Mälardalen University Press Doctoral Theses
No.84

Enabling Timing Analysis of Complex Embedded Software Systems

Johan Kraft

August 2010

School of Innovation, Design and Technology
Mälardalen University

ISSN 1651-4238
ISBN 978-91-86135-76-8
Printed by Mälardalen University Press
Distribution: Mälardalen University Press

Abstract

Cars, trains, trucks, telecom networks and industrial robots are examples of products relying on complex embedded software systems, running on embedded computers. Such systems may consist of millions of lines of program code developed by hundreds of engineers over many years, often decades.

Over the long life-cycle of such systems, the main part of the product development costs is typically not the initial development, but the *software maintenance*, i.e., improvements and corrections of defects, over the years. Of the maintenance costs, a major cost is the verification of the system after changes has been applied, which often requires a huge amount of testing. However, today's techniques are not sufficient, as defects often are found post-release, by the customers. This area is therefore of high relevance for industry.

Complex embedded systems often control machinery where timing is crucial for accuracy and safety. Such systems therefore have important requirements on timing, such as maximum response times to different events. However, when maintaining complex embedded software systems, it is difficult to predict how changes may impact the system's run-time behavior and timing, e.g., response times. Analytical and formal methods for timing analysis exist, but are often hard to apply in practice on complex embedded systems, for several reasons. As a result, the industrial practice in deciding the suitability of a proposed change, with respect to its run-time impact, is to rely on the subjective judgment of experienced developers and architects. This is a risky and inefficient, trial-and-error approach, which may waste large amounts of person-hours on implementing unsuitable software designs, with potential timing or performance problems. This can generally not be detected at all until late stages of testing, when the updated software system can be tested on system level, under realistic conditions. Even then, it is easy to miss such problems. If products are released containing software with latent timing errors, it may cause huge costs, such as car recalls, or even accidents. Even when such

problems are found using testing, they necessitate design changes late in the development project, which cause delays and increase costs.

This thesis presents a framework for impact analysis with respect to runtime behavior such as timing and performance, targeting complex embedded systems. The impact analysis is performed through optimizing simulation, where the simulation models are automatically generated from the system implementation. This approach allows for predicting the consequences of proposed designs, for new or modified features, by prototyping the change in the simulation model on a high level of abstraction. This could be to simply increase the execution time of a particular task. Thereby, unsuitable designs can be identified early, before implementation, and a late redesigns are thereby avoided, which improves development efficiency and predictability, as well as software quality.

The contributions presented in this thesis are within four areas related to simulation-based analysis of complex embedded systems: (1) simulation and simulation optimization techniques, (2) automated model extraction of simulation models from source code, (3) methods for validation of such simulation models and (4) recording techniques for model extraction, impact analysis and model validation purposes. Several tools has been developed during this work, of which two are in commercialization in the spin-off company Percepio AB.

Note that the Katana approach presented in Chapter 5 is subject for a U.S. patent application – patent pending.

Sammanfattning

Mobiltelefoner, bilar, tåg, automationssystem och industrirobotar är exempel på produkter som är beroende av komplexa inbyggda mjukvarusystem, ofta bestående av milliontals rader programkod som utvecklats under många år. Dessa mjukvarusystem har möjliggjort helt nya funktioner, men även gjort produktutveckling mer komplex. När nya funktioner läggs till komplexa system är det stor risk att fel uppstår, på grund av svårigheten att överblicka alla konsekvenser av ändringarna. Trots att produktföretagen lägger mycket tid och pengar på testning upptäcks inte alla fel vilket orsakar stora kostnader, t.ex. i form av återkallade bilar. Stora summor kan sparas och bättre produktkvalitet uppnås genom nya typer av utvecklingsverktyg som bättre identifierar mjukvaruproblem så tidigt som möjligt i produktutvecklingsprocessen. Vissa typer av mjukvarufel är extra svåra att hitta och återskapa eftersom de bara uppstår i mycket speciella situationer, som t.ex. när datorns processor inte hinner köra en viss programkod inom avsedd tid. För vanliga PC datorer är sådana fördröjningar vanliga, men orsakar oftast inte några större problem. För industriella mjukvarusystem, ofta tidskritiska, kan dock fördröjningar mätta i millisekunder orsaka allvarliga fel. Därför vill man tidigt i utvecklingen av nya funktioner kunna förutse hur CPU belastning och svarstider kommer att påverkas. Med denna analys kan produktföretag minska sina kostnader eftersom man kan förutse och undvika problem som annars orsakat kostnader, och man förbättrar produktens tillförlitlighet genom man minskar risken att införa svårfunna fel. För komplexa industriella system kräver denna analys en analyserbar modell som beskriver hur systemets delprogram utnyttjar delade resurser, som t.ex. processorn, och de möjliga kommunikationerna mellan delprogrammen samt med omgivningen. En sådan modell kan sedan analyseras i ett simulatorprogram, utvecklad i för detta syfte, som visar effekten av föreslagna förändringar. Avhandlingen beskriver metoder och verktyg för att automatiskt skapa sådana modeller, baserat på analys av programkod och in-

spelningar av mjukvarusystemet i drift, metoder för att analysera de skapade modellerna, metoder för att spela in information från simuleringar eller från det skarpa mjukvarusystem under drift, samt metoder för att jämföra simuleringsresultat med verkliga inspelningar från det modellerade mjukvarusystemet. De viktigaste delarna av detta har utvärderats på ett skarpt industriellt system, ett styrsystem för industrirobotar från ABB; dock finns ännu ingen integrerad helheltslösning som möjliggör skarp användning av analysramverket. Dellösningar är dock under separat kommersialisering i författarens företag, Percepio AB. Observera att lösningen som presenteras i Kapitel 5, Katana, är under patentering i USA.

Till Birgitta och Gabriel

Preface

This work has been supported by ABB, Bombardier Transportation, the Knowledge Foundation (KKS), and the Swedish Foundation for Strategic Research (SSF), through the strategic research center PROGRESS.

This thesis concludes a long and probably quite unusual journey which started back in 2002, in my Magister thesis project together with Jonas Neander, at ABB Robotics. My main supervisor, Christer Norström, was at the time in a position as development manager at ABB Robotics. From his background in academic real-time systems research he realized their need for real-time analysis support and initiated a quite open magister thesis project in that direction. In that work we first investigated simulation as a means for timing analysis and proposed a simulator solution named ART-ML. An interesting story from the magister thesis project is the reactions from experienced developers when we showed them recordings of their system's run-time behavior; even highly skilled, senior developers were surprised by some details. After working some time with embedded software development at ABB Robotics, as a consultant, in 2004 I got the opportunity to develop a new solution for trace recording and trace visualization. This resulted in the Tracealyzer tool and the trcrec recorder module, which quickly became an integrated part of their control system. The Tracealyzer is still (2010) used for monitoring and troubleshooting purposes at ABB Robotics and is now also a commercial product of Percepio AB. The initial purpose of the Tracealyzer was however trace visualization in the context of simulation-based timing analysis and it is still a key part in the timing analysis framework presented in this thesis.

In April 2003 I enrolled as a PhD student at MDH with support from ABB and ASTEC, a Vinnova competence center, initially working 50/50 at ABB Robotics and MDH. The first years were quite straight-forward; I developed some tools, including the first version of the RTSSim simulator, and outlined a process for (manual) simulation modeling and validation, which lead to a

licentiate thesis presented in 2005.

Since January 2006 I have been employed 100 % at MDH, up until 2009 in the EXTRACT project, supported by KKS in collaboration with ABB and Bombardier Transportation, and thereafter in PROGRESS, supported by SSF. After my licentiate thesis (2005) I started working on methods and tools for automated model extraction since it was realized that manual modeling is not realistic for large industrial systems. Initially I worked on a semi-automated approach where some manual modeling still was required. However, during 2007 I realized that also the semi-automated approach would require too much manual modeling to be realistic and a fully automated model extraction tool would be necessary for realistic applicability on large industrial systems. Finally, I ended up spending 18 months (almost full time) on developing the Katana algorithm and implementing a tool using this approach. I am however very happy with the end result; a U.S. patent application has recently been filed regarding Katana. I hope the reader will find this thesis as interesting to read as I found it interesting to write!

I would like to thank my current and former supervisors, especially Christer Norström for trying to keep me focused and for sharing his industrial insights, and Anders Wall for the many creative discussions during these years. Björn Lisper contributed as assistant supervisor up until my licentiate thesis and provided many interesting ideas. I greatly appreciate the enthusiastic support from people at ABB, especially Peter Eriksson, Anders Wall, Goran Mustapic and Magnus Larsson, and from Bombardier Transportation, through Erik Gyllenswärd, Christer Persson and Anders Östmark. Our discussions during these years has provided a lot of valuable input from an industrial perspective and given me a much better understanding of software development for complex embedded systems. Your enthusiasm and positive spirit have been very supportive. Thanks a lot. I also want to thank Anders Öberg, Stefan Rönning and Mikael Åkerholm[1] at CC Systems[2] for all your help and enthusiasm.

I greatly appreciate the feedback I have received on my thesis drafts, from more people than expected. Apart from my supervisors, also Jan Carlsson, Stefan Bygde, Holger Kienle, Daniel Sundmark, Joel Huselius, and Bill Dittmann (Quadros Systems, Inc.) have provided great feedback! Thanks!

I really enjoyed working with Joel Huselius during 2004 – 2007, especially during the Sydney trip! I hope that we can stay in touch in the future, as friends and hopefully also as colleagues. Since 2006 I have worked a lot with Yue Lu, a nice and interesting collaboration with significant cultural differences from I

[1] Now at ABB

[2] Now Cross Control

have learned a lot. I hope we continue this in the future.

In February 2008 I met Markus Bohlin at the hospital where our respective wives were recovering after delivering Idun and Gabriel. Markus and I knew each other briefly from our undergraduate studies, but this started a collaboration which lead to quite interesting results, presented in Chapter 4. This collaboration, which also involved Yue Lu, Per Kreuger and Thomas Nolte, was very interesting and fun, and I think we can do great things in the future.

I would like to thank all current and former colleagues at the department, including Joel Huselius, Thomas Nolte, Markus Bohlin, Holger Kienle, Hans Hansson, Yue Lu, Farhang Nemati, Mikael Åsberg, Moris Benham, Daniel Sundmark, Anders Pettersson, Mikael Åkerholm, Johan Fredriksson, Jonas Neander, Stefan Bygde, Dag Nyström, Jan Carlsson, Andreas Hjertström, Stefan Cedergren, Joakim Fröberg, Jukka Mäki-Turja, Stig Larsson, Kaj Hänninen, Sara Dersten, Peter Wallin, Rikard Lindell, Hüseyin Aysan, Hongyu Pei-Breivold, Rikard Land, Christer Sandberg, Andreas Ermedahl, Sigrid Eldh, Filip Öhman, Gunnar Widforss, Malin Rosquist, Ivica Crnkovic, Mikael Sjödin, Mats Björkman, Mikael Ekström, Martin Ekström, Jörgen Lidholm, Monica Wasell and Harriet Ekwall. Working with you has been great and I hope to see you around also in the future. The “SAVE-IT rockers” deserve special thanks for nice company during our morale-boosting school trips, like Grenoble!

My dear friends and fishing buddies Christian Hultman, Christian Andersson and Rickard Söderbäck deserve many thanks for all the fun (with or without fishing gear) and for keeping me connected to reality during the periods I have been deep into my algorithms.

My parents, Lennart and Susanne, and sisters, Josefin and Kim, deserve many thanks for your love and support during all the years. Thanks for helping me in so many ways. I love you! My mother-in-law, Margareta, has been curious and supportive throughout this journey. I am very grateful that you raised such a great girl and sent her to Västerås for me to find! Last but definitely not least, Birgitta and Gabriel, my beloved wife and son. Being with you give me inspiration and energy and without you, life would not be the same. I love you with all my heart!

Johan Kraft
June 2010

Publications

The thesis author has previously authored or co-authored the following publications. Note that the thesis author was named Johan Andersson up until October 2006.

Theses

Modeling the Temporal Behavior of Complex Embedded Systems – A Reverse Engineering Approach, Johan Andersson, Licentiate Thesis, Mälardalen University Press, June, 2005.

Timing analysis of a robot controller, Johan Andersson and Jonas Neander, Magister Thesis, Mälardalen University, October, 2002.

Articles in Collection

Decreasing Maintenance Costs by Introducing Formal Analysis of Real-Time Behavior in Industrial Settings, Anders Wall, Johan Andersson and Christer Norström. In "Leveraging Applications of Formal Methods, Lecture Notes in Computer Science (LNCS) 4313", p 130 – 145, Springer, November, 2006.

A Framework for Analysis of Timing and Resource Utilization targeting Complex Embedded Systems, Johan Andersson, Anders Wall and Christer Norström. In "ARTES – A network for Real-Time research and graduate Education in Sweden 1997 – 2006", p 297 – 329, Uppsala University, Editor(s): Hans Hansson, 2006.

A Dependable Open Platform for Industrial Robotics – A Case Study, Goran Mustapic, Johan Andersson, Christer Norström and Anders Wall. In "Architecting Dependable Systems II, Lecture Notes in Computer Science (LNCS) 3069", Editors: Rogerio de Lemos, Cristina Gacek, Alexander Romanovsky, 2004.

Conferences and Workshops

Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects, Johan Kraft, Anders Wall and Holger Kienle. To appear in Proceedings of the 1st International Conference on Runtime Verification, Malta, November, 2010.

A Statistical Approach for Validation of Task Simulation Models with Intricate Temporal Execution Dependencies, Yue Lu, Johan Kraft, Thomas Nolte and Christer Norström. In Proceedings (Work-In-Progress track) of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10), Stockholm, Sweden, April, 2010.

System-specific Static Code Analyses for Complex Embedded Systems, Holger Kienle, Johan Kraft and Thomas Nolte. In Proceedings of the 4th International Workshop on Software Quality and Maintainability (SQM'10), Madrid, Spain, March, 2010.

Statistical-based Response-Time Analysis of Systems with Execution Dependencies between Tasks, Yue Lu, Thomas Nolte, Johan Kraft and Christer Norström. In Proceedings of 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'10), St. Anne's College, University of Oxford, March, 2010.

Statistical-based Response-Time Analysis of Systems with Execution Dependencies between Tasks, Yue Lu, Thomas Nolte, Johan Kraft and Christer Norström. In Proceedings (Work-In-Progress track) of the 30th IEEE Real-Time Systems Symposium (RTSS'09), Washington, DC, USA, December, 2009.

Simulation-Based Timing Analysis of Complex Real-Time Systems, Markus Bohlin, Yue Lu, Johan Kraft, Per Kreuger and Thomas Nolte. In Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09), p 321 – 328, Beijing, China, August, 2009.

Transformational Specification of Complex Legacy Real-Time Systems via Semantic Anchoring, Yue Lu, Antonio Cicchetti, Stefan Bygde, Johan Kraft, Thomas Nolte and Christer Norström. In Proceedings of the 2nd IEEE International Workshop on Component-Based Design of Resource-Constrained Systems (CORCS'09), p 510 – 515, IEEE Computer Society Press, Seattle, Washington, USA, July, 2009.

Approximate Timing Analysis of Complex Legacy Real-Time Systems using Simulation Optimization, Yue Lu, Markus Bohlin, Johan Kraft, Per Kreuger, Thomas Nolte and Christer Norström. In Proceedings (Work-In-Progress track) of the 29th IEEE Real-Time Systems Symposium (RTSS'08), p 29 – 32, Barcelona, Spain, December, 2008.

Towards Migrating Legacy Real-Time Systems to Multi-Core Platforms, Farhang Nemati, Johan Kraft and Thomas Nolte. In Proceedings (Work-In-Progress track) of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'08), p 717 – 720, IEEE Industrial Electronics Society, Hamburg, Germany, September, 2008.

Validation of Temporal Simulation Models of Complex Real-Time Systems, Farhang Nemati, Johan Kraft and Christer Norström. In Proceedings of the 1st IEEE International Workshop On Component-Based Design Of Resource-Constrained Systems (CORCS'08), Turku, Finland, July, 2008.

A Metaheuristic Approach for Best Effort Timing Analysis targeting Complex Legacy Real-Time Systems, Johan Kraft, Yue Lu, Christer Norström and Anders Wall. In Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08), St. Louis, MO, USA, April, 2008.

Extracting Simulation Models from Complex Embedded Real-Time Systems, Johan Kraft, Joel Huselius, Anders Wall and Christer Norström. Real-Time in Sweden 2007, Västerås, August, 2007.

Evaluating the Quality of Models Extracted from Embedded Real-Time Software, Joel Huselius, Johan Kraft, Hans Hansson and Sasikumar Punnekkat. In Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, p 577 – 585, IEEE, Tucson, USA, March, 2007.

Extracting Simulation Models from Complex Embedded Real-Time Systems, Johan Andersson, Joel Huselius, Christer Norström and Anders Wall. In Proceedings of the 2006 International Conference on Software Engineering Advances, IEEE, Tahiti, French Polynesia, October, 2006. **Best Paper Award**.

Automatic Generation and Validation of Models of Legacy Software, Joel Huselius, Johan Andersson, Hans Hansson and Sasikumar Punnekkat. In Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06), p 342 – 349, Sydney, Australia, August, 2006.

Model Synthesis for Real-Time Systems, Joel G Huselius and Johan Andersson. In Proceedings of the 9th 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 and 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 and 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 and 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 and 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 and 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 and 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 and Christer Norström. In Proceedings of the ICSE 2003 Workshop on Software Architectures for Dependable Systems, Portland, Oregon, USA, May, 2003.

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

Technical Reports

Best-Effort Simulation-Based Timing Analysis using Hill-Climbing with Random Restarts, Markus Bohlin, Yue Lu, Johan Kraft, Per Kreuger and Thomas Nolte. MRTC report, ISSN 1404-3041, ISRN MDH-MRTC-236/2009-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June, 2009.

A Framework for Real-Time Systems Migration to Multi-Cores, Farhang Nemati, Johan Kraft and Thomas Nolte. MRTC report, ISSN 1404-3041, ISRN MDH-MRTC-235/2009-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May, 2009.

RTSSim – A Simulation Framework for Complex Embedded Systems, Johan Kraft. Technical Report, MRTC, March, 2009.

Legacy Issues in Industrial Software Development, Johan Kraft and Joel Huselius. MRTC report, ISSN 1404-3041, ISRN MDH-MRTC-213/2007-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May, 2007.

Experimental Model Synthesis for Timing Analysis of an Industrial Robot, Joel Huselius, Johan Andersson, Hans Hansson and Sasikumar Punnekkat. MRTC report, ISSN 1404-3041, ISRN MDH-MRTC-193/2005-1-SE, Mälardalen University, November, 2005.

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 and 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 and Christer Norström. Technical Report, MRTC, August, 2004.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

When most people hear the word “computer” they think of a PC. However, over 99 % of all computer processors manufactured are used in *embedded computer systems* [127]. These are specialized computers, integrated in many types of products, from advanced industrial products such as cars, trains, airplanes, telecom switches, industrial robots, factory automation systems, medical equipment to consumer electronics such as mobile phones, wireless routers, TVs, cameras and toys. Software development is today dominating the product development for many companies and most new innovations today are implemented in software. As an example, Volvo (the truck company), estimates that 90 % of their new innovations are in the field of electronics, and 80 % thereof is software [84]. In total, Swedish industry spends 60 billion SEK (6 billion Euros) annually on software development. In the ten largest Swedish companies, software development accounts for 60 % of the research and development budgets. Ericsson alone spends 25 billion SEK yearly on software development – 80 % of its research and development budget [85].

Product companies are using embedded computers in their products for several reasons. In several domains, older electromechanical solutions are being replaced with software solutions, running on small and cheap embedded computers, in order to reduce unit cost, size, weight or allow for new, advanced functionality previously not possible, such as vehicle stability control or advanced fuel injection with reduces fuel consumption. Cheap, high performance embedded computers have also enabled new products, previously not possible (or very expensive) such as GPS navigation systems, portable media players and advanced mobile phones.

Embedded computers come in all sizes, from very small and simple 8-bit single-chip computers, with 1 kilobyte of memory, to gigahertz 64-bit multi-core computers with the performance and resources of a PC. This thesis focuses on complex embedded software systems, which typically runs on relatively powerful hardware. Examples of such systems are industrial robot control systems, automation systems and telecommunication systems. Such systems often consist of millions of lines of source code, which have been developed and maintained by hundreds of engineers over many years. A system of this size is too large and complex for any single person to understand in detail. In this thesis, such software systems are labeled *complex embedded systems*.

Embedded computer systems are typically in control of machinery and thereby often safety-critical and/or business-critical systems. They therefore have requirements on dependability, such as safety, reliability and availability. Moreover, many embedded systems are *real-time systems*, i.e., systems with strict requirements on timeliness; they must respond to input from its environment in a timely manner. For non-real-time computers such as home PC's the focus is on the average performance, while for real-time systems another property is much more important: a predictable *worst case response time*, i.e., the highest possible latency of an event, from the input signal to the corresponding reaction. A violation of a temporal requirement may cause a system failure; it is therefore of great importance that the worst case response time is known for each time-sensitive system function.

Another aspect of complex embedded systems (and of large software systems in general) is their long life-cycles, many years, often decades [106]. Since the development of such a system represents a major investment for a company, often hundreds of person years, it is seldom economically feasible to "start over" and redesign such a system from scratch. Consequently, systems of this type are maintained over many years during which thousands of changes are made in order to, e.g., implement new requirements, correct errors, improve performance or improve the software design.

Conceptually, the life cycle of a complex software system can be divided into three different phases as depicted in Figure 1.1: (1) the early phase, (2) the evolution phase, and (3) the legacy phase. The curve shows a conceptual illustration of the increasingly harder development; the horizontal axis shows the total, accumulated development effort invested in the system (the cost), while the vertical axis shows the achieved system functionality (the resulting value).

In the early phase (1), a lot of effort is required for development of the platform, which causes a relatively low productivity with respect to customer

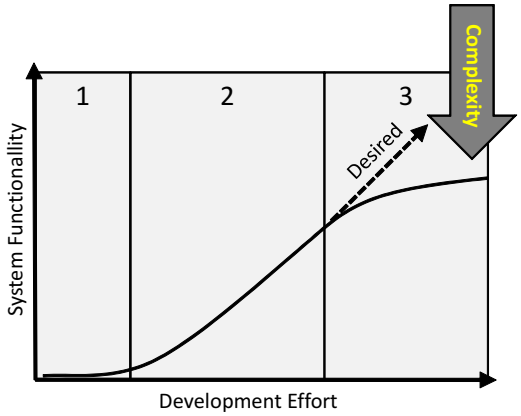

Figure 1.1: System functionality as a function of total development effort

value. This investment however pays off in the evolution phase (2), where the developed platform facilitates development of new features, which gives high productivity. Large amount of changes are made which cause the system to evolve from its original design and gradually become larger and more complex. The legacy phase (3) begins when the complexity approach the practical limit of the development tools and methods used, which causes a decrease in productivity. Adding new features is now much harder than it used to be.

Lehman et al. [107] recognizes the phenomenon of increasing complexity as the second[1] law of software evolution: "As an E-type[2] system evolves its complexity increases unless work is done to maintain or reduce it".

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, and documentation, tends to degrade as changes are made over the years, often in a less than optimal manner due to time pressure or inadequate design documentation. Another reason is that not all software developers have backgrounds in computer science or software engineering. In industry many developers have their expertise in domain-specific technology, such as combustion engines, control theory or welding techniques. Moreover, for long-lived systems, the requirements typically change over time due to factors such as legislation, standards, technology development and competitor initiatives, and

[1] The Lehman paper includes eight "laws" in total (observations really).
[2] A system solving a problem or addressing an application in the real world.

it is often hard to adapt existing architectures in a good way without making large (and thereby time consuming) changes. Yet another issue is the personnel turnover over the long system life-cycle. Experienced developers leave, together with their knowledge, and newly employed inexperienced developers have to take over.

In order to combat the increasing complexity, i.e., to extend the period of efficient software development, developers need better means for finding latent defects, for predicting the consequences of proposed changes and for avoiding bad design decisions. Today, most companies that develop complex embedded systems rely heavily on code inspection and testing, which are necessary but not sufficient. Huge amounts of testing are performed on each release version in order to capture as many defects as possible, but it is still common that defects are missed and instead experienced by the customers.

According to a 2002 study from by the National Institute of Standards and Technology at the U.S. Department of Commerce, software defects cost the U.S. economy an estimated $59.5 billion annually [4]. The study concluded that more than a third of these costs could be eliminated by improved testing infrastructure that enables earlier and more effective identification and removal of software defects, i.e., by finding an increased percentage of errors closer to the development stages in which they are introduced. Moreover, over half of all errors are not found until “downstream” in the development process or during post-sale software use.

1.1 Problem and Possible Solutions

When maintaining complex embedded 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, e.g., simply to correct a minor defect, it might cause the response time to exceed the specified limit, the *deadline*.

In a worst case scenario, a maintenance operation will cause the system to violate its temporal requirements in rare situations only. 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 software defect could cause an industrial robot to fail and thereby halt a car production line for hours,

causing a large monetary loss.

Developers of complex embedded systems today rely on system-level testing in order to detect timing errors, but this is inefficient, since the search space is so large and since such errors depend on program execution times, which typically vary from time to time. It is well known that timing errors are often hard to detect and to reproduce using testing [89, 105], but existing methods for timing analysis are often hard to apply on complex embedded systems for a number of reasons, as discussed in the following sections.

The risk of introducing timing errors when changing a system can be significantly reduced if the impact of the change can be predicted at an early stage of development. The ability to perform *impact analysis* of proposed changes (e.g., new features) with respect to important run-time properties could allow developers to produce software of higher quality, in less time and more reliably.

To manually analyze the timing impact of a change is difficult since the details and dependencies of a software system's temporal behavior is not visible in the source code. Studying recordings of the system can help, but this view of the run-time system is however difficult to extrapolate with respect to proposed changes, due to the system complexity.

Industrial use of timing impact analysis requires tools and models which give analyzability with respect to relevant system properties, such as response times. Introducing such analysis support for existing systems can be achieved in two ways, either intrusively or non-intrusively. In an intrusive approach, the system is changed in order to be more predictable and analyzable. This could for instance mean to change from an event-triggered to a time-triggered software architecture. The major problem with an intrusive approach is the large effort and risks involved in a major redesign. This would be very costly, will most likely introduce new defects in the system, and would probably be hard to motivate economically. Moreover, the current software architecture has probably been selected for a good reason, so changing it for improved analyzability might mean other drawbacks, such as lower performance or reduced flexibility.

The work in this thesis follows a non-intrusive approach, where the focus is to find or develop means for analysis which can be applied to the existing system, without changing[3] the implementation.

[3]Even though introduction of a software-based trace recorder (cf. Chapter 7) is technically intrusive, this is a negligible change compared to a major re-design of the system.

1.1.1 RTA – Response Time Analysis

Several analytical methods have been proposed for *response-time analysis* of real-time systems [98, 99, 101, 37, 112, 113, 114, 115]. In this thesis, such methods are commonly labeled RTA. Such methods however all use a rather simplistic system model and have several assumptions which makes them inapplicable or highly pessimistic for embedded software systems which have not been designed with such analysis in mind. For instance, there are industrial systems which violate the assumptions of RTA by containing tasks which

- trigger other tasks in complex, often undocumented chains of task activations depending on input,
- share data with other tasks, e.g., through global variables or inter-process communication (IPC),
- have radically different behavior and execution time depending on shared data and input,
- change priorities dynamically, e.g., as a solution to previously identified timing problems,
- have timing requirements expressed in functional behavior rather than explicit task deadline, such as availability of data in input buffers at task activation.

In general, RTA methods are overly pessimistic for complex embedded systems since they do not take behavioral dependencies between tasks into account, for instance the above listed types. Analyzing complex embedded systems requires a more detailed system model which includes relevant behavior as well as resource usage of tasks. Two approaches are presented in the following sections where detailed behavior models are used: model checking and discrete event simulation.

1.1.2 Model Checking

There are several formal analysis tools which can be used to verify different properties of models through *model checking*, such as UppAal [129], KRONOS [131] and ComFoRT [130]. Model checking implies an exhaustive search of the state-space of the analyzed model and can therefore allows for identification of worst-case behaviors, e.g., for response times. However, model checking is not widely used in industry, apart from domains with extreme dependability

requirements, such as aerospace systems, military systems or nuclear power plants, where the software systems where the consequences of a failure are catastrophic. Such systems are typically quite small and are rarely modified after deployment.

Model checking is however difficult to apply on large industrial software systems for a number of reasons. One problem is the size and complexity of these systems which implies an astronomic amount of possible scenarios. This makes exhaustive analysis techniques, like model checking, extremely resource demanding. In many cases, the analysis will not terminate, as the model state-space is too large to search in realistic time.

Another issue is that formal analysis tools requires an analyzable model in a formal notation, such as timed automata [82]. The formal methods community tends to assume a model-driven development approach, where the model to analyze is the system specification and is used to automatically generate (synthesize) the run-time system. However, with few exceptions, industrial software systems are still developed in a traditional code-oriented manner; there are no analyzable models available. These would therefore have to be created based on the implementation in a major modeling project. Moreover, even if an initial modeling effort could be managed, e.g., by hiring a team of experts in formal modeling, the developers would still need to keep the model up-to-date with the system source code as it evolves. This is likely to be neglected, e.g., due to focus on short term development goals, which would invalidate the analysis model. If updating the model requires too much effort, or is too hard, it is likely that the developers would stop maintaining and using the analysis model.

1.1.3 Discrete Event Simulation

Discrete event simulation (hereafter just “simulation”) is an approach to timing analysis which avoids the state-space explosion problem by sacrificing the guaranteed safety of the result. In a simulation, the state space of the model is sampled, typically in a random manner, rather than searched exhaustively. The random sampling (simulation) is repeated for a certain duration (e.g., over night), or until a problem is found. This means that it is not possible to identify worst-case situations using simulation, since not all possibilities are necessarily explored. Note that the worst-case might have occurred in a simulation, but the simulation result does not tell if this is the case.

A general problem when using simulation is to determine the simulation budget, i.e., how many simulations to run and the length of the individual simulations, and the resulting confidence in the simulation results.

Simulation can however be used for predicting typical performance (or responsiveness) and for finding *extreme cases*, i.e., situations far from the typical, but not necessarily the worst case. This can be enough in many cases. If the extreme case is a requirement violation, e.g., an exceeded deadline, a real problem has been found (assuming a correct simulation model) which otherwise could have been missed during system testing. A negative result from a simulation-based analysis, i.e., that no problem is found, is however no guarantee of correctness. The simulation approach is closer to testing than formal analysis, since it can only show the presence of errors, not their absence.

Simulation is however a much more efficient method for finding timing problems than system-level testing, the dominating method in industry today. Moreover, simulation is applicable for any system, also for complex industrial systems where model checking or RTA falls short. Furthermore, by using simulation it is possible to eliminate the modeling problem, as simulation models are typically expressed in normal programming languages, like C, and can be automatically generated (extracted) from the original source code, as demonstrated in this thesis. Note that simulation is a very general technique, which can be used for predicting any measurable run-time property, not just response-time. Examples of other properties of interest are occurrence of certain error conditions, such as buffer under-run, and the utilization of dynamically allocated resources.

Monte Carlo simulation (random sampling) is best suited for performance analysis, i.e., average-case behavior. Analysis of extreme value properties can however be made efficiently using *Simulation Optimization*, an iterative process where repeated simulations of a model are guided using heuristic optimization techniques in order to maximize a specified property, such as highest response times of specific tasks.

1.2 Vision

A framework for simulation-based analysis is envisioned, illustrated by Figure 1.2, which to a large extend has been implemented in this thesis work. The framework is integrated in the system development process and is highly automated. An updated simulation model is always available, using an automatic model extraction tool which is either executed on demand or during every system build. In the latter case, the simulation model is a development artifact that is available as naturally as the compiled executable files.

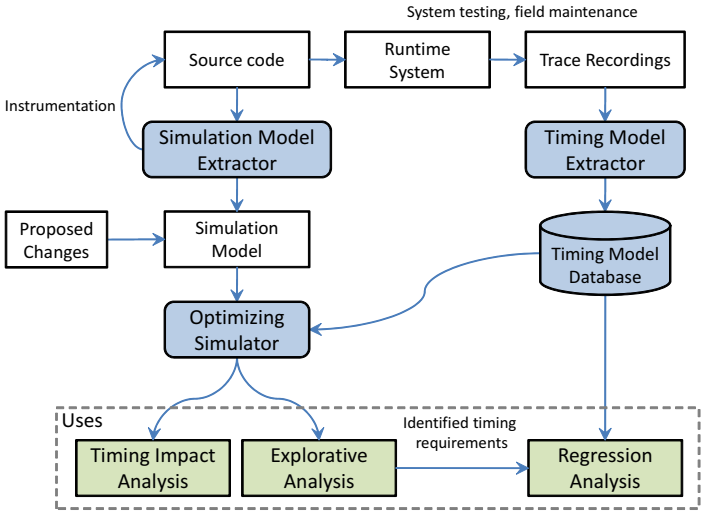

Figure 1.2: The envisioned analysis framework

The simulation model is a filtered version of the original code, containing only the statements of importance for timing and usage of specific resources. In order to reduce the size of the simulation model and thereby the simulation runtime, modeling abstractions may be provided manually, e.g., as code annotations which becomes an integrated part of the system source code. Timing-accurate simulation is achieved using timing data from a software *trace recorder*, permanently integrated and always active in the modeled system. The model extraction tool instruments the source code with calls to the recorder, in order to record three kinds of quantitative run-time information: (1) task-switches, i.e., the task scheduling trace, (2) execution of instrumentation

points, for execution time profiling, and (3) relevant system input events, such as commands/requests.

The run-time information is systematically collected during in-house system testing and possibly also during field maintenance, e.g., if troubleshooting a system in customer operation, labeled with the system version and configuration, and stored in a central database of the development organization, the timing model database. When performing simulation, this database is used by the simulator to get the quantitative information needed by the simulation model. This framework for simulation-based analysis allows for at least three kinds of analysis:

- **Impact analysis**, i.e., "what-if" analysis on specific proposed changes. This is used on demand, before implementing large changes, such as adding new tasks or changing a task's priority.
- **Explorative analysis**, in which a large amount of simulations are performed, with random changes in the quantitative information (execution times, inputs), in order to find the limits where errors start to occur, i.e., the timing requirements. The analysis would generate a report describing what parts of the system that are most timing sensitive, i.e., where timing error might occur today or might occur after a small change of the system. This analysis would be performed in the verification phase, as soon as a new version is available for system-level testing.
- **Regression analysis**, in which differences and trends are identified between different versions of the same system. In this analysis, the timing requirements identified during the explorative timing analysis is used as reference for comparison; the analysis gives a warning if an execution time is getting close to violating such a timing requirement. This analysis is a part of long term quality assurance.

The properties in focus can be related to either typical, average-case behavior or extreme-case behavior, where the latter is accomplished using simulation optimization, e.g., as presented in Chapter 4.

Model validation is integrated in this framework, but is however not included in the illustration. The automated approach to model extraction does not eliminate the model validity issue, it just moves the problem to ensuring the correctness of the model extraction tool and its configuration. After major changes of the system, such as major architectural changes or new hardware, model validation is performed through *sensitivity analysis*, where different types of changes are tested in order to verify that the analysis framework

can accurately predict their impact. This is an impact analysis with respect to dummy changes where the prediction is compared against the actual impact, measured on the real system after the dummy change has been implemented.

Most components of the envisioned framework have been implemented and separately evaluated in this thesis work, partly on industrial cases. A case study using the complete framework on real industrial cases remains for future work.

1.3 Research Questions

The research questions of this thesis are related to the feasibility and practical applicability of the above vision with respect to complex embedded systems. Concretely, three research questions are stated:

- **Q1:** Can simulation models be extracted automatically from C source code, with sufficient efficiency and accuracy for scaling to complex embedded systems?
- **Q2:** Is simulation optimization an efficient approach for predicting extreme cases in the temporal behavior of complex embedded systems, compared to existing methods for timing analysis?
- **Q3:** Is software trace recording generally applicable on common commercial operating systems for embedded systems with respect to implementation feasibility and run-time overhead?

Given that a solution for automated model extraction can be developed, the important questions are the scalability of the model extraction solution, i.e., how the runtime of the tool relates to the amount of code to analyze, as well as the size of the extracted simulation models in relation to the original program. The model size is important for the simulation speed and thereby model coverage of simulation optimization methods.

Simulation optimization is a promising method which may allow for more efficient prediction of extreme-case behavior compared to system-level testing and Monte Carlo simulation, intended for systems where analytical or formal methods are difficult to apply. This is a “best effort” approach, but if sufficiently efficient this can extend the applicability of simulation to also include efficient search for extreme-case timing behaviors.

Trace recording is a key component in this analysis framework but requires permanent integration of a software recorder, unless dedicated hardware is used for this purpose. If adding a trace recorder to the system on demand only, e.g.,

during testing, there is a risk of *probe effects* [89], i.e., that the system behavior changes as a result of the added recording overhead. Thereby, the modeled, analyzed system is not identical to the release version, used by customers, which threatens the validity of the analysis. An easy answer to this problem is to keep the trace recorder in the system, always active, and treating it as an integrated part of the system. This also has the advantage of improved debugging support, since traces are always available, but has a cost in the form of a CPU and RAM overhead. The size of the recorder overhead is of great importance to the applicability of this approach, since product companies naturally wants to get as much product performance as possible while keeping hardware costs down.

1.4 Scientific Contributions

The contributions of this thesis are addressing the research questions presented in Section 1.3 and are necessary components in realizing the vision presented Section 1.2. The contributions are:

- **C1:** An algorithm for automatic extraction of simulation models from source code, *Katana*, which is based on a new approach to program slicing. This method was developed to answer **Q1**.
- **C2:** A prototype implementation of Katana, MXTC, and an evaluation of MXTC on industrial code, which indicates that **C1** is indeed an answer to **Q1**, i.e., that the Katana approach is sufficiently efficient and scalable for complex embedded systems.
- **C3:** An efficient simulation framework for analysis of embedded systems, RTSSim, which is a key component in answering **Q2**.
- **C4:** Two methods for simulation optimization, HCRR and MABERA, with performance evaluations. Together with **C3**, this answers **Q2**.
- **C5:** Experiences regarding trace recorder implementation feasibility and typical run-time overheads, from five industry collaboration projects where trace recorders were implemented. This answers **Q3**.

The experiences of trace recorder implementations on several industrial systems should be of general interest for the embedded community. However, the results are not related to the scientific body-of-knowledge, since monitoring is a vast research area but, at the same time, not the research focus.

The issue of model validity is discussed in Chapter 8, which also presents ideas on trace comparison methods for general use in the envisioned analysis framework. This is however not counted as a formal scientific contribution due to the lack of evaluation but should rather be considered as work-in-progress.

All other contributions in this thesis are innovations of the author alone, except for the HCRR algorithm in **C5**. The author however initiated the collaborative project which resulted in the HCRR algorithm and played a significant role in the discussions leading to HCRR as well as in the HCRR evaluation.

Chapter 4 is based on two publications, the first publications of MABERA [15] and HCRR [14]. Chapter 1, Chapter 2 and Chapter 8 are based on the author's licentiate thesis [95]. The remaining chapters (i.e., 3, 4, 6, 7 and 9) have been written specifically for this thesis and present previously not published results.

1.5 Research Method

This thesis presents engineering research performed in collaboration with ABB and Bombardier. In this type of research, technical solutions are created or identified in response to industrial problems. The solutions are evaluated in order to determine their suitability and find possibilities for further improvement. The academic body-of-knowledge is used and extended throughout the process, through publications and conference presentations. The research follows an iterative model where the problem definition and solution design are refined through prototype development, evaluation and industrial feedback, as illustrated by Figure 1.3. Note that the edges leading "backwards", e.g., from internal evaluation to problem definition, corresponds to feedback from development and evaluation phases, when the proposed solution has been found suboptimal. This ranges from minor adjustments of the proposed solution design, to reformulating the overall research focus.

