uses of the system. Since the tasks B and C have the highest scheduling priority in the system, they may only be disturbed by interrupts. Task C has however a mid priority and there are tasks other than B and C which may disturb task A. The focus of this modeling study is the motion control subsystem, and the properties of interest are if any of the two queues can become empty while the robot is active, and in that case, in which situation this may occur.

Bibliography

[AB93] R. S. Arnold and S. A. Bohner. Impact Analysis - Towards a Framework for Comparison. In *Proceedings of the Conference on Software Maintenance (ICSM '93)*, pages 292–301. IEEE Computer Society, 1993.

[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.

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

[ABRT93] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, pages 284–292, 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.

[ACD93] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AN02] J. Andersson and J. Neander. Timing Analysis of a Robot Controller. Master's thesis, Mälardalen University, Västerås, Sweden, 2002.

[AWN04a] J. Andersson, A. Wall, and C. Norström. Decreasing Maintenance Costs by Introducing Formal Analysis of Real-Time Behavior in Industrial Settings. In *Proceedings of the First International Symposium on Leveraging Applications of Formal Methods (ISoLA '04)*, 2004.

[AWN04b] J. Andersson, A. Wall, and C. Norström. Validating Timing Models of Complex Real-Time Systems. In *Proceedings of the Fourth Conference on Software Engineering and Research Practice in Sweden (SERPS '04)*, 2004.

[Bal90] O. Balci. Guidlines for Successful Simulation Studies. In *Proceedings of the 1990 Winter Simulation Conference*. Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 2061-0106, U.S.A., 1990.

[BCP02] G. Bernat, A. Colin, and S. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd IEEE International Real-Time Systems Symposium (RTSS '02), Austin, TX, USA*, 2002.

[BCP03] G. Bernat, A. Colin, and S. Petters. pWCET: a Tool for Probabilistic Worst Case Execution Time Analysis of Real-Time Systems. Technical Report YCS353, University of York, Department of Computer Science, United Kingdom, 2003.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. In *In proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04), LNCS 3185*, 2004.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer-Verlag, 1998.

[BG97] B. Bellay and H. Gall. A Comparison of Four Reverse Engineering Tools. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '97)*, page 2, Washington, DC, USA, 1997. IEEE Computer Society.

[BLL+95] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software (SPIN '01)*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[BTMG02] R. I. Bull, A. Trevors, A. Malton, and M. W. Godfrey. Semantic grep: Regular expressions + relational abstraction. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, 2002.

[But97] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. ISBN: 0-7923-9994-3. Kluwer Academic Publisher, 1997.

[CDH+00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 439–448, 2000.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CFV99] A. Cimitile, A. R. Fasolino, and G. Visaggio. A Software Model for Impact Analysis: A Validation Experiment. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE '99)*, 1999.

[CGP02] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 431–441, New York, NY, USA, 2002. ACM Press.

[CI90] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 66, Washington, DC, USA, 1995. IEEE Computer Society.

[DY00] A. David and W. Yi. Modelling and analysis of a commercial field bus protocol. In *Proc. of 12th Euromicro Conference on Real-Time Systems*, pages 165–172. IEEE Computer Society Press, 2000.

[EH84] E. A Emerson and J. Y. Halpern. Sometimes and Not Never Revisited: on Branching Versus Linear Time. Technical report, University of Texas at Austin, Austin, TX, USA, 1984.

[ENE] ENEA website, http://www.enea.com.

[HJ94] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[HJMS03] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN '03)*, *LNCS 2648*, 2003.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[Hol00] G. J. Holzmann. Logic Verification of ANSI-C Code with SPIN. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 131–147, London, UK, 2000. Springer-Verlag.

[Hol03] G.J. Holzmann. *The SPIN MODEL CHECKER - Primer and Reference Manual*. ISBN: 0-321-22862-6. Pearson Education, Addison-Wesley, Inc, 2003.

[HS99] G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proceedings of the 21st international conference on Software engineering (ICSE '99)*, pages 597–607, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[Jen98] P. K. Jensen. Automated Modeling of Real-Time Implementation. Technical Report BRICS RS-98-51, University of Aalborg, December 1998.

[Jen01] P. K. Jensen. *Reliable Real-Time Applications. And How to Use Tests to Model and Understand*. PhD thesis, Aalborg University, February 2001.

[Joh93] R. Johansson. *System Modeling Identification*. ISBN: 0-13-482308-7. Prentice-Hall, 1993.

[Kat98] J. Katoen. Concepts, algorithms and tools for model checking, lecture notes of the course mechanised validation of parallel systems, friedrich-alexander university at erlangen-nurnberg, 1998.

[KRO] Kronos website, http://www-verimag.imag.fr/temporise/kronos.

[KSS+02] R. Kollman, P. Selonen, E. Stroulia, T. Syst, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE '02)*, page 22, Washington, DC, USA, 2002. IEEE Computer Society.