The problem described in the introduction was initially identified by ABB Robotics, a manufacturer of industrial robots and robot control systems. An on-site study was conducted on the subject in the form of a Master's thesis [94]. Between September 2002 and December 2004, the thesis author worked in embedded software development at ABB Robotics, which resulted in a good understanding of the challenges of developing and maintaining complex embedded software systems. In April 2003, the author enrolled as an industrial PhD student, working 50 % at ABB and on 50 % continuing the research initiated in the Master's thesis project [92, 93, 91, 90]. This initial work outlined

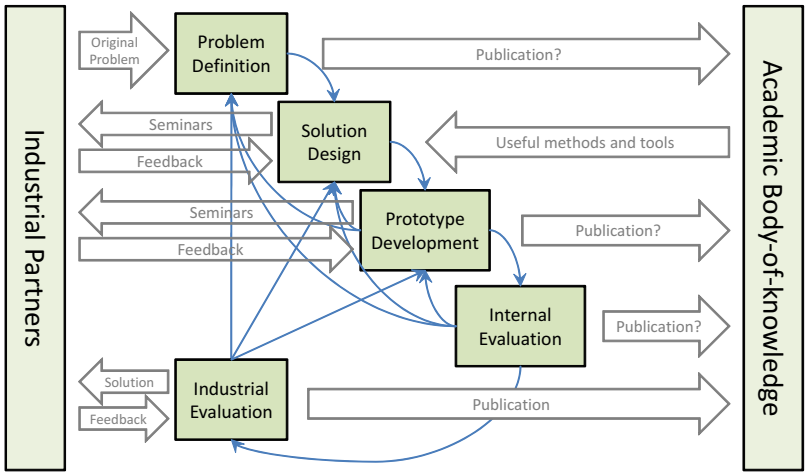

Figure 1.3: The research method

the overall approach, which resulted in a licentiate thesis on the subject [95]. With that, the core problems had been identified (see Section 1.3) and in the following years several solutions where developed, which are presented 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 Corporate Research, ABB Robotics and Bombardier Transportation, as well as researchers from other universities. Further, several publications on this subject have been presented on international scientific conferences in mainly two areas: Software Engineering [56, 108, 58] and Real-Time Systems [15, 14, 92, 93, 91, 64].

The methods and tools developed in this work have been evaluated on real cases provided by the industrial partners. The thesis author has visited both ABB Robotics and Bombardier Transportation for shorter industrial stays during which parts of these evaluations have been performed. The author has also been given access to proprietary source code and equipment. The evaluations of trace recording techniques, presented in Chapter 7, were performed on cases from four systems from four different companies: ABB Robotics, Bombardier Transportation, a major telecom company and a developer of automated welding equipment. In a fifth project a trace recorder was developed in close col-

laboration with Quadros Systems, Inc. [137], a U.S. company developing a real-time operating system.

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: real-time systems, simulation, model validation, model checking, reverse engineering and program slicing.

The work presented in this thesis is primarily influenced by the analysis challenges of ABB's control system for industrial robots, but there is nothing "robot-specific" in the proposed approach; it is general and can conceptually be applied on any software system, although the need is greatest on complex embedded systems, with highly dynamic run-time behavior.

1.6 Thesis Outline

This thesis is organized in nine chapters. Chapter 2 presents a state-of-the-art report on related research areas. Then follows five contribution chapters, focusing on different aspects of the envisioned analysis framework. Chapter 3 presents the use of simulation for timing analysis and introduces the RTSSim simulation framework. Chapter 4 presents two techniques for simulation optimization, MABERA and HCRR, together with an evaluation and comparison with other methods for timing analysis. Chapter 5 presents Katana, a method for automated extraction of simulation models from source code, through a novel approach to program slicing. An evaluation of Katana on industrial source code is presented in Chapter 6. Chapter 7 presents techniques for trace recording, experiences from five industrial trace recorder implementation projects, as well as ideas for how to obtain and model simulation timing models using trace recording. Chapter 8 discusses the threats against model validity and presents a process for comparison of trace data, which can be used in model validation and impact analysis. Finally, Chapter 9 concludes the thesis and outlines future work.

Chapter 2

Timing Analysis, Modeling and Model Validation

This chapter gives a background regarding related work in six areas of academic research, where some are alternative methods for similar purposes. The overall purposes of the described works fall into three main categories: timing analysis, modeling and model validation. The structure of this chapter is as follows:

- Real-Time Systems and Timing Analysis
- Timing Analysis using Model Checking
- Timing Analysis using Simulation
- Modeling using Source Code Analysis
- Modeling using Dynamic Analysis
- Model Validation

The first three sections (Section 2.1 – Section 2.3) describes different methods for timing analysis proposed in academia, including response-time and execution-time analysis. Since it is believed that detailed models are necessary in order to allow sufficiently accurate timing analysis of complex embedded software systems, and since such systems are too large for manual modeling,

a central problem is to find methods for automated generation of such models from existing implementations. The next two sections (Section 2.4 and Section 2.5) therefore describe works of relevance for automated modeling of existing implementations, using source code analysis and run-time measurements, respectively. The last section, Section 2.6, is about how to ensure the validity of the system model that is used for the analysis describes results related to model validation, both subjective methods and methods based on statistics. The chapter is concluded by Section 2.7, which discuss how the approach proposed in this thesis relates to the works (and research areas) presented in this chapter.

2.1 Real-Time Systems and Timing Analysis

This section describes the area of real-time systems and different types of timing analysis, i.e., methods of direct relevance for the overall research goal.

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 with requirements 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's research is how to verify that a system meets its temporal constraints.

Real-time systems are often composed of *tasks*, which are threads or processes implementing a particular system function, which typically execute periodically. An activity is often broken down into several tasks and tasks therefore often communicate, e.g., in order to forward their results as input to other tasks, or in order to send commands/request to other tasks.

Obviously, *periodic* tasks are tasks which are activated periodically, i.e., with a constant *inter-arrival time*. Many systems however also contain tasks which are recurring but not strictly periodic, which are typically labeled *sporadic* or *aperiodic* tasks. The difference between these is that sporadic tasks have a minimum inter-arrival time, aperiodic tasks does not. Such may correspond to interrupt service routines, triggered by hardware signals from an external system, e.g., a network.

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 affected 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 priorities 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 *transaction* is a chain of related tasks with *precedence constraints* which dictate their execution order. This is often realized using *offsets*, which changes the phasing of the task periods. A deadline over a chain of related events, like a transaction, is known as an *end-to-end* response time.

The temporal constraints of a real-time system is usually expressed as *deadlines*, which specify the highest response time allowed. If a real-time system is unable to finish a task before its deadline, it is a *deadline violation*. Such are often the result of an *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. However, a deadline violation may also occur in other situations, e.g., due to a deadlock causing indefinite blocking 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 violation, *hard* and *soft* real-time systems. A soft real-time system allows some occasional deadline violations. An example is a telecom system. The system's 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 frequently. 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.

Note that "real-time" does not have to mean "fast", even though many real-time systems run at high speeds. The critical concern is the importance of the timing constraints and the consequences if they would be violated.

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 "fly-by-wire" airplanes, such as the Swedish fighter-jet JAS 39 "Gripen" or the Boeing 777, where there is no physical connection (e.g., hydraulics) between the pilot's controls and the rudders, etc., but only electronic signaling. Another example in a different domain is a 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.

Most real-time systems are *multi-tasking* systems where the CPU[1] is shared by several tasks. This is achieved by using a multi-tasking operating system where a *scheduler* decides which task that should execute at any particular moment.

A large area within real-time research is *scheduling theory*, i.e., methods for setting task scheduling attributes, such as priorities, in order to fulfill a (multi-tasking) system´s timing requirements. The scheduling algorithms can be divided into off-line and on-line scheduling. When using on-line scheduling, the scheduling decisions are taken during run-time. An off-line scheduled system makes no decisions regarding the execution order of the tasks during run-time, instead a pre-calculated schedule is used. However, in such systems it is not possible to create new tasks during run-time since adding of new tasks to the system requires reconstructing the schedule. A more flexible scheduling policy is on-line scheduling. In this case, no schedule exists, but the operating system makes all the scheduling decisions during run-time. There are however operating systems where both types of scheduling can be used in parallel, such as Rubus from Articus Systems [120].

A very common algorithm for on-line scheduling is Fixed Priority Scheduling (FPS). 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. Most commercial real-time operating systems use *preemptive* fixed priority scheduling, meaning that the executing task can be preempted by higher priority tasks at any time.

The other major paradigm in on-line scheduling is Earliest Deadline First (EDF), an approach which is less common in commercial operating systems but often assumed in academic research. An EDF scheduler always select the task with least time left until its 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 its deadline. EDF can however be combined with other scheduling algorithms, such as overload handling or aperiodic server algorithms such as Total Bandwidth Server [104] or Constant Bandwidth Server [50]. Such server algorithms allocate a certain percentage of the CPU time, the server bandwidth, to aperiodic or sporadic tasks, if such are pending.

[1] For multi-core CPUs, each CPU core is shared by several tasks.

2.1.1 Schedulability- and Response-Time Analysis

A variety of analytical methods exists for schedulability analysis, i.e., methods which determine if a real-time system is schedulable with respect to the system deadlines. This section presents the major results within scheduling theory and the analytical response-time analysis methods.

The most well known result in the real-time community is the one by Liu and Layland from 1973 [99], in which 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 the highest rate (shortest period) receives the highest priority. The value of the Liu-Layland bound is dependent on the number of tasks in the system, but as the number of tasks increase, this bound approaches *ln 2*, approximately 69.3 %. For systems containing only tasks with harmonic periods, the bound is 100 %. Harmonic periods means that all task periods are multiples of the shortest task period.

The *Exact Analysis* was presented by Joseph and Pandya [101] 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 a fix-point 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 *critical instant*, where all tasks are ready to execute at the same time, with their individual worst-case execution time. The method has later been extended to handle, e.g., semaphores [37], deadlines longer than the periods [112], variations (jitter) in the task periodicity [113, 114] and distributed systems [115]. This family of analytical methods for response time analysis is commonly known as RTA.

2.1.2 Execution Time Analysis

When modeling a real-time system for analysis of timing related properties, the model needs to contain execution time information, i.e., how much CPU time needed by each task individually, if executing undisturbed. A common method in industry 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 the coverage; it is very hard to select test cases which generate high execution times and it is not possible to know if the worst case execution time (WCET) has been observed. Some companies compensates this to some extent through a "brute force" approach, where they systematically collect statistics from deployed systems, over long periods of real operation. This is however very dependent on how the system has been used and is still an "optimistic" approach, as the real WCET might be higher than the highest value observed.

Measuring is however not the only way to find execution time information. *Execution time analysis* is a well studied area in the intersection of program analysis and real-time systems research, where the focus is mainly on *WCET analysis*. Static WCET analysis tools such as AIT [139], Bound-T [141] and the (local) SWEET tool [140] strive to compute a safe, but tight, upper bound for the execution time of a program, given a specific hardware. On complex hardware architectures, with cache memory, pipelines, branch prediction tables and out-of-order execution, estimating a tight but safe WCET is however difficult. Complex embedded systems however often use relatively powerful and complex CPUs, such as Intel Pentium III, Pentium M or PowerPC 750 (cf. Chapter 7).

Since static WCET analysis depends on a model of the hardware, which however cannot predict every detail and therefore requires conservative, worst case assumptions in order to report a safe WCET estimate. Due to these assumptions the estimated WCET becomes pessimistic. However, there are several groups doing WCET research and during the last years there is an open exchange format for such tools, ALF [3], which is now being adopted by academic research groups and WCET tool vendors.

The WCET approach by Bernat et al. [39, 40] is however quite different. Their solution, probabilistic WCET (or pWCET), combines program analysis with execution-time measurements on basic-block[2] The execution time data is used to construct a probabilistic WCET for each basic block, i.e., an execution time with a specified probability of not being exceeded. The block pWCETs are combined using the static analysis in order to produce a total pWCET for the specified code. This is today a commercial WCET and profiling tool, RapiTime, from Rapita Systems, Ltd. [136].

The pWCET approach is not dependent on a model of the hardware, as in the case with static WCET analysis, but instead relies on execution time measurements. The dependence on a hardware timing model is a major criticism

[2] A basic -block is a sequence of unconditional instructions, i.e., without jumps.

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, meaning that the WCET estimate might be too low. Bernat et al. [39] however argue that static WCET analysis for real complex software, executing on complex hardware, is "extremely difficult to perform and results in unacceptable levels of pessimism". Static WCET analysis tools are today limited to fairly simple CPUs, while the pWCET approach is applicable on any CPU.

The pWCET approach is today implemented in the RapiTime product of Rapita Systems, Ltd. [136]. This solution can be combined with a hardware monitoring tool, the RTBx data logger. The RapiTime source code analysis adds source-code instrumentation points (IPoints), which write an identifier to a generic I/O port of the CPU, to which the RTBx is connected. According to Rapita Systems, an IPoint only require two CPU instructions. The RTBx is a separate computer, equipped with a large hard drive and a data acquisition card which samples the I/O port at a very high frequency. This solution however requires the existence of a generic I/O port, and the RTBx is a quite large and expensive device mainly intended for lab use.

2.2 Timing Analysis using Model Checking

Model checking is a method for verifying that a model (of a system) meets formally specified 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 and communication protocols. In recent years model checkers for software have been developed and proposed as method for software verification. Many case studies have been performed where defects have been identified in existing software by using model checking. This section will describe the basic concepts of model checking and temporal logics, a widely used model checking tool as well as two model checkers especially targeting real-time systems.

2.2.1 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 modeled system by 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 automata 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 complete system.

The properties that are to be checked are usually specified in a temporal logic, such as CTL [71] or LTL [72]. Temporal logics allow 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 CTL safety property is:

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

which states that A and B may never be true at the same time, using the temporal operator AG (“always”). Imagine that the logical propositions A and B describe valves in a chemical production plant, the proposition is true if the valve is open, false if closed. This formula then states that they may not be open at the same time (since that would cause, e.g., a dangerous spill). CTL contains several temporal operators, apart from AG, and is presented further in Section 2.2.3.

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 Katoen [74], 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., 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, meaning that the number of possible states in the system easily becomes very large as it grows exponentially with the number of parallel processes. This is a serious problem, as model checking tools often 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.

Another problem is the need for a detailed analysis model in a formal notation, which typically is specific for each tool. Model checking has great potential in model-driven development, where the verified model is a specification

for automatic code generation tools. However, for complex embedded systems developed in a traditional code-oriented manner, no analyzable models are available and model checking therefore typically[3] requires a major modeling effort, which is very time consuming and may introduce errors in the model.

2.2.2 The model checker SPIN

SPIN [75] is a well established tool for model checking and simulation of software. SPIN supports simulation (random, guided and interactive) and model checking of formulas in the temporal logic LTL [72]. According to the SPIN website [117], 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 by Holzmann [76] containing tutorials on using SPIN and Promela, as well as reference material.

Promela

A Promela model roughly consists 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, in the same way as in common programming languages, but if several behaviors are enabled, i.e., have guard conditions which are true, one is selected in a non-deterministic way. As an example, consider the following:

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

[3] Unless an automatic model extraction tool can be used, such as Modex (cf. Section 2.4.3) which generates models for the model checker SPIN (cf. Section 2.2.2). SPIN is however not suitable for timing analysis of real-time systems, since it does not have a notion of quantitative time.

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 larger than 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, which blocks until the send and receive operations can occur 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 straightforward. 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, i.e., like the "main" function in common programming languages. The atomic-statement creates a critical section, which ensures that the contained statements executes sequentially, without preemptions. The two "run" commands creates two new processes, which are released at the same time, when leaving the atomic section.

LTL

One way[4] to specify the properties for SPIN to verify is linear temporal logic (LTL), which is classic propositional logic extended with temporal operators. Using LTL for program verification was first proposed by Pnueli [72]. The LTL operators supported by SPIN are:

```
[]  always              &&   logical and
<>  eventually          ||   logical or
!   logical negation    ->   implication
U   strong until        <->  equivalence
```

As an example, the following LTL formula specifies that the logical proposition L should remain true until E becomes true:

```
[](L U E)
```

The logical propositions L and E could be electrical signals, e.g., in a washing machine, where L is true if the door is locked, and E is true if the machine is empty of water, and thereby safe to open. The LTL formula in the above example then means "the door must never open while there is still water in the machine".

2.2.3 Model Checking for Real-Time Systems

Model checkers such as SPIN do not have a notion of quantitative 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 [129] and KRONOS [131], both described later in this section. These tools analyze models described in *timed automata* using variants of the temporal logic CTL.

[4] Another method is to insert "assert" commands in the Promela model.

Timed Automata

The concept of timed automata was first proposed by Alur and Dill [82], who 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 [81]. 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, from the UppAal tool.

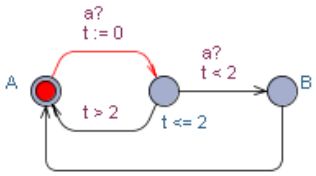

Figure 2.1: A small example of (UppAal) timed automata

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 use variants Computation Tree Logic [71], or CTL, which is a branching-time temporal logic. This means 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 cannot 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* [87], 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.

Apart from ordinary propositional logic, CTL contains four temporal operators:

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

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

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

UppAal

The tool UppAal [77, 78, 79] 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 has now (2010) reached version 4.0. The tool is available for many platforms including Windows and Linux, and can be downloaded without charge from the UppAal website [129].

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, which allows for 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.

TIMES

Times [122] is a tool set for modeling, simulation, schedulability analysis and synthesis (code generation) of systems that can be described as a set of periodic or event-triggered tasks. The Times tool was first proposed by Amnell et al. [13] and allows for specifying both the triggering mechanisms of tasks

and the internal behavior of tasks using timed automata models, extended with data variables and the notion of tasks [57]. The verification parts of Times is based on UppAal, developed by the same group. The Times tool can synthesize source code from the developed models, but currently only for the LegoOS operating system. This tool can be regarded as a special version of UppAal for real-time systems analysis.

KRONOS

Another model checker for real-time system is Kronos [81, 80] 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 [83], or TCTL, an extension of CTL to include quantitative time for the purpose of specifying timeliness properties, i.e., liveness properties with a deadline. Kronos also allows for checking safety properties and 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 [131].

2.3 Timing Analysis using Simulation

Another method for analysis of response times of software systems, and for analysis of other properties, is the use of discrete event simulation, or simulation for short. 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 [103], ARTISST [88] and VirtualTime [136] simulation frameworks, outlined below. The rich modeling languages allow modeling of the semantic dependencies between tasks in the system, e.g., communication, synchronization and shared state variables. Simulation models may contain non-deterministic or probabilistic selections. By using probabilistic selections, task execution times can be modeled with high realism, as probability distributions.

A problem with simulation is the lower confidence in the result in comparison to formal or analytical methods. A simulator executes the model and randomly explore the possible execution scenarios. Even though it is possible to perform a large amount of simulations in short time, the number of possible execution scenarios, i.e., the state space, is often too large for an exhaustive

analysis. On the other hand, simulation allows for an analysis, even though not exhaustive, in situations where other analysis methods fail.

2.3.1 STRESS

The STRESS environment [102] is a collection of tools for analysis and simulation of hard real-time applications, based on a special-purpose modeling language, essentially a procedural programming language, in which the behavior of the modeled system is described. It is focused on tasks and intended as a tool for testing various scheduling and resource management algorithms. It can also be used to study the general behavior of applications.

2.3.2 DRTSS

The DRTSS simulation framework [103] allows for easy construction of discrete event simulators describing complex distributed real-time systems. Compared to STRESS, DRTSS is a more generic tool, not only focusing on hard real-time. Unlike STRESS, DRTSS does not define an own modeling language for behavior, but instead models the task behaviors using executable code expressed in C, C++ or any other language which can be linked with C++. The framework consists of three major components: a search controller, which selects parameters for each individual simulation run, an execution engine and SETI, “the System for Extraction of Timing Information”, which analyzes the simulation output. The DRTSS framework is a member of the PERTS family of timing-oriented prototyping and verification tools, which also contain tools for analytical schedulability analysis.

2.3.3 ARTISST

The ARTISST simulation framework [88] targets performance evaluation of “complex dynamic real-time systems made of tasks performing arbitrary computations and exhibiting a complex and realistic pattern for their arrival law, synchronization relations, and execution time.”, which is essentially also the focus of all other known simulation frameworks. Like STRESS, the ARTISST solution was initially intended for evaluation of different scheduling algorithms. The specific solution is very similar to VirtualTime and RTSSim (cf. Chapter 3). ARTISST is task-centric and allows for specifying behavior of tasks in C or C++. Time is advanced in an explicit manner using an API

function called “hold_cpu”, which has direct correspondence in both VirtualTime and RTSSim. The authors behind ARTISST emphasize its modular, object-oriented design; it is not dedicated to a particular operating system but fully customizable allowing for simulation of systems using different operating system APIs. The name ARTISST is a recursive acronym, “ARTISST is a Real-Time System Simulation Tool”.

2.3.4 VirtualTime

An example of a commercial simulation framework is VirtualTime [136]. 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. [136], there are few limitations to the models that can be produced using VirtualTime. VirtualTime is specifically targeting the OSE operating system from ENEA [134], mainly used in the telecom domain. Rapita Systems, Ltd. is a spin-off company from the Real-Time Systems Research Group at the University of York, UK.

2.4 Modeling using Source-code Analysis

This section describe works related to automated analysis of software systems. For large industrial systems, it is not realistic to construct detailed analysis models by hand; the models must be generated using automatic analysis tools. There are two primary information sources when analyzing existing systems: the source code itself and measurements of the run-time system. This section presents related works in source code analysis, while Section 2.5 presents related works in *dynamic analysis*, i.e., analysis of information recorded during run-time. Some solutions uses both static and dynamic information. This section presents results from three fields of research: program slicing, reverse engineering and formal verification of source code.

2.4.1 Program Slicing

Program slicing, first proposed by Weiser [20], is a type of program analysis which identifies the statements of a program of relevance for a particular slicing criterion, typically the value of a certain variable at a particular point in

the program. This analysis is highly relevant for automated extraction of simulation models from source code and is used for this purpose in the approach presented in Chapter 5.

The most common type of program slicing is *backward slicing*, the process of identifying all statements which might affect a particular variable, typically at a particular point in the program. A less common analysis is *forward slicing*, which identifies the statements which might be affected by a particular variable. Unless otherwise stated, the term *program slicing* hereafter refers to backwards slicing.

An example of program slicing is given in Figure 2.2. The example code is a C function which counts the number of characters and line-breaks in a string. The code in the left part is the original program, while the right side code is the backwards slice with respect to the variable *lines*, used in the last *printf* call (in blue text). In the slice illustration, red statements are those of direct relevance for the slicing criterion, i.e., modifications of *lines*, while the remaining code corresponds to indirectly relevant statements.

```
void count(char* text)
{
   int i = 0;
   int lines = 1;
   int chars = 0;
   while (text[i])
   {
      if (text[i] == '\n')
         lines = lines + 1;
      else
         chars = chars + 1;
      i = i + 1;
   }
   printf("chars: %d\n", chars);
   printf("lines: %d\n", lines);
}
```

```
void count(char* text)
{
   int i = 0;
   int lines = 1;

   while (text[i])
   {
      if (text[i] == '\n')
         lines = lines + 1;


      i = i + 1;
   }


}
```

Figure 2.2: An example of program slicing

The approach to program slicing proposed by Weiser [20, 26] is an iterative process operating on the *control-flow graph* of the program and produced backwards slices of the program. A control-flow graph (or CFG) is a representation of a program as a directed graph, where the vertices represents the program´s

statements and the edges represent possible control flow. This first approach to program slicing was however restricted to slicing within a single subroutine (i.e., intraprocedural slicing), and did not address issues such as data structures or pointers. During the 1980´s there were however many results following up Weiser's first approach. Weiser later proposed an extension [25] to his original approach, which allowed for *interprocedural* program slicing, i.e., slicing of more realistic programs divided up in several subroutines (functions). Leung and Reghbati later proposed a set of corrections of Weiser's extended approach [24].

In 1988, Horwitz, Reps and Binkley proposed the *System Dependence Graph*, SDG, as a base for program slicing [23]. The SDG is an extension of the *Program Dependence Graph*, first proposed by Ottenstein and Ottenstein [35, 36]. The main difference is that the SDG allowed for representing call dependencies between functions, while the PDG represented each function/procedure separately and thereby only allowed for intraprocedural slicing. Both the PDG and SDG represent a program as a directed graph, with vertices corresponding to statements and edges representing control-flow and data-flow dependencies. In the SDG there are also special vertices and edges representing the data flow between function call arguments and the corresponding formal parameters of the callee, and similar for data flow through function return values. On this representation, program slicing is realized through a reachability search starting from the specified program point, i.e., SDG vertex. An illustration of a SDG is given in Figure 2.3. The corresponding source code is:

```
void main()
{
   int sum = 0;
   int i = 1;
   while (i<11)
   {
      sum = add(sum,i);
      i = add(i, 1);
   }
   printf("sum=%d\n", sum);
   printf("i=%d\n", i);
}

int add(int a, int b)
{
   return (a+b);
}
```

In Figure 2.3, the blue arcs represent control dependencies, while the green arcs

represent data dependencies. The dotted green arcs represent interprocedural dataflow dependencies, through function parameters and return values.

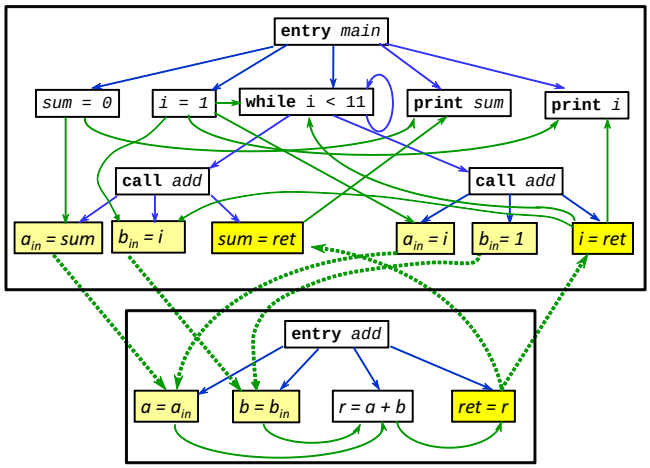

Figure 2.3: An example of a System Dependence Graph (SDG)
(Published with permission from GrammaTech, Inc. [123])

The SDG representation was patented in 1992 [22]. The patent however only concerns the construction of the SDG, not the actual slicing algorithm. An improved version on the SDG slicing method [21] was used in the commercial tool CodeSurfer, developed by GrammaTech, Inc. [123]. According to the research group website of Horwitz and Reps [121], the CodeSurfer tool scales to maximum 200 000 lines of code.

In *conditional program slicing* [30], additional input is provided in the form of known facts about variable values, which results in smaller slices. This can be regarded as something between traditional static slicing and dynamic program slicing methods, which uses recorded execution history (cf. Section 2.5).An implementation of a conditioned program slicer, ConSIT, has been presented by Danicic et al. [31]. ConSIT uses traditional static slicing together with symbolic execution and theorem proving.

Amorphous program slicing was proposed by Harman et al. [32]. An amorphous slice of a program only preserves the semantics, not the syntax. The idea is to produce a simplified subprogram, which is semantically identical to the original program with respect to a set of selected variables. Amorphous pro-

gram slicing is more like program transformation than traditional static slicing.

By relaxing the requirement on preserved syntax, smaller slices can be obtained. This is not suitable when the correlation with the original code is important, like in debugging, but can be an efficient technique for purposes such as program comprehension and reengineering. A method for amorphous slicing was later proposed by Harman et al. [29]. The method is interprocedural, unlike earlier proposed methods for amorphous slicing, and operates directly on the abstract syntax tree (AST) representation of the program. The amorphous slicer is part of an analysis framework called GUSTT.

Sandberg et al. [27] presents an alternative method for program slicing. The overall purpose of this work is to speed up WCET analysis through program slicing, by removing statements which cannot affect the program flow. The paper introduces an alternative method for program slicing, named SimpleSlice. Starting from a set of variables, a fix-point iteration is performed over the assignments in the code, where all statements possibly affecting the variable are added to the output set. SimpleSlice requires as input lists of all assignments and all variables in the code. SimpleSlice handles pointers in the same way as most other slicing methods; it is assumed that a *points-to analysis* has been performed and produced a *points-to set* for each pointer and program point. The points-to set for a specific pointer contains all variables possibly pointed to by the pointer. A common method for points-to analysis is the method proposed by Steensgard [73], which is interprocedural and fast; it has almost linear time complexity with respect to the program size. It does however not take control-flow into account. A more accurate method is the one proposed by Andersen [86], which however is considerably slower.

The SDG-based slicing methods are typically *control flow-sensitive*, which means that they can exclude irrelevant assignments of relevant variables. The SimpleSlice approach is however *flow-insensitive*, which means that it does not analyze the control flow in order to exclude irrelevant assignment, but instead includes all assignments of relevant variables. This is less accurate, but much faster and gives a less complex implementation. SimpleSlice treats data structures and arrays as single variables, for which all assignments are treated equally, independent of referenced field or index. The SimpleSlice approach is only presented for intraprocedural slicing.

Another approach to program slicing is the work presented by Bent et al. [69, 70, 68], implemented in a prototype tool called *Sprite*. It has been developed with scalability as a primary concern. It represents the control-flow of each function in an intraprocedural manner and uses a separate call-graph for representing interprocedural control flow. The data-flow analysis is per-

formed using the control-flow graph (CFG) representation. Like SimpleSlice, the Sprite tool uses the Steensgard algorithm [73] for pointer analysis. An interesting aspect of this tool is that it computes all program representations on demand to the extent this is possible. The CFGs and the points-to information are calculated on the first slice computation, while control-dependencies and data-flow information is calculated on demand during the slicing. Sprite is part of a larger package called ICARIA, which is a C specific instance of a generic program analysis system called PONDER. In [69], it is compared with CodeSurfer [123] with respect to slice size and runtime.

Espresso [65] is described as a *slicer generator* which, given a program to analyze outputs a multi-threaded Java program which produce static slices for the program at hand. Due to the multi-threaded approach, the solution benefits from multi-core CPUs, which allows for parallel processing. Each Java thread correspond to a node in the CFG of the program to analyze, and communicates with other threads according to the edges of the CFG. The messages sent between the CFG nodes (Java threads) contains variable names and node IDs. Espresso assumes that all expressions are free from side-effects, and that there are no unstructured control-flow due to goto statements.

Jackson and Rollins [67] proposed the concept of *Chopping* as a generalization of program slicing, where the slicing (or chopping) criterion contains two sets, *sinks* (uses) and *sources* (definitions). Chopping a program means to identify all statements involved in dependencies from sources to sinks. A backwards slice corresponds to a "chop" of the program where the sink corresponds to the traditional slicing criteria (a variable at a certain program point) and where the source set contains all statements of the program. An interprocedural method for chopping was later proposed [66].

For further reading, a good start is the survey on program slicing techniques by Tip [18] and the more recent slicing overview by Xu et al. [8].

2.4.2 Reverse Engineering

The process of extracting information from an implementation is commonly referred to as *reverse engineering*. A related term is *reengineering*, which according to the "horseshoe model of reengineering" [42] is the process of reverse engineering an implementation into a higher level of abstraction, *restructuring* the result of the reverse engineering, and finally *forward engineering* in order to implement the new requirements and/or new architecture. An extensive annotated bibliography is presented by van den Brand [43] describing around 100 works in the area of Reengineering and Reverse Engineering.

There are several tools and results focusing on reverse engineering for improved comprehension of software. Bellay and Gall [44] presented a comparison of 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 supported source languages 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 displays 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.

The Bellay and Gall paper [44] references a tool called Refine/C, "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.

The second tool presented in the Bellay and Gall paper [44] is Imagix 4D (cf. Chapter 6). This is a tool for analysis of C and C++ programs and is a commercial product from Imagix Corp. [135]. This tool can, among other things, perform control flow analysis and program slicing (the "Calculation Tree" feature). It can identify unused variables, present metrics of the individual routines in the code, such as line count, McCabe complexity, fan in, etc. The Imagix tool also allows for using 3D visualizations, which in many cases can produce a more compact view, for, e.g., function call graphs.

The third tool studied by Bellay and Gall [44] 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 datatypes 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+, a commercial development environment from Wind River [132], supporting reverse engineering activities. Sniff+ targets embedded solutions but is no longer maintained by Wind River.

A study by Kollmann et al. [45], 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, such as the number of classes reported.

Reverse engineering tools of a more lightweight nature are *Revealer* [47] and *Semantic Grep* [48]. 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 in the traditional sense, e.g., by building a parse tree, instead it performs a high-level scan for syntactic 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 [48] allows queries on the source code, for instance "*show all functions in parser.c*" or "*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.

Understand, from Scientific Toolworks, Inc. [138], is a reverse engineering tool available for a large set of programming languages, including C and C++. This tool allows for studying control-flow, inside and between functions. The tool can also present header-file relationships, where variables are defined and used and many other similar tasks. The tool is focused on assisting maintenance of large software systems and provides open, well documented APIs in Perl and C, which enable implementation of custom analyses. Understand does however not provide support for program slicing or other forms of data-flow analysis, like CodeSurfer or Imagix 4D does. The strength of this tool is its performance on parsing large amounts of code, together with flexibility and extension possibilities through the open APIs. A performance evaluation of Understand is presented in Chapter 6.

2.4.3 Formal Verification Tools using Source Code Analysis

There are many works related to reverse engineering in the area of formal verification of programs. There are model checkers for software which 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 [51], BLAST [52], FeaVer/Modex [55] and Bandera [53].

SLAM is a toolkit developed at Microsoft Research for checking safety properties of system software. A case study has been presented [51] where the SLAM toolkit has been used to verify Windows NT device drivers. SLAM 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 non-deterministic selections. 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 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 [52], is another solution for checking safety properties of C programs. The safety property to check is specified by adding a special *error location* 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.

According to Henzinger et al. [52], BLAST has been used in case studies to verify safety properties of, e.g., Windows and Linux device drivers. In some cases, defects have been found and in other cases BLAST proved that the drivers correctly implemented a specification.

An interesting result is the tool Modex [55, 59], a model extractor for the SPIN model checker described in Section 2.2.2. There is a name confusion regarding this model extraction tool. Modex is an acronym of Model Extractor, a tool for extracting verification models from ANSI C. Modex was previously known as AX (Automata Extractor). FeaVer is the user interface of Modex. 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 effort from manual modeling to defining the table of rules, which specify patterns for what statements that should be included in the model (Promela allows for including C statements) and what to ignore. There are standard rules that can be used, but the user may add their own rules to improve the quality of the resulting model.

Bandera [53] 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 [117] and SMV [133]. The authors 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 program slicing 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 important since 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, this is feasible.

A different approach to model checking is the one used in VeriSoft [54]. It is not a traditional model checking tool, in the sense that no explicit model is required. VeriSoft instead 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. Even though Linux is growing as platform for embedded systems, this solution is not possible to apply on closed platforms, such as VxWorks [132] or OSE [134].

2.5 Modeling using Dynamic Analysis

The use of dynamic analysis techniques for the modeling of complex embedded systems is interesting as the approach allows for capturing realistic timing information. This kind of information is generally not possible to obtain using static analysis, at least not for more advanced hardware platforms. As discussed in Section 2.1.2, static WCET analysis is restricted to rather simple, low performance CPUs. Moreover, research in this area focuses on providing safe upper bounds for the *worst case* execution time, but also *typical* execution times are of relevance for performance analysis.

One interesting study is the one presented by Marburger and Westfechtel [28]. They report on a set of reverse engineering tools, developed in cooperation with Ericsson Eurolab Deutschland, which include 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 impact analysis, unfortunately this study focuses on telecom system and the Ericsson-specific PLEX language.

Related to the Marburger and Westfechtel work is that of Systä and Koskimies [34] where state diagrams are synthesized from traces. The source code of the system in focus is instrumented in order to generate a trace, which is fed into the SCED tool which in turn generates a minimal state diagram corresponding to the observed behavior. The work does however not address real-time systems, as no timing information is recorded.

A system called DiscoTech was presented by Yan et al. [46], which based on run-time observations generates an architectural view of the system. 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 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.

Jensen [49, 116] proposed a solution for automatic generation of behavioral models from recordings of a real-time systems behavior, i.e. model synthesis

from traces. The resulting model is expressed as timed automata for the UppAal model checking tool [77, 78, 79]. The aim of the tool is verification of 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, which instead of WCET uses a measure called Reliable Worst Case execution time, or RWC, a statistical measure 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 (CPUs). The allocation of subtasks to servers is derived from requirements such as periodicity and deadline. Thus, each *job* of a task is a sequence of interactions between *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. According to the author´s own experiences of trace recording and execution time measurement on real industrial systems (cf. Chapter 7), task-level execution times often follow complex multi-modal distributions, with several "peaks", corresponding to different control-flow branches. The distributions are rarely normal distributed.

Another type of dynamic program analysis is dynamic program slicing, proposed by Korel and Laski [33]. In contrast to program slicing based on source code analysis, dynamic program slicing is performed using a trace describing a specific execution of the program. Since the resulting slice only takes a specific execution into account, the slices are typically smaller. This approach is mainly intended for facilitating debugging. A potential problem with this approach is that the recorded execution history (the trace) often becomes very large, since many details must be recorded.

2.6 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.

The validity of models have been studied in the simulation community. Law and McComas [60] define model validation as "*the process of determining whether a simulation model is an accurate representation of the system, for the particular objectives of the study*". They address validation of simulation models in general, e.g., of models describing a physical process. A book by Law [63], on simulation studies, includes one chapter on model validation which 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 results 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.

Balci [61] presented guidelines for simulation studies, including a simulation study life cycle with 10 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 Balci [61] 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 Balci [61] and the author's own 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, which is not always possible or practical.
- Sensitivity Analysis: This implies to systematically apply changes to the model or model input variables and observing the effect on model behavior. The idea is that unexpected effects may reveal flaws in the model. This analysis is further discussed in Chapter 8.
- 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.

Balci [61] 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 by Sargent [62]. 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. A simplified version of Balci's modeling process is proposed, 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.

Law and McComas [60] discuss many aspects of the validity of models in general and describe a seven-step approach for conducting a successful simulation studies. This approach is specified on 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 emphasizes the importance of a definite problem formulation, comparisons between the model and the system, and the use of sensitivity analysis.

2.7 Conclusions

Many analytical methods for response-time analysis (RTA) have been proposed in research literature. However, the system models used by such methods are not expressive enough in order to capture the behavior of large and complex systems. They do not consider the behavior of the tasks, only their individual worst-case execution time, which can make it very pessimistic for large industrial systems as their software architectures often violate the assumptions of analytical methods regarding independence between tasks. RTA analyzes the "critical instant", i.e., the worst case scenario when all tasks attempts to execute at the same time, and with their individually highest execution times. If this situation can be managed, the system is truly safe. However, on complex embedded systems, this scenario might not be possible, or so extremely improbable that it will not occur in practice. For instance, the individual worst case execution times of different tasks may be mutually exclusive if they depend on different states of the same shared state variable. For systems which violate the RTA assumptions, only a positive RTA result is useful (i.e., that response times are below deadline requirements), while a negative result does not say much, since it is not known if the analyzed worst case scenario is feasible.

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 possible to derive task deadlines from such requirements, but this is often difficult. 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 cannot 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 due to some application specific condition.

Compared to the simplistic system models used in RTA, model checking allows for using detailed system models expressed in rich modeling languages such as Promela [117, 76] or timed automata [82, 79], where functional behavior and task dependencies can be specified. Timed automata moreover allows for the modeling of real-time systems, where a notion of quantitative time is

important. However, model-checking does not scale properly to larger systems due to the state-space explosion problem.

Another problem with model checking is the need for a detailed analysis model, which describes the relevant aspects of the system in a formal notation, which typically is specific for each tool. The exception is the model checker SPIN [117, 76], which has support for Promela model extraction from C code (the Modex tool), but SPIN does not support timing analysis of real-time systems, since it has no concept of quantitative time, only relative event order.

Unless an automatic model extraction tool is available, using model checking on large, existing systems implies a major, error-prone modeling effort, and in the end, the state-space explosion problem might make the resulting model useless if it cannot be analyzed with realistic memory and run time constraints. Model checking is today mainly used for small systems with extreme requirements on dependability, where the consequences of a failure are catastrophic. One can argue that model checking will be possible on more and more complex systems as computers are getting faster and faster, and model checking tools better and better. That is true, but the systems out in industry also benefit from the trend of ever faster CPUs as a result grow larger and larger.

The custom modeling required makes formal methods quite expensive to use and it might not give the best return on investment in terms of software quality. Even though timing analysis is important for many companies, most of their quality problems are likely due to “ordinary” (functional) errors. For such companies, the value of timing analysis is proportional to the number of system failures it prevents. Investments in increased software quality through timing analysis must be economically motivated compared to other software quality investments, like refactoring, improved test frameworks or extended code reviews.

Discrete event simulation is another approach which like model checking also allows for using detailed models of the system. Simulation does not suffer from the state-space explosion problem, at least not in the same way, since no exhaustive search is performed. The state-space is instead randomly sampled. A disadvantage of the simulation approach is the lower confidence in the result. No guarantees can be given regarding the properties of the analyzed models since the whole state-space is typically not explored. In fact, the size of the state-space or the number of explored states are typically not known since they are not represented explicitly. The confidence issue is obviously a larger problem for models with very large state-space and can be seen as the state-space explosion problem applied to the simulation approach.

Simulation is more like testing in the sense that it can show the presence

of (timing) errors, but not guarantee their absence. However, even though a simulation is not an exhaustive analysis and thus might miss the worst case situation, it may still point out potential problems and assist the developers in making the right decisions.

Based on this analysis, the author decided to focus on the possibilities of using simulation-based analysis, an area that has received little attention. In the general case simulation suffers from the same modeling problem as model checking and has confidence problems instead of scalability problems. Possible solutions have however been identified and are addressed by the research questions of this thesis.

Research question **Q1** address the modeling issue, concretely the possibility of automated model extraction. Chapter 5 proposes a method for automated model extraction, Katana, using a new approach to program slicing. In response to research question **Q2** two methods for *simulation optimization*, which improves accuracy and confidence, are presented in Chapter 4.

With these contributions, simulation models can be automatically generated from an existing system and analyzed in an efficient manner in order to quickly assess the typical performance or to identify extreme scenarios which may constitute timing errors. This approach is therefore quite cheap to use in terms of man hours required for training and application, especially compared to formal analysis methods. Imagine that timing errors constitute 5 % of all errors for a system, and 80 % of these are found using a simulation-based approach, compared to 100 % when using formal methods such as model checking. This means that the advantage of using model checking compared to simulation would be only a 1 % increase in the ratio of detected errors, while the cost to achieve this would be large even for *small* projects, several hundred hours, and extremely high for larger projects. This automated approach to simulation-based timing analysis should be realistic to deploy in industrial settings and could in that case improve software quality and reduce costs.

Program slicing is a highly interesting techniques for extraction of simulation models from the source code of existing systems. The desired simulation model can be regarded as an executable slice, with respect to a specific slicing criterion. However, the existing tools for program slicing, such as CodeSurfer [123] and Imagix 4D [135], do not scale sufficiently. As presented in Section 5.5, the scalability of such tools are limited to systems with at most 200 000 lines of code (as stated by the research group behind the CodeSurfer tool [121]). According to experiments presented in Chapter 6, the practical applicability is however even more restricted, and industrial systems often consists of many hundred thousands of lines of code, or even millions [106].

There are however reverse engineering tools which scale much better, but which offers less functionality. For instance, Understand [138] scales to several millions of lines of code (as presented in Chapter 6), but has no support for program slicing. Many reverse engineering tools have however APIs, which allows for implementation of custom algorithms. For instance, the API of Understand has been used to implement the MXTC prototype presented in Chapter 6.

Timing-accurate simulation also requires quantitative data from the system, e.g., execution times. Profiling of embedded systems is however a fairly mature area, and apart from software solutions (e.g., those described in Chapter 7) there are also hardware solutions which allows for days or weeks of continuous execution time recording, such as the RTBx data logger from Rapita Systems, Ltd. [136]. Such a solution could be used for populating simulation models with execution time data.

The advantage of software recorders is the possibility of performing execution time recording at all times, also post-release, since no extra hardware is required. This imposes an overhead, i.e., additional CPU and RAM usage, and amount of RAM available limits the amount of data which can be stored. Research question **Q3** was therefore formulated to verify the practical applicability of this approach on common operating systems in the embedded systems domain and to investigating the overhead of trace recording on real systems. The results are presented in Chapter 7, based on experiences from five industry collaboration project where trace recorders have been developed and evaluated.

Model validation is the process of assuring that a model correctly describes the intended system. Techniques for model validation are important also when using automated model extraction, as the model validation also verifies the model extraction and simulation tools and their configuration. Moreover, the coverage of the execution time measurements is another validity threat which needs to be investigated as a part of the model validation process. If manual abstractions are allowed, e.g., through code annotations, model validation is critical in order to ensure the validity of the abstractions made. Works exist regarding validation of models in the general 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 reverse-engineering activity (automated or manual), the model validity cannot be assumed. The results found in the simulation community include two main classes of model validation techniques, subjective techniques (i.e., in-

spection) and those based on statistics. Both can be used to validate models in this approach. Chapter 8 presents a five-step process for model validation which should be suitable for this approach.

Chapter 3

Timing Analysis using Discrete Event Simulation

This chapter presents discrete event simulation as a method for timing- and performance analysis of complex embedded systems and introduces the simulation framework *RTSSim* developed for this purpose. This type of analysis can be used for predicting any run-time property where the logic involved is implemented in software and available as source code. Run-time properties strongly dependant on the hardware architecture, e.g., cache hit ratio, are not supported or targeted by this work.

Simulation is a broad term which easily can be misunderstood, even in the context of analysis of embedded systems. Simulation is the process of imitating key characteristics of a system or process, and typically implemented as computer programs. One type of simulation is used during design of physical structures, e.g., for predicting wind forces for a bridge. Such a simulation model consist of mathematical equations. However, in the context of this work, the term *simulation* implies *discrete event simulation*, unless otherwise stated.

Law and Kelton [63] defines discrete event simulation as *"modeling of a system as it evolves over time by a representation in which the state variables change instantaneously at separate points in time"*. This naturally includes simulation of computer-based systems.

Simulation can be performed on different levels of abstraction. In one end of the scale, simulators such as Virtutech Simics [124] are found, which simulates software and hardware of a computer system in detail. Such simulators are used for low-level debugging, where a very detailed view is necessary, or

for hardware/software co-design, i.e., when developing software for new hardware that is not yet physically available, but can be modeled using a simulator. This type of simulation is considerably slower than normal execution, typically magnitudes slower, but gives an exact analysis which takes every detail of the behavior and timing into account.

In the other end of the scale we find scheduling simulators, who abstract from the actual behavior of the system and only analyzes the scheduling of the system's tasks, specified by scheduling attributes and execution times. One example in this category is the approach by Samii et al. [12]. Such simulators are applicable for strictly periodic real-time systems. However, complex embedded systems often contain aperiodic tasks, triggered by messages from other tasks or interrupts. Moreover, tasks may have different behaviors, and thereby execution times, depending on message content. A simulator must therefore take relevant aspects of the task behavior into account in order to accurately simulate a complex embedded system.

In the simulation approach presented in this chapter, RTSSim, the source code of the analyzed system is used as a base for constructing a simulation model, expressed in the same programming language as the original system. In the domain of embedded systems this typically implies C/C++. RTSSim has therefore been developed in C, which allows for both C and C++ models.

Even though the implementation details of RTSSim may be of interest for some people, RTSSim is not proposed as a novel contribution in itself, at least not conceptually. Similar approaches are "Virtual Time", from Rapita Systems, Ltd. [136] and the ARTISST simulator [88]. Additional works in the area can be found in Section 2.3. The purpose of this chapter is mainly to give the reader an understanding of the type of simulation in focus and thereby to set the context for the following chapters which present results in simulation optimization, simulation model extraction and model validation.

3.1 Motivations for Simulation

Compared to other methods for timing analysis, simulation has the advantage of not posing restrictions (by making assumptions) regarding the design of the software system. Simulation allows for analysis of any measurable runtime property, in contrast to the analytical methods for response time analysis [45, 47, 46] which have many assumptions and are specialized for a specific property, task response times.

Simulation does not have the state-space explosion problem in the same

way as model checking tools, like UPPAAL [48] or KRONOS [49]. Such tools attempt to search the model state-space exhaustively, which for large industrial systems will require more time and memory than realistically available. In contrast, simulation is a best-effort analysis which randomly explore the possible behaviors of the model for as long time as allowed. A simulation-based analysis can therefore not identify worst-case scenarios, since only a random subset of the state-space is explored. Note that the worst-case scenario might have been encountered during a simulation, but the simulation result does not tell if this is the case.

The state-space explosion problem exists in the context of simulation as well, but due to the best-effort approach instead manifests in lower state-space coverage and thereby simulation results of lower confidence. This is however better than no results at all.

Simulation using randomized input, i.e., *Monte Carlo simulation*, will mainly explore the typical behaviors, while rare, extreme scenarios are less likely to be found. The efficiency of using simulation for finding extreme behaviors can however be significantly increased through *simulation optimization*, which is presented in Chapter 4.

In the perspective of analyzing an existing software system, simulation does not require formal modeling, like most model checking tool does (at least those targeting real-time systems), since a simulation model can be automatically extracted from the system implementation, e.g., using the method presented in Chapter 5.

Simulation-based analysis should be regarded as a form of specialized testing, in this case focusing on timing and resource usage, and used as a complement to traditional testing. Since a simulation model only need to include the code of relevance for the properties in focus, and since a PC is typically much faster than embedded hardware, simulations can be performed in much less time than required to run the corresponding test cases on the real system, which means that more scenarios can be explored in the same time. If using a PC with a multi-core CPU, multiple simulations can be executed in parallel. Moreover, a simulation can explore scenarios which are hard to generate on a real system, can be extensively monitored without causing probe effect problems[1] (discussed by Schutz [89]), and a simulation is completely deterministic and reproducible, as explained in Section 3.2.6. In contrast, for multi-tasking systems, system-level testing is not always deterministic and repeatable since execution times and input timing varies between test runs.

[1] A probe effect is an accidental change in system behavior due to altered execution times when activating or deactivating the recording.

3.2 The RTSSim Simulation Framework

RTSSim was developed for the purpose of simulation-based analysis of run-time properties related to timing, performance and resource usage, targeting complex embedded systems where such properties are otherwise hard to predict. RTSSim has been designed to provide a generic simulation environment which provides functionality similar to most real-time operating systems.

The RTSSim simulation framework allows for simulating an embedded software system on a standard PC, many times faster than normal execution on the embedded hardware (i.e., testing), with approximately correct timing and with complete control and reproducibility.

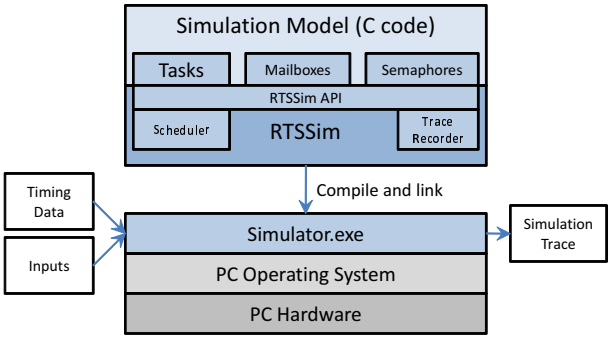

Figure 3.1: The RTSSim framework

As depicted by Figure 3.1, a simulation is performed by executing the simulation model in the RTSSim environment, which emulates a real-time operating system on a PC and works as a "sandbox" with respect to timing. The real timing of the simulator executable, which naturally is affected by the host PC, does not impact the simulated timing (i.e., the simulation result) since CPU usage is modeled explicitly. All time-triggered events in RTSSim are controlled by a simulation clock, which is incremented by explicit *Execute* statements in the simulation model (cf. Section 3.2.3), using timing data recorded from monitoring of the modeled system. The timing of the modeled system is thereby preserved in the simulation, or at least a good approximation. This is however not guaranteed to be 100 % identical to the real timing, i.e., when executing the code on the intended hardware. This is the case since the simulation model is probabilistic with respect to execution times; it abstracts from details in the

hardware platform and instead describe the execution time between relevant points in the source code in a probabilistic manner. An approach for execution time profiling and modeling, for this purpose, is presented in Section 7.5.

The simulation input decides the simulation length and the outcome of any stochastic selections in the model, e.g., execution-time variations (cf. Section 3.2.6). The input which decides stochastic selections be a seed value, used to initialize a pseudo-random number generator (cf. Section 3.2.7, or a data set specifying the outcome of each stochastic selection individually.

The tasks of an RTSSim simulation model are scheduled using preemptive fixed-priority scheduling, as described in Section 3.2.4, and are assumed to share a single CPU core.

A simulation model can theoretically contain the full source code of the analyzed software system, but the intension of the RTSSim framework is to use it together with a *model extraction* tool, which produces a simulation model only containing the source code of relevance for the properties in focus, together with added *Execute* statements. The execution time of the excluded code is still taken into account by the simulation model since the execution time measurements are performed on the original system. A method for simulation model extraction is presented in Chapter 5 and an evaluation of this approach on industrial code is presented in Chapter 6.

RTSSim can produce two kinds of output, a detailed simulation trace and a text file containing selected statistics on task timing, such as highest response time observed for the selected task. The simulation trace is produced using a *trace recorder*, i.e., an event logger, very similar to the one described in Section 7.4.5, which outputs a simulation trace for the Tracealyzer tool, presented in Section 7.3, including task scheduling, task communication and synchronization events. The trace recorder in RTSSim also supports “user events”, i.e., user-specific events and data, logged through explicit calls to the trace recorder from the simulation model.

An RTSSim simulation is performed by compiling and linking the simulation model, which is expressed in C code, together with the RTSSim library and running the resulting executable, as depicted by Figure 3.1. The RTSSim framework is implemented in C using the Win32 library of Windows XP, and has only been tested in this environment. Porting to other operating systems is possible, but requires that the current use of Win32 fibers (cf. Section 3.2.4) is replaced with a more portable solution, e.g., by using the POSIX thread library.

3.2.1 The Simulation Model

An RTSSim simulation model is focused on tasks, which in C describe behavior of relevance for timing, i.e., timing, scheduling, communication, synchronization and relevant state changes. The current RTSSim implementation allows for using tasks, mailboxes and semaphores, but other operating system features can quite easily be added since the core functionality is in place. The RTSSim API is presented in detail by Appendix B.

Tasks

RTSSim stores each task in a list of *task control blocks*, or *TCBs*, which include the following attributes:

- Name: an identification string used for logging purposes.
- Status: READY, BLOCKED, WAITING or DORMANT.
- Priority: an unsigned 8-bit integer, where 0 is the highest priority and 255 the lowest priority.
- Period: the (minimum) inter-arrival time of the task.
- Offset: the activation time of the first instance.
- Jitter: allows for adding a stochastic inter-arrival time jitter.
- Entry function: the main function of the task.

The task scheduling of RTSSim is described in greater depth by Section 3.2.4.

Mailboxes and Semaphores

A *mailbox* is used for asynchronous message passing between tasks and contains a fixed-size FIFO buffer where the messages are stored. A message is a 32-bit integer value, typically a message code or a pointer to a data-structure. A *semaphore* is a basic binary semaphore for mutual exclusion, which initially is unlocked. No resource management protocol has been implemented for preventing priority inversion.

Mailboxes and semaphores may block the execution of tasks. An attempt to send a message to a full mailbox, or to receive a message from an empty mailbox, or an attempt to lock an already locked semaphore will block the executing task until the operation is successfully completed or, optionally, until a specified timeout expires. If the timeout is set to zero (0), the timeout will occur immediately, without blocking, if the operation cannot complete directly. The timeout option is disabled by using -1 as timeout duration; the task may

in such cases be blocked indefinitely, i.e., until simulation termination, if the resource does not become available. As can be expected, if more than one task are waiting to acquire a specific resource (e.g., putting a message in a mailbox, or locking a semaphore), the task with highest priority will get the resource when it becomes available.

Time and Scheduling

In RTSSim simulation models, time is discrete and is represented by an integer simulation clock, a global variable named *clk*. All time-triggered events in an RTSSim simulation depend on *clk*, which in turn depends on *Execute*, which models the consumption of CPU time by incrementing *clk* as described in Section 3.2.3. The *clk* variable may be used in order to read the current time, and it is also used by the RTSSim trace recorder for time-stamping of events.

As mentioned, tasks are scheduled using preemptive fixed-priority scheduling, as described in Section 3.2.4, and are assumed to share a single CPU core. Task-switches can however only occur inside RTSSim API functions which invokes the scheduler, such as *Execute*. Other model code always execute in an atomic manner, i.e., without preemptions, since the scheduler is only invoked by explicit RTSSim API calls. In order to allow for preemptions between two "normal" code statements, it is necessary to insert an *Execute* statement in between.

Unlike VirtualTime [136], RTSSim does not yet support simulation of multiple CPU cores in parallel, i.e., parallel/multi-core computers or distributed systems. RTSSim however allows for simulating individual CPU cores of a parallel (or distributed) system in isolation, by modeling input from other CPU cores as "system environment", as described in Section 3.2.5.

3.2.2 A Small Example Model

Next follows an example of a simple RTSSim model containing two interacting tasks: *Sender* and *Receiver*. The *Sender* task begins by calling *Execute* in order to consume between 130 – 150 units of CPU time. A message is then sent to *Receiver*, using the mailbox *MBox*. The *Execute* corresponds to real system calculations found irrelevant for the model and therefore only included with respect to execution time.

As specified in *model_init*, the *Sender* task is activated periodically (every 2000 time units) and thereby uses 7 % of the CPU time since each task instance in average consumes 140 time units. The *Receiver* task executes less frequently

(every 5000 time units) and reads all messages stored in the mailbox (note the second *RecvMessage*, in the "while" loop). Each message is associated to a processing time of 500 time units in *Receiver*, but the number of messages in the mailbox when *Receiver* starts will vary due to inharmonic task periods.

```
void Sender(TCB* task)
{
  Execute(130 + getRandomValue(20));
  SendMessage(MBox, m++, FOREVER);
}

void Receiver(TCB* task)
{
  int msg = 0;
  msg = RecvMessage(MBox, 0);
  Execute(10);
  if (msg < 0)
  {
    UserEvent(ue_no_msg);
  }
  else
  {
    while (msg >= 0)
    {
      Execute(500);
      msg = RecvMessage(M, 0);
    }
  }
  Execute(10);
}

void model_init()
{
  MBox = CreateMailbox("M", 10);
  CreateTask("Sender", 1, 2000, 500, 0, Sender);
  CreateTask("Receiver", 2, 5000, 0, 0, Receiver);
  ue_no_msg = CreateUEChannel("No msg");
}
```

Since the *Sender* task produces a message every 2000 time units, and since all messages produced are at some point processed by *Receiver*, using 500 time units each, the average CPU load of *Receiver* should be just over 25 %. Note that *Sender* has an offset of 500 time units, which causes the first instance of *Receiver* to find an empty mailbox, which fires the user event "No msg" at time 10.

Figure 3.2 shows a Tracealyzer[2] view over the first 12 000 time units of an RTSSim simulation of the above example model. The sending and receiving of messages is shown as system events, and at time 10, the user event registered by *Receiver*, to indicate that mailbox *MBox* was empty, is shown in yellow. The

[2]The tool used in the figure, RTXCview, is a commercial version of the Tracealyzer.

CPU load view shows that the CPU usage is around 33 % in total, of which *Receiver* uses about 7 % and *Sender* 26 %, which matches the expectations. The Tracealyzer tool is presented further in Section 7.3.

A larger example of an RTSSim simulation model is found in Appendix C. This simulation model was used in the evaluation of simulation optimization methods, as presented in Chapter 4.

3.2.3 Execute

The *Execute* function is used to model the consumption of CPU time by incrementing the simulation clock, *clk*. The amount of CPU time to consume, i.e., to add to *clk*, is given as a parameter ($incr$ in the below code). The *Execute* function is also responsible for the processing of time-triggered simulation events, such as activation of time-triggered tasks, timeouts, and terminating the simulation. Such events are created by different mechanisms in RTSSim, e.g., by the scheduler. The event list, where such events are stored, keeps the events sorted by their scheduled time of activation. The core functionality of *Execute* is presented in C code below:

```
void Execute(TCB* tcb, int incr)
{
  while(incr > 0)
  {
    event = getNextEvent(clk + incr);
    if (event == NULL)
    {
      clk = clk + incr;
      incr = 0;
    }
    else
    {
      do{
        incr = incr - (event->time - clk);
        clk = event->time;
        processEvent(event->action);
        event = getNextEvent(clk);
      }while(event != NULL);
      schedule(tcb);
    }
  }
}
```

The *getNextEvent* function gives the earliest simulation event scheduled to occur at latest on the specified time. If no event are scheduled to occur within the time-window $[clk, clk + incr]$, the current time *clk* jumps directly to $clk + incr$, and the *Execute* is finished. If there is at least one event in the specific time-window (e.g., an activation of a time-triggered task) *clk* is

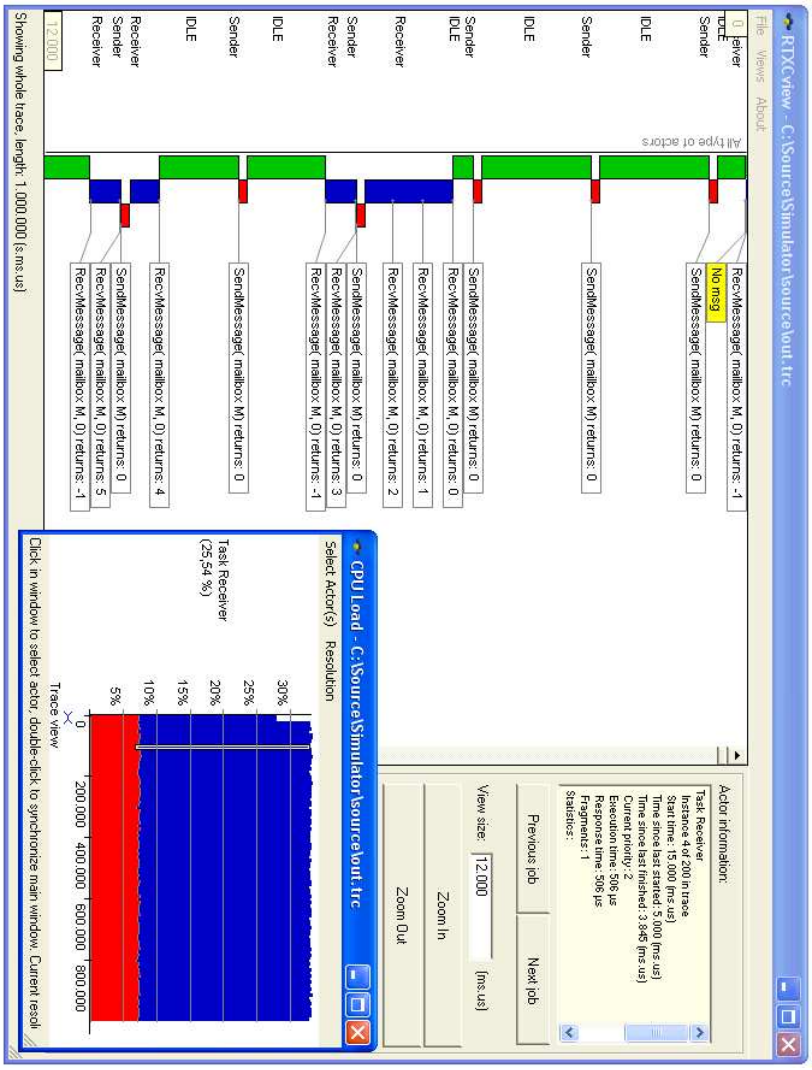

Figure 3.2: A simulation trace from the example model

advanced to the time of the first event. The remaining amount of CPU time to consume, *incr*, is decreased accordingly. The event is thereafter executed and the inner loop makes sure that all events scheduled at the same time instant (e.g., task activations) are processed before the scheduler is called. When the amount of CPU time remaining to consume (*incr*) reaches zero, the *Execute* function is completed and returns.

RTSSim contains a pre-defined idle task at lowest priority (255). This task is always ready to execute if no other task is, since it contains a single *Execute* statement placed in an infinite loop. The idle-task is important since it prevents that RTSSim goes into a deadlock state if no tasks are ready to execute at some point in the simulation.

3.2.4 Task and Scheduling Implementation

In RTSSim, each simulation model task is mapped to a *fiber*, a concept in Microsoft's Win32 API which implies a lightweight thread which the application is responsible for scheduling, i.e., within a single thread. This fits the RTSSim framework perfectly, since a separate, explicit scheduler is desired. A context-switch between fibers is achieved by letting the currently running fiber call the Win32 API function *SwitchToFiber*, with the handle of the new fiber as parameter. This is faster than normal context-switching, i.e., between threads, as fibers have less state information than threads or processes.

The scheduling in the RTSSim framework is explicit, in the sense that the tasks calls the scheduler, although not directly but through *Execute* and most other RTSSim API functions also call the scheduler. The core functionality of *Schedule* is presented below in C code:

```
void Schedule(void)
{
  TCB* NewTask = SelectTask();
  if(NewTask != RunningTask)
  {
    RunningTask = NewTask;
    SwitchToFiber(NewTask->fiberHandle);
  }
}
```

The scheduler begins by looking up the currently ready task of highest priority in the TCB list (using *SelectTask*). If the selected task is not identical to the currently executing task (*RunningTask*), the scheduler performs a task-switch by updating the *RunningTask* pointer and then calling the Win32 API function *SwitchToFiber*, which blocks the caller task (fiber) and activates the

new. In this manner, only one task (fiber) executes at any given time, while all others fibers are blocked by the *SwitchToFiber* call in *Schedule*.

The scheduling attributes are stored in the TCB of each task and thereby possible to read and modify from the task code, in order to implement custom scheduling algorithms on top of the normal scheduler. The scheduling attributes of tasks in an RTSSim model are:

- priority,
- period (minimum),
- offset, and
- jitter (maximum).

The highest priority in RTSSim is 0 and the lowest priority is 255. In the current implementation, tasks are expected to have unique priorities. If two tasks have identical priorities and are ready at the same time, their order in the TCB list will decide. This depends on the order of creation and is not by design, but more of a side-effect of the implementation. A perhaps better solution would be to perform round-robin scheduling for tasks with the identical priorities, since this allows for fair scheduling of tasks of equal importance.

The second attribute, period, specifies the periodicity of tasks and if they should be recurring (periodic/sporadic) or one-shot tasks. This also involves the jitter attribute. If the jitter attribute is zero, the result is a periodic task, which is activated (becomes ready to execute) every $period$ time units. If a non-zero maximum jitter is specified, a random value in the range $[0, jitter)$ is added to the inter-arrival time of each task instance, thereby creating a random but bounded inter-arrival time variation. The period thus specifies the minimum inter-arrival time. Periodic and sporadic tasks must terminate after each instance, i.e., return from their entry-function, since the activation event of the next instance is not created until the previous instance has finished. Non-terminating tasks are common in many industrial systems. Such tasks contain "infinite" main-loops where a loop iteration corresponds to a task instance. Such tasks are realized in RTSSim using one-shot tasks, which is the result if specifying a period of -1 (or any other negative value).

The offset sets the activation time of the first task instance and thereby shifts the activation time of the later instances with that amount. Note that the activation time is the time when the task becomes ready to execute, not the time when it actually starts to execute, and is therefore predictable for time-triggered

tasks. The activation time is calculated using the following formula:

$$AT_i = \begin{cases} AT_{i-1} + period + rand(jitter) & if(i > 0) \\ offset + rand(jitter) & if(i = 0) \end{cases}$$

where AT_i refers to the activation time of instance i and $rand$ provides a "stochastic" jitter selection (as described in Section 3.2.6).

3.2.5 Environment Modeling

RTSSim allows for modeling external systems, e.g., other connected computer systems (or CPU cores), sensors, operator controls, etc., using *environment tasks*. These are RTSSim tasks that do not consume CPU time and therefore only impact the simulation by the input events they generate, e.g., IPC messages sent to other tasks or modified global variables.

Environment tasks are "invisible" during the simulation in the sense that they does not affect the scheduling or show up in the simulation trace output. Environment tasks may use all RTSSim API functions except for *Execute*. The set of environment tasks used in a simulation can be regarded as an *environment model*, a necessary subset of a complete simulation model. As an example, the larger RTSSim model presented in Appendix C includes three environment tasks, those with the suffix "ENVTASK".

3.2.6 Stochastic Selections

An RTSSim simulation model may contain "stochastic" selections, which are not decided by the simulation model but by inputs to the simulation, either directly or through the pseudo-random number generator. The most visible type of stochastic selection is the random variations in task release time specified by the jitter attribute. Other types of stochastic selections are execution time variations (stochastic increment of the simulation clock) and stochastic behavior selections (typically in environment tasks). RTSSim determines stochastic selections using either pseudo-random numbers, resulting in Monte Carlo simulation, or by using explicit selection values, specified as input. In the latter mode, RTSSim is completely deterministic and can be controlled by an external tool for simulation optimization purposes, as described in Chapter 4. In order to realize Monte Carlo simulation, RTSSim obtains a "seed" value from a high resolution timer which used to initiate the pseudo-random number generator. The seed value is reported in the output which makes it possible to replicate previous simulations. This is discussed in greater depth in Section 3.2.7.

3.2.7 Pseudo-Random Number Generation

A pseudo-random number generator is an algorithm which generates numbers that are seemingly random, typically according a uniform probability distribution. However, a produced sequence of pseudo-random numbers is not truly random (hence "pseudo") since the sequence is completely determined by the seed value used to initialize the pseudo-random number generator. Thus, given a specific seed, a specific sequence of seemingly random values is produced. A good approximation of truly random values can be produced by using a seed value from a high resolution hardware clock.

However, the standard library function for generating pseudo-random numbers in the Win32 API, *rand*, only produces 15-bit values, i.e., in the range $[0, 32767]$. This is a problem since a simulation model may pseudo-random number larger than 32767, e.g., as execution time or inter-arrival time jitter. One solution is to compensate this by merging two 15-bit values into a 30-bit value, but this requires calling the *rand* twice for each 30-bit random number. This was used in an earlier version of RTSSim, but was later replaced with a faster solution, a custom random number generator based on the AS 183 algorithm [10] which produces 32-bit values.

In a benchmark test, the AS 183 solution required 56 seconds for producing 2 billion ($2 * 10^9$) 32-bit random numbers. The earlier approach, i.e., when combining the results of two calls of the standard library function *rand* into a 30-bit random number, required 88 seconds for the same amount of random numbers, i.e., 57 % longer time.

A more recently proposed algorithm for generating pseudo-random numbers is the Mersenne Twister [19]. It is claimed to be fast and generate pseudo-random numbers of very high quality. An interesting direction of future work could be to compare the performance of RTSSim using different pseudo-random number generators, AS 183, Mersenne Twister, and other solutions available (many exists).

3.3 Conclusions

This chapter has presented simulation as method for timing analysis of complex embedded systems, including motivations and limitations for the approach. It has moreover presented a technical solution for this purpose, the simulation framework RTSSim, how it works and roughly how it is used. The RTSSim API is presented in detail by Appendix B and a larger example of an RTSSim model is presented in Appendix C. Ideas on recording and generation of *timing profiles* containing timing data for RTSSim models are presented in Section 7.5.

In a thesis perspective, this chapter is not to be considered a novel research contribution on its own; at least four similar solutions are described in Section 2.3. The RTSSim simulation framework is however the technical platform of this research and a conceptual understanding of RTSSim is therefore important for a good understanding of the following chapters, which in different ways all relate to the simulation-based timing analysis.

The next chapter describes novel results in simulation optimization, a method in which RTSSim (or a similar simulator) is controlled by an optimization algorithm in order to provoke as extreme behaviors as possible, with respect to a specific property of the simulation model. Chapter 5 describes a novel approach to automated extraction of simulation models through static analysis and Chapter 7 presents techniques for monitoring (trace recording) of embedded systems, which has been used in RTSSim, and a method for generating timing profiles for RTSSim models, using trace recording. Chapter 8 discuss validity and validation of simulation models, and presents methods for comparing traces from recorded during simulation or real system execution.

Chapter 4

Simulation Optimization

Simulation is a promising approach to timing analysis of complex embedded systems. As presented in Chapter 3, simulation-based analysis is applicable to software systems of any design and scales to large, complex systems. The downside of simulation is the confidence in the predictions; a traditional Monte Carlo simulation corresponds to a random search and is not suitable for worst-case timing analysis, since only a random subset of the possible scenarios are explored. Unlike formal analysis methods, results from simulation-based timing analysis cannot be used to guarantee that a system meets its timing requirement. This is similar to the problems of general software testing; the method can only be used to show the presence of errors, not to prove the absence of errors. Nonetheless, a simulation-based analysis can identify extreme scenarios, e.g., very high response-times which may violate the system requirements, even though worst case scenarios are not identified.

This chapter presents two alternative methods for *simulation optimization* which allows for efficient identification of extreme scenarios with respect to a specified measurable run-time property of the system, e.g., related to timing or resource usage, using a combination of discrete event simulation, as described by Chapter 3, and heuristic search methods. The two approaches, *MABERA* and *HCRR*, both use a simulator (RTSSim) as a deterministic function which given a set of parameters defining the scenarios to explore, returns the most extreme value observed for the run-time property in focus. In the implementations and evaluations presented in this chapter, the property in focus is the highest response time observed for a specific task.

Both approaches use the simulation results in an iterative analysis where

the simulation parameters are gradually refined in order to provoke as extreme results as possible. Like traditional Monte Carlo simulation, this is still a best-effort approach, but significantly more efficient.

An evaluation is presented where the two approaches to simulation optimization are compared to Monte Carlo simulation, with respect to analysis time and the discovered response times. The results indicate that both MABERA and HCRR are significantly more efficient in finding extreme response times for a particular task than Monte Carlo simulation, but also that HCRR, which uses a variant of the hill climbing approach [111], is vastly more efficient than MABERA, which uses a genetic algorithm [7].

The use of search algorithms for different types of test case generation has also been studied for quite some time. Alander et al. [5] used genetic algorithms to generate test cases for a software relay system used in power girds, in order to provoke high response times of the software, executed in a simulation environment. Nossal et al. [6] describe various extensions of the traditional genetic algorithm to better suit the type of problems in the real-time domain. Samii et al. [12] present a work where they attempt to find extreme response times for distributed systems by optimizing a set of simulation parameters for models containing temporal attributes and communication. They use a genetic algorithm to explore combinations of task execution times in order to maximize end-to-end response time. Task behavior is however not considered. Their results depend on the method developed by Racu and Ernst [11] for identifying situations where decreased execution times can lead to increased response times.

4.1 MABERA

MABERA is an abbreviation of "Metaheuristic Approach for Best Effort Response-time Analysis". Metaheuristics are generic solution methods for iterative approximation of search/optimization problems. MABERA is a *genetic algorithm* [7], which is the most well-known type of metaheuristics. The MABERA algorithm is designed to use an RTSSim executable (i.e., the simulator framework and the simulation model) as a black-box function, which given a set of simulation parameters (cf. Section 4.2) outputs the highest response-time found during the specified simulation. The objective of the MABERA algorithm is to find the set of parameters which gives the highest response time for the task in focus. This is performed by iteratively creating and evaluating a set of independent candidates, a *generation*, where each candidate is a simulation.

The specification of a simulation, i.e., the set of parameters to RTSSim, is here named a *parameter set*, formally defined in Definition 3. The evaluation of a parameter set corresponds to running an RTSSim simulation, which gives a *simulation result* as defined by Definition 4.

The MABERA algorithm uses an indirect representation of the simulations to perform, using *seed schedules*. A seed schedule specifies the seed values to use for generation of pseudo-random numbers and thereby the outcomes of all non-deterministic selections during the simulation. Thus, the combination of a seed schedule and time instant corresponds to an exact specification of a simulation state. Given a seed schedule and a simulation length, an RTSSim simulation is thereby a deterministic function. The concept of seed schedule is defined by Definition 2.

Definition 1. *A seed change event is a pair $\langle t, s \rangle$, where t and s are integer values. The t value specifies the simulation time instant when the seed value s should be applied. A seed value of 0 specifies that a randomly selected seed should be used.*

Definition 2. *A seed schedule is a list of seed change events, sorted in ascending order with respect to the t attribute the included seed change events, in ascending order. The first seed change event of a seed schedule is always at time 0.*

Definition 3. *A parameter set is the specification of a simulation, represented as a tuple $\langle T, l, S \rangle$, where T is the task in focus, l the simulation length and S the seed schedule to use.*

Definition 4. *A simulation result is a tuple $\langle rt, et, pc, t_{rt}, t_{et}, t_{pc}, S \rangle$, where rt, et and pc is the highest observed response time, execution time and preemption count, respectively, during the simulation specified by the seed schedule S. The properties t_{rt}, t_{et} and t_{pc} are the start times of the task instances corresponding to rt, et and pc, respectively.*

The first generation of simulations are (random) Monte Carlo simulations. From each generation, a set of promising simulations are selected as *parent simulations* and used to create the next generation, where each simulation is created by mutation of a (single) parent. The algorithm iterates in this manner until a termination condition is reached.

Formally, MABERA can be described as a (non-deterministic) function

$$r = MABERA(s, p, l, tt, T)$$

where s specifies the population size (the number of simulations per generation), p the number of parent to select per generation, l the length of each individual simulation, tt the "termination threshold", and where the result r is the highest response time found for the specified task T. This may vary between analyzes (with identical parameters) due to the random simulations involved.

The *termination threshold* parameter, tt, decides how many "unsuccessful" generations that are allowed before termination, i.e., generations that failed to discover a response-time higher than the highest response time of the previous generations. Since the population size is constant for each generation, each parent simulation should result in s/p child simulations. The MABERA algorithm is presented in detail in Section 4.1.3.

The child simulations will explore a subset of the model's state-space, the *offspring state-space*, which is reachable from the state corresponding to a specific time instant during the parent simulation: the *restart time*. Together with the seed schedule of the parent simulation, the restart time specifies the starting state of the child simulation. The restart time is randomly selected in a specific time interval of the parent simulation, as described in Section 4.1.2. A child simulation reaches the specified starting state by using the same seed schedule as the parent simulation up until the restart time, where instead a randomly selected seed is applied in order to explore other parts of the offspring state-space. This is likely to contain a response time for T higher than the highest response time for T of the parent simulation, unless the worst-case response time of T has already been found.

To explain the concept of offspring state-space, think of the state-space of a Monte Carlo simulation model as a tree, where each node corresponds to a state in the model where a non-deterministic selection is made, e.g., selecting an execution time for an *Execute* statement. An individual simulation is a specific path through the tree, which ends at a state decided by the simulation length. The offspring state-space, which the child simulations explore when applying the random seed, is the sub-tree rooted in the state corresponding to the restart time of the parent simulation.