[Leh90] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS '90)*, pages 201–212, December 1990.

[LK93] A. M. Law and W. D. Kelton. *Simulation, Modeling and Analysis*. ISBN: 0-07-116537-1. McGraw-Hill, 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.

[LM01] A. M. Law and M. G. McComas. How to Build Valid and Credible Simulation Models. In *Proceedings of the 2001 Winter Simulation Conference*. Averill M. Law and Associates,Inc., P.O. Box 40996, Tucson, AZ 85717, U.S.A., 2001.

[MH89] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

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

[MW03] A. Marburger and B. Westfechtel. Tools for understanding the behavior of telecommunication systems. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 430–441, Washington, DC, USA, 2003. IEEE Computer Society.

[MWN+04] G. Mustapic, A. Wall, C. Norström, I. Crnkovic, K. Sandström, J. Fröberg, and J. Andersson. Real World Influences on Software Architecture - Interviews with Industrial Experts. In IEEE, editor, *Proceedings of IEEE Working Conferance on Software Architectures (WICSA '04), Oslo, Norway*. IEEE, 6 2004.

[NIS02] The Economic Impacts of Inadequate Infrastructure for Software Testing, Planning Report 02-3, Prepared by RTI for the U.S. National Institute of Standards and Technology, 2002.

[OSE] OSE website, http://www.ose.com.

[PFGJ02] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A Lexical Pattern Matcher for Architecture Recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE '02)*, pages 170–178, 2002.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*, 1977.

[QVWM94] J. P. Queille, J. F. Voidrot, N. Wilde, and M. Munro. The Impact Analysis Task in Software Maintenance: a Model and a Case Study. In *Proceedings of International Conference Software Maintenance (ICSM '94)*, pages 234–242, 1994.

[RSW] Rapita systems website, http://www.rapitasystems.com.

[Sar99] R. G. Sargent. Validation and Verification of Simulation Models. In *Proceedings of the 1999 Winter Simulation Conference*. Department of Electrical Engineering and Computer Science, College of Engineering and Computer Science, Syracuse University, Syracuse, NY 13244, U.S.A., 1999.

[SB94] M. Spuri and G. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Proceedings of the 15th IEEE Real-Time System Symposium (RTSS'94)*, pages 2–21, 1994.

[SCG+79] S. Schlesinger, R. E. Crosbie, R. E. Gagne, G. S. Innis, C. S. Lalwani, and J. Loch et al. Terminology for Model Credibility. *Simulation*, 32(3):103–104, 1979.

[Sch91] W. Schutz. On the Testability of Distributed Real-Time Systems. In *Proceedings of the 10th Symposium on Reliable Distributed Systems, Pisa, Italy*. Institut f. Techn. Informatik, Technical University of Vienna, A-1040, Austria, 1991.

[Sho02] M. El Shobaki. On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications*. IEEE, March 2002.

[SK98] T. Systä and K. Koskimies. Extracting State Diagrams from Legacy Systems. In *Proceedings of the ECOOP Workshops on Object-Oriented Technology (ECOOP '97)*, pages 272–273, London, UK, 1998. Springer-Verlag.

[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.

[SPI] ON-THE-FLY, LTL MODEL CHECKING with SPIN, spin website, http://spinroot.org.

[TC94] K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

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

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[UPP] Uppaal website, http://www.uppaal.com.

[vdBKV97] M. G. J. van den Brand, P. Klint, and C. Verhoef. Reverse Engineering and System Renovation: an Annotated Bibliography. *SIGSOFT Software Engineering Notes*, 22(1):57–68, 1997.

[Wal03] A. Wall. *Architectural Modeling and Analysis of Complex Real-Time Systems*. PhD thesis, Mälardalen University, Sweden, 2003.

[WAN+03a] 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 of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA'03)*. Department of Computer Science and Engineering, Mälardalen University, P.O. Box 883, S-721 23 Västerås, Sweden, 2003.

[WAN03b] A. Wall, J. Andersson, and C. Norström. Probabilistic Simulation-based Analysis of Complex Real-time Systems. In *Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time distributed Computing*. Department of Computer Science and Engineering, Mälardalen University, P.O. Box 883, S-721 23 Västerås, Sweden, 2003.

[Wei81] M. Weiser. Program slicing. In *In the Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[WRW] WindRiver website, http://www.windriver.com.

[WW96] M. A. F. Wagner and J. R. Wilson. Recent Developments in Input Modeling with Bezier distributions. In *Proceedings of the 1996 Winter Simulation Conference*, pages 1448–1456, New York, NY, USA, 1996. ACM Press.

[YGS+04] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 470–479, Washington, DC, USA, 2004. IEEE Computer Society.

[Yin03] R. K. Yin. *Case Study Research - Design and Methods*. ISBN: 0-7619-2552-X. Sage Publications, 2003.