The state-space exploration of MABERA is illustrated by Figure 4.1, simplified to a 2-dimensional state-space. In practice, the state-space will have a large number of dimensions (equal to the number of independent variables)

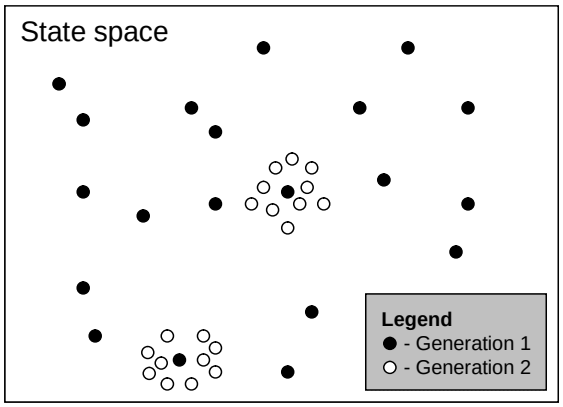

Figure 4.1: MABERA – conceptual

and more than two iterations are typically made. In this example the population size (s) is 20 and the number of selected parents (p) is 2.

The selection of parents to produce the next generation, described in Section 4.1.1, is very important for the efficiency of MABERA. There is always a risk of not finding the global maximum, i.e., the worst case response time, as the algorithm might "get stuck" at a local maximum, where no child simulation can be found that is more extreme than the parent (although higher response times are possible in other scenarios). To reduce this risk, the MABERA algorithm is designed to select several parents from each generation, at least two are recommended. Thereby, if one parent "gets stuck" at a local maximum, there is still a chance that the other parents find better results.

MABERA uses no recombination/crossover operation, which otherwise is common in genetic algorithms, since the meaning of the seeds used during a simulation depends on the simulation state when the value is used. They can therefore not be used as independent chromosomes, which can be recombined with preserved semantics.

Note that the connection between a parameter set and the simulation result is unknown in this approach since the simulator is considered a black-box. It is thereby not possible to optimize the result by, in some way, selecting "good" seeds for the initial generation; there is no way of assessing the potential of a seed schedule without running a simulation.

MABERA was never intended as an optimal solution. It is the result of a first investigation for assessing the potential of simulation optimization methods in the context of best-effort response-time analysis. The MABERA results were however quite interesting and motivated a continued effort which resulted in the HCRR approach presented in Section 4.4.

4.1.1 Selection Heuristics

The SEL function, presented in Section 4.1.3, implements the heuristic selection of parents simulations from a set of simulation results, which are used to produce the next generation of parameter sets. The selection ranks all simulation results in the current generation with respect to the three properties rt, et and pc, i.e., the highest response time, execution time and preemption count, respectively, of the task in focus. The ranking assigns each simulation result a positive, non-zero integer value, which tells how many simulation results that have higher values, for the specific property. Simulations with equal values receive the same rank, with respect to the specific property.

The three rank values of each simulation result are multiplied in order to obtain a total fitness score for the simulation result. The best fitness score is 1, which corresponds to a simulation result that holds the record for all three properties. The returned set of simulation results contains a specified number of simulation results with best (lowest) fitness scores.

The execution time and preemption count properties are included in the selection heuristics due to their potential for impacting response time, e.g., a task instance with very high execution time but relatively low response-time is also interesting since a different preemption pattern may result in a higher response time.

The method of combining the three rank values into a total fitness score is not claimed to be optimal. It gives equal importance to the three indicators, response time, execution time and preemption count. Moreover, the ranking hides the absolute differences in property values between candidates with adjacent ranking. Investigation of other selection heuristics is part of future work.

4.1.2 Mutation

The GEN function, presented in Section 4.1.3, is responsible for creating a new generation of parameter sets through mutation of the selected parent simulations. Each parameter set is created through mutation of the seed schedule of a specific parent simulation result by adding an additional seed change

event in the end of the seed schedule, with a zero as seed which specifies that a randomly selected seed should be applied, i.e., used to re-initialize the pseudo-random number generator. This makes the simulation leave the path of the parent simulation and instead explore the path associated with the new, randomly selected seed for the remaining part of the simulation. The mutation algorithm is described in Algorithm 2 in Section 4.1.3.

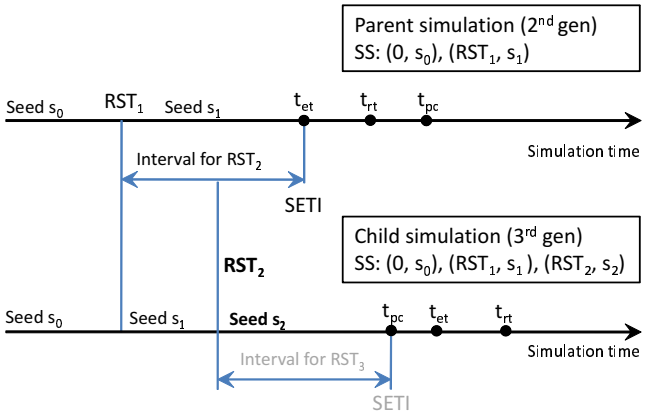

Figure 4.2: Seed schedule mutation in MABERA

The time of the new seed change event, i.e., the restart time, is randomly selected in a time interval where the lower bound is the restart time of the parent and the upper bound is the parents *Start time of the earliest Extreme Task Instance* (or *SETI*), where *extreme* refers to the task instances that have the highest value of at least one of the following properties: response time, execution time and preemption count. The seed schedule mutation, including selection of restart time, is illustrated by Figure 4.2. In this illustration, the bold labels (RST_2 and Seed s_2) corresponds to the mutation.

As specified by Algorithm 2 in Section 4.1.3, there is a special case if the parent's SETI is found to be earlier than its restart time. This indicates that the mutation performed to produce the parent simulation resulted in a less extreme scenario than observed in the parent´s parent, since task instances before the restart time now are more extreme than those after. In this case, the restart time of the parent is reused with a new randomly selected seed in order to that restart time a second chance.

4.1.3 The MABERA Algorithm

This section presents the MABERA algorithm and the remaining definitions on which it relies. Note that the concepts of seed change event, seed schedule, parameter set and simulation result are defined by Definition 1 to Definition 4 in Section 4.1.

Definition 5. *$R = SIM(P)$ represents a simulation according to the parameter set P, where the output R is a simulation result. A seed value of zero (0) in the seed schedule of the parameter set is an instruction to apply a randomly selected seed value, which also replaces the zero seed in the simulation result seed schedule.*

Definition 6. *$RT(R)$ gives the rt property of a simulation result R, i.e., the highest response time found for the task in focus in the specific simulation.*

Definition 7. *$ET(R)$ gives the et property of a simulation result R, i.e., the highest execution response time found for the task in focus in the specific simulation.*

Definition 8. *$PC(R)$ gives the pc property of a simulation result R, i.e., the highest preemption count found for the task in focus in the specific simulation.*

Definition 9. *$TRT(R)$ gives the t_{rt} property of a simulation result R, i.e., the start time of the task instance corresponding to the rt property of R*

Definition 10. *$TET(R)$ gives the t_{et} property of a simulation result R, i.e., the start time of the task instance corresponding to the et property of R*

Definition 11. *$TPC(R)$ gives the t_{pc} property of a simulation result R, i.e., the start time of the task instance corresponding to the pc property of R*

Definition 12. *$SS(R)$ gives the seed schedule used in the simulation which produced the simulation result R.*

Definition 13. *$RST(R)$ gives the restart time used in the simulation which produced the simulation result R, i.e., the time of the last seed change event in $SS(R)$.*

Definition 14. *$RAND(a, b)$ gives an integer value x, such that $a \leq x < b$, randomly selected according to a uniform probability distribution.*

Definition 15. *$APPEND(A, E)$ gives a seed schedule which is the result from appending the seed change event E to the end of the seed schedule A.*

Algorithm 1: The parent selection procedure of MABERA

SEL($\mathbf{R}, p$)
  foreach $res \in \mathbf{R}$
    er $\leftarrow |\{q \in \mathbf{R} \mid \text{ET}(q) \geq \text{ET}(res)\}|$
    rr $\leftarrow |\{q \in \mathbf{R} \mid \text{RT}(q) \geq \text{RT}(res)\}|$
    pr $\leftarrow |\{q \in \mathbf{R} \mid \text{PC}(q) \geq \text{PC}(res)\}|$
    rank$_p$ $\leftarrow$ *er* $*$ *rr* $*$ *pr*
  Order $\mathbf{R}$ as $r_1, r_2, \ldots$ according to *rank*$_{p_i}$.
  return $\{r_1, r_2, \ldots, r_p\}$

Algorithm 2: The mutation procedure of MABERA

MUTATE(p, l, T)
  SETI $\leftarrow$ MIN(TET(p), TRT(p), TPC(p))
  if *SETI* < RST(p)
    $M \leftarrow \langle 0, \text{RST}(p)\rangle$
  else
    $M \leftarrow \langle 0, \text{RAND}(\text{RST}(p), \textit{SETI})\rangle$
  return $\langle T, l, \text{APPEND}(\text{SS}(p), M)\rangle$

Algorithm 3: The procedure for populating a new generation in MABERA

GEN($\mathbf{P}, s, l, T$)
  $\mathbf{G} \leftarrow \emptyset$
  foreach $p \in \mathbf{P}$
    for $i = 1$ **to** $\lfloor s/|\mathbf{P}| \rfloor$
      $\mathbf{G} \leftarrow \mathbf{G} \cup \{\text{MUTATE}(p, l, T)\}$
  return $\mathbf{G}$

Algorithm 4: The overall MABERA algorithm

MABERA(s, p, l, tt, T)
  $G \leftarrow \emptyset$
  tc $\leftarrow 0$
  best $\leftarrow 0$
  for $i = 1$ **to** s
    $G \leftarrow G \cup \langle T, l, \langle 0, 0\rangle\rangle$
  while $tc < tt$
    found $\leftarrow 0$
    foreach *si* $\in G$
      $sr \leftarrow$ SIM(*si*)
      if RT(*sr*) $>$ *best*
        best $\leftarrow$ RT(*sr*)
        found $\leftarrow 1$
      $R \leftarrow R \cup \{sr\}$
    if *found* $= 0$
      tc $\leftarrow$ *tc* $+ 1$
    else
      tc $\leftarrow 0$
    $P \leftarrow \text{SEL}(R, p)$
    $G \leftarrow \text{GEN}(P, s, l, T)$
  return *best*

4.2 The MABERA Parameters

The MABERA algorithm has four parameters which impact the thoroughness and runtime of the analysis. These parameters are:

- l: The **l**ength of each individual simulation.
- p: The number of selected **p**arents from each generation.
- tt: The **t**ermination **t**hreshold.
- s: The population **s**ize.

To maximize the efficiency of MABERA it is important to select good values for these four parameters. This section discuss how they are related and how they impact the performance of MABERA. Note that the MABERA parameters should not be confused with the parameter set used to define an individual (RTSSim) simulation within MABERA.

Parameter l The simulation length, l, is the value of the simulation clock when the simulation should terminate. The l parameter naturally impacts the runtime of a simulation and should therefore not be longer than necessary, which depends on the scenario under analysis, e.g., a specific system test case. Even though longer simulations may find higher response times, as they might contain multiple instances of the relevant scenario, the resulting increase in runtime can instead be used to increase the population size, s, or the termination threshold, tt, which also impacts the runtime.

Parameter p The p parameter is the number of simulations from each generation to select as parents for the next generation (created trough mutation of single parents). This decides how much to trust the selection heuristics. If the heuristics could be trusted to always point out the truly most "promising" simulation result, i.e., that is closest to the true worst case scenario, the analysis could rely on a single parent. However, since the selection heuristics is not a perfect oracle, several parents should be selected in order to reduce the risk of bad heuristic decisions.

However, the important property is not the absolute number of parents, but rather the relative amount of parents in relation to the population size, i.e., the p/s quota. For instance, in Figure 4.1, two simulation results are selected ($p = 2$) from each generation of 20 ($s = 20$), which gives a p/s quota of 0.1 and 10 child simulations per selected parent.

A smaller p/s quota implies more simulations based on each selected parent, i.e., a more thorough analysis of the selected cases, but also means that fewer simulation results are selected to become parents for the next generation, i.e., a higher trust in the selection heuristics and an increased risk of getting stuck in local maxima. A larger p/s quota implies a wider search, which may converge slower, but with less risk of getting stuck in local maxima. Since a balanced p/s quota is important, the parameter selection process in Section 4.3 is therefore focused on the relative number of parents (p/s), not the absolute value (p).

Parameter tt The tt parameter, i.e., the termination threshold, impacts the number of generations analyzed and thereby the runtime of the analysis. The tt parameter decides the number of "unsuccessful" generations (MABERA iterations) allowed before MABERA should terminate. An "unsuccessful" generation is a generation which did not contain any simulation with higher response time result than the highest result found so far, in previous generations. The MABERA algorithm includes a termination counter (the variable tc in the MABERA pseudo code presented in Section 4.1.3) which initially is 0. Unsuccessful generations will increment the termination counter by 1, while a successful generation resets the counter to 0. When the termination counter reaches the termination threshold, tt, the MABERA algorithm terminates and reports the highest observed response time in any generation.

Thus, with a higher tt value, the risk that a good parent is rejected due to "bad luck" is reduced, but the runtime is increased by the extra iterations. A higher tt value may compensate the negative effects of a lower p/s quota by allowing for additional iterations, at least to some extent. It is however important to find a balanced value for tt, as the extra runtime required for larger tt values can instead be used to increase the population size.

Parameter s The population size, s, is the number of simulations to perform in each iteration. The larger population size, the more thorough analysis. Thus, s should preferably be as large as possible, but since it impacts the runtime of the analysis, which in practice is limited, it is necessary to find an upper bound for s that gives a runtime below (but close to) the desired runtime. When starting a large, over-night analysis, one would like to know that the analysis is finished by the morning, but preferably not much earlier in order to best utilize the available analysis time.

4.3 Selecting Parameters for MABERA

A three-step process is proposed for finding good parameters values for a specific simulation model. The process contains a set of experiments presented together with examples performed on Model 1 (cf. Section 4.5.1), one of the simulation models used in the evaluation, in Section 4.6. Note that the second major step of this process is divided into four parts, as presented below.

Since the parameter selection process requires several time-consuming experiments it should only be performed initially, when a simulation model has first been constructed, or after major architectural changes in the modeled system which impacts the model. One should note that the parameter values presented in this section are not necessarily optimal for other simulation models. Good parameter values are believed to be dependent on the characteristics of the model under analysis, specifically, the amount and type of stochastic selections in the model. Good values for the four parameters of MABERA are selected using the following process:

1. Select simulation length.
2. Select p/s quota and tt value. This is divided into four parts:
   (a) Specify candidate values.
   (b) Determine sufficient replication count.
   (c) Determine comparable parameter combinations.
   (d) Compare comparable parameter combinations.
3. Select population size.

4.3.1 Step 1: Selecting Simulation Length

The value for the l parameter is decided first since it does not depend on the other parameters but is needed in the other steps of the process. The l parameter is determined through a manual analysis of the model, either by studying the model code or by studying traces from test simulations of the model. The challenge is to find the minimum simulation length which includes the scenario of interest. For Model 1, a suitable l value was found to be 650 ms. This length included the system's processing of the events relevant for the scenario under analysis, as well as a safety margin if the scenario length varies between simulations.

4.3.2 Step 2: Selecting p/s quota and tt value

There is a dependency between the p/s quota and the tt value, as a higher tt value may compensate, to some extent, for the negative effect of a higher p/s quota, i.e., the decreased number of child simulations based on each parent. Suitable values for these parameters can therefore not be selected individually but need to be evaluated together, in combination. This section describes a four-step method for finding a good combination of p/s quota and tt value experimentally.

Step 2.A: Specify candidates

The first step is to specify a set of candidate values for p/s and tt. Each combination of these parameter values will be compared in the last step of this method. A straight-forward approach is to perform a set of experimental MABERA runs on the simulation model at hand and observe what range of values that seem to give good result. For Model 1, the tt candidates were limited to 2, 3 and 4. A value of 1 implies no tolerance, and values above 4 do not seem to improve the performance of MABERA. The best results were observed with p/s quotas below 0.05, i.e., at least 20 child simulation per parent, so the values of 0.005, 0.01, 0.02 and 0.04 were selected as candidates for the p/s quota.

Step 2.B: Determine replication count

The second step is to decide the number of data points (replications) necessary per MABERA configuration in order to ensure the reliability of the parameter comparisons in later steps of this process. This is important in order to avoid that the selection of good parameter values is obscured by random variations.

Table 4.1: Test of MABERA reliability

	p/s = 0.01	p/s = 0.04
Comparison 1	7 886	7 932
Comparison 2	7 889	7 940
Comparison 3	7 901	7 956

An experiment is proposed using a two-column table, illustrated by Table 4.1, where the columns correspond to different MABERA parameter com-

binations. Each cell contains a statistical measure, $Mean_{Q4}$, calculated over r independent runs of MABERA, where r is the candidate number of replications, using the parameter combination specified by the column. The $Mean_{Q4}$ measure implies the mean value of the 25 % highest results, i.e., the fourth quartile. This is selected since a typical use of the MABERA analysis would imply several replications and a focus on the highest result found. The lower results can safely be ignored in this comparison, since they are more likely to contain random "noise" caused by unsuccessful MABERA runs.

In this table, each row represent an independent comparisons based on independent data sets, each containing r data points. If the differences between columns of the same row are significantly larger than the differences between rows of the same column, this indicates that this number of replications gives sufficient reliability. The population size can be quite small to speed up this experiment and the tt value is not that important in this case, as it should not impact the reliability significantly. The important parameter here is the number of replications. It should however be constant for all columns.

This is however not a sufficient measure of reliability; it is necessary to verify that the differences indicated by $Mean_{Q4}$ correspond to statistically significant differences between the underlying data sets. If the two data sets of a row are not significantly different, the replication count should be increased in order to avoid inconclusive results later in the parameter selection process. An appropriate statistical test for this purpose is the two-sample Kolmogorov-Smirnov test [63], hereafter the KS test. This test is non-parametric and distribution-free, i.e., it makes no assumptions on the underlying distribution of the data, which is necessary in this case as the response-time data is not normally distributed. The KS test should be applied on the fourth quartile of the MABERA results, in line with the motivation behind the $Mean_{Q4}$ measure.

For Model 1, 200 replications was found to give reliable results for a comparison of the p/s quotas of 0.01 and 0.04, as presented in Table 4.1. The differences between the two columns (p/s quotas) are about 3 times larger than the differences between the rows, which indicate a significant difference. The data sets of each row were compared using the KS test and the differences between the cells of each row was found to be statistically significant at a confidence level of 99.9 %.

Step 2.C: Comparable parameters

The third step of this method is to calculate the *cost index* for each combination of candidate values for p/s and tt, which is a relative measure of the

average runtime (i.e., cost) of a MABERA configuration. The purpose of this cost index is to allow for a fair comparison of different parameter combinations, which may have considerable differences in their average runtime. Even if using the same population size, simulation length and termination threshold, the p/s quota impacts the speed of the convergence. If two parameter combinations produces similar results, but one is considerably faster, it is possible to increase the population size for the faster one and thereby obtain better results.

The first activity in this step is to run MABERA analysis of each parameter combination, replicated the number of times decided in step 2, and collect the average iteration count for each of the parameter combinations. This can be used as a measure of the runtime, since these are directly proportional due to the constant population size. The population size should be the same in all cases and should be a multiple of the number of parents implied by the candidate p/s quotas.

When this experiment was performed on Model 1, the difference in average iteration count was significant. As presented in Table 4.2, the most time-consuming combination of p/s and tt (p/s = 0.04, tt = 4) required 88 % more iterations (CPU time) than the least time-consuming combination (p/s = 0.005, tt = 2).

Table 4.2: Average iteration count of MABERA in different configurations

	tt = 2	tt = 3	tt = 4
p/s = 0.005	6.30	8.40	9.89
p/s = 0.010	6.64	9.16	10.18
p/s = 0.020	7.41	9.43	10.83
p/s = 0.040	7.27	10.06	11.87

The cost index of each candidate parameter combination is calculated by dividing the average iteration count of the specific case with the highest average iteration count of all cases, in this case 11.87. From the cost indices it is possible to calculate a comparable population size, s_c, for each candidate parameter combination. The comparable population size is calculated for each candidate parameter combination in order to give equal runtimes of MABERA, which allows for a fair comparison of the candidate parameter combination. This is essentially a normalization with respect to runtime. The comparable population size of a parameter combination is calculated by dividing a *reference population size* with the cost index of the parameter combination. To maintain

the relative number of parents it is necessary to calculate a comparable number of parents, p_c, by multiplying s_c with the desired p/s quota. Since p_c and s_c should be integers and thus needs to be rounded, the p_c/s_c quota will not be identical to the desired p/s quota. However, by selecting the reference population size carefully, it is possible to reduce these errors. In this case, a reference population size of 1000 was found to give quite small errors, below 5 %. For other, smaller, reference population sizes, errors up to 17 % were observed compared to the desired p/s quota.

When applying this process to the runtime data of Table 4.2, i.e., on Model 1, the following cost index results were obtained for the candidate combinations of tt and p/s quota.

Table 4.3: Comparable MABERA parameters

tt	p/s	Cost index	s_c	p_c	p_c/s_c
2	0.005	0.531	1 883	9	0.00478
2	0.01	0.560	1 787	18	0.0101
2	0.02	0.624	1 602	32	0.0199
2	0.04	0.613	1 632	65	0.0398
3	0.005	0.708	1 413	7	0.00495
3	0.01	0.772	1 295	13	0.0100
3	0.02	0.794	1 259	25	0.0199
3	0.04	0.847	1 180	47	0.0398
4	0.005	0.834	1 200	6	0.005
4	0.01	0.858	1 166	12	0.0103
4	0.02	0.912	1 096	22	0.0201
4	0.04	1	1 000	40	0.04

Step 2.D: Comparison

The part of step 2 is to execute MABERA for each candidate parameter combination, using the comparable population size (s_c) and the comparable number of parents (p_c) and the number of replications decided in step 2.B, in this case 200. The simulation length (l) should be decided according to step 1, described in Section 4.3.1. The best parameter combination is decided with respect to $Mean_{Q4}$, i.e., the mean value of the fourth quartile. If the difference between the top candidates is small in comparison to the variance indicated by

the earlier reliability test, the KS test should be used to verify the statistical significance of the difference. If no significant difference is found, one can either reduce the confidence level of the KS test or perform a focused comparison of the top candidates using a higher number of replications.

Results from this experiment is presented in Table 4.4, which indicates that the best parameters for Model 1 is p/s = 0.01 and tt = 3. The difference between this parameter combination and second best (p/s = 0.005 and tt = 2) was statistically significant according to the KS test, at a confidence level of 75 %.

Table 4.4: MABERA results using comparable parameters

	tt = 2	tt = 3	tt = 4
p/s = 0.005	8 194	8 155	8 105
p/s = 0.01	8 184	8 231	8 179
p/s = 0.02	8 156	8 172	8 192
p/s = 0.04	8 193	8 162	8 101

4.3.3 Step 3: Selecting Population Size

Once suitable values for the other parameters have been established, the last parameter s ultimately decides the runtime of MABERA. The larger population size, the more thorough analysis. Thus, s should preferably be as large as possible but since it impacts the runtime of the analysis, which in practice is limited, it is necessary to find suitable population size which limits the runtime to the runtime allowed. For instance, if starting an over-night analysis, it is important that the analysis is finished by the morning, but preferably not much earlier in order to best utilize the available analysis time.

If a runtime of several hours is desired, finding a suitable population size by using a trial-and-error would be quite time consuming. A better way of determining an appropriate s value that corresponds to a desired (quite long) runtime is through extrapolation of a reference case with a relatively small population size. This reference case should use the parameters found suitable in previous steps of this process.

In order to find a suitable population size for the reference case, start with a very small population size, e.g., 100, and measure the runtime. If very short, increase the population size, measure, and repeat until the runtime is significant but manageable, e.g., a few minutes. The desired population size s is

approximated using a linear extrapolation:

$$s = (t/t_r) * s_r$$

where t is the a runtime desired, s_r the population size of the reference case, and t_r the measured runtime of the reference case.

As mentioned, the reference case should have the p/s quota, tt and l values identified in previous steps. The p/s quota is especially important to maintain. If changing s without adjusting p, the changed p/s quota will cause the selected parents to be more or less extensively analyzed, which is likely to impact the number of iterations before termination and thereby cause a non-linear runtime increase. This is supported by Table 4.2, where the average iteration count has a positive correlation with the p/s quota.

4.4 Hill Climbing with Random Restarts

This approach, abbreviated HCRR, was developed to address shortcomings of the MABERA approach and is the result of a collaborative project with researchers at SICS[1] specialized in optimization methods. The goal of HCRR is the same as for MABERA, to find as high response time as possible for a specific task by optimizing the simulator input. The HCRR method uses hill-climbing [111], which has the advantage of being one of the simplest meta-heuristics available. It is based on the idea of starting at a random point and then repeatedly taking small steps pointing "upwards", i.e., to nearby input combinations giving higher response times. If no nearby input combination gives an improved result, a local maximum have been reached, possibly the global maxima. Random restarts are used to avoid getting stuck in local maxima. HCRR operates on a more detailed and system-dependent set of simulation parameters compared to MABERA. As demonstrated by the evaluation presented in Section 4.6, this method typically yields substantially better results than both Monte Carlo simulation and the MABERA approach.

4.4.1 Simulator Input Representation

A major difference between MABERA and HCRR is the different representations used for simulator input. Instead of the indirect approach of MABERA,

[1] The Swedish Institute of Computer Science, www.sics.se.

where the simulator input is used to initialize a pseudo-random number generator, the HCRR approach uses an explicit representation where each stochastic selection during the simulation is directly decided by a separate parameter value. Thus, in the HCRR approach each simulation may require hundreds, or thousands of inputs. Model 1, used in the evaluation section, requires over 600 inputs per 650 ms simulation.

The simulator is assumed to contain three types of stochastic selections: execution time variations, arrival-time jitter (e.g., caused by external interrupts) and environmental input stimulus (e.g., for determining non-deterministic selections of task behaviors or inputs in the simulation model).

A *simulation instance* is represented as a set of sequences of integers, where each sequence is associated with either an arrival jitter of a specific task, a specific execution time, or a specific environmental input stimulus. Each value of one of these sequences decides a specific stochastic selection, e.g., the release jitter of a specific task instance, or a specific selection of execution time. The advantage of this approach is that the direct relationship between representation and model properties makes it possible to locally refine specific aspects of a given simulation instance.

Let J_i be a sequence of actual jitter values $j_{i,r}$ experienced by instance r of a task T_i, where $j_{i,r}$ are integer values in the interval $[0, ub(J_i)]$, where $ub(J_i)$ is an upper bound on jitter for task i in units of the smallest measurable time interval (clock ticks) for the target system. Furthermore, let X^k be a sequence of values for a certain environmental input stimulus or execution time in the simulated program, and X^k_j be the j^{th} value in X^k. It is assumed that all stimulus and execution times X^k_j are of integer type and have upper (ub) and lower bounds (lb), so that $lb(X^k) \leq X^k_j \leq ub(X^k)$ for all k, j. A simulation instance S, defining a fully deterministic simulation of the model, is therefore a set

$$J_1, J_2, ..., J_n, X_1, X_2, ..., X_m$$

where n is the number of tasks which have non-zero jitter and m is the number of environmental stimulus and *Execute* statements. Denote by N_i and M_k the number of values that are used to represent jitter sequence J_i and input sequence X_k. N_i and M_k can be determined empirically by tracing how many values the simulator uses for each value. In theory, N_i and M_k can be unbounded, and for long simulations, the values for N_i and M_k needed may grow to unacceptable levels. If there are not enough input values in the sequence, the simulator should report a warning and reuse values, e.g., by starting over from the beginning of the sequence. For the evaluated models in this chapter, N_i and

M_k were long enough to represent all values used.

4.4.2 The HCRR Algorithm

HCRR uses a combination of a local improvement algorithm, which quickly converges to high response times, and two diversification mechanisms allowing the search to escape from local maxima, either by jumping back to an earlier explored candidate or by making a full random restart. Thus, HCRR works on a single solution candidate. In contrast, MABERA used a population of candidates evaluated in parallel, to reduce the risk of getting stuck in local maximas.

Algorithm 5: Hill Climbing with Random Restarts (HCRR)

```
HCRR(nofsims, m, l, k, nB, nR)
  hrt ← 0
  while m > 0
    curr ← rnd_inst()
    SIM(curr, l)
    nofsims ← nofsims − 1, m ← m − 1
    if RT(curr) > hrt
      best ← curr, hrt ← RT(curr)
  E ← {best}
  nonimp ← 0
  while nofsims > 0
    if nonimp > nR
      curr ← rnd_inst(), E ← {curr}, nonimp ← 0
    else if (nonimp + 1) mod nB = 0
      curr ← random element in E
    next ← NBH(curr, ⌊k · len(curr)⌋)
    SIM(next, l)
    nofsims ← nofsims − 1
    if RT(next) > hrt
      hrt ← RT(next), best ← next
    if RT(next) > RT(curr)
      curr ← next, E ← {next}, nonimp ← 0
    else
      nonimp ← nonimp + 1
      if RT(next) = RT(curr) then E ← E ∪ {next}
return best
```

The HCRR algorithm is given in Algorithm 5. The simulation budget,

i.e., the allowed number of simulations, is denoted *nofsims*, and $\mathrm{RT}(q)$ is the highest response time for the task in focus of analysis, performed using *SIM*, with respect to the simulation instance *q*. Like in the MABERA algorithm, *SIM* refers to running an RTSSim simulation, although the interface is different.

The simulation time instant when a simulation input X_i^j is consumed is expressed as TM_i^j and $q[X_i^j]$ is the value of X_i^j in the simulation instance *q*. A random simulation instance is generated using the function $\mathrm{rnd\_inst}()$.

The HCRR algorithm begins by choosing as starting point the best simulation instance from m randomly selected candidates, which are evaluated by performing simulations using *SIM*. Then, in each iteration, $k \cdot \mathrm{len}(\textit{curr})$ random values of the current simulation instance *curr* consumed before $\mathrm{ET}(\textit{curr})$, are selected and modified using the neighborhood procedure NBH, shown in Figure 4.3.

In this description, $ET(\textit{curr})$ denotes the end time of the task instance which produced the highest response time for the task in focus in the simulation instance *curr*, and $\mathrm{len}(\textit{curr})$ denotes the total number of input values in the simulation instance $curr$.

NBH(*inst*, n)
  for $k = 1$ **to** n
    X_i^j = random element in *inst*, $\mathrm{TM}_i^j < \mathrm{ET}(\textit{inst})$
    $V = \{\mathrm{lb}(\mathbf{X}_i) \ldots \mathrm{ub}(\mathbf{X}_i)\} \setminus \{\textit{inst}[X_i^j]\}$
    inst$[X_i^j] \leftarrow$ random value in V

Figure 4.3: Neighborhood procedure of HCRR

The response time for the task under analysis is measured by running a simulation using the $\mathrm{SIM}(\textit{next})$ call on a neighbor *next*. The modifications suggested by NBH are accepted only if they increase the response time. Modifications that have equal response time are rejected but saved for future reference, as described below.

A pure hill-climbing procedure is susceptible to getting stuck in local maxima, and can therefore exhibit less than satisfactory performance on many problems. In order to improve the probability of finding a true global maximum, two different diversification mechanisms are used. First of all, the algorithm jumps back to a previously encountered, randomly selected simulation instance with an equal response time to the current instance after a number of non-improving simulations, denoted *nB*. This distributes focus over a number of equal instances, which can help in avoiding small local maxima. The second

mechanism performs a full restart of HCRR from a random point after a number of non-improving simulations, denoted by *nR*. We call *nB* the *jump-back threshold* and *nR* the *random-restart threshold*.

4.5 Evaluations of MABERA and HCRR

This section present evaluations of MABERA and HCRR and a comparison between these two methods, traditional Monte Carlo simulation and an analytical method for response-time analysis, RTA [101]. This is done using three simulation models: two representing industrial cases and one simplified validation case, which unlike the other two models can be analyzed using RTA. The models have similar architecture and analysis problems as two industrial real-time applications in use at ABB [118] and Arcticus Systems [120]. Although the simulation models contain relatively few tasks, at most 11, their behavioral complexity is significant due to, e.g., shared variables, sporadic events and dynamic priority changes.

Model 1 is representing a control system for industrial robots developed by ABB Robotics, a complex embedded system which violates the assumptions of analytical response time analysis methods through several types of intricate task dependencies.

Model 2 is constructed from a test application used by Arcticus Systems [120], which develops the Rubus RTOS used in many vehicular systems.

A less complex version of Model 1, labeled MV, is used for validation purposes. Unlike the other models, MV is possible to analyze using RTA [101]. The purpose of this model is to investigate how close the response times found by MABERA and HCRR are to the true worst-case response times, derived using RTA.

The scheduling policy is preemptive priority-based scheduling for all models. Model 2 and the validation model uses strictly fixed priorities, while Model 1 contains one task that alters between two priority levels depending on the system state.

4.5.1 Model 1

This model is inspired by the IRC 5 control system for industrial robotics, developed by ABB [118], and is described in detail by Appendix C. The ABB Robotics system is quite large, containing around 3 million lines of code and is not analyzable using traditional analytical methods, such as RTA. Model 1 is

of much smaller scale but is designed to include behavioral mechanisms from the ABB system which RTA cannot take into account:

- tasks with intricate dependencies in temporal behavior due to IPC and shared state variables;
- the use of buffered message queues for IPC, where triggering messages may be delayed;
- tasks that change scheduling priority or periods dynamically, in response to system events.

The modeled system controls a set of (fictive) electric motors based on periodic sensor readings and aperiodic events. The calculations necessary for a real control system are, however, not included in the model; the model only describes behavior with a significant impact on the temporal behavior of the system, such as resource usage (e.g., CPU time), task interactions and important state changes.

4.5.2 Model 2

This model is based on a test application from Arcticus Systems, developers of the Rubus RTOS [120] which is used in heavy vehicles. This model uses a pipe-and-filter architecture, where tasks trigger other tasks through trigger ports, forming transactions. The model contains three periodic transactions and one interrupt-driven task, in total 11 tasks. The interrupt has a small jitter, while the other transactions are strictly periodic.

This model is less complex than Model 1 in the sense that there exist no shared variables or IPC via message passing which can impact the tasks' timing and functional behavior. Instead, the tasks have large variations in execution times, which makes the state space of this model very large. For this model, the evaluation focuses on the end-to-end response time of the transaction which contains the tasks with the lowest priority. More details of the model can be found in [16].

4.5.3 The Validation Model

Simulation-based methods for response-time analysis have in common that the result is not guaranteed to be a safe upper bound on the response time. We therefore constructed a validation model, analyzable using RTA, with the purpose to investigate how close the response times given by HCRR are to the

worst-case response times derived using RTA. Hence, RTA provides an upper bound on the worst-case response time that the simulation-based results should approach but not exceed. The validation model is based on Model 1, but simplified in that 1) shared state variables have been removed, 2) priority and period is strictly static for all tasks, and 3) static loop bounds have been added manually.

As a consequence, the validation model has considerably lower complexity, and exhibits quite different timing properties when compared to Model 1. For instance, the worst-case response time of the CTRL task (the task in focus in Model 1 and MV) is in MV only 52 % of the highest response times found for this task in Model 1. Note that this response time is known to be the true worst case for MV, since it could be verified using RTA.

Direct application of RTA yielded a worst-case response time of 5 982. However, after reviewing the results of running HCRR on the model, it was realized that a refinement was possible in order to reduce the pessimism of the RTA. The DRIVE task was modeled as two separate tasks, which represent two different WCETs of the DRIVE task. The higher WCET may only occur if a rare sporadic event has just occurred, where the minimum inter-arrival time is known and much larger than the period of the CTRL task and the DRIVE task. Therefore, only one such case may occur during a single CTRL task instance, while, in contrast, several normal DRIVE task instances (i.e., with the lower WCET) may preempt a CTRL task instance. This refinement of the model had a major impact with respect to RTA, yielding a worst-case response time of 4 432 (refined model) instead of 5 982 (without refinement). However, it is important to realize that such model refinements are hard to apply in practice, for real industrial systems, as the temporal behavior of such systems are rarely documented in detail.

Note that MV is a separate case with quite different behavior and not comparable with Model 1. The RTA results from MV are only intended as a reference for MABERA and HCRR on the same model, i.e., MV, they are not applicable to Model 1 (e.g., as approximations).

4.6 Experimental Evaluation

This section presents an evaluation of accuracy, convergence and scaling properties of HCRR, MABERA and Monte Carlo simulation using in total six different versions of the three models described in Section 4.5.

The goal of the analysis is to find extreme response times for a specific

task in the model. The results are obtained from running 100 samples of each algorithm and test case, each sample being allowed to run 10 000 simulations, except for in the case shown in Figure 4.4. The simulation budget was considered reasonable due to the convergence of HCRR on the most realistic and complex model, Model 1. The experiments were performed on a computer equipped with a Intel Core 2 Duo CPU at 2.33 GHz, with 2 GB of RAM.

The MABERA parameters used for the evaluation was a population size of 1 250 and 12 parents in each generation, which reflects the p/s quota of 0.01 found suitable in earlier experiments on Model 1. In order to ensure that MABERA used exactly 10 000 simulations in total, to be comparable with the HCRR results, the original termination threshold was changed. Instead of using a termination threshold, the termination occurs when the simulation budget of 10,000 individual simulations has been used up. The population size of 1 250 was selected since it allows for 8 generations, which is the average number of generations per MABERA run in the initial MABERA evaluation [15].

Regarding the parameters of HCRR, the jump-back threshold (*nB*) should be relatively small to spread the search over the set of equal candidate solutions found so far. However, the random restart threshold (*nR*) should be larger in order not to erase any progress made so far, but small enough to force restart from a local maximum as soon as possible. The fraction *k* of input values changed in each iteration should provide a good balance between power (larger fractions) and low dimensionality (smaller fractions).

To decide the HCRR parameters, *k*, *nB* and *nR*, three experiments were performed using Model 1 where one parameter at a time was determined, as described below. For each evaluated parameter combination a measure of convergence rate was calculated, C, which is the mean value over 20 sample runs, on the highest response time found after j simulations. Formally the convergence measure is defined as:

$$C = \frac{\sum_{i=1}^{20} \sum_{j=1}^{S} R_i^j}{(20 \cdot S)}$$

where S is the number of simulations and R_i^j denotes the response time found after j simulations in sample run i. The simulation budget used in these experiments was 500 for *nB* and *k* and 3 000 for *nR*. The parameters giving quickest convergence ($nB = 2$, $nR = 300$, and $k = 0.02$) were then used for all experiments. The results of the experiments are shown in Table 4.5.

To show the effects of scaling on the three algorithms, more complex models are created by instantiating several independent instances of Model 1, as

Table 4.5: Parameter selection for HCRR

$nB = nR = \infty$		$k = 0.02, nR = \infty$		$k = 0.02, nB = 2$	
k	C	nB	C	nR	C
0.01	7 796.76	100	7 931.37	1 000	8 308.11
0.02	**8 010.90**	50	7 902.86	300	**8 312.05**
0.03	7 988.83	20	7 939.70	100	8 304.17
0.04	7 976.14	10	7 972.72	50	8 254.26
0.05	7 961.80	7	7 992.25		
0.07	7 944.69	5	7 944.27		
0.10	7 761.59	4	8 001.89		
0.15	7 645.62	3	7 919.24		
0.20	7 604.48	2	**8024.98**		
0.30	7 483.33	1	7 944.27		

"subsystems", where each subsystem is a complete model as described in Section 4.5. The only dependency between the subsystems is that they share the same CPU and therefore will interfere with respect to timing. The subsystems are however time-separated by relative offsets of 20 000 time units in order to even out the CPU usage, and have reassigned priorities since RTSSim assumes unique task priorities. All execution times (inputs for *Execute* calls) were scaled to avoid overload. Scaling factors used were $1.0, 1/1.5, 1/1.8$ and $1/2.2$ for $1, 2, 3$ and 4 subsystems, respectively. The scale factors were found experimentally, with the criteria to give an interesting, complex task-level behavior while avoiding overload and task input starvation.

4.6.1 Results

The response time results from Monte Carlo simulation (MC), MABERA (MAB) and HCRR were obtained in the following manner, using a simulation budget of 10 000 individual simulations unless otherwise stated.

MC: The traditional Monte Carlo approach, giving the highest response time found during a specific number of simulations.

MAB: The MABERA algorithm presented in Section 4, using a population size (s) of 1 250 and 12 parents per generation ($p = 12$), which gives a p/s quota of 0.0096.

HCRR: The HCRR algorithm presented in Section 4.4.2, using parameters $nB = 2$, $nR = 300$, and $k = 0.02$.

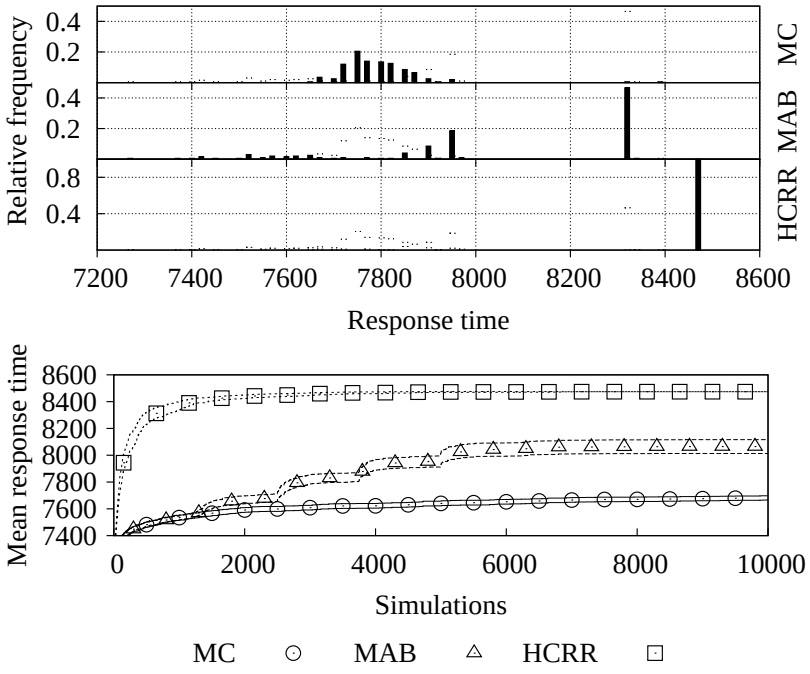

Figure 4.4: Final RT distributions and convergence for Model 1

Figure 4.4 shows the results obtained for Model 1 from Section 4.5.1. The top of the figure contains the response time distributions of the three algorithms, where the results for MABERA and MC are taken from the original evaluation of MABERA [15], produced using 200 sample runs (replications), which in total required 16 280 000 individual simulations.

For HCRR, only 100 sample runs were used, using 10 000 individual simulations each. This gives a total of 1 000 000 simulations. This was considered sufficient, since all 100 runs of HCRR found the highest known response time, 8 474, for the CTRL task of Model 1. None of the MABERA or MC runs found this response time. The bottom of Figure 4.4 shows convergence (mean RT and 95 % confidence intervals) for the three algorithms, using 100 replications of each algorithm, each using 10 000 simulations.

The highest response time found by MABERA was 8 349, and this value was only found one single time in 200 runs, using in total 16 280 000 simula-

tions. However, a value of 8 324 was found in 47 % of the cases. MC only produced two results are over 8 000, while the rest follows a seemingly gaussian distribution with a median around 7 800. The highest MC outlier, 8 390, is however higher than the MABERA results. Note that this is found only once in over 16 million simulations, with a total run time of 24 hours. With a significantly smaller (more typical?) simulation budget, MC would most likely not have found any of the outliers, only results below 8 000, while MABERA would most likely have found the 8 324 case even if using few replications (e.g. three or four, instead of 200), since it was found in almost 50% of the runs.

HCRR is however far superior to MABERA and MC, using only about 6 % of the number of simulations used by the other methods. Every HCRR run found the highest response time, 8 474, using only 10 000 simulations which means that HCRR was 1 628 times faster than MABERA in this case, since only a single HCRR replication is necessary in order to find 8 474 while MABERA requires 200 replications in order to find its highest result, 8 349. The runtime per sample run was 7 minutes in the earlier MABERA evaluation [15], where each MABERA sample used on average 81 400 simulations. This translates to 5 ms per individual simulation, including the optimization algorithm code. Since the length of the simulated scenario was 650 ms, this means that the simulation speed was 123 times faster than real execution in this case. For the cases where 10 000 simulations was used per replication, both MABERA and HCRR required less than 3 minutes per run.

Figure 4.5 shows the obtained results for Model 2 (Section 4.5.2). In this model, the tasks have large variations in execution times, which makes the state space very large. We can see that HCRR yields a result approximately 5 % higher than what is obtained from the two other methods. Interestingly, it looks like HCRR was still slowly progressing towards higher response times at 10 000 simulations, while both MABERA and MC seems to have converged quite early to a much lower result. MABERA seems to give lower results than MC in average but finds a higher maximum value. For Model 2, all algorithms finished in less than one minute per sample.

In Figure 4.6, we can see the results for the validation model (MV) described in Section 4.5.3, again using the standard parameters. In addition, we show the RTA results. Here, HCRR could find a response time of 4 432 in every sample run, which was also confirmed by RTA to be the worst-case response time. The difference between MABERA and MC appears to be quite small in this case, but MABERA found the worst case in a few cases, while MC did not.

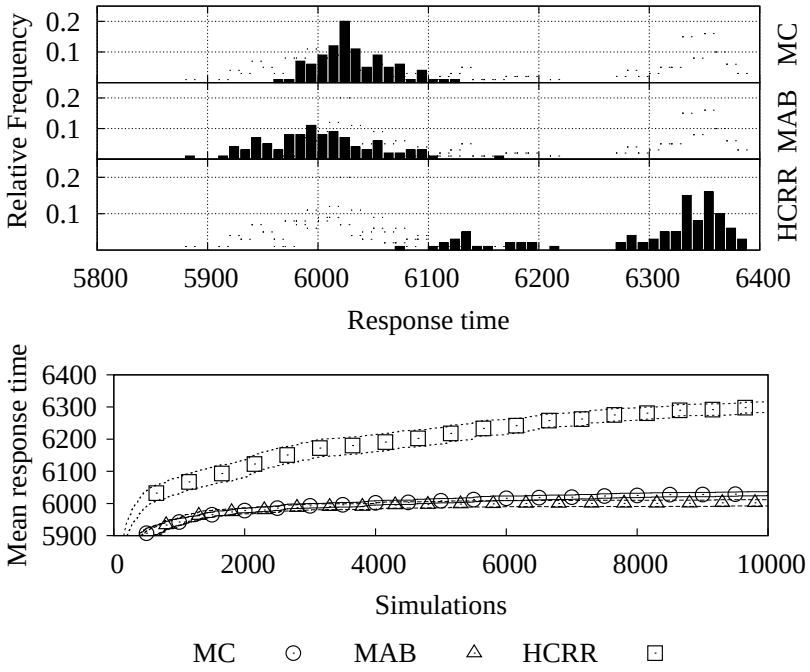

Figure 4.5: Final RT distributions and convergence for Model 2

Figure 4.7 shows how the different methods scale to larger systems, by illustrating the convergence for Model 1 when increasing the model size to 2, 3 and 4 subsystems (model instances). As expected, since the state space increases with number of subsystems, all three algorithms converge slower when system size is increased. HCRR is consistently the best, while MABERA and MC give quite similar values, although MABERA gives slightly higher average results than MC in all cases, and significantly higher for the middle case, with three subsystems. As the number of subsystems increase, the difference in performance between the methods decrease, although HCRR produced on average 4.7 to 11 % higher results than both MC and MABERA. For 4 subsystems, none of the methods appear to have converged. However, during the 10 000 simulations, HCRR progressed more quickly to higher response times than both MC and MABERA. Table 4.6 presents the run times (in minutes) for a single run of the algorithms on the three models (M1-2 to M1-4), which

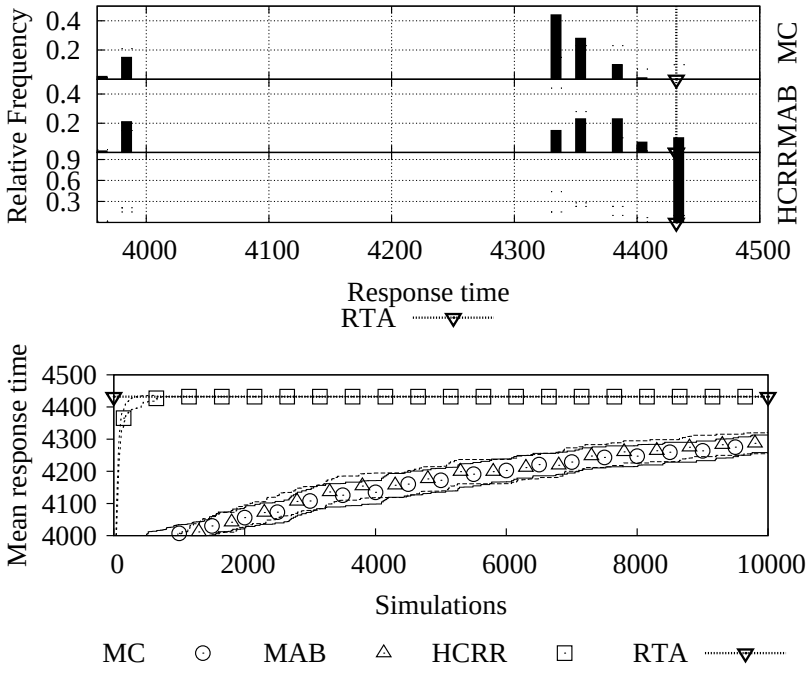

Figure 4.6: Final RT distributions and convergence for the validation model

shows the impact of increasing model complexity on run time, caused by longer run times of the individual simulations.

Note that the same parameters for MABERA and HCRR were used for all models, although these parameters were selected with respect to Model 1. Their suitability for the other models are not known. If the best parameters were re-evaluated for each model, better results might have been achieved.

The average end results are summarized in Table 4.7. The last column also shows the average number of simulations needed for HCRR to reach the end result of the second best method (usually MABERA), obtained in 10 000 simulations. Overall, HCRR reached the second-best result 13 to 112 times faster than the second-best method did. For all models, it took HCRR less than 800 simulations to reach the results of the other methods, which corresponds to less than 1.5 minutes of computation time on the PC used for experiments.

Table 4.6: Run times of Monte Carlo, MABERA and HCRR (minutes)

	MC	MABERA	HCRR
M1-2	4	7	5
M1-3	5	10	6
M1-4	8	16	10

Table 4.7: Average end result of Monte Carlo, MABERA and HCRR

	MC	*MABERA*	*HCRR*	*Passes 2^{nd} best*
M1-1	7 682	8 065	8 474	224
M1-2	9 693	9 750	10 844	238
M1-3	13 555	13 789	14 672	521
M1-4	15 235	15 298	16 013	764
M2	6 031	6 002	6 299	634
MV	4 286	4 288	4 432	89

4.6.2 Average Convergence

To measure average convergence more exactly, we use the relative difference in average response-time results over the last d simulations. We say that a method has for practical purposes converged (on average) when

$$\frac{1 - \overline{R}^{(k-d)}}{\overline{R}^{(k)}} \leq \varepsilon$$

where $\overline{R}^{(k)}$ is the average response-time result at simulation k for a set of samples. Using this definition, convergence will never be detected before at least d simulations has been performed. In order to measure convergence for the evaluation presented in this paper, d obviously needs to be less than the number of simulations (10 000) performed in each sample. We therefore use $d = 1\,000$ for the convergence comparison. For the tolerance parameter, we chose a value of $\varepsilon = 0.001$. In other words, if the average progress in 1 000 simulations is lower than 0.1 %, we declare that the method has converged on average. It should be pointed out that different parameters will give radically different results on convergence. Detecting true true convergence would require that $\varepsilon = 0$ and d is infinite (or in practice, at least very large).

Table 4.8 summarizes the convergence results obtained with the parameters above, for Model 1 with 1 – 4 subsystems (M1-1 to M1-4), Model 2

Table 4.8: Convergence of Monte Carlo, MABERA and HCRR

	MC		*MABERA*		*HCRR*	
	k	$\overline{R}^{(k)}$	k	$\overline{R}^{(k)}$	k	$\overline{R}^{(k)}$
M1-1	7 632	7 670	7 356	8 062	4 090	8 466
M1-2	4 806	9 660	6 518	9 728	7 093	10 830
M1-3	3 527	13 502	7 801	13 773	5 568	14 578
M1-4	3 410	15 175	5 104	15 271	6 948	15 881
M2	3 656	5 997	3 552	5 991	9 556	6 295
MV	–	–	–	–	1 661	4 432

(M2), and the validation model (MV). In general, we can see that HCRR converged to significantly higher response times than MABERA and MC. For the validation model, the only method to converge within 10 000 simulations was HCRR. Overall, the results are mostly consistent with what can be seen in Figure 4.4 to Figure 4.6, but also classified the slow progress for HCRR on M2 in Figure 4.5 as convergence. Running the algorithm longer would either yield slightly higher results or confirm convergence.

For M1-4, convergence of HCRR is also detected in iteration 6 948 after a slow progress between simulation 6 000 and 8 000, but as we can see in Figure 4.7, more average progress is made after simulation 8 000. Sampling more than 100 runs for M1-4 would most likely even out the slope after simulation 6 000. In any case, HCRR has clearly not converged after 10 000 simulations, and running the algorithm longer would likely yield even higher results.

4.7 Conclusions

The results presented in this chapter indicate that simulation optimization algorithms such as MABERA and (especially) HCRR has the potential to provide engineers with accurate extreme value predictions regarding run-time properties of embedded systems, such as task response times, also for complex systems not conforming to classical real-time analysis models such as RTA. Note however that the simulation-based approach implies a best-effort analysis, which only provides a lower bound for the worst-case response time, i.e., the highest response time found. This is not necessarily the worst-case response time. In the evaluation performed, six different simulation models were used, developed to represent analysis challenges of real industrial real-time systems.

The results indicate that MABERA is significantly more efficient than Monte Carlo simulation, but HCRR was found to be 4 – 11 % more accurate than MABERA and between 13 to 112 times quicker in reaching the end result. An analysis of convergence indicates that for two cases out of six, even higher response times could be achieved by allowing HCRR more simulations.

Both HCRR and MABERA require parameters, which impact their performance. Finding suitable parameter values for MABERA is quite time-consuming, since the MABERA parameters are not independent and therefore has to be evaluated in combination. Suitable parameters for HCRR can be found much faster since each parameter can be optimized independently.

MABERA seems to be more dependent on good parameters than HCRR; while MABERA performed quite well on Model 1 (for which the parameters had been optimized) it was only marginally better than Monte Carlo simulation on the other models. This might be due to larger state space in some cases, but MABERA is only marginally better than Monte Carlo also for the MV model, which is less complex than Model 1. It is likely MABERA would have performed better on MV (and the other models) if the parameters would have been tuned. The possible parameter sensitivity is a serious drawback of MABERA and speaks for HCRR. It would be interesting to repeat the MABERA and HCRR runs with parameters optimized for each model, but this is quite time-consuming work and it is quite clear that HCRR is superior compared to MABERA and Monte Carlo simulation.

Future work includes evaluating HCRR (or an improved method) on models extracted from real industrial systems, using the model extraction approach presented in the coming two chapters. The models used for the evaluation in this chapter is very small compared to such systems, which makes the efficiency of the model extraction very important, i.e., the size and complexity of the resulting models. Even though HCRR seems to be very efficient, it will not perform as well on models which are several magnitudes larger.

If HCRR would turn out to have insufficient scalability with respect to large industrial systems, improvements are however possible. HCRR performs a quite simple type of optimization, without any knowledge of the dependencies in the simulation model. This has shown to work well, but could possibly be made “smarter” by logging additional information during the simulations, regarding the context in which each input value is used. Thereby, it would be possible to put more focus on optimizing the input values with high likelihood of being relevant, such as inputs used by the task in focus (e.g, to decide execution time variations) or by other relevant tasks which preempt, block or communicate with the task in focus.

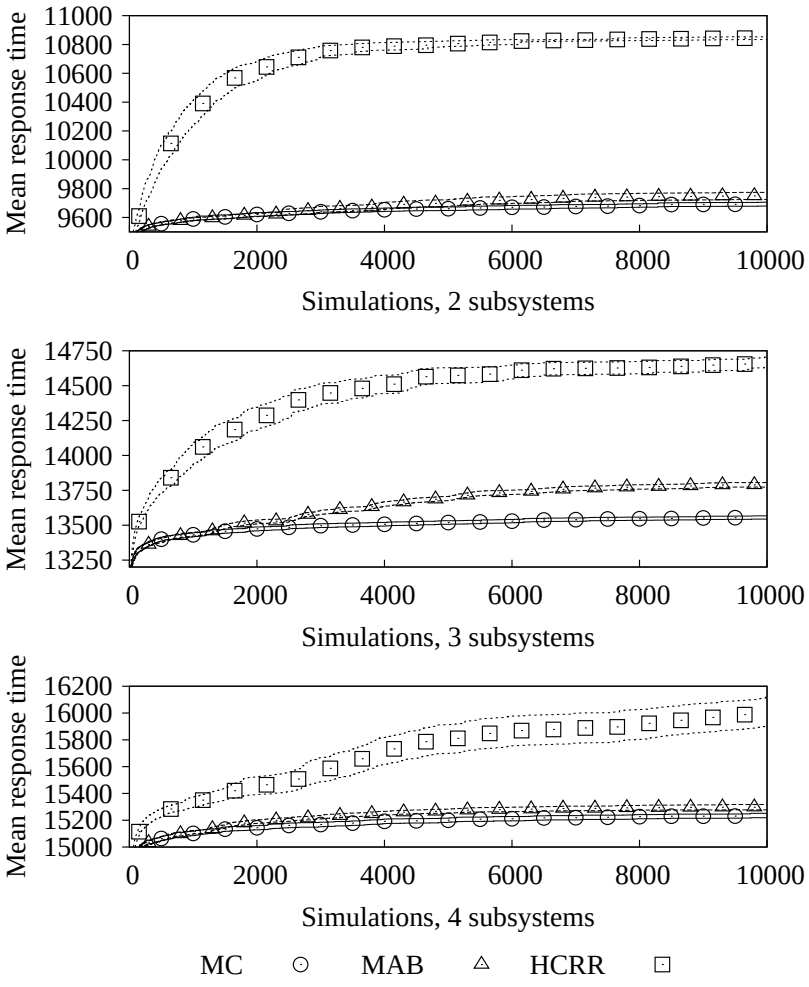

Figure 4.7: Convergence for Model 1 using 2-4 subsystems

Chapter 5

A Method for Automated Model Extraction

Patent Pending

This chapter presents a method for automated extraction of simulation models from complex embedded software systems implemented in C. The method is intended to be realized as a software which as input takes the system source code and execution-time measurements, and which outputs a filtered, more abstract version, an RTSSim simulation model as described in Chapter 3.

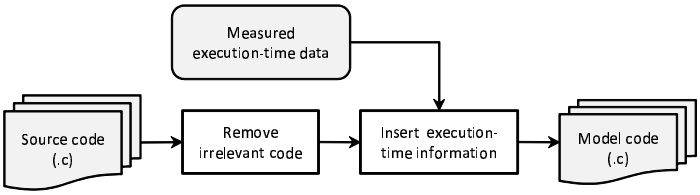

Figure 5.1: Overview – simulation model extraction

The model extraction process contains two steps, as illustrated by Figure 5.1, where the first step identifies the source code of relevance for the simulation model, while the second step allows for timing-accurate simulation using target system execution-time measurements. This thesis focuses on the first step, the source code “filtering”, which serves to reduce the size of the simulation model.

Note that the first step can theoretically be omitted, by using the full source code as simulation model (extended with timing information). There are however several good reasons for reducing the amount of model code:

- Faster simulations and smaller model state-space, allowing better accuracy and confidence when using methods like MABERA or HCRR, presented in Chapter 4.
- Probably fewer uses of library functions (i.e., without source code), which might need to be manually replicated in the simulator framework.
- Fewer instrumentation points necessary for execution-time measurements.
- Better system understanding, at least if the reduction in size is large.

The second step enables timing-accurate simulation through automatic insertion of *execute* statements, i.e., calls to an RTSSim function which advances the simulation clock, according to execution-time information from measurements. The measurements are performed between instrumentation points, inserted in the original code at locations derived during Step 1.

The process of removing irrelevant code is commonly known as *program slicing*, a concept first proposed by Weiser [26] which implies an analysis which given a program and a *slicing criterion* identifies the program statements of relevance. The most common type of program slicing, backwards slicing, identifies all statements which may impact the value of a particular symbol (e.g., a variable) at a particular point in the program were the symbol is used. The area of program slicing is further described in Section 2.4.1.

The existing methods and tools for program slicing are however not scalable enough (cf. Section 6.4) for analysis of large industrial software systems, which may consists of millions of lines of code. Moreover, program slicing for model extraction requires a different type of slicing criteria. A new method for program slicing named[1] *Katana* has therefore been developed.

Katana identifies all statements of a program that impacts the execution of a set of functions, the *model focus functions*, which constitute the slicing criteria and are provided as input. They are the functions which directly impact the run-time properties in focus. They typically correspond to operating system services. In the context of this thesis, the model focus functions should include all functions which impact the task scheduling, e.g., functions causing task triggering, blocking or priority changes.

[1] Katana is named from the very sharp Japanese sword (another type of efficient slicer).

Note that this approach assumes that all relevant source code is available. In cases where "black-box" components are used, like an SQL database, they might also perform actions of relevance to the model, e.g., spawning tasks or locking a semaphore. In order to model black-box components, there are approaches to modeling based on dynamic analysis techniques, i.e., based on run-time monitoring, such as the works by Huselius et al. [56] and Jensen [49, 116]. Such methods could be used as a complement to model extraction from source code. This is however not further explored in this thesis.

The Katana algorithm has been implemented in a prototype tool named *MXTC*, an abbreviation of **M**odel e**X**traction **T**ool for **C**. MXTC is presented in Chapter 6 together with an evaluation of MXTC on industrial code.

This chapter is organized as follows: Section 5.1 presents the Katana algorithm in an informal manner using examples and illustrations, and with focused subsections highlighting different aspects of the solution. Section 5.2 discusses the efficiency of the proposed solution, with respect to runtime and memory usage. Section 5.3 gives a formal definition of the core parts of the Katana algorithm, and Section 5.4 presents the lower layer functionality assumed by the formal definition. Section 5.5 relates the Katana approach to existing academic and commercial solutions for program slicing. Finally, Section 5.7 concludes this chapter with a discussion on current limitations and future work.

5.1 The Katana Approach to Program Slicing

The dominating approach to program slicing since the mid 1990's is the *System Dependence Graphs* [23], or *SDG*, which describe the dependencies between a program´s statements and symbols. A SDG vertex represents a statement, e.g., a condition, assignment or function call, while an SDG edge corresponds to a dependency in control-flow or data-flow. The SDG approach and other types of program slicing are presented in Section 2.4.1. In the SDG approach, program slicing is performed by traversing the system dependence graph in a reachability search, starting from the nodes matching the slicing condition. To allow analysis of programs containing pointers, a separate "points-to" analysis is also required before the program slicing, where all pointers are analyzed in order to find all possible variables they may refer to at each reference of the pointer. The construction of the SDG and the points-to analysis implies a very detailed analysis of the entire program, which for large programs can take very long time and requires vast amounts of memory, as presented in Section 5.5.

Katana does not use the SDG approach, but instead uses a *Symbol Database*

as program model, as depicted by Figure 5.2 and Figure 5.4. The symbol database can be regarded as an index over the source code and can be constructed using a lexical scan, possibly after applying a preprocessor. Since no advanced models of the code are constructed, like CFGs or even ASTs, this is a very fast analysis, which is done in a matter of minutes also for systems with millions of lines of C code, as demonstrated by the evaluation results presented in Chapter 6.

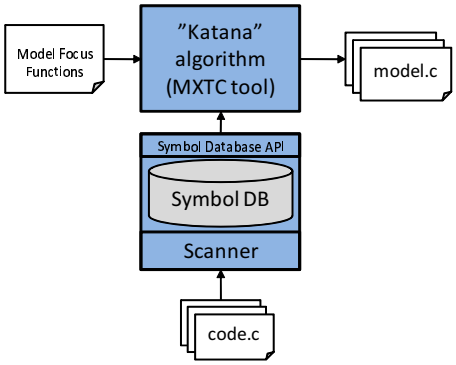

Figure 5.2: The context of the Katana algorithm

The symbol database contains three types of entries: *symbols*, *references* and *lexemes*. A *symbol* corresponds to a variable, function or user-defined datatype. The *lexemes* represent the analyzed source code in a tokenized manner according to the syntax of the programming language, in this case C. A lexeme contains the corresponding source code text, location (line and column number) and references to the previous and following lexeme. Lexemes are classified in types, such as keywords, operators, delimiters, identifiers and literals. A *reference* is an entry which connects a symbol with the code locations (lexemes) where rge symbol is used. Reference have types, indicating the context in which the symbol is used, e.g, if the symbol is assigned, used or called.

The different types of entries are connected by bidirectional links, which connects symbols with references and references with lexemes. The structure (or metamodel) of the symbol database assumed by Katana is presented in Figure 5.3. Note that the presented attributes is not a complete list, but only examples.

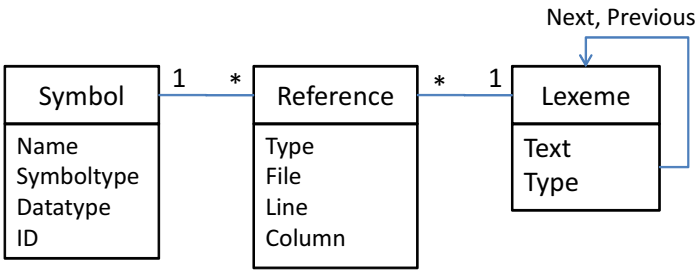

Figure 5.3: The structure of the symbol database representation

As depicted by the UML class diagram in Figure 5.3, each symbol may have multiple references, but a reference always has exactly one symbol and one lexeme. Note that a lexeme may have multiple references if they are of different types and to the same symbol. An example is the following case:

```
foo = bar++;
```

In the above example, the lexeme bar and the corresponding symbol bar will be connected by two references, one "use" (in assignment source) and one "modify" reference. An example of a symbol database structure is given in Figure 5.4. The structure of the symbol database is inspired by the database API of the commercial reverse-engineering tool "Understand for C++" [138]. This API was used as base for the Katana prototype implementation, the MXTC tool presented in Chapter 6.

The symbol database makes it easy to look up a specific symbol and find all locations (as lexemes) where the symbol is used in a particular way, e.g., assigned. Each location (statement) can then be analyzed on the lexeme level in order to identify other symbols of relevance, on which the first symbol depends. These new symbols are thereafter analyzed recursively. This continues until all relevant symbols have been analyzed. This approach has some similarities with the original Weiser approach [26], which however used a very different program model, control-flow graphs, and handles pointers and function calls quite differently.

The target of the Katana method, in the context of model extraction, is to find all statements which impact the execution of the model focus functions

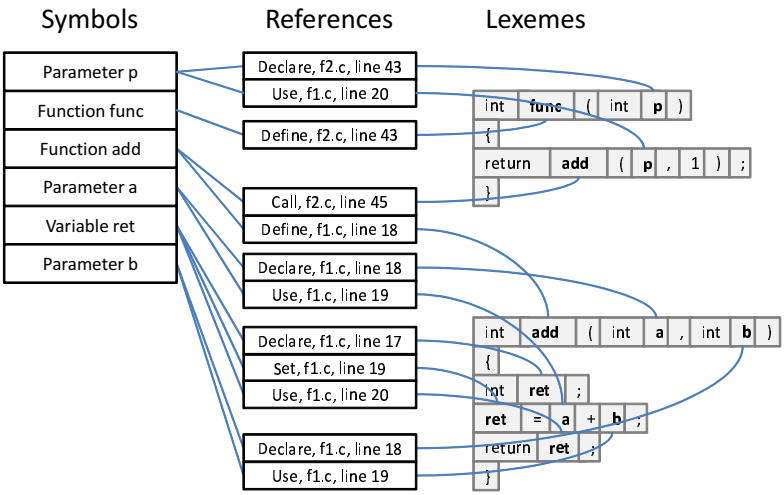

Figure 5.4: An example of a symbol database

(i.e., the slicing criteria). The output should contain all statements which directly or indirectly decide when the control flow reaches a model focus function, and all statements which directly or indirectly impact the arguments used in calls of model focus functions.

The term *model statement* is used to specify the statements found to be relevant, i.e., that should be included in the output. Concretely, a model statement is represented using the first lexeme of the statement. When a conditional statement (i.e., a loop or selection) is recognized as a model statement, this does not include the enclosed statements within the associates block(s), but only the condition, block delimiters and relevant keywords.

A *model symbol* is a symbol in the symbol database, of types variable, parameter or function, which is found in at least one model statement. A more precise definition is given below.

A *model function* is a function containing at least one model statement. Formally, model functions are however represented as model symbols of function type. This since a function name may refer to different things depending on the context, the function definition, the function address, or the function return value. Katana treats function return values as a two-step assignment, where the

function symbol is a temporary variable between the returned expression and the use of the return value in the calling function.

There are two types of model functions, *globally model-relevant* (GMR) functions and *locally model-relevant* (LMR) functions. A GMR function is a function containing side-effects of relevance for the model:

- model focus functions (specified as input),
- functions containing assignments of global variable model symbols, or
- functions calling other GMR functions.

Thus, all calls to GMR functions are relevant for the model. The other type of model functions is locally model-relevant (LMR) function. For LMR functions, only calls found in model statements are relevant. For instance, consider that the variable `foo` is known to be relevant and the following model statement is found:

```
foo = bar(i);
```

The assignment depends on the return value of `bar()`. If the formal parameter of `bar()` is found relevant for the return value, only the argument of the specific call is analyzed. Other calls (and arguments) of `bar()` are ignored in this case, but will naturally be included if found model-relevant for other reasons.

Given the above introductions, a model statement is defined as any statement matching at least one of the following rules:

- Rule A: Calls of model focus functions.
- Rule B: Calls of globally model-relevant (GMR) functions.
- Rule C: Assignments of model symbols (direct or by reference).
- Rule D: Conditions guarding the execution of model statement(s).
- Rule E: Return statements in any model function.
- Rule F: Break or continue statements where the closest encapsulating loop is a model statement.
- Rule G: Statements obtaining or forwarding pointers to model symbol(s).
- Rule H: Declarations of model symbols or model symbol datatypes.

The above definition of model statements depend on the concept of a model symbol. A *symbol* is in this context a parameter, variable, constant or function return value, of any datatype (including pointers/arrays). A *model symbol* is defined as a symbol matching any of the following rules:

- Rule I: Symbols used in a condition of a conditional model statement.
- Rule J: Symbols used in assignments or initiations of model symbols.
- Rule K: Symbols used in a function call argument where the corresponding formal parameter is a model symbol.
- Rule L: Symbols used in return statements of functions which return values are used in at least one model statement.
- Rule M: Pointers to a model symbol S and used in a dereference assignment of S, or in data-flow leading to such a dereference assignment of S.

Note that the above list is not an algorithm description, but rather a specification of the statements and symbols that Katana should identify.

5.1.1 An Overview of the Katana Algorithm

The purpose of this section is to provide a conceptual understanding of the algorithm, a foundation for the later descriptions of individual algorithm aspects. Note that a detailed description of the Katana algorithm is provided in Section 5.6.

The first step in Katana model extraction is to identify all calls of model focus functions (MFFs). The MFF calls constitute the initial set of model statements and the functions in which these calls occur are globally model-relevant (GMR) model functions. Since all callers of GMR functions also are GMR functions, all functions and function calls in the incoming call-graph of the model focus functions are included in the model. Thereby, all function call paths leading to a model focus function have been captured.

The next step is to identify the variables and statements which decides when the control-flow reaches a model focus function and with what arguments. This is achieved using two types of program slicing, labeled *Slice* and *SmtSlice* (Statement Slice), which are recursively dependent as illustrated by Figure 5.5.

Slice takes a symbol as input and returns all statements which may impact the symbol (typically a variable) at any point. *SmtSlice* takes a statement as

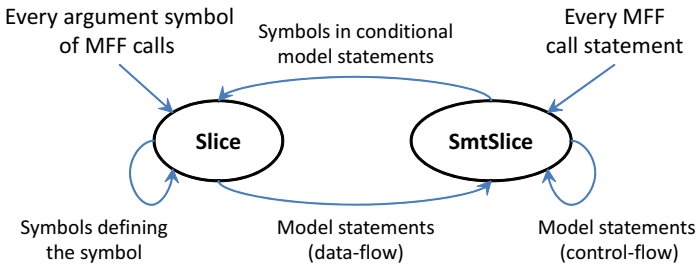

Figure 5.5: A high-level view of the Katana algorithm

input and returns all statements which may impact if and when the statement is executed. The analysis can start in either one, depending on the context. *Slice* is the starting point when analyzing the symbols used in arguments of MFF calls, while *SmtSlice* is applied to the MFF call statements.

Given a symbol, *Slice* analyzes all symbol references of relevant types in order to find statements where the symbol value is updated (assigned or modified), or where pointers to the symbol are created. For each identified statement, the symbols involved are analyzed recursively. If the symbol provided to *Slice* is a formal parameter, the relevant function calls are identified and the corresponding call arguments are analyzed recursively.

For each statement found relevant, *SmtSlice* identifies all enclosing control-flow statements (if, for, while, switch, etc.), which also are model statements. Their conditions are analyzed in order to identify symbols, which become model symbols. Each such symbol is analyzed recursively using *Slice*.

In order to avoid analyzing the same statement or symbol multiple times, the analyzed symbols are stored in a suitable data structure, e.g., a hash table, which is checked before starting each Slice operation. If an identical analysis of the symbol has already been performed, the Slice operation is aborted.

5.1.2 Katana on Example Code

A small example is used to illustrate the Katana algorithm, model statements and model symbols. The example program contains only two functions, `main()` and `controlTemp()`, which implements a very simple temperature supervision system. The program also uses some library routines for which the source

code is not available. In this case, they are safely ignored. The handling of library functions is presented in Section 5.1.8

```
 1: double controlTemp(double maxTemp)
 2: {
 3:     double currentTemp = readTemp();
 4:     if (currentTemp > maxTemp)
 5:     {
 6:         alarm();
 7:     }
 8:     return currentTemp;
 9: }
10:
11: void main()
12: {
13:     double mTemp = 80;
14:     double currTemp = 0;
15:     while(1 == 1)         // forever
16:     {
17:         currTemp = controlTemp(mTemp);
18:         displayTemp(currTemp);
19:         delay (500);
20:    }
21: }
```

In the above example, the function `alarm()` is a model focus function. By applying the earlier definitions of model symbol and model statement together with the informal algorithm description presented in the previous section, the following model statements are found:

1. Line 6: Due to Rule A, the call of the model focus function `alarm()` is a model statement. Thereby, `controlTemp()` is a globally model-relevant (GMR) function, since it calls a model focus function.
2. Line 8: The return statement is a model statement according to Rule E.
3. Line 17: Since `controlTemp()` is a GMR function, this call is a model statement according to Rule B, and `main()` is thereby a GMR function.
4. Line 15: The while-loop is a model statement since the loop encloses the previously found model statement at line 17.
5. Line 4: The if-statement is relevant according to Rule D, since it encloses the previously found model statement at line 6.
6. Line 4: When analyzing the if-statement condition, `currentTemp` and `maxTemp` are found and thereby become model symbols due to Rule G.

7. Line 3: When searching for assignments of the new model symbols from the condition at line 4, this assignment of `currentTemp` is found which thereby becomes a model statement according to Rule C.
8. Line 3: When analyzing the source of the assignment, the function `readTemp()` is found and therefore a model symbol according to Rule J. Specifically, this symbol is a locally model-relevant (LMR) function.
9. Line 17: The model symbol `maxTemp`, found at line 4, is found to be a formal parameter of `controlTemp()`. In the call at line 17, the corresponding argument expression is therefore relevant. This is found to contain `mTemp`, which thereby becomes a model symbol due to Rule K.
10. Line 13: When analyzing the new model symbol `mTemp`, the assignment at line 13 is found to be a model statement due to Rule C.

Note that the `readTemp()` function found at line 3 is a library function for which no source code is available, and since no argument is used, no further dataflow analysis is performed in that case. Functions assumed to be library functions (i.e., if they have declarations but no definition), must have corresponding implementations in the targeted simulator framework.

In this example, almost all statements are found to be relevant and thereby included in the output. This does however not reflect the typical performance of the model extraction on real programs, of non-trivial size. In the experiments on industrial code, presented in Chapter 6, the output size is between 3 – 59 % of the input program size.

5.1.3 Producing the Simulation Model

The recursive search of the symbol database identify model statements, which are represented by a reference to their first lexeme. In order to produce a compilable simulation model, the model statements (i.e., the lexemes representing them) are sorted based on their source file, line and column number. For each source file where model statements have been found, its model statements are written, lexeme by lexeme, to an output (simulation model) source file. Since also declarations of relevant variables, functions, and datatypes are model statements, the issue of declaring the model symbols (in the right order) is solved automatically. The file structure for the resulting simulation model source code thus reflect the file structure of original source code.

A last analysis step is however necessary before outputting the lexemes of the model statements. Model statements may include irrelevant symbols, which does not impact the behavior of the model but which may cause compile errors if not handled correctly. For instance, in the previous section presenting

the Katana example, the variable `currTemp`, declared at line 14, is not relevant for the model, since its value is never used in any model statement, but it is assigned in the model statement at line 17. In order to produce compilable simulation models, there cannot be references to variables which are not included in the model, since they would not be declared. Therefore, each lexeme of identifier type in every model statement needs to be checked if referencing an irrelevant symbol, i.e., a symbol which is not a model symbol. There are two alternative solutions for handling irrelevant symbol references: (1) transform the statement in order to remove any references of irrelevant symbols or (2) include irrelevant symbols as "dummy" symbols which declarations are model statements. Note that this issue has not yet been addressed in the MXTC tool (cf. Chapter 6). This is one of the main remaining issues before working simulation models can be produced by the MXTC tool.

5.1.4 Control Flow Sensitivity

Katana is not fully control-flow sensitive and may therefore produce simulation models which contain some irrelevant statements. Katana assumes that all assignments of a relevant variable are relevant, independent of where in the code they occur. Moreover, Katana assumes that every return statement in model functions are relevant since a conditional return statement may prevent later model statements in the local function from being executed. This is an over-approximation, since some of these statements might actually not impact the relevant statements. For instance, consider the following example, where a local variable is used for two different, unrelated purposes. Similar cases have been encountered several times when studying industrial code.

```
 1: void func(int p)
 2: {
 3:    int status;
 4:    status = foo(p);
 5:    if (status < 0)
 6:    {
 7:       query = 1;
 8:    }
 9:    status = bar(p);
10:    if (status < 0)
11:    ...
```

The assignment of the variable `query`, at line 7, is beforehand known to be relevant and is the starting point of the example. The condition at line 5 is

also relevant, since it guards the statement at line 7 (Rule D). This makes the variable `status` relevant, since it is used in the condition at line 5. Since Katana considers all assignments of a model symbol as relevant (Rule C), the assignment of `status` at line 9 will also be included, which implies that the function `bar()` is also included, without being actually relevant. The return statements of `bar()` may in turn include many other irrelevant statements.

Note however that Katana does not completely disregard control-flow, since it identifies all conditional statements which guard identified model statements. This is however a quite simple analysis, performed on demand, by searching backwards in the lexeme list. Katana does not construct any control-flow graph or similar for this purpose.

5.1.5 Handling of Function Calls

A problem in early approaches to program slicing, such as the interprocedural solution by Weiser [25], was the handling of function calls, i.e., to correctly follow dataflow through function call parameters and return values. According to Horwitz et al. [23], the "chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure". For instance, if the return value of a function `foo` is used in a relevant assignment or condition, it naturally makes the specific call of `foo` relevant, but this does not mean that all calls of `foo` are automatically relevant. Therefore, when analyzing a formal parameter in order to identify the relevant call arguments, it is important to include the relevant calls only.

Katana solves the context sensitivity problem in the following way. As previously introduced, Katana classifies model functions either as LMR (locally model-relevant) or GMR (globally model-relevant). A LMR function is a function which is not a GMR function, but found model-relevant since its return value is used in a model statement, or since the function takes a pointer to a model symbol as argument, used to modify the model symbol in the current function or in any callee, direct or indirect.

A GMR function is a function which calls a model focus function, or which assigns/modifies a global variable, or which calls another GMR function. Thus, the GMR functions correspond to the combined incoming call graph of (1) the model focus functions and (2) the functions assigning model-relevant global variables. The identification of GMR functions caused by model focus functions is the first step of Katana and is performed through a recursive upstream call-graph search, i.e., from callee to caller, starting in the model focus functions. Every function found in this search is a GMR function and the func-

tion calls followed are thereby model statements. The base condition is when a function is already marked GMR, or when no callers exists (i.e., the entry function). In this way, additional attempts to mark a function as GMR will not cause recursion, but only a single lookup on GMR status.

The main purpose of the LMR/GMR classification is when analyzing formal parameters. When a formal parameter is recognized as a model symbol, the analysis should naturally follow the dataflow upstream in order to analyze the corresponding call arguments, which implicitly assign the formal parameter. If the formal parameter belongs to a GMR function, all calls are relevant by definition. However, for LMR functions, only the call where the LMR function was encountered is considered relevant and thereby explored.

Note that a LMR function can change status to GMR, if it assigns a global variable which later is recognized as a model symbol, or if a callee later becomes GMR. In that case, the relevant formal parameters of the function needs to be analyzed again, in order to include all calls of the function. The calls already analyzed under LMR assumptions can be excluded from this analysis.

If an encountered LMR function already has been analyzed in a previous model statement, no *Slice* operation is performed on the function. This is detected using the *analysis cache* (cf. Section 5.2) and if this is the case, the dataflow analysis jumps directly to the function call arguments corresponding to those formal parameters already recognized as model symbols.

Recursive calls are handled correctly by Katana. The calls are recognized as model statements if relevant, but the functions are processed at most once, due to the analysis cache (cf. Section 5.2). Recursion is not really a problem in Katana — since no value analysis is used, there is no need for separating different instances of local variables during recursion.

5.1.6 Data structures

Katana allows for analysis of individual members of data structures through the use of *symbol expressions* and *symbol reference filters*.

A symbol expression is constructed from a symbol reference and represents an expression consisting of one or several identifier lexemes. The first identifier is labeled the *primary symbol*. This is followed by a list of data-structure field identifiers, if any, reduced into a minimal form as later described in this section. A symbol expression is represented by a list of *symbol specifiers*, each consisting of the symbol's datatype name (with any typedefs resolved) and the symbol name.

A symbol expression may represent a function call with respect to its return value, using the function name as the primary symbol. When the *Slice* function detects that the primary symbol is a function, it continues the recursive analysis with all symbols used in return statements of the function. Function arguments are considered as separate symbol expressions compared to the statement where the function is called (and the return value used). Arguments symbols are therefore not included in function symbol expressions. Any array index expressions also constitute separate symbol expressions.

A symbol expression is constructed from a symbol reference through a scan for identifiers, on the lexeme level. The scan starts at the lexeme corresponding to a reference of the primary symbol, and adds all encountered identifiers to the symbol expression until a disallowed lexeme is found, such as a semicolon or a comma. If a left bracket (of an index expression) or a left parenthesis (of a function-call argument list) is found, the scan jumps to the matching right parenthesis/bracket. The scan may find "odd" right parenthesis, where the corresponding left parentheses is before the lexeme where the scan started, e.g., due to type casting. Such right parenthesis are accepted without action.

Thus, a symbol expression corresponds to a list of adjacent identifier lexemes which only may be separated by (any valid combination of) the following lexemes:

- member operators, e.g, `foo.bar`" or `foo->bar`
- function call argument blocks, e.g., `foo(arg1, arg2)->bar`
- index blocks, e.g., `foo[i][j]->bar`
- right parentheses, e.g., `((MyType*)foo)->bar`

Note that the expressions within call argument blocks or index blocks are not ignored completely by Katana, but they become separate symbol expressions. Code examples with corresponding symbol expressions follow below.

Source code example	Symbol expressions
`int foo = bar;`	`(foo),(bar)`
`int* foo = & bar->f2;`	`(foo),(bar,f2)`
`foo[i] = func(j)`	`(foo),(i),(func),(j)`
`foo->bar[func(arg)].val`	`(foo,bar,val),(func),(arg)`
`func(arg)->val`	`(func,val),(arg)`

All examples contain at least two symbol expressions, which in the right column are delimited using parentheses. Note that the symbol type information is omitted from the field specifiers in this example.

Linked data structures, such as graphs or linked lists, may cause problems since they allow for multiple ways of referring to the same data structure field, through the same primary symbol. For instance, in a linked list there is no point in separating between `p->next` and `p->next->next`. There is naturally a semantic difference between the expressions, but Katana only identifies potential dependencies between symbols and cannot separate the linked lists nodes, since the possible values of variables (e.g., link pointers) are not known. Katana therefore abstracts from repeated (or cyclic) link references. This is achieved by reducing symbol expressions into a minimal form, by "folding" (i.e., removing) any duplicated field specifiers while preserving their relative order. Thereby, both `p->a->a` and `p->a->a->a` will be folded and treated like `p->a`. An expression on a cyclic linked data structure will preserve the first unique identifers, but fold any repeated sequences. This means that `p->a->b->a->b` is folded into `p->a->b`, assuming that `a` and `b` are link pointers, i.e., of the same datatype as `p`. This way, the number of possible symbol expressions for any given symbol is finite and limited by the defined datatypes.

A *symbol reference filter* determines if a particular symbol expression is of relevance for the current dataflow analysis (*Slice* operation). Only symbol references where the referenced symbol expression matches the current filter are considered relevant and thereby analyzed further. The main motivation behind the filter system is to separate between relevant and irrelevant data structure fields. The filter is a list of identifiers, in the same manner as symbol expressions. Filters are constructed in the same way as symbol expressions; symbol reference filters do not include function call arguments or array index expressions, and expressions containing repeated use of link pointers are folded into a minimal form. Thereby, the folding is applied symmetrically, to both the symbol reference filters and the symbol expressions, which makes the matching correct.

A particular symbol expression *SE* matches a symbol reference filter *F* if *SE* is identical to, or a prefix of *F*, i.e., if every field specifier in *SE* exists in *F* at the same location.

Thus, a filter *(foo, bar)* will accept the expressions `foo` and `foo->bar`, but not longer expressions like `foo->bar->val`. The prefix rule is motivated since a primary symbol of pointer type defines the context (address) of any following field symbols. For instance, for the filter *(foo, bar)* also assignments of `foo` are relevant since the value of `foo` is used in relevant assignments of `foo->bar`. Another reason why a data-structure pointer like `foo` is relevant by itself is that all aliases must be analyzed in order to find other

pointer assignments of relevant fields, in this case `bar`.

It is however possible to specify a less strict symbol reference filter by using the code `ANY`. The reason for this is the analysis of arguments to model focus functions. If a data structure pointer is passed as argument to a a model focus function, e.g., `ipc_send(msg)`, then all fields of the data structure are relevant for the model. (The possibility to relax this assumption is discussed in Section 5.7). For filters ending with `ANY`, symbol expressions containing the specified identifiers plus any additional field specifiers are also considered relevant. Thereby, given a filter *(foo, bar, ANY)*, then the expressions `foo` and `foo->bar` will match the filter, as well as other symbol expressions beginning with the primary symbol `foo`, followed by `bar` as the first field reference. Note that *ANY* is only allowed in the end, as the last field specifier of a filter.

The symbol reference filter for newly discovered model symbols are constructed by "projecting" the current filter with respect to the source and target symbol expressions of the dataflow dependency[2] at hand. The projection is a transformation where symbol specifiers are added and/or removed from the current filter, in order to produce a filter for the new primary symbol about to be analyzed.

The projection of symbol reference filters is performed when *Slice* calls itself recursively in order to explore a symbol which has been found to impact the model symbol under analysis. An interesting property is that the projection is symmetrical, it is performed in the same way independent if the analyzed dataflow dependency is "upstream" (from target to source) or "downstream" (from source to target). In both cases, the projection is a transformation of the filter string, which can be described as a function

$$NF = getNewFilter(CSE, CF, NSE)$$

which returns the new symbol reference filter, *NF*, given:

- *CSE:* The current symbol expression, i.e, a reference of the currently analyzed model symbol.
- *CF:* The current filter (symbol reference filter), i.e., which *CSE* matched.
- *NSE:* The new symbol expression to use when analyzing the new model symbol (i.e, through a recursive call of *Slice*).

[2]Examples of dataflow dependencies are assignment statements and function calls, i.e., the dependency between a function call argument and the corresponding formal parameter.

Note that "current" and "new" have different meanings depending on whether the dataflow dependency is analyzed "upstream" or "downstream". In the upstream case (from dataflow target to dataflow source), "current" refers to the dataflow target, while "new" refers to a dataflow source. In downstream analysis, i.e., when following forwarded pointers, the meaning of "current" and "new" is swapped. The new filter is calculated in the following way:

1. If there is a new field specifier in *NSE*, i.e., which does not exist in *CF*, then *NF* equals *NSE*. This is the case when a new dataflow analysis begins and the filter is empty.

2. If all field specifiers in *NSE* do exist in *CF* (i.e., not counting the primary symbol) or if *NSE* only contains a single identifier (the primary symbol), the new filter NF depends on *CF* and *CSE* in the following way:

   (a) Construct the first part of *NF* by adding each identifier in *NSE* to *NF*, unless a symbol specifier of the same datatype already exists in *NF*. The relative order of the identifiers must be preserved.

   (b) Extract *UF*, the unused (and rightmost) part of *CF*, from *CF* by removing all identifiers from *CF* which occur in *CSE*. This should remove at least the primary symbol, i.e., the first identifier of *CSE*. If *CSE* equals *CF*, i.e., a complete match, then *UF* will be empty, as there are no unused parts.

   (c) Append each identifier in *UF* (if any) to the end of *NF*, unless a symbol specifier of the same datatype already exists in *NF*. The relative order of the identifiers must be preserved.

The projection of symbol reference filters (getNewFilter) is illustrated using a code example, presented below. The analysis begins with the call of the model focus function "alarm". The relevant statements are thereafter identified according to the Katana algorithm, formally defined in Section 5.3.

The first symbol expression (step 1) is found in the if-statement condition. Since *s* is the first model symbol of a new data-flow analysis, there is no previous filter, so the new filter (NF) to use when analyzing the reference to the new symbol *s* will equal *NSE*, i.e., the symbol expression found in the condition.

In step 2, the only reference of *s* is analyzed and found relevant, since it matches the filter from step 1, i.e., "*s, sn, pos*". The source expression in the assignment, *s1*, becomes a model symbol with the filter "*s1, pos*", as *pos* was not used by *CSE*, i.e., "*s1, sn*".

The model symbol *s1* leads to the inclusion of three model statements, in steps 3, 4, and 5, but only in the statement of step 5 a new model symbol is found: the formal parameter *fp* in the function *update*.

```
typedef struct{
    short pos;
    short * adr;
 } SENSOR;

typedef struct{
    SENSOR * sn;
} SYSTEM;

SENSOR s1 = {0, 0x0210};

void update(SENSOR* fp)
{
 fp->pos = bar(fp->adr);
}

void foo()
{
 SYSTEM s;
 s1.pos = 0;
 s.sn = & s1;
 update(&s1);
 if (s.sn->pos == 0)
   alarm();
}
```

Step	CSE	CF	NSE	NF
3	*s1*	*s1, pos*	n/a	n/a
6	*fp, pos*	*fp, pos*	*bar*	*bar*
4	*s1, pos*	*s1, pos*	n/a	n/a
2	*s, sn*	*s, sn, pos*	*s1*	*s1, pos*
5	*s1*	*s1, pos*	*fp*	*fp, pos*
1	n/a	n/a	*s, sn, pos*	*s, sn, pos*
MFF call				

The statement at step 5 is relevant since a pointer to a model symbol (*s1*) is passed as argument (Rule G). The formal parameter receives the filter "*fp, pos*", which means that the assignment at step 6 matches the filter and thereby is a model statement. At this point we end this example (which otherwise would have continued with the symbols used in the return statement(s) in the `bar()` function).

Unions is a potential problem for this approach, i.e., overlayed data structures, where the same memory address may be accessible using multiple symbols (i.e., union members). Since Katana is a symbolic method, i.e., unaware of the actual memory layout, unions are treated like normal data structures. This is a valid simplification in most cases, for well structured code, but it is possible to construct examples where Katana would miss relevant statements due to unstructured use of (conflicting) union expressions. This is however bad programming style and is believed to be unlikely in industrial software systems. Solving the union issue requires that each data structure member is translated to a memory address offset, which requires a memory model specifying the size of primitive datatypes as well as padding/alignment policies of the compiler. This is not addressed in this thesis, but could be a future extension.

5.1.7 Pointers, Arrays and Function Pointers

The pointer analysis is a major difference between Katana and previous approaches for program slicing. The traditional approach is to resolve pointers before performing program slicing, i.e., a points-to analysis which finds the variables which each pointer may refer to. In Katana, pointers are handled much like any other variable in the sense that no separate pre-analysis is performed, they are instead analyzed on demand. During the evaluation of Katana, each model symbol is searched for assignments of any type (direct or pointer dereference), as well as locations where the address of the model symbol is obtained using the address-of operator (i.e., `ptr = &foo;`). Thereby, the definition of relevant pointers are captured, as well as all dereference assignments through these pointers (i.e., `*ptr = bar;`). As defined by Rule M, such pointers become model symbols they are used in a relevant dereference assignment, or if they are part in a dataflow leading to such an assignment.

In studies of industrial code, it has been observed that many functions takes as parameter a pointer to a large data structure, but only modify a small subset of the fields. The additional condition in Rule M, i.e., that the forwarded pointer must actually be used in a relevant dereference assignment, is an optimization in order to reduce the number of falsely included, irrelevant statements. Otherwise, if considering every case of pointer forwarding from model symbols as relevant, many unnecessary function calls would therefore be included, where the called functions does not contain any model statements. This might not sound like a big problem, since the model versions of these functions would be empty. However, if the calls are guarded by conditional statements, all symbols found in those conditions would falsely be considered relevant and thereby cause other irrelevant statements to be included.

For each case of pointer forwarding detected, Rule M implies that Katana inspects the local analysis result, i.e., the local recursive branch, originating in the forwarded pointer and current filter. If no model statements have been found (i.e., due to at least one relevant dereference assignment), the pointer forwarding statement is deemed irrelevant.

The Katana approach to handling pointers is not waterproof, since it is symbolic and unaware of the run-time memory layout (as discussed earlier, regarding unions). It is therefore possible to construct examples where Katana will fail to identify relevant pointer dereference assignments. Katana requires that there exist some statement which “connects” the model symbol and the pointer, i.e., by assigning the address of the symbol to the pointer. If a pointer is defined without using the model symbol it will not be detected. For instance,

consider the following case:

```
1: int a = 0, b = 0;
2: int *ptr = & a;
3: ptr++;
4: *ptr = 1;
```

If `b` is a model symbol, the assignment at line 4 is actually relevant since the pointer `ptr`, which originally pointed at `a`, is incremented at line 3 and after that points at the following memory address, where `b` resides. This cannot be detected by Katana since there is no obvious connection between the symbols `a` and `b`; the dependency is only due to the adjacent memory addresses. This is however not a unique limitation of Katana, but a problem for all tools doing symbolic pointer analysis. For instance, the commercial tools CodeSurfer [123] and Imagix 4D [135] cannot detect such dependencies either. These tools are compared against Katana in Chapter 6.

A pragmatic solution to the above exemplified problem, i.e., when pointers are modified using arithmetics, is to simply detect such statements and report them as warnings. Since pointer arithmetics is hardly good programming style and hopefully not very common, it is reasonable to suggest that such code, if found, is refactored in order to make it analyzable and more generally maintainable.

Arrays are treated differently than data structures, even though both are used to structure data. The difference is that array expressions typically involve index variables. Since Katana is not aware of the possible values of symbols, it is not possible to separate between different array elements when a variable is used in an index expression. Katana therefore treats all references of an array as equal references to a single variable. There is however one exception: two array expressions can be separated if the index expressions in both cases are single constant only. In such cases, the names of the index constants are treated like data-structure field names. This is most likely an unusual way to use arrays, but was observed in industrial code during the evaluation and this exception was therefore implemented in the MXTC tool.

Function pointers can be resolved in two ways. In order to find any function pointer calls of already known model functions, e.g., GMR functions, the solution is straight-forward and similar to how normal pointers are handled in Katana. Whenever a model function is detected, for any reason, all statements are identified where the address of the model function is assigned to a function pointer. The forwarding of the function pointer is then explored (i.e., all aliases of the pointer) with the purpose of finding function pointer calls, which

are added to the callers list of the model function. The second case is when a function pointer call is found in a potentially model-relevant statement, i.e., an LMR function pointer call. In this case, the called function pointer might not be previously known, and must in that case be backtracked in order to find all possible functions to which the function pointer may refer. This is a bit similar to general pointer analysis, but unlike traditional pointer analysis, this is not a whole-program analysis which resolves every pointer in the program. This is a limited analysis of specific function pointers which have been encountered in relevant statements.

Note that the proposed solution for handling function pointers is however not yet implemented in the MXTC prototype. In the MXTC evaluation presented in Chapter 6, the source code was modified in order to replace all function pointer calls with equivalent static function calls.

5.1.8 Library Routines

Library routines, for which the source code is not available, can be ignored by Katana if they are reentrant, i.e., without side-effects. Examples include most math library functions, e.g., `abs()`, `sqrt()` and `cos()`. When calls to reentrant library functions are found in a model-relevant statement, Katana will skip the function body and instead directly explore the arguments of the function. This assumes that identical library functions are available in the target simulation environment. Calls to non-reentrant library functions like “memcpy” and “sprintf”, which write to memory specified via an input pointer, are relevant when a pointer to a model symbol is used as argument. In such cases, all call arguments become model symbols and are analyzed accordingly. Functions which have other “hidden” side-effects (e.g., “malloc”) should be specified as model focus functions if their side-effects are relevant for the model. The Katana input should include lists of reentrant and non-reentrant API functions. If a function call is found to an undefined function which is not in this list, it is reported as an error.

5.2 Algorithm Efficiency

Since a particular symbol may be used at many locations in the source code, it is crucial for efficiency to avoid repeating already performed analysis jobs, i.e., *Slice* operations with specific parameters. This is achieved using an *analysis cache*, a hash table where the parameters of each started analysis job are stored

at the very beginning of the analysis job, before any recursive calls are made.

Before a new analysis job is started, the analysis cache is checked in order to determine if the job has been performed already, or at least been started in another branch of the recursion. In that case, the specific analysis job is redundant and will therefore not be started.

The lookup in the analysis cache naturally require some computational time, both for constructing the key based on the parameters and for searching for the key in the analysis cache hash table. But the important thing is that the recursion is stopped, so that repeated redundant analyses are avoided.

Since the analysis cache is implemented using a hash table, in the theoretical worst case, a lookup may correspond to a linear search, where the match is not found until all entries have been checked. This is however extremely pessimistic; in the average case a hash lookup is quite fast, much faster than repeating the analysis of an expression every time it occurs.

Since each symbol reference is analyzed at most once and since these recursive analysis jobs represent the core operation of the Katana algorithm, the runtime of Katana is typically linear to the output (slice) size. The memory usage of Katana mainly depends on:

- the size of the symbol database, which typically is dominated by, and directly proportional to the program size, specifically the number of lexemes, symbols and symbol references,
- the recursion stack size, bounded by the number of symbols, and
- the analysis cache size, also bounded by the number of symbols.

Thereby, the memory usage is typically linear to the program size, since only identifier lexemes have symbol references, and usually only one each. The theoretical worst-case complexity is however harder to reason about. The number of analysis jobs is in worst case equal to the number of unique symbol expressions in the code, which is a measure of program size, but we do not know how the runtime of an individual analysis job depends on the program size, since the lookup operations on the symbol database are only defined on a high level, leaving much to the implementation. The algorithmic complexity these, on which the algorithm relies on, is therefore unknown. In the prototype implementation of Katana, this depends on the API of the Understand tool for which the source code is not available for study.

5.3 The Katana Algorithm

Formally, the Katana algorithm can be described as a function, *Katana*, which takes as input a set of model focus functions, and returns a set containing the relevant statements (i.e., the program slice). This main function depends on a set of functions with recursive dependencies, as depicted by Figure 5.6.

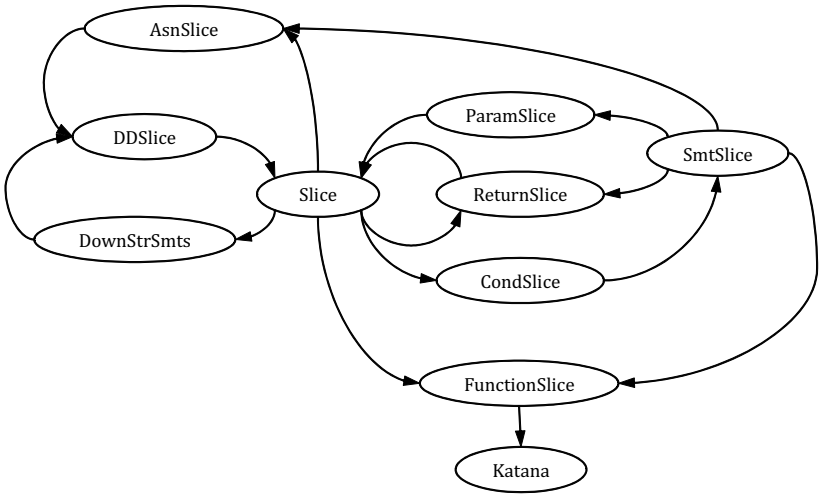

Figure 5.6: The Katana algorithm illustrated

Figure 5.6 illustrates the relations between the Katana functions. In this graph, nodes correspond to the functions presented later in this section, and edges to call-by relations, i.e., the propagation and accumulation of analysis results. The edge from *FunctionSlice* to *Katana* means that *FunctionSlice* return results to *Katana*, and thereby implies that *Katana* calls *FunctionSlice*. Note that this description does not include all aspects of Katana. In order to simplify the conceptual understanding several details have been omitted, for instance the analysis cache, the handling of symbol reference filters and details regarding detection and considerations of LMR and GMR functions. These aspects have however been extensively described previously in this chapter. This section mainly focuses on the recursive exploration of the program dependencies, through the symbol database representation.

In the following function definitions, the following datatypes are used: *Symbol*, *Statement* and *Boolean*. Braces are used to indicate sets of items with a specific datatype (i.e., $\{Datatype\}$). The following notation is used to specify the datatypes of function parameters and result:

$$Name : Datatype_1\ \ Param_1,\ \ \ldots,\ \ Datatype_N\ \ Param_N \rightarrow Datatype$$

First presented is the functions *OnEach* and *OnEach2*, commonly used in the later algorithm description. These are not visible in the illustration (Figure 5.6) in order to make the algorithm illustration more readable. However, most edges in the illustration correspond to an OnEach operation. In the below definitions, *Item* represents any datatype. F_1 and F_2 represent any function which return a set of statements and which takes one or two parameters, respectively.

$OnEach:\ \ Function\ \ F_1,\ \ \{Item\}\ \ S \rightarrow \{Item\}$
OnEach applies a function F to every element in the set *S*, and returns the union of the function results.

$$OnEach(F, S) = \bigcup_{x \in S} F(x)$$

$OnEach2:\ \ Function\ \ F,\ \ \{Item\}\ \ S,\ \ Item\ \ a \rightarrow \{Item\}$
OnEach2 allows for functions with two parameters (F), where the second, *a*, is common for all items in *S*.

$$OnEach2(F, S, a) = \bigcup_{x \in S} F(x, a)$$

Several other supporting functions are used in the below function definitions. These are presented informally in Section 5.4 and correspond to low level analyses using the symbol database functionality, typically lookups of references and various analyses performed on lexeme-level.

Katana

Katana : $\{Symbol\}\ \ MFFs \rightarrow \{Statement\}$

The main function of the Katana algorithm, which returns all model statements of relevance, with respect to a set of model focus functions represented as Symbols.

$$Katana(MFFs) = OnEach(FunctionSlice, MFFs)$$

FunctionSlice

FunctionSlice : $Symbol\ \ Sym \rightarrow \{Statement\}$

Returns all statements of relevance for the execution of a model focus function, represented as a Symbol. This include all statements involved in deciding when the function is called and all statements which impact the values of the arguments.

$$\begin{aligned} &FunctionSlice(Sym) = \\ &\qquad OnEach(Slice, AllCallArgs(Sym)) \ \cup \\ &\qquad\qquad OnEach(SmtSlice, AllCallers(Sym)) \end{aligned}$$

Slice

Slice : $Symbol\ \ Sym \rightarrow \{Statement\}$

Returns all statements of relevance for the specified symbol, i.e., the backwards slice. Since the slicing is not control-flow sensitive, the result includes all statements which may impact the symbol at any point in the program.

$$Slice(Sym) = \begin{cases} DDSlice(Sym) & \text{if } \neg IsFunc(Sym) \wedge \\ & \neg IsParam(Sym) \\ DDSlice(Sym) \ \cup OnEach2(ParamSlice, & \\ CallerSmts(DefFunc(Sym)), Sym) & \text{if } IsParam(Sym) \\ DDSlice(Sym) \ \cup & \\ OnEach(ReturnSlice, ReturnSmts(Sym)) & \text{if } IsFunc(Sym) \end{cases}$$

DDSlice

DDSlice : *Symbol Sym* $\rightarrow \{$*Statement*$\}$

Returns all statements of relevance for direct or downstream assignments of the specified Symbol. A downstream assignment is an assignment through a pointer dereference, where the specific pointer has been assigned the address of the specific symbol, either directly using the "address-of" operator, or indirectly, from another pointer variable.

$$\begin{aligned} &\mathit{DDSlice}(\mathit{Sym}) = \\ &\qquad \mathit{OnEach}(\mathit{AsnSlice}, \mathit{AsnSmts}(\mathit{Sym})) \ \cup \\ &\qquad\qquad \mathit{OnEach}(\mathit{DownStrSmts}, \mathit{PtrUseSmts}(\mathit{Sym})) \end{aligned}$$

ParamSlice

ParamSlice : *Statement Smt*, *Symbol Sym* $\rightarrow \{$*Statement*$\}$

Returns all statements of relevance for the execution of the specified function call (Smt) as well as statements of relevance for the function argument matching the specified formal parameter, Sym.

$$\mathit{ParamSlice}(\mathit{Smt}, \mathit{Sym}) = \mathit{SmtSlice}(\mathit{Smt}) \cup \mathit{Slice}(\mathit{ArgOfParam}(\mathit{Smt}, \mathit{Sym}))$$

SmtSlice

SmtSlice : *Statement Smt* $\rightarrow \{$*Statement*$\}$

Returns a set of statements containing the input statement (Smt) and all other statements of relevance for the execution of Smt, i.e., all guarding conditions and all statements of relevance for these conditions.

$$\mathit{SmtSlice}(\mathit{Smt}) = \{\mathit{Smt}\} \cup \mathit{OnEach}(\mathit{CondSlice}, \mathit{CondSmts}(\mathit{Smt}))$$

CondSlice

CondSlice : *Statement Smt* $\rightarrow \{$*Statement*$\}$

Given a statement, CondSlice gives a set of statements, which for a condition statement includes all statements of relevance for the condition. For other statements (without conditions), the set only contains the input statement.

$$\textit{CondSlice}(\textit{Smt}) = \begin{cases} \{\textit{Smt}\} \cup \textit{OnEach}(\textit{Slice}, \textit{Symbols}(\textit{Smt})) & \text{if } \mid \textit{CondSmts}(\textit{Smt}) \mid > 0 \\ \{\textit{Smt}\} & \text{if } \mid \textit{CondSmts}(\textit{Smt}) \mid = 0 \end{cases}$$

AsnSlice

$AsnSlice$: $Statement\ Smt,\ Symbol\ Sym \rightarrow \{Statement\}$

Returns all statements of relevance for a specific assignment statement (Smt). They are:

1. All local control-flow statements (conditions) who directly impact when the control-flow reaches the specific statement. This corresponds to the use of SmtSlice on Smt.
2. All statements of relevance for the source parts of the assignment, excluding arguments to function calls. Call arguments are analyzed later, when formal parameters of the called function have been found relevant with respect to the function return value.
3. If the Symbol is a global variable, all calls of the scope function and all statements controlling these calls are also included, since they are GMR functions. Otherwise, relevant function calls might be excluded, when they do not have parameters.

$$\textit{AsnSlice}(\textit{Smt}, \textit{Sym}) = \begin{cases} \textit{SmtSlice}(\textit{Smt}) \cup \\ \textit{OnEach}(\textit{Slice}, \textit{Symbols}(\textit{Smt})) \cup \\ \textit{OnEach}(\textit{SmtSlice}, \textit{AllCallers}(\textit{Smt})), & \text{if } \textit{IsGlobal}(\textit{Sym}) \\ \textit{SmtSlice}(\textit{Smt}) \cup \\ \textit{OnEach}(\textit{Slice}, \textit{Symbols}(\textit{Smt})), & \text{if } \neg\textit{IsGlobal}(\textit{Sym}) \end{cases}$$

DownStrSmts

DownStrSmts : *Statement Smt*, *Symbol Sym* $\rightarrow \{$*Statement*$\}$

Returns all statements of relevance due to dereference assignments of pointers "forwarded" from the symbol Sym, i.e., when the address of Sym (or an address stored in Sym) is used in an assignment, function call or function return. The result is an empty set if the "forwarded" pointer is not used in any relevant dereference assignments.

1. If Smt is an assignment, where the address of Sym is obtained and assigned to a pointer variable, then Sym is analyzed recursively.

2. If Smt contains a function call where Sym is part of a call argument, where the corresponding formal parameter is of pointer type, then Sym is analyzed recursively.

3. If Smt is a return statement where Sym is returned and the function is of pointer datatype, then all calls of the function, i.e., all uses of the function return value, are analyzed recursively.

$$
\begin{aligned}
&\mathit{DownStrSmts}(\mathit{Smt}, \mathit{Sym}) = \\
&\begin{cases}
\mathit{OnEach}(\mathit{Slice}, \mathit{AsnTargets}(\mathit{Smt}, \mathit{Sym})), & \text{if } \neg \mathit{IsCallArg}(\mathit{Smt}, \mathit{Sym}) \vee \\
& \quad \neg \mathit{IsReturned}(\mathit{Smt}, \mathit{Sym}) \\
\mathit{OnEach}(\mathit{Slice}, \mathit{AsnTargets}(\mathit{Smt}, \mathit{Sym})) \cup & \\
\mathit{Slice}(\mathit{ParamOfArg}(\mathit{Smt}, \mathit{Sym})), & \text{if } \mathit{IsCallArg}(\mathit{Smt}, \mathit{Sym}) \\
\mathit{OnEach}(\mathit{Slice}, \mathit{AsnTargets}(\mathit{Smt}, \mathit{Sym})) \cup & \\
\mathit{Slice}(\mathit{ContextFunc}(\mathit{Smt})), & \text{if } \mathit{IsReturned}(\mathit{Smt}, \mathit{Sym})
\end{cases}
\end{aligned}
$$

ReturnSlice

ReturnSlice : *Statement Smt* $\rightarrow \{$*Statement*$\}$

Returns all statements of relevance for the specified return statement.

$$\mathit{ReturnSlice}(\mathit{Smt}) \quad = \quad \mathit{SmtSlice}(\mathit{Smt}) \ \cup \ \mathit{OnEach}(\mathit{Slice}, \mathit{Symbols}(\mathit{Smt}))$$

AllCallers

AllCallers : *Symbol* $F \rightarrow \{$*Statement*$\}$

Returns the function call statements corresponding to the callers graph of the specified function, i.e., all direct and indirect "incoming" function calls.

$$\textit{AllCallers}(F) \;=\; \textit{CallerSmts}(F) \cup \textit{OnEach}(\textit{AllCallers}, \textit{CallerSmts}(F))$$

5.4 Supporting Functions

The following supporting functions are assumed by the definition of the Katana algorithm in Section 5.3. They correspond to low level analyses using the symbol database functionality, typically lookups of references and analyses on lexeme level.

- *AsnSmts* : $Symbol \rightarrow \{Statement\}$

  All statements where the specified model symbol is assigned.

- *IsGlobal* : $Symbol \rightarrow Boolean$

  True if the symbol is a global variable, else false.

- *Symbols* : $Statement \rightarrow \{Symbol\}$

  All symbols in the specified statement, excluding function call arguments.

- *PtrUseSmts* : $Symbol \rightarrow \{Statement\}$

  All statements where a pointer to the specified symbol is created.

- *AsnTargets* : $(Statement, Symbol) \rightarrow \{Symbol\}$

  All symbols assigned from the specified symbol in the specified statement. In most cases there is only one target symbol, but there may be multiple symbols for statements like *a = b = c;*

- *ParamOfArg* : $(Statement, Symbol) \rightarrow Symbol$

  Gives the formal parameter symbol corresponding to the specified function call argument symbol in the specified function call statement.

- *ArgOfParam* : $(Statement, Symbol) \rightarrow Symbol$

  Gives the call argument in a specified call statement corresponding to the specified formal parameter symbol.

- *AllCallArgs* : $Symbol \rightarrow \{Symbol\}$

  Gives all arguments (symbols) used in any call of a particular function, specified as a Symbol.

- *ContextFunc* : $Statement \rightarrow Symbol$

  The function where the specified statement is located.

- *DefFunc* : $Symbol \rightarrow Symbol$

  The function in which the provided Symbol is declared.

- *IsCallArg* : $(Statement, Symbol) \rightarrow Boolean$

  True if the specified Symbol is an argument in a function call in the specified statement, else false.

- *IsReturned* : $(Statement, Symbol) \rightarrow Boolean$

  True if the specified Symbol is referenced in the specified return statement, else false.

- *IsParam* : $Symbol \rightarrow Boolean$

  True if the specified Symbol is a formal parameter of a function, else false.

- *IsFunc* : $Symbol \rightarrow Boolean$

  True if the specified Symbol is a function call, else false.

- *CondSmts* : $Statement \rightarrow \{Statement\}$

  Gives the statements of direct relevance for the *execution* of the specified statement, in the local function. This includes:

  1. all conditions of enclosing selections and loops,
  2. all break and continue statements of enclosing loops, and
  3. all return statements.

- *CallerSmts* : $Symbol \rightarrow \{Statement\}$

  Returns a list of statements corresponding to all direct calls of a specified function (the Symbol).

- *ReturnSmts* : $Symbol \rightarrow \{Statement\}$

  Returns a list of statements corresponding to all return statements in a specified function (the Symbol).

5.5 Katana Compared to Related Work

Sandberg et al. [27] propose a program slicing method called SimpleSlice which like Katana does not depend on a pre-calculated dependency graph such as an SDG. SimpleSlice starts from a set of variables and performs a fix-point iteration over all assignments in the code. Any statement which possibly may affect the specified variables are added to the output set. The purpose of SimpleSlice is to allow for faster WCET analysis by computing a slice for all variables used in control-flow conditions of the program. Any statements not in that slice cannot impact the program control-flow and can thereby be excluded from the flow analysis step of static WCET analysis, in order to reduce the computation time.

Like most existing slicing methods which support pointers, SimpleSlice assumes that a points-to analysis has been performed and produced a points-to set for each pointer, containing the variables possibly pointed to by the specific pointer. This is a major difference compared to Katana. SimpleSlice and SDG-based methods resolve pointers beforehand, in a separate "points-to" analysis, e.g., using the Steensgard approach [73]. In contrast, Katana does not keep track of what symbols a particular pointer may refer to. Katana simply includes all statements which forward the address of already identified model symbols and where the pointer forwarding may reach a relevant pointer dereference modification (e.g., assignments like `*ptr = x;` or some library function calls like `memset(ptr, ...);`).

Enabling support for function pointers however requires a simple form of points-to analysis, as discussed in Section 5.1.7. The function pointer extension of Katana is however not applied on the whole source code like the Steensgard method, but only on relevant statements and pointer variables, which forwards addresses of newly identified model functions. The analysis is similar to the analysis of normal (data) pointers, but instead of searching for dereference assignments, the analysis searches for function calls through the function pointer, in order to add the call to the caller list of the model function.

Unlike Katana, SimpleSlice treats data structures as single variables. Thus, if SimpleSlice find a particular field of a data structure relevant, it will include any statement assigning any field of that data structure. Moreover, the SimpleSlice approach is only presented for intraprocedural slicing, i.e., within the scope of a single function, even though it is claimed that it can be extended for interprocedural slicing using standard methods.

Another example of a slicing tool which does not use the SDG approach is the research prototype *Sprite* [69, 70, 68]. In Sprite, the control-flow of each

function is only represented in an intraprocedural manner and a separate call-graph is used for representing interprocedural control flow. The data-flow analysis is performed using the control-flow graph (CFG) representation. Like SimpleSlice, the Sprite tool is flow-insensitive and uses the Steensgard algorithm [73] for pointer analysis. An interesting aspect of this tool is that it computes all program representations on demand, to the extent it is possible; the CFGs and points-to information is calculated on the first slice computation, while control-dependencies and data-flow information are calculated on demand during the slicing. This is also the approach of Katana, but Katana is even more "on demand" than Sprite, only the symbol database is pre-calculated. Katana does not use the CFG representation but instead performs its (limited) control-flow analysis on the lexeme level.

Compared to traditional SDG approaches, which typically are flow-sensitive, both SimpleSlice and Katana are *flow-insensitive*, meaning that they do not take control-flow (fully) into account. Katana, for instance, assumes that all assignments of a relevant symbol are relevant, even though some assignments may be "killed" by later assignments of the same symbol. In contrast, a *flow-sensitive* approach uses an exact model of control-flow and can thereby remove more statements due to the control flow constraints. Katana can be extended to allow for flow-sensitive slicing using the same conceptual approach, but it has not been investigated how that would impact the efficiency of the algorithm or the accuracy of the result. This is planned for future work.

The author is aware of two commercial tools with support for program slicing or dataflow analysis: Grammatech CodeSurfer [123] and Imagix 4D [135]. CodeSurfer explicitly supports program slicing, using a variant of the SDG approach proposed by Reps et al. [21]. The CodeSurfer tool is a spin-off from the Wisconsin program slicing group, headed by Horwitz and Reps, who originally proposed the SDG-based approach to program slicing. At their website [121] they state that CodeSurfer is limited to a maximum of 200.000 lines of code.

Imagix 4D is not claimed to do program slicing, but has a feature called "Calculation Tree" which identifies the statements which are involved in the dataflow which define a specific variable. This is very similar to program slicing. The difference is that Imagix 4D does not produce a full executable slice, since it does not perform analysis of relevant conditions for the identified statements. It is not known what method that is used in Imagix 4D, as it has not been published, from what the author could find. It seems to be a trade secret of the Imagix Corporation. It however seems likely that also this tool uses the SDG or similar approach. The SDG approach has dominated the research focus in

program slicing since the early 1990's and Imagix 4D seems to have similar scalability problems as CodeSurfer, which is known to use the SDG representation. Using the Calculation Tree analysis of Imagix 4D requires many hours of analysis for larger code-bases, as presented in Chapter 6. During this time, the tool probably generates some type of model of the whole program, since this step is very time consuming but only necessary once for each project, and once this is completed there is no significant delay when using the Calculation Tree feature.

A big difference between Katana and other slicing approaches is the use of a symbol database as program representation, instead of more advanced representations, such as the CFG, PDG or SDG. The Katana approach avoids computationally costly analyses of the whole program, such as constructing an SDG. In Katana, only statements which are known to be (or most likely) relevant are analyzed in detail, while the others are ignored. The only whole-program analysis is the initial generation of the symbol database, but this is very fast, as indicated by the performance evaluation presented in Chapter 6.

5.6 Limitations of Katana

This section presents the conceptual limitations of the Katana algorithm, which is not the same as the current status and implementation limitations of the MXTC prototype implementation, presented in Section 6.1. The main conceptual limitation of Katana is that it is unaware of the real memory addresses of symbols and functions, which makes it impossible to resolve pointers assigned direct explicit addresses, i.e., from literal expressions, or pointers which are modified using arithmetics. For the same reason, statements which modify relevant data trough accidental buffer overflow or "rouge pointers" are not detected. Typecasting between different datastructures can not be followed with maintained accuracy; all fields of the new data structure are considered as relevant in this case. Unions are treated like normal data structures (struct), and unstructured, mixed use of union overlays, where different identifiers refer to the same data, will not be handled correctly.

These limitations are however not unique for Katana but also present for tools using more detailed but less scalable analysis methods, for instance the commercial tools Imagix 4D [135] and CodeSurfer [123]. However, cases of pointer arithmetics, problematic typecasting and unstructured union usage can at least be detected by a tool implementing Katana. Such statements are often "shortcuts" which could have been implemented in a more structured (and an-

alyzable) manner. It is therefore fair to assume that the user could modify such statements in order to make the code analyzable and to improve its general maintainability.

Katana is not fully control-flow sensitive, but this is not a conceptual limitation of the algorithm. It does not limit the applicability and it is possible to extend Katana to support at least intraprocedural control-flow analysis without conceptual changes of the algorithm. This change only impacts the implementation of `AsnSmts` and `PtrUseSmts`, which are "supporting functions" of Katana, as presented in Section 5.4.

5.7 Possible Extensions of Katana

This section presents possible future improvements of the Katana algorithm on a conceptual level. Note that these extensions do not reflect the current implementation status of the MXTC prototype. The status of the MXTC tool and remaining issues can however be found in Chapter 6.

Flow-Sensitivity: The Katana approach (and the MXTC prototype) is in the described version flow-insensitive, meaning that all assignments of a relevant variable are considered relevant. This is an over-approximation, since not all assignments may reach the relevant statement where the specific symbol is used. During studies of industrial code, several cases have been observed where a single local variable is used for several purposes, e.g., to store return values from different, possibly completely independent functions. In such cases, a control-flow sensitive approach would give a more accurate result (smaller output models). Even though the control-flow analysis will require extra analysis in order to exclude some statements, this may actually reduce runtime when the reduction in output size is large, as the runtime of MXTC is mainly dependent on output size, not the total program size.

Task Dependency Analysis: In many embedded systems, tasks communicate using various methods for *interprocess communication*, or *IPC*, typically message queues. An IPC message is typically a data structure, containing several fields. Normally, a set of "standard" fields are always available, such sender ID and message code, i.e., how to interpret the rest of the message. Currently, Katana does not treat IPC functions differently than other model focus functions, so all arguments of IPC operations are considered relevant, including all parts of sent IPC messages. However, it is not certain that all fields are actually used by the receiving task, especially in the code resulting from the model extraction. An interesting optimization would be to, instead of as-

suming that all fields of outgoing IPC messages are relevant, identify the fields in outgoing IPC messages actually used by the receiving tasks in the resulting simulation model. This way, unused message fields can be excluded from the analysis, which most likely leads to reduced model size.

Manual Abstractions: An interesting extension is to provide support for manually specified *modeling abstractions*, which may reduce the output size significantly. An obvious type of abstraction would be to replace a condition with a constant, or a probabilistic expression, in order to exclude the condition variables from the model and thereby possibly all statements which impact those variables. However, in order to benefit from such abstractions, *all* references to a specific variable must be removed. It is important that the modeling abstractions are valid with respect to the purpose of the model, but typically one could exclude error-checking conditions if the model is for average case performance analysis. Based on seven years of research during which several industrial software systems have been studied with simulation model extraction in mind, it is the author's belief that a small amount of carefully selected modeling abstractions in many cases can reduce the model size significantly. The modeling abstractions can be stored as code annotations (comments) adjacent to the condition in focus. Thereby, the modeling abstractions remain in the code for automated reuse during future model extractions.

Supporting C++: Another possible extension is to support C++. Katana is currently limited to C. An extension to full C++ support, or other object oriented languages, is probably possible but most likely hard. C++ is often claimed to be a nightmare for reverse engineering tools due to language features like templates, virtual functions, operator overloading, multiple inheritance and exceptions. However, in many embedded systems, only small parts are written in C++ and these parts often only use basic features of the C++ language. A realistic extension of Katana in this direction would be to support a limited subset of C++, e.g., Embedded C++ [128]. This could be sufficient for many systems.

Chapter 6

A Model Extraction Tool and Evaluations

This chapter presents an implementation of the Katana algorithm presented in Chapter 5. The tool is named *MXTC* – **M**odel e**X**traction **T**ool for **C**. This chapter presents this implementation and three evaluations performed using industrial code:

- A performance evaluation of "Understand for C++", a commercial tool used for constructing the *symbol database*, i.e., the program representation used, as discussed in Chapter 5.
- An evaluation of MXTC for simulation model extraction, performed using two 3rd party software systems, where one is a subsystem in ABBs highly complex control system for industrial robots.
- A comparison between MXTC and two commercial program analysis tools, CodeSurfer and Imagix 4D, with respect to general program slicing, specifically the scalability and accuracy of the tools.

As introduced in Section 2.4.1, program slicing is a type of program analysis which identifies all program statements of relevance for a particular slicing criterion, typically a particular variable at a particular point in the program. Program slicing is a key analysis in this approach to automated simulation model extraction and is realized using a novel approach with better scalability compared to existing approaches.

6.1 MXTC – Model Extraction Tool for C

MXTC is a research prototype tool with at least two uses: simulation model extraction and general program slicing, e.g., for program comprehension purposes. The tool is designed for analysis of ANSI C programs only, C++ is not yet supported. The common GCC compiler implements several extensions of ANSI C, of which some may cause problems for MXTC in its current implementation, such as nested functions and loop statements inside expressions.

When used for model extraction, its original purpose, the tool takes as input a list of *model focus functions*, as presented in Chapter 5. Being a prototype, MXTC does not yet output executable model code; the current version outputs a log file, which lists the statements and symbols of relevance for the model, i.e., the *model statements* and *model symbols*[1]. The MXTC tool is however intended to produce a set of code files containing a simulation model for the RTSSim simulation framework, presented in Chapter 3. The simulation model is a filtered version of the original code, with additional *Execute* statements added to model the tasks' consumption of CPU time.

When used for program comprehension, the tool takes as input a single symbol reference, i.e., a symbol name and a program point where a specified symbol (e.g., a variable or function) is used. The program point is used to uniquely identify the symbol in the symbol database. The analysis is not control-flow sensitive, so any reference of a specific symbol gives the same result: all statements possibly affecting the value of the symbol, at any location.

The MXTC tool can produce graphical output, an image showing a dependency graph over the identified model statements and symbols variables. This was initially a feature intended to facilitate debugging of MXTC, but has been found to be interesting for general program comprehension. An example of the dependency graph output is provided in Figure 6.1. The graph image is generated automatically using the DOT tool [126], based on a text file (in the DOT input format) produced by the Katana analysis.

In the generated visualization, there are two main node types. Statement nodes (blue) representing model statements and symbol nodes (red) representing model symbols, except for function return values. Note that this illustration does not show the statement execution order, only data dependencies and control-flow dependencies with respect to conditions. Diamond-shaped statement nodes (i.e, blue) represent conditional statements, i.e., loops and selections. Hexagon-shaped symbol nodes (i.e., red) represents formal parameters

[1] As introduced in Chapter 5, a model symbol refers to a variable, parameter, constant or function discovered in a model statement, i.e., a statement found to be relevant for the model.

of functions, while ellipse-shaped symbol nodes represent function return values. There are two main types of edges. Blue edges represent control-flow dependencies between conditional statements and enclosed (guarded) statements, while red edges correspond to data-flow dependencies. A special case is dotted red edges, which represent dependencies between symbols and conditional statements where they are used. The # in the statement nodes indicates the line number where the statement starts. There are many opportunities for improving the graphical output of MXTC since the visualization currently produced is a result of only a minor development effort.

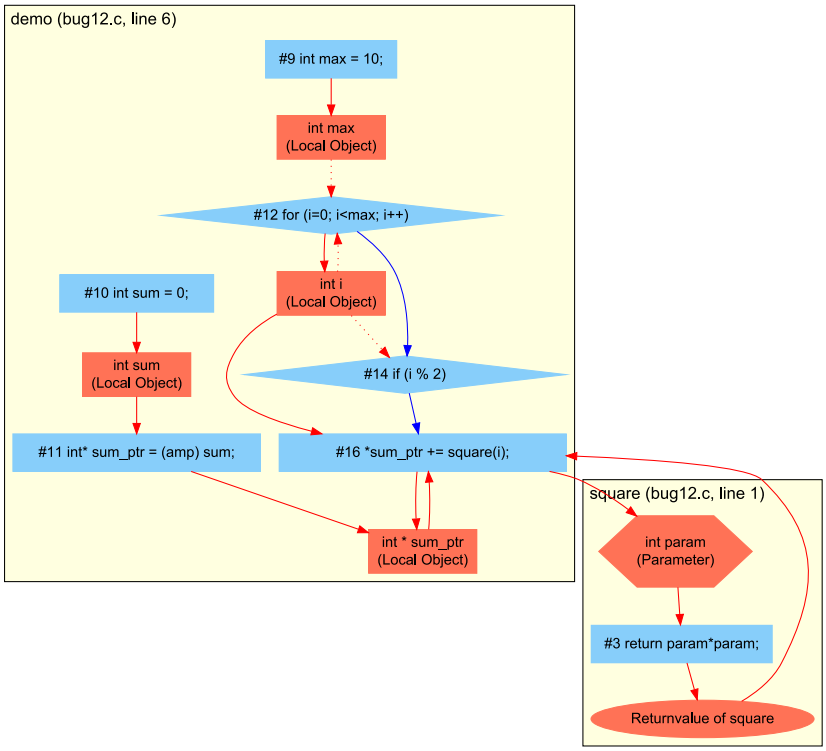

Figure 6.1: An example of the graphical output of MXTC

MXTC is a Perl implementation relying on the Perl API of the tool "Understand for C++" [138]. The Understand tool is a commercial reverse engineering tool which parses source code and constructs a database, what the author calls a *symbol database*, an index over the source code describing all *symbols* in the code, such as functions, variables and parameters, and where they are declared, assigned and used. The symbol database also contains the source code in tokenized form (lexemes). The symbol database constitutes the program model on which the Katana algorithm is based, as described in Chapter 5.

The Understand tool was selected as a base for MXTC since it has shown to be very fast in processing large amounts of source code and since it is provides both C and Perl APIs with good documentation and many examples. The choice of Perl was mainly since the Perl API has better documentation; the author was initially not aware of the C API. Perl is however not a bad choice, it a widely used and mature script programming language, for which there are many code examples and free libraries available. Perl has native support for regular expressions and hash tables, which has been valuable when implementing MXTC. While being a scripting language, Perl is claimed to be quite fast since the interpreter uses Just-In-Time (JIT) compilation. However, in a benchmark [143] comparing C++, Perl, Python and PHP, C++ is about 8 times faster than Perl. In another set of benchmarks [144], the C++ implementations were 64 – 120 faster than the corresponding Perl implementations. These benchmarks were however trivial programs containing a single operation, executed in a loop. However, it seems likely that porting MXTC from Perl to C/C++ would reduce runtimes significantly.

The development of the Katana algorithm and the MXTC tool has required a substantial effort, even though the author did this alone. The development started in January 2008 and lasted until October 2009, when the evaluation and documentation phase started. Well over one person-year was put into this development during this time — quite a lot for an academic research prototype. MXTC consists of around 9 000 lines of Perl code, not counting the Understand API. The relatively long development time was partially since the overall algorithm was fundamentally changed two times during the process. This since it was hard to identify all requirements of the solution and take them into account during the high level algorithm design. Problems were instead realized later during implementation which caused major re-designs. It took three attempts before a suitable algorithm design was found. Another problem was that the Katana algorithm turned out to be quite hard to implement and verify,

since most analyzes are made on demand, in a single[2] pass analysis. This is however also believed to be the key to scalability.

As mentioned, MXTC does not yet output simulation models; it only lists the relevant statements. In order to reach a state where the tool can output compilable RTSSim simulation models, the following issues remain:

1. A solution for handling references of irrelevant symbols in otherwise relevant model statements, as discussed in Section 5.1.3. This is the main issue preventing output of compilable simulation models.

2. Identifying and reporting all occurrences of pointer arithmetics, i.e., modifications of pointers using arithmetics, as discussed in Section 5.1.7. MXTC/Katana is not aware of the memory layout, so any such pointer manipulations of relevance would be missed, which is a model validity threat.

3. Resolving function pointers, using the method proposed in Section 5.1.7. This is important for the model validity, since function calls through function pointers otherwise would be missed.

4. Allowing for timing accurate simulation. From the perspective of MXTC, this requires:

   (a) Identification and enumeration of *time synchronization points* (TSPs), i.e., points in the simulation model where the simulation clock should be updated with respect to run-time measurements. These corresponds to calls of model focus functions and all kinds of task inputs and outputs, e.g, IPC and use of global variables shared between tasks.

   (b) Adding *Execute* calls in the model code at the TSP locations, in order to models the consumption of CPU time as presented Chapter 3. In the envisioned solution, this data is sampled from a *timing profile*, constructed from recordings as presented in Section 7.5.

The MXTC tool is currently limited to preprocessed code, which requires that the code is preprocessed before constructing the symbol database using the Understand tool. In this activity, which is the first step performed by C compilers, all preprocessor directives are resolved, such as macros, and each source file is merged with the included header files. The limitation to preprocessed

[2] If not counting preprocessing and symbol database construction.

code is necessary since complex macros, e.g., containing several statements, otherwise may cause serious problems for MXTC since the analysis is performed on source-code level. Most compilers can output preprocessed code, including GCC and Microsoft Visual C++, and typically quite quickly. This is therefore not a problem if all header files are available.

6.2 An Evaluation of "Understand for C++"

The Understand tool was selected as a base for the MXTC tool since it is claimed (and was also found to be) capable of processing large amounts of source code in little time. In order to investigate the scalability of the symbol database approach used by MXTC/Katana, a performance evaluation has been performed of Understand, focusing on the time to construct the symbol database from source code. The specific version of the Understand tool was version 1.4, build 449. The results are presented in Table 6.1.

Table 6.1: Measured parsing times of "Understand for C++"

Name	LOC	ExSmt	Functions	Files	Runtime (s)
ABB-1	1 083 604	448 963	9 728	1 699	118
ABB-2	183 492	81 203	3 116	416	16
ABB-3	136 537	83 118	3 125	89	14
ECU	41 320	18 515	1 169	324	5
RTSSim	3 802	1 572	152	58	2

The computer used for this experiment was a Dell Latitude E6400 laptop from 2009, equipped with an Intel P8400 CPU (2.26 GHz), 4 GB of RAM, a Western Digital WD1600 hard drive and used Microsoft Windows XP SP3. The first three cases correspond to different subsystems of the IRC 5 control system for industrial robots, developed by ABB Robotics [118]. The ECU case is the whole source code for a vehicular control unit, provided by an anonymous company. The RTSSim case is the simulator framework presented in Chapter 3. The *LOC* metric, Lines Of Code, is the number of source code lines containing actual code, i.e., excluding comments and empty lines. The metric *ExSmt* is the number of executable statements, which also excludes type- and variable declarations.

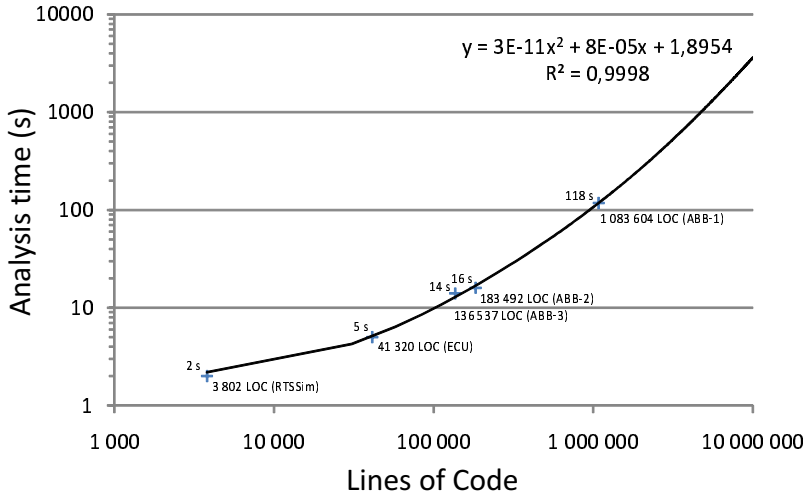

Figure 6.2: Parsing times of "Understand for C++", observed and extrapolated

Figure 6.2 shows a plot of the data from Table 6.1 with a regression curve generated using Microsoft Excel, with the equation $3 * 10^{-11} * x^2 + 8 * 10^{-5} * x + 1.8954$. A second degree polynomial function was found to produced the best fit. The R^2 value is an indication of how well the curve fits the data; the scale is from 0 – 1 where 1 corresponds to a perfect fit. The reported R^2 of 0.9998 indicates that the curve fits the data very well. However, only five data points were used. Since three data points is the minimum number for defining a second degree polynomial, there are only two additional "control points", which is a bit low. A few more would strengthen the validity of this analysis.

Assuming that this equation is correct, the runtime is essentially linear for small- and medium-sized systems (note that the diagram scale is not linear but logarithmic). There is a quadratic term but its coefficient is very small, which makes its impact negligible for smaller amounts of code. For instance, at 100 000 lines of code (100 KLOC) the x^2 term is only accountable for about 3 % of the predicted runtime, 10.2 seconds. The x^2 contribution does not reach 50 % of the total until at around 2.67 MLOC.

The predicted runtime at 10 MLOC is 3 802 seconds, i.e., about 63 minutes. This is probably most acceptable for large industrial systems, since this is typically much less than the build time for a system of this size (often many hours)

and it is only necessary once per system version. Moreover, the Understand tool allows for incremental update. This means that once a complete symbol database has been constructed, the following updates only have to analyze the modified source files. This reduces the analysis time considerably.

The author believes that performance of the Understand tool comes from its light-weight program representation; it does not seem to generate abstract syntax trees (ASTs) or control-flow graphs (CFGs) during the analysis, but instead operates directly on the lexeme level, i.e., tokenized source code. This is not known for sure, but seems likely since the performance is significantly higher compared to tools using heavy-weight program models (cf. Section 6.4), and since the Understand API provides a lexeme library, but no corresponding AST or CFG functionality.

6.3 An Evaluation of Model Extraction

This section presents an evaluation of the proposed approach for automated simulation model extraction, Katana, and of the MXTC tool implementing this algorithm, on two 3rd party, multi-threaded software systems: a proprietary industrial system (ABB IRC 5) and an open-source web server (Mongoose). The large ABB system has however not been studied in whole, only a quite small subsystem. The evaluation focuses on the model extraction runtime and the size of the resulting model. The amount of code is relatively small compared to the typical systems targeted by this work, but cases are real applications of non-trivial size and complexity.

The results of this evaluation indicate that the MXTC tool, and the Katana approach in general, scales to much larger programs than the ones used in this study, since the runtimes are short and, more importantly, mainly depend on the number of statements of relevance for the simulation model, i.e., the number of model statements, rather than the total size of the tasks. The only explicit dependency between runtime and total program size is for the very first step, the generation of the symbol database using the Understand tool. This is however a lightweight analysis which has been shown to be very fast also for millions of lines of code, as presented in Section 6.2.

6.3.1 Case 1 - SAF - A Subsystem of ABB IRC 5

IRC 5 is an advanced control system for industrial robots, developed by ABB [118]. It is a very large and complex software system, containing about 3 million

lines of code, mainly in C, and 50 – 100 tasks depending on configuration. It is structured in about 10 subsystems, roughly corresponding to subject areas, e.g., motion control, fieldbus communication, welding techniques, user interface, etc.

The MXTC tool has been applied on the safety subsystem, *SAF*, which is responsible for monitoring a large set of signals related to safety and system integrity. Some signals come from physical safety devices, such as light/laser sensors which detect if a human (or other obstacle) enters the working area. In such cases, the robot must stop immediately. Other signals are related to the physical state of the computer, such as CPU temperature and available disk space. If an unexpected signal value (or combination of) is observed, a suitable action is taken, which usually means to stop the robot and log the error.

The choice of the SAF subsystem for this evaluation was suggested by ABB personnel since it was considered relatively independent and not as business sensitive than other parts, e.g., motion control. SAF is probably the smallest subsystem in IRC 5 and contains, in the version studied, in total 6 061 lines of code or 3 994 executable statements. This includes conditions, assignments, function calls, etc., but not declarations or initializations. This code is distributed over 6 tasks and 196 functions. In contrast, there are several tasks in the ABB system where each task contain thousands of functions and hundreds of KLOC. Although small, the SAF code is however rather complex. The amount of conditional statements is almost twice as high in SAF as in other subsystems studied.

Four SAF tasks out of the six are used in the evaluation. Two tasks are excluded since they only contained about 50 lines of code each, too little to be representative cases. The model focus functions specified as input to MXTC included all functions for IPC communication (in the *IPC* class), the semaphore operations, and *taskDelay*, in total 9 functions.

6.3.2 Case 2 - MG - The Mongoose Web Server

Mongoose [119] is an open-source[3] web server and is included to broaden the scope of this evaluation with a rather different system. Mongoose is a multi-threaded application developed in C. It is a complete solution of manageable but non-trivial size, 2 410 lines of code distributed over 107 functions and three threads. It contains a large amount of conditional statements, 323 in total, corresponding to 24.9 % of its 1 297 executable statements.

[3]Mongoose is released under the liberal MIT license.

For this case MXTC is configured for another type of model extraction, focused on dynamic memory usage which implies three model focus functions, *malloc*, *calloc* and *free*, i.e., the standard functions for allocating and releasing heap memory.

6.3.3 Results

This section presents results from applying MXTC on the two described cases. Figures are presented per task/thread as well as in total, for all tasks of the specific system. The author is not allowed to reveal the task names in the SAF case. The SAF tasks are therefore labeled SAF-T1 ... SAF-Tn, and SAF-ALL refers to the whole SAF code. For symmetry, the Mongoose (MG) tasks are labeled in the same manner. The results are presented in Table 6.2.

Table 6.2: Results from MXTC on Case 1 (SAF) and Case 2 (MG)

Task/Case	LOC	FI	SI	CI	FO	SO	CO	RT
MG-ALL	2 410	107	1 297	323	77	604	179	26
MG-T1	1 969	85	1 010	246	62	531	166	23.2
MG-T2	705	31	379	71	15	97	27	6.6
MG-T3	223	15	142	36	2	13	4	1.8
SAF-ALL	6 061	196	3 994	1 003	137	1 967	643	186.2
SAF-T1	3 592	131	2 479	629	81	1 137	402	109.1
SAF-T2	2 710	64	1 666	477	52	990	343	123.6
SAF-T3	880	23	479	138	13	145	49	6.9
SAF-T4	550	15	290	80	2	9	3	2.1

The columns of Table 6.2 have quite cryptic labels in order to make them short enough. Their meanings are:

- LOC - Lines Of Code: a measure of program size excluding only comments and empty lines.
- FI - Function In: the number of functions in the input code.
- SI - Statements In: the number of executable statements in the input code. This excludes comments, empty lines and declarations.
- CI - Conditions In: the number of conditional statements, i.e., selections and loops, in the input code.
- FO - Functions Out: the number of model functions, i.e., functions found relevant, which are included in the output.
- SO - Statements Out: the number of model statements, i.e., statements found relevant, which are included in the output. Like for SI, only executable statements are counted.

- CO - Conditions Out: the number of conditions in the output, i.e., conditional model statements.
- RT - Runtime: the time required by MXTC to finish, in seconds.

Regarding the number of statements (LOC, SI and SO), note that some functions are used by more than one task, so the total number of statements is often smaller than the sum of the task sizes.

Diagrams of the Table 6.2 data are presented next in order to facilitate interpretation of the results. Figure 6.3 shows the number of executable statements in total (SI), for each task and case, as well as the number of model statements identified by MXTC (SO). Figure 6.4 shows the number of model statements (SO) in relation to the total number of executable statements (SI), per task and in total. In most cases, the identified model statements (SO) correspond to between 26 – 59 % of the input code (SI), but in two cases the fractions are much smaller, 3 % and 9 %. These cases correspond to the smallest tasks in the study, with only 290 and 142 executable statements, respectively. The reason why these tasks are so heavily abstracted is that they have quite simple behavior and contain only few calls of the specified model focus functions.

Figure 6.5 presents the runtimes of MXTC on the different tasks and in total. Figure 6.6 shows another view of the runtimes, plotted together with the total size of the analyzed tasks (SI) and the resulting model size (SO), with a linear scaling applied on the runtimes in order to fit them into the same scale. From this visualization (i.e., Figure 6.6) it seem likely the runtime is mainly dependent on the resulting model size (SO) rather than on the total size of the tasks (SI). The scale factors, 10 for SAF and 20 for MG, were chosen to line up the runtime data points next to the model size (SO) data points, in order to graphically compare their relation, since the hypothesis is that they are strongly dependent.

One can argue that by using another scale factor it would be possible to fit the runtime data to the total size (SI) data instead, since there seem to exist a correlation between these as well. In order to objectively investigate the correlation between the data sets SI, SO and Runtime, Microsoft Excel was used to calculate correlation factors (i.e., a value between 0 – 1, where 0 means no correlation and 1 means total correlation). The reported correlation factor for model size (SO) and runtime is in this case 0.979, while the correlation factor between total size (SI) and runtime is only 0.913. Thus, there is a correlation also in the latter case, but it is weaker, meaning that the runtime is primarily dependent on SO, i.e., the number of model-relevant statements in the code. This is very important for the scalability of the approach.

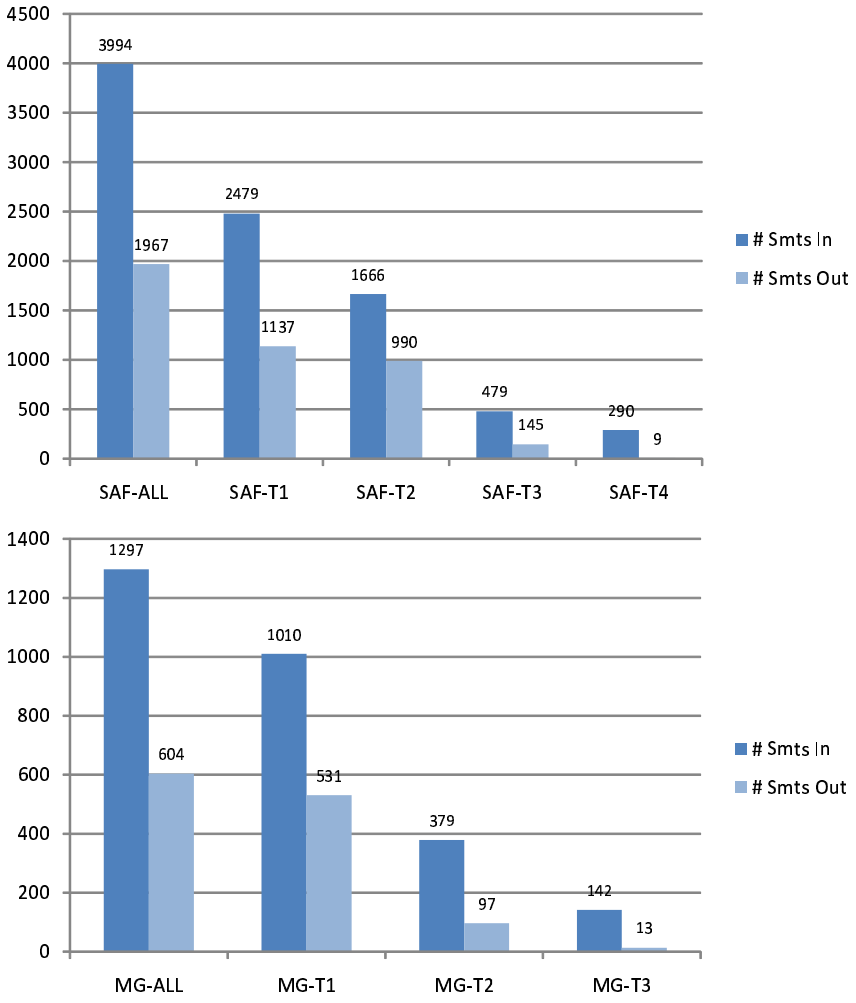

Figure 6.3: Total size (Statements In) and model size (Statements Out)

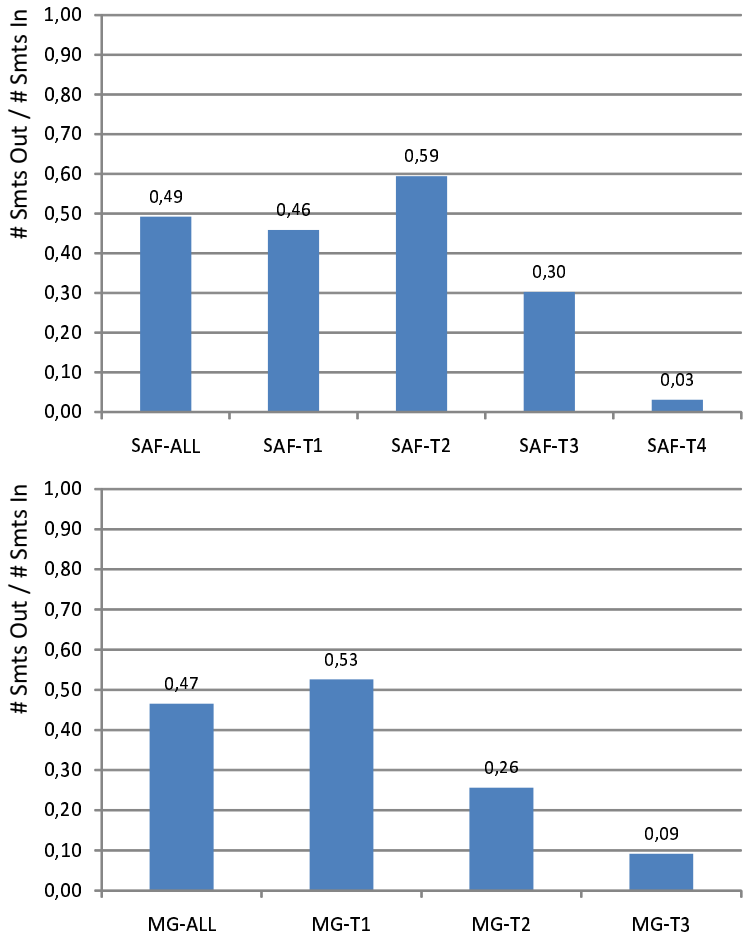

Figure 6.4: Relative model size (Statements Out/Statements In)

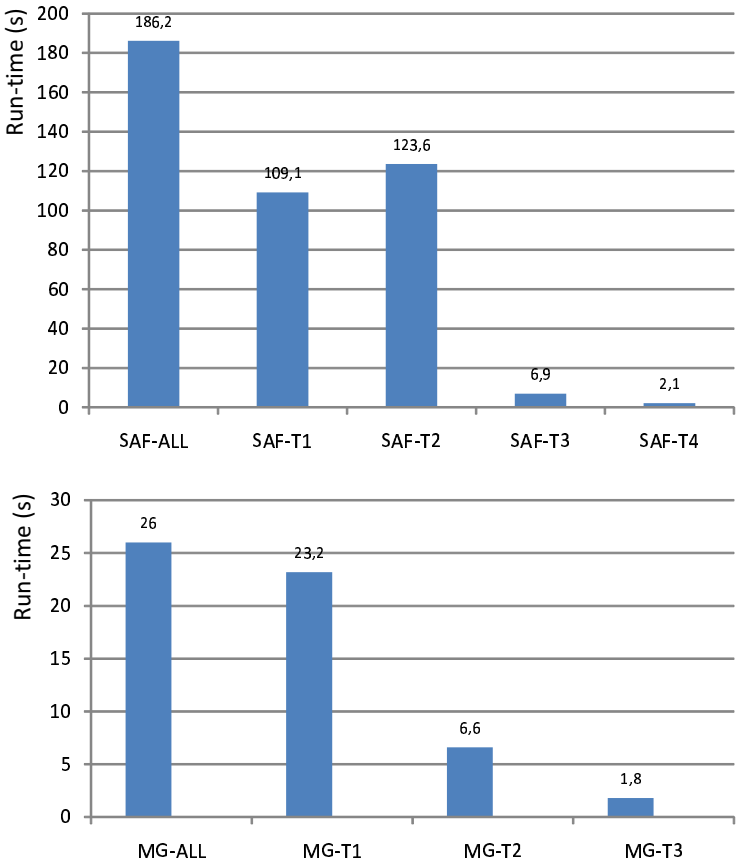

Figure 6.5: Runtimes of MXTC, for individual tasks and in total (seconds)

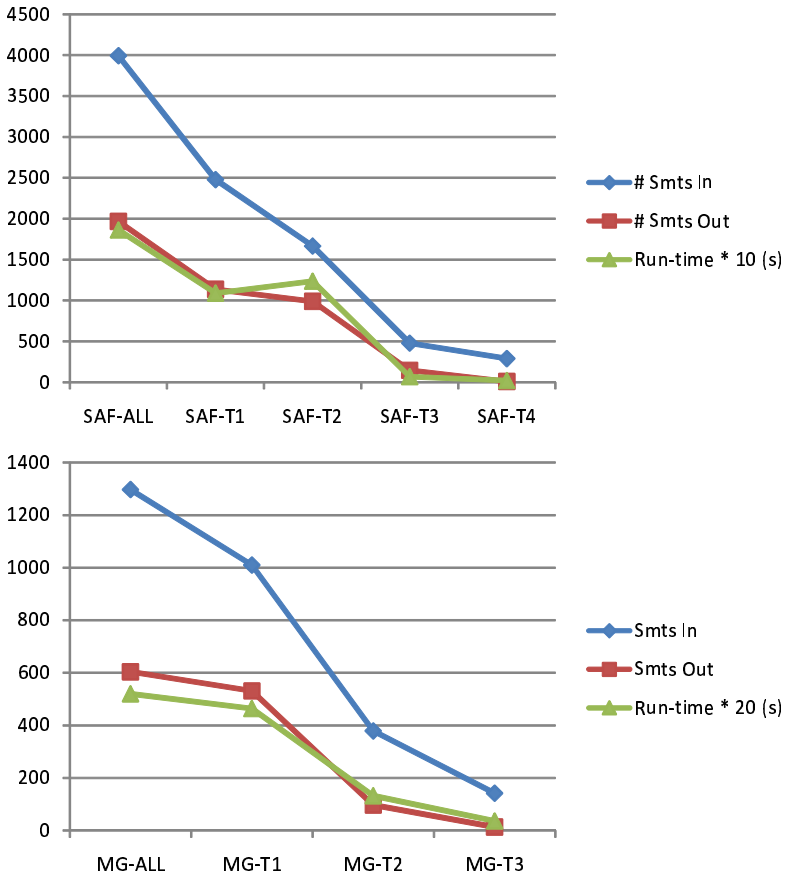

Figure 6.6: Runtimes (scaled), total size (SI) and model size (SO)

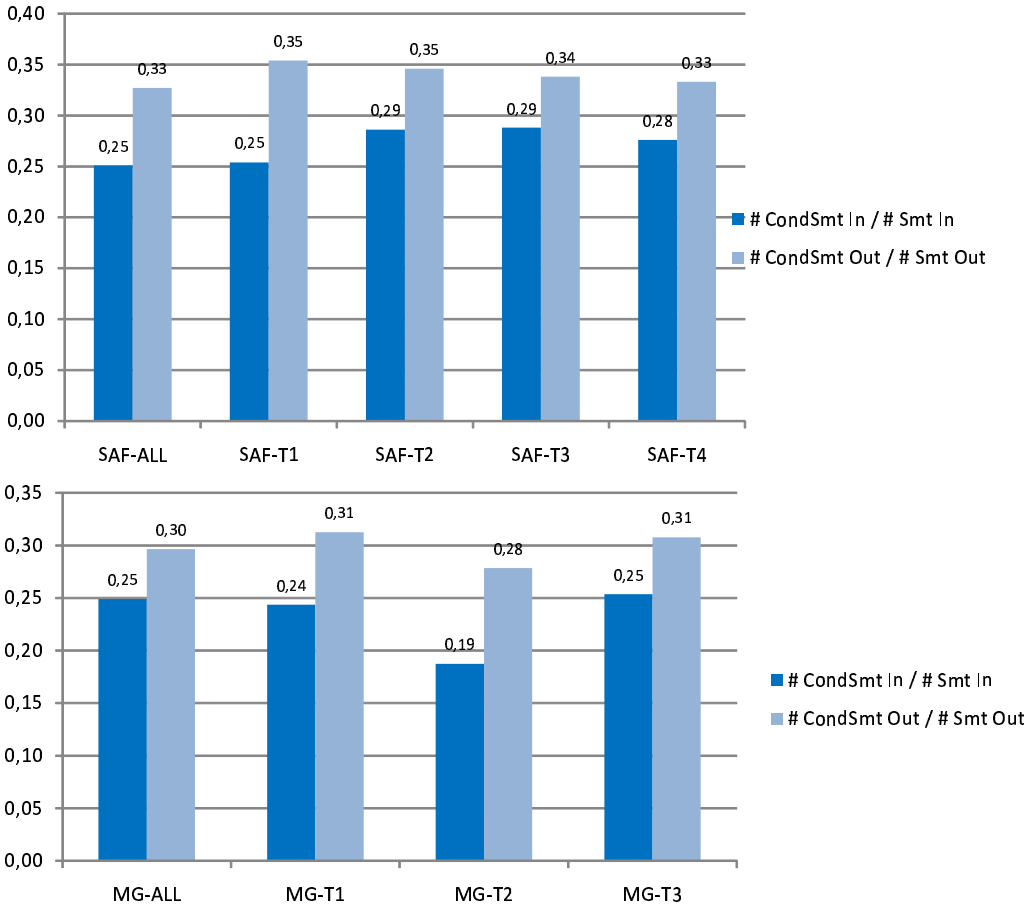

Figure 6.7: The amounts of conditional statements

Figure 6.7 shows the amount of conditional statements in the input code (CI/SI) and in the identified model statements (CO/SO). Observe the general increase in condition density caused by the model extraction. Conditional statements are more likely be model-relevant than statements in general, since any model statement enclosed by a conditional statement will cause the conditional statement to be relevant as well. Both the SAF and MG cases contain a large amount of conditions, the percentage is 25 – 29 % for SAF and 19 – 25 % for MG (i.e., 100*CI/SI). As a comparison, a large subset of the motion control (MOC) subsystem was studied, containing 84 156 executable statements in 3 829 C functions. This code would have been very interesting as a larger test case for the MXTC tool, but this was not possible since it was not available in full. The condition density was however found to be 14 – 17 % for the MOC code, i.e., about half the SAF condition density. The major difference in condition density could mean that MXTC would perform better on MOC than on SAF, in the sense that the resulting simulation models would be (relatively) smaller. This since fewer conditional statements in total probably mean fewer model-relevant conditions, which probably means fewer model symbols found due to such conditions. With fewer model symbols, fewer statements should be of relevance for the model.

The possibility for a correlation between condition density and relative model size was studied on the SAF code, in order to see if the condition density could be used to as a predictor of model size. No such correlation could however be found, maybe because the variance in condition ratios (25 – 29 %) was not large enough in comparison to the “noise” caused by their individual differences, which may be large.

The relative model size does however not depend directly on the number of conditions, but rather on what variables used in the model-relevant conditions. If a model condition only refers to already included variables it does not increase the model size at all. However, in a worst case scenario, a single condition could depend on every variable and statement of the program. Related to this is a study by Harman et al. [17], which reported that large *dependence clusters*, i.e., sets of interdependent statements, are very common. Harman et al. analyzed the source code for 45 open source programs, including common programs such as cvs, gcc, sendmail and ftpd, and they report that most of the studied programs contain large dependence clusters.

6.4 Katana Slicing vs. Commercial Tools

Program slicing is a general program analysis method with many uses. The author is aware of two commercial program comprehension tools with support for program slicing or data-flow analysis: Grammatech CodeSurfer [123] and Imagix 4D [135]. This section presents an evaluation of these tools on industrial code and a comparison with MXTC, with respect to scalability, accuracy, and type of analysis provided.

6.4.1 CodeSurfer

CodeSurfer, from Grammatech, Inc. [123], is a spin-off from the Wisconsin program slicing group, headed by Horwitz and Reps who, together with Binkley, originally proposed the approach to program slicing [23] based on the System Dependence Graph, or *SDG*, as discussed in Section 2.4.1. This is a very detailed program model, which can be time-consuming to construct for larger amounts of code. They patented the SDG representation in 1992 [22]. CodeSurfer is based on an improved version [21] on the original SDG slicing method. The website [121] of the research group headed by Horwitz and Reps states that CodeSurfer is limited to maximum 200 000 lines of code. Complex industrial systems may however contain millions of lines of code, and it is unclear if this is a practical limitation, in order to get a reasonable runtime, or an absolute maximum assuming no practical restrictions on runtime.

CodeSurfer can be configured in many different ways in order to trade accuracy for scalability. There are five level of accuracy, or *build presets*: “superlite”, “lite”, “medium”, “high”, and “highest”. Interprocedural program slicing is only supported if using build presets “high” (CS-High) or “highest” (CS-Highest). The exact meaning of the CodeSurfer build presets is explained in detail by the CodeSurfer documentation and from this it can be concluded that CS-High is actually less accurate than MXTC in some aspects, but more accurate in others:

- CS-High is more accurate than MXTC with respect to intraprocedural control-flow, since MXTC is not fully control-flow sensitive. CS-High can exclude irrelevant assignments of relevant variables, e.g., assignments which are “killed” by later assignments of the variable before the impact reaches the relevant code location. MXTC assumes that all assignments of a relevant variable are relevant.

- MXTC is more accurate than CS-High with respect to interprocedural

data-flow, since it is "context-sensitive" with respect to function calls. This means that MXTC separates between irrelevant and relevant call-sites when analyzing formal parameters.

- MXTC is more accurate than CS-High with respect to data structures since it allows for analysis of individual fields. In contrast, CS-High treats all fields of a data structure as if they where a single variable.

Which tool that is more accurate, CS-High or MXTC, thus depends on the source code analyzed. CS-Highest however employs very accurate methods which should make it more accurate in all aspects compared to MXTC. However, as presented later in this section, the scalability of CodeSurfer has been found to be severely restricted and not suitable for analysis of large industrial systems, at least if using build presets "High" or "Highest", which are required for program slicing.

The CodeSurfer documentation claims: *"With build presets high and above, 400 KLOC should usually be possible, given enough time and disk space."*. As previously mentioned, the research group behind CodeSurfer states at their website [121] that 200 KLOC is the upper limit. This figure may however be a few years old. Moreover, there are different ways of measuring lines of code (LOC). The LOC figures reported by the thesis author is from the "Understand for C++", specifically its "Lines Code" metric, which includes declarative statements and executable statements, but not comments or blank lines. The author has however not found any information regarding how the 200/400 KLOC limits for CodeSurfer has been measured. CodeSurfer does not have a similar metric from what the author could find, but instead reports program size measured in "program points", which is more exact but not directly comparable to lines of code. A statement may correspond to several program points and the number of program points may vary drastically between statements.

The scalability of CodeSurfer (version "2.1p1") has been investigated using industrial code from ABB's control system for industrial robots, IRC 5. The whole system contains some 3 million lines of code, but this study was limited to a subsystem containing 183 KLOC, which in Section 6.2 is labeled ABB-2.

CodeSurfer crashed repeatedly when trying to analyze this code using build preset "highest" (CS-Highest). It reported an error message stating that it failed to allocate memory. When using the less accurate build preset "High" (CS-High) it did not crash, but the analysis did not finish in reasonable time. After about 92 hours it was terminated and the log file indicated that it was still on one of the first analysis steps, labeled *"Computing call graph and modified globals (CG)"*, which had executed for about 90 hours before it was aborted. When

analyzing smaller cases, several more time-consuming steps followed this one, so even if this step would have been finished in just a few hours more, the whole analysis would probably take much longer, probably several additional days, or even weeks. Up until the point where it was aborted, CodeSurfer had stored 2.8 GB of intermediate data and used 900 MB or RAM.

In comparison, the tool used for constructing the symbol database for MXTC, "Understand for C++", processed the 183 KLOC code (ABB-2) in about 16 seconds, as presented in Section 6.2. MXTC could then successfully be used for performing backwards slicing. As an example, a slice on a randomly selected variable in the ABB-2 identified 24 statements in 1.7 seconds. The statements were distributed over three functions. In another example, the runtime was only three minutes for calculating a large slice consisting of about 2 000 statements in about 140 functions.

CodeSurfer however worked fine on a smaller test case, the RTSSim simulation framework presented in Chapter 3, which contains 3 830 LOC. This allowed for a comparison between CodeSurfer and MXTC. The comparison was made by querying a commonly used global variable in RTSSim, the simulation clock, *clk*, which was expected to produce a large slice. The results are summarized in Table 6.3. In this table, "Parse Time" refers to the time required in order to construct the program model from the source code, i.e., an SDG for CodeSurfer and a symbol database for MXTC (using Understand). "Slicing Time" refers to the time required for calculating the backwards slice for the global variable *clk* in RTSSim, based on the respective program model.

Table 6.3: MXTC/Understand compared to CodeSurfer

	Parse Time (s)	Slicing Time (s)
MXTC/Understand	2	51.6
CS-High	158	< 1
CS-Highest	160	< 1

As previously presented in Section 6.2, the Understand tool requires only about 2 seconds for creating a symbol database over this code. The slice computed by MXTC contained 568 executable statements, i.e., about 37 % of the 1 586 executable statements of RTSSim. The runtime for MXTC was 51.6 seconds and its peak RAM usage was 18.7 MB. The runtime for CS-High was 158 seconds, during which it stored 312 MB of intermediate data. Its peak RAM usage was about 240 MB, almost 13 times more than for MXTC. The

runtime of CS-Highest was 160 seconds, only slightly higher than the runtime of CS-High. Note however, after CodeSurfer has constructed its program model, backwards slicing and other types of program analyzes is performed very quickly, without noticeable delay. This is not the case for MXTC, which performs most calculation on demand, given a specific query.

The slice produced using CS-Highest contained 8 253 out of 37 144 program points, i.e., 22.2 % of the analyzed program. When using CS-High the slice is considerably larger, 11 770 program points, corresponding to 31.7 % of the program. Note that CodeSurfer and MXTC present the results in different ways, program points vs. statements, so this figure is not directly comparable.

The slice size ratios should however give an indication of accuracy and it seems as CS-High is more accurate (31.7 %) than MXTC (37 %) in this case, although the validity of this comparison is questionable. MXTC was however about three times faster than CodeSurfer in either configuration; the total analysis time of MXTC/Understand (parse + slice time) is only 33.5 – 33.9 % of the parse time of CodeSurfer in this case.

6.4.2 Imagix 4D

Imagix 4D, from the Imagix Corporation [135], is a similar type of tool as CodeSurfer, but no information is available regarding the analysis algorithm used by Imagix 4D; it seems to be a secret of the Imagix Corporation. It is however likely that also this tool uses the SDG approach, or a similar dependency graph model. This since Imagix 4D has similar scalability problems as CodeSurfer, many hours for larger amounts of code, and since the SDG approach seems to have been the de-facto solution for program slicing since the mid 1990's.

The company behind Imagix 4D, Imagix Corporation, does not present any figures regarding scalability; an evaluation therefore been performed using the ABB-2 case, like for CodeSurfer. As previously mentioned, this is a subsystem from ABB's control system for industrial robots, IRC 5, consisting of 416 source files containing in total 183 492 lines of code, not counting whitespace or comments, and 3 116 functions.

The comparison was made with respect to randomly selected variables from different parts of the ABB code. The "Calculation Tree" feature of Imagix 4D (version 6.6.1) was used to find the relevant dataflow dependencies. Unlike MXTC and CodeSurfer, Imagix 4D does not produce full executable slices; it identifies the directly relevant dataflow but does not analyze conditions guarding relevant statements. In order to allow for a fair comparison, the MXTC tool

was configured to perform the same kind of analysis as Imagix 4D by ignoring conditions and thereby any statements on which they depend.

The Imagix 4D tool gave a warning regarding the size of the project; indicating that the analysis could take several hours. It required almost 7 hours, and the peak memory usage was 1.5 GB. The analysis time should be similar for any other variable since the main cause of the long runtime is (probably) the construction of a detailed whole-program model, i.e., an SDG or similar. Like for CodeSurfer, once this program model has been constructed, analyzes are very quick. However, the many hours required for constructing the program model is obviously a serious scalability problem.

When using MXTC on this variable, the runtime was only 2 seconds. If also including the 16 seconds required for constructing the symbol database, as presented in Section 6.2, MXTC is about 1 400 times faster than Imagix 4D in this case. This experiment was repeated on all four selected variables and the runtime was around two seconds in all cases. The peak RAM usage of MXTC was 33 MB, i.e., about 45 times less than for Imagix 4D. The identified slices were quite small, around 10 statements, since only the direct dataflow was included. This is typically only a small subset of the total slice obtained when taking conditions into account. Both MXTC and Imagix 4D gave similar results with respect to the number of identified statements, but a detailed comparison was however never made.

6.5 Conclusions

This chapter has demonstrated that the MXTC tool, based on the Katana algorithm, works on real industrial code and scales to large software systems. Even though the cases used in this evaluation are relatively small, they are real systems of non-trivial size and complexity. The runtimes of MXTC are short, just over 3 minutes in total for four tasks containing in total 6 000 lines of code, and, more importantly, mainly depends on the number of statements found to be of relevance for the simulation model, i.e., the number of model statements, rather than the total program size of the tasks. If extrapolating these numbers, the MXTC prototype would process a 1.2 MLOC system in 10 hours. This is of course longer than desired, but the tool is still a prototype and is implemented in Perl, a scripting language. According to benchmark comparisons [143, 144] between C++ and Perl, C++ has been observed to be 8 – 120 times faster Perl. If a C++ version of MXTC could be made 10 times faster than today, it would be able to process the whole ABB IRC 5 code in about 3 hours, assuming the previous extrapolation.

The amount of model statements can be quite small, in one case as little as 3 % of task code, but is most cases in the range 25 – 50 %. For the task with largest relative model size 59 % of the executable statements were identified as model statements. However, the relative model size can most likely be reduced significantly by taking intraprocedural control-flow fully into account. The current solution, which is not fully control-flow sensitive, may in some situations include irrelevant statements, which in turn may have dependencies to other irrelevant symbols and statements, which thereby are included as well.

For general program slicing, the MXTC tool has shown to outperform the commercial tools CodeSurfer and Imagix 4D with respect to scalability. MXTC could successfully operate on a 183 KLOC code base, requiring only 16 seconds for constructing the program model (the symbol database, using Understand), and 1.7 seconds for identifying a program slice consisting of 24 statements in three functions. In practice, the runtime of MXTC seems to scale linearly with the number of statements found relevant. For instance, an analysis identifying about 2 000 statements, distributed over 142 functions, required only 3 minutes. In contrast, CodeSurfer could did not terminate in 92 hours when attempting to process the 183 KLOC code-base. Imagix 4D, which provides a less detailed analysis, was able to analyze this code in about 7 hours. MXTC was 1 400 times faster than Imagix 4D in this case, and used 45 times less memory.

On a smaller example which CodeSurfer could handle (the RTSSim code,

3 800 LOC) MXTC was found to be three times faster and used 13 times less memory. As expected, CodeSurfer was however more accurate than MXTC, even though an exact comparison is not possible since different size metrics are used by the tools. Using the most accurate configuration, CodeSurfer found 22.2 % of all program points to be relevant for a particular variable and if using the more scalable but less accurate configuration (build preset "High"), CodeSurfer found 31.7 % of the program points relevant for the same case. In comparison, MXTC found 37 % of the executable statements to be relevant for this variable. This probably corresponds to more than 22.2 % or 31.7 % of the *program points*, even though these measures of program size (program points vs. executable statements) are not directly comparable as an executable statement may contain several program points.

This chapter has also demonstrated that the program representation used by MXTC/Katana, the symbol database, can be constructed within minutes also for large software systems, containing millions of lines of code. The Understand tool, which MXTC uses for constructing the symbol database, has proved itself able to process 183 KLOC in 16 seconds and 1 084 KLOC of C code in just under 2 minutes. The predicted time for analyzing a very large code-base of 10 MLOC is 63 minutes. Thus, constructing the symbol database is not a threshold which limits the scalability of the approach, unlike tools using the heavy-weight SDG program model.

Chapter 7

Uses and Experiences of Software Trace Recording

This chapter discuss recording of event traces during execution of embedded software systems, or *trace recording* for short, with a primary focus on task scheduling. The chapter has three main purposes:

1. to explain the role of trace recording in the context of the envisioned analysis framework,

2. to present techniques for trace recording, including own, specific solutions and more general methods, and

3. to present experiences from five industry collaboration projects performed where software trace recorders have been developed for different industrial software systems and evaluated with respect to CPU and RAM overhead.

This chapter is mainly motivated by research question **Q3**, presented in Section 1.3. Since trace recording is a key component of the envisioned analysis framework, presented in Section 1.2, the general applicability of trace recording is naturally of high importance. This has two aspects, implementation feasibility on common real-time operating systems, and the overhead with respect to CPU and RAM usage. The scientific contributions of this chapter are mainly the presented experiences from industrial collaboration projects, with respect to the two aspects of trace recording applicability.

The chapter also presents an approach for recording of *timing profiles* for the RTSSim simulation framework. This section shows how a full implementation of the envisioned analysis framework could use trace recording in order to populate the simulation models with timing data recorded from the real system.

7.1 Uses of Trace Recording

The approach of this thesis requires trace recording for several purposes. Trace recording is necessary in order to do execution-time measurements under realistic circumstances, since the systems in focus typically are multitasking and use preemptive scheduling. In order to measure execution-time, it is necessary to monitor the context-switching and only account for the CPU time actually consumed by the specific task. Trace recording is also necessary during simulations, in order to record a detailed simulation trace for later analysis and comparison. Moreover, the three analyses presented in Section 1.2 (impact analysis, model validation and regression analysis) all requires methods for comparing trace recordings. This naturally assumes that such traces can be recorded in the first place, either from a real system or from a simulation. Trace comparison is discussed further in Chapter 8.

Another aspect of trace recording is *trace visualization*, which can be used for comparison of traces and for inspecting details in simulations or real system traces. A trace visualization tool has been developed in this research, the Tracealyzer, which originally targeted the simulation framework. However, it can also be used to study traced recorded from real systems and this use of the Tracealyzer has served as a "low hanging fruit" for industry collaboration, since industrial developers often can have immediate use of this tool for debugging, performance optimization and general system understanding.

7.2 Trace Recording Fundamentals

This work focuses on software-based recording, where code instrumentation is inserted at suitable code locations in order to log the desired information. This typically implies adding function calls to a software recorder module, which is integrated in the base platform of the system. There are however also hardware solutions [109] and hybrid solutions [136], using both hardware support and software probes. The added code instrumentation for registering events are often referred to as *software probes*. Thus, *events* refer to the instances of

recorded data, while *probes* refers to the code responsible for recording the events.

In the context of this thesis, the events in focus of recording are primarily scheduling events where the operating system scheduler switches the currently running task, i.e., the *task-switch* event, but also other types of event may be stored, such as calls of operating system services (e.g., semaphores or IPC operations) and application-specific "user events". In this work task-switches are considered as instantaneous actions and only a single time-stamp is stored for each task-switch event. The context-switch time (from the OS) is thereby accounted to the execution time of the tasks.

It is possible to detect scheduling events on most real-time operating systems, either by registering callbacks (hooks) on system events like task-switches, task creation and termination, or by modifying the kernel source code. The callback approach is possible on at least VxWorks (from Wind River) and OSE (from ENEA). Operating systems with available kernel source code, e.g., Linux and RTXC Quadros[1], can be modified to call the trace recorder module on relevant events. Åsberg et al. [1] has shown that for Linux (2.6 kernel), the only kernel modification required is to remove the "const" keyword from a specific function pointer declaration. It is however possible to realize Linux trace recording without kernel modifications, if using a custom scheduler like RESCH [2].

Software trace recorders typically operate by storing relevant events in a circular RAM buffer, as binary data in fixed-size records. In this manner, the recorder always holds the most recent history. In all implementations presented in this paper, a single ring-buffer is used for storing all types of events. An alternative strategy is to store events of different types in different buffers, to be able to use different sized events for better memory efficiency. Such a solution however becomes more complex and thereby less robust, and the extra logic required, and buffer dimensioning problems, may cancel out the benefits of this solution. Storing different types of events in the same ring buffer requires that the events have a common location for storing an *event code*, typically the first byte, which indicates how the data should be interpreted. By storing all types of events in the same buffer, the relative order of the events is maintained automatically and it is thereby possible to use relative time-stamps, as discussed later.

Each task-switch event corresponds to exactly one *execution fragment*, i.e., the interval of uninterrupted execution until the next task-switch event. For

[1] `www.quadros.com`

each execution fragment, the following information needs to be registered:

- What task that executes, i.e., a task ID.
- When the execution fragment started, i.e., a time-stamp.
- Why the task-switch occurred, i.e., the scheduling status of the task after the execution fragment has ended.

The rest of this section will discuss these three aspects (what, when and why) of task-switch event recording and finally also recording of other types events, such as calls of operating system services. This provides a foundation for the following descriptions of the five industry collaboration projects where trace recorders have been developed for different industrial software systems.

7.2.1 Task Identity (the "What")

The most obvious and fundamental piece of information in a task trace is the task identity of each execution fragment. Most operating systems use 32-bit IDs for tasks, even though many embedded system only contain a handful of tasks, at most a few hundred. It is therefore often a good idea to introduce a short task ID, *STID*, using 8 bits or 16 bits only in order to make the task-switch events less memory consuming. Saving 2 or 3 bytes per event might not sound like a big deal, but the recorder solutions described in this chapter only require 4 – 8 bytes per event in total, so the reduction is significant. With an 8-bit STID it is possible to handle up to 256 unique tasks, which is sufficient for many systems. In other cases a 16-bit STID would surely be enough, as it allows for over 65,000 unique tasks.

The STIDs needs to be allocated on the creation of tasks and quickly retrieved when storing task-switch events. This can be achieved by storing the task's STID in its task control block (TCB), either by modifying the kernel (possible in Linux and RTXC Quadros [137]) or by using an unused "spare" field of the TCB data structure, which are available in VxWorks [132]. In OSE [134] there is a "user area" of each process which can be used for this purpose.

Complex embedded systems with event-triggered behavior, such as telecom systems, often create and terminate tasks dynamically. In this context, it is important to recycle the STIDs, otherwise the recorder will sooner or later run out of STIDs. This means that the termination of tasks must be registered in order to mark the particular STID as no longer in use. An STID may however not be reused for newly created tasks as long as there are references to a

particular STID in the event ring buffer, unless the newly created task is identical to the last task referred by the STID. Otherwise, this would change the meaning of the older task-switch events and make the trace incorrect.

7.2.2 Time-stamping (the "When")

For timing analysis purposes each execution fragment must have a time-stamp, typically stored as an integer value. Since the trace contains the complete sequence of execution fragments, a single time-stamp is sufficient per fragment, either when it started or when it ended, it does not matter as long as it is consistent. From here on, it is assumed that time-stamps refer to the start of execution fragments. This implies that the operating system overhead, i.e., the execution time of the context-switch code, is accounted to the task execution times.

Obtaining a time-stamp is normally a trivial operation, but standard libraries typically only allow for getting clock readings with a resolution of maximum 1 or even 10 milliseconds, depending on the tick rate of the OS. This is too coarse-grained for embedded systems timing analysis, since many tasks, and especially interrupt routines, often have execution times measured in microseconds. Fortunately, embedded systems usually have hardware features for getting more accurate time-stamps, such as real-time clocks (*RTC*). In other cases, if the CPU frequency is constant, it is possible to use a CPU instruction counter register.

In order to reduce the memory usage when storing the events, a good method is to encode the time-stamps in a relative manner, i.e., to only store the time passed since the previously stored event, i.e., the durations of the execution fragments. If the absolute time of the last stored event is kept, it is possible to recreate absolute time-stamps during off-line analysis. This allows for correlating the trace recording with other time-stamped logs created by the system, which can be important for troubleshooting purposes.

The relative time-stamp encoding allows for using fewer bits for storing time-stamps, typically between 8 – 16 bits per event. A problem however occurs in cases where the duration of an execution fragment exceeds the capacity of the time-stamp field, i.e., 255 or 65535 time units. Handling the overflow issue for relative time-stamps introduces a tradeoff between memory usage and recorder-induced jitter (i.e., predictability). The most reliable but least efficient solution is to use enough bits for this purpose so that the overflow does not occur. A more efficient solution is to reduce the number of time-stamp bits to better fit the typical fragment duration, and instead introduce an alternative handling of the few cases where the number of time-stamp bits are insufficient.

In this case, an extra "XTS" event (eXtended Time-Stamp) is inserted before the original event, carrying the time-stamp using enough (32) bits. This however introduces a control branch in the task switch probe, which might cause timing jitter in the recorder overhead and thereby additional timing jitter in the system as a whole, which can be bad for testability and predictability. We however believe that this jitter is negligible compared to other sources of jitter, such as execution time variations. The XTS approach is used in all five recorder implementations presented in this chapter.

Note that the higher resolution used for storing time-stamps (e.g., nanoseconds instead of microseconds), the higher RAM usage. This is due to either the need for a wider time-stamp field (more bits) or more frequent XTS events. However, if using a too low time-stamp resolution (e.g., milliseconds), some execution fragments may get a zero duration and thus becomes "invisible" in off-line visualization and analysis, e.g., with respect to CPU usage contribution.

7.2.3 Task-switch Cause (the "Why")

In preemptive fixed-priority scheduling [98, 99] a task-switch may occur for several reasons: the running task might have been blocked by a locked resource, it might have suspended itself, terminated, or a task of higher priority might have preempted the task. This information is necessary to record in order to allow grouping of execution fragments into task *instances*, also known as task *jobs*. A task instance corresponds to one logical execution of the task, i.e., the processing of one work-package. The end of an instance is referred to as the *instance finish*, and corresponds to the termination of the task, i.e., exit from main function, or for non-terminating tasks when the task has performed one iteration of the main loop and enters a blocked or waiting state awaiting the next task activation, i.e., the start of the next instance.

From a trace perspective, a task instance corresponds to one or several consecutive execution fragments of the same task, possibly interleaved by execution fragments of other tasks, where the last fragment is ended by the instance finish, and where any previous fragments of the same instance is ended by preemption or blocking. The concepts of instances and execution fragments are illustrated by Figure 7.1, using an example with three tasks, where task *H* has the most significant priority and task *L* the least significant priority. Each execution fragment is labeled $T_{i,f}$, where *T* is the task name, *i* the instance number and *f* the execution fragment number within the instance. The upper row indicates the task-switch cause: preemption (*P*) or termination (*T*).

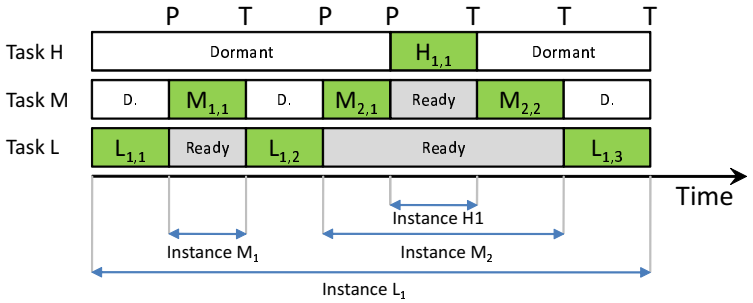

Figure 7.1: Execution fragments and task instances

What counts as an instance finish for non-terminating tasks is system specific and depends on the software architecture. For non-terminating tasks there are two options for detecting instance finish: using the scheduling status or using code instrumentation. If a certain scheduling status can be unambiguously associated with the inactive state of a task, a task-switch due to this scheduling status can be regarded as the instance finish. The next execution fragment of this task is thereby the start of the next instance. This approach is however difficult if the task may be blocked for other reasons (other semaphore or message queues), since the scheduling status at best tells the type of resource causing the blocking, but not the identity of the specific resource.

A pragmatic solution is to add code instrumentation in the task main loop, immediately before the operating system call corresponding to the instance finish. A problem with code instrumentation in the application code is that the application developer needs to be aware of the recorder, maintain the instrumentation points properly and also adding new instrumentation when adding new tasks to the system.

7.2.4 Recording Operating System Services and User Events

Apart from recording the task scheduling trace, there may be other events of importance for recording, such as calls of certain operating system services and application-specific “user events”.

Operating systems calls can easily be recorded if the system has an OS isolation layer, which contain wrappers for operating system features, or if the kernel source code is available for modifications, like for Linux and RTXC

Quadros [137]. For potentially blocking OS services, like attempting to lock a semaphore, it is a good idea to use two probes, an “entry” probe immediately before entering the OS service and and “exit” probe directly after the return. An off-line analysis or visualization tool can thereby identify the beginning and corresponding end of any operating system blocking.

It is possible to detect blocking from the recorder’s event trace, either off-line or during run-time, by checking if any task-switch event exists between the entry and exit probes. The chance of a task-switch occurring for other reasons than blocking between the entry and exit probes of a OS service is very small. This risk can however be eliminated completely by a control in the recorder‘s task-switch event handling routine: if the last stored event is an entry event for a particular OS service and the task scheduling status also indicates “blocked”, it is for sure a blocking call of the specific OS service. If the task scheduling status however is “ready”, it was actually a preemption which happened to occur between the entry and exit probes.

One can argue that a simple solution of preventing such unlikely preemptions, and thereby make the task-switch control described above unnecessary, is to disable interrupts before storing the entry event and enable them after storing the exit event. This would however increase the interrupt latency and the OS kernel may enable interrupts during the processing of the particular service, so preemptions might still be possible before the exit probe has been stored.

User events correspond to explicitly logged information from the application code, which is stored using *user probes*, typically inserted by the application developer. This can be used in order to log events or data of importance for, e.g., troubleshooting purposes. It can however also be used for fine grained execution-time measurements (between any two points in the program code) and for monitoring application-specific limited resources.

A concept of *probe channels* has been used to avoid enable logging of named events. A probe channel connects a string name to a numeric handle during system initiation. The numeric handles, or *probe channel IDs* is then used to label the later stored user events. A similar technique can be used for storing names for particular probe values, so that the name can be displayed instead of a numeric value in off-line visualization and analysis tools. This is be valuable when monitoring e.g., state variables, which always hold the value of a named constant, since the state names are typically more familiar to the developer than the corresponding numeric codes, which may change during the system evolution.

7.3 The Tracealyzer

Tracealyzer is a visualization tool with analysis capabilities for various timing and resource usage properties. It is mentioned at several locations in this thesis as it can be used for studying the output of the RTSSim simulator, as presented in Chapter 3, and for model validation purposes, as presented in Chapter 8.

The main view of the tool displays a task trace using an novel visualization technique. Other trace visualization tools, such as the Wind River WindView, uses a trace visualization technique similar to a logic analyzer or Gantt-style charts, where the status of every task is displayed at all times, with one row or column per task. Such visualizations become hard to comprehend when zooming out to overview a longer scenario and the user may need to scroll in two dimensions.

In contrast, the visualization used by the Tracealyzer focuses on the task preemption nesting and only shows the currently active tasks, as depicted by Figure 7.2. This makes the trace easier to overview, especially long and complex scenarios with many tasks involved. The tool also provides a CPU load view over the entire trace. The two views are synchronized; the time window display in the main window is indicated in the CPU load overview and by clicking in the CPU load overview the trace view displays the corresponding time window. The tool has advanced features for searching, with several filters, and can also generate a report with detailed timing statistics for each task. The tool also allows for exporting timing data regarding tasks and other events to text format. More information about the tool is available at `www.percepio.se` where a demo version can be downloaded.

The Tracealyzer was originally developed as a means for verifying the trace recorder developed for the robotics control system of ABB system, as described in Section 7.4.1. The tool was however soon found valuable by ABB developers, for troubleshooting and performance analysis, due to the possibilities for visualizing and analyzing recordings of the system in operation. Even though the ABB developers had access to a commercial trace tool, WindView, from the operating system developer Wind River, they decided already in 2005 to integrate the newly developed Tracealyzer recorder in the base software platform and to have it active by default, also in the production version. This means that it is actively monitoring all industrial robots delivered by ABB since 2005, several thousand per year. The Tracealyzer is today used systematically at ABB Robotics; at least 30 developers have used it at some point, and many use it frequently.

The second generation Tracealyzer (i.e., version 2.0) is since 2009 in com-

mercialization by Percepio AB, in collaboration with Quadros Systems, Inc. who develops the real-time operating system RTXC Quadros. A special version for RTXC Quadros will be released in September 2010 under the name RTXCview, which becomes the official tracing tool for the RTXC Quadros platform. For more information about RTXCview, see `www.quadros.com`. Note that the first generation Tracealyzer, i.e., up until version 1.31, is still freely available for non-commercial use, and for existing commercial users according to prior agreements.

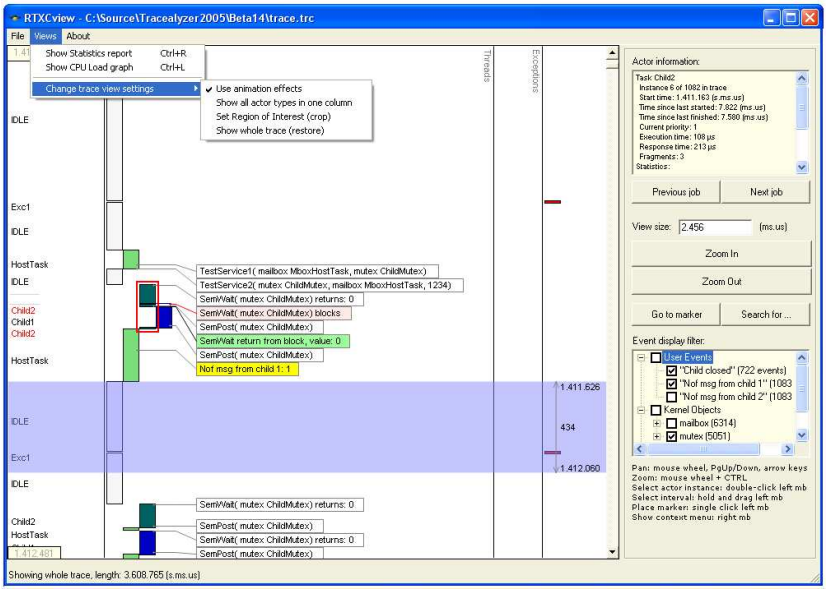

Figure 7.2: The Tracealyzer/RTXCview

7.4 Five Industrial Trace Recorder Projects

Another purpose of this chapter is to document experiences from five industry collaboration projects where trace recording solutions have been developed for industrial software systems and from these experiences attempt to summarize a list of observations or recommendations. The five projects have been performed in collaboration with respective companies and, in the end, at least

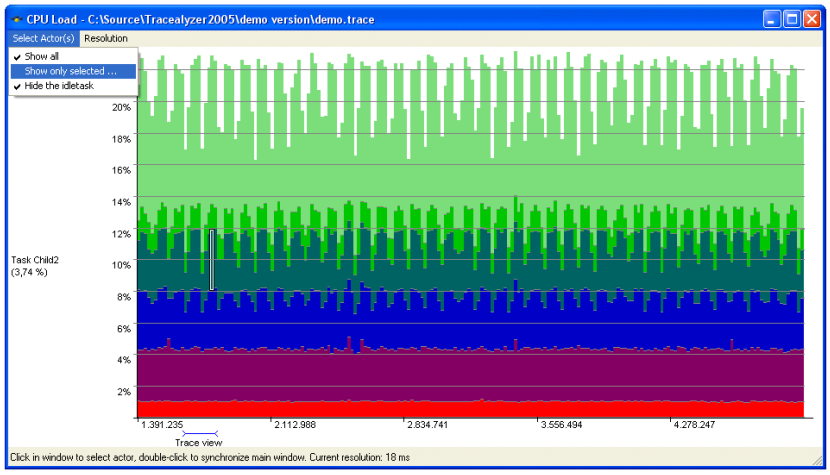

Figure 7.3: The Tracealyzer/RTXCview, CPU load view

evaluated by system developers. Three of the projects have lead to industrial deployment of the results, in one case as the official tracing tool for a commercial real-time operating system.

The purpose of these projects have varied slightly, but all have included trace recording and visualization using the Tracealyzer, described in Section 7.3. The research motivation for these projects have been to verify the applicability of trace recording techniques on different embedded systems platforms, since trace recording is a key enabler for the timing analysis in focus of this work.

7.4.1 The RBT Project

ABB develops a control system for industrial robots, IRC 5. This is a large and complex embedded software system, consisting of around 3 million lines code. The operating system used is VxWorks, and the hardware platform is an Intel-based Industry PC. At the time of the evaluation, this system used an Intel Pentium III CPU and had 256 MB of RAM. It moreover has a flash-based hard drive, a network connection and an onboard FTP server.

Since VxWorks has features for registering callbacks on task-switch, task creation and task deletion, these events could be captured without kernel modifications. The task-switch callback function receives pointers to the task con-

trol blocks (TCBs) of both the previously executing task and for the task that is about to start. The developed recorder uses 8-bit STIDs, stored in an available "spare" field in the TCB by the task create callback routine. The task names are stored at creation time in a list of tasks, indexed by the STID.

All types of events are stored in a single ring buffer, using a fixed event size of 6 bytes. This required the use of bit-wise encoding in order to fit the desired information into the 48 bits available. The two first bytes are used to store two pieces of information in an asymmetric manner, where 2 bits are used for the event code and 14 bits for a relative time-stamp, obtained from an instruction counter of the Intel CPU used by this system. Since the time-stamp resolution used in this recorder is 1 μs, this solution allows for a execution fragment duration up to 2^{14} μs (16.4 ms). This is typically more than enough for this system; there are usually several task-switch events every millisecond. However, in some system modes, such as during system startup, the task-switch rate is much lower and the 14 bits may then be insufficient. As a precaution, an additional "XTS" event (eXtended Time-Stamp) is stored if the relative time-stamp does not fit in 14 bits. The XTS event stores the relative time-stamp using 32 bits and overrides the time-stamp field of the associated (following) event.

Recording inter-process communication events was considered important and this was accomplished by adding code instrumentation in the OS isolation layer. Semaphore operations are however not instrumented; they are very frequent in this system and it was feared that monitoring these would cause a major additional recording overhead. The event rate of the ABB system when recording task scheduling and IPC operations was found to be around 10 KHz. A ring buffer capacity of 100 000 events (600 000 bytes) therefore gives a trace history of around 10 seconds. The runtime of a recorder probe was found to be on average 0.8 μs, which at the typical event-rate of 10 KHz translates into a CPU overhead of 0.8 %.

As mentioned, ABB Robotics personnel decided after this project to integrate the recorder in their control system IRC 5 and to keep it active by default, also in the production version. The Tracealyzer is today used systematically at ABB Robotics for troubleshooting and for performance measurements. The recorder is triggered by the central error handling system, so whenever a serious problem occur a trace file is automatically stored to the system's hard drive. A trace file is in this case only about 600 KB and can therefore easily be sent by e-mail for quick analysis, e.g., if a customer experiences a problem.

7.4.2 The ECU project

The system in focus of this project was the software of an ECU, i.e., a computer node in a vehicular distributed system developed by Bombardier Transportation[2]. Since also this system used VxWorks a similar recorder design could be used as in the RBT project. The company developers were mainly interested in the CPU usage per task, as well as for interrupt routines, during long-term operation of the vehicle. The hardware platform was a Motorola[3] PowerPC 603 running at 80 MHz.

In initial experiments using the Tracealyzer tool, the main problem was the endianness; the Motorola CPU uses big endian encoding, while the Tracealyzer expected little-endian encoding. In the first experiments in using the Tracealyzer for this system, the solution was a recorder design where all data is stored in little-endian format during run-time, by assigning each byte explicitly. This is far from optimal with respect to the CPU overhead of the recording and should be avoided. The latest version of the Tracealyzer assumes that the recorder writes the data to a binary file in native format and therefore detects the endianness, and converts if necessary, while reading the trace file. The endianness is detected by using a predefined 32-bit value, where the four bytes have different values, which is written to a predefined file location by the recorder, typically in the very beginning. An off-line analysis tool can then find the endianness from the order of these values.

Unlike the RBT project, this project included recording of interrupt routines. The operating system VxWorks does not have any callback functionality or similar for interrupts, but the interrupt controller of the CPU allowed for this. Interrupt routines could thereby be recorded as high-priority tasks, by adding task-switch events to the main ring buffer in the same way as for normal tasks.

An interesting requirement from Bombardier was that the recorded information should survive a sudden restart of the system and be available for post-mortem analysis. This was accomplished by using a hardware feature of the ECU; the event buffer was stored in Non-Volatile RAM (NVRAM). During the startup of the system, the recorder recovers any trace data stored in the NVRAM and writes it to a file, thereby allowing for post-mortem analysis. The ECU was equipped with 4 MB of NVRAM which is plenty since the company only needed a 2.5 second trace history. Since it was only desired to log task-switch events in this project, i.e., no IPC events like in the RBT case, it was possible to reduce the event size from six to four bytes per event.

[2] `www.bombardier.com`
[3] Now Freescale

A recorder and a company-specific analysis tool was developed in a Master's thesis at Bombardier [97], but the Tracealyzer was not used after the initial tests leading to the thesis project. One of the students was however employed by the company after the thesis project.

7.4.3 The WLD Project

This system is also an ECU-like computer, although not in the vehicular domain and the company is anonymous in this case. The computer system in focus is a node in a distributed system, with the overall purpose of automated welding for production of heavy industrial products. The computer in focus controls an electrical motor and is connected to a set of similar computer nodes over a field bus. The CPU used was an Infineon XC167, a 16-bit CPU running at only 20 MHz. The operating system used was RTXC Quadros.

Since the kernel source code of RTXC Quadros is available for customers, the recorder could be integrated in a custom version of the kernel. It was however not trivial to find the right location where to add the kernel instrumentation, especially for the task-switch events, since parts of the context-switch handling is written in assembly language. Time-stamps were obtained from the real-time clock (RTC) feature of the Infineon XC167 CPU and stored in a relative manner in the same way as in the previous cases.

There was no need for using short task IDs (STIDs) for reducing memory usage, since RTXC Quadros already uses 8-bit task handles. However, dynamic creation of tasks required an indirect approach, involving a lookup table, as the task handles of the operating system are reused. The lookup table contains a mapping between the RTXC task ID and the index of the task in an recorder-internal list of tasks, which is included in the generated trace file. The recorder task list contains the name and other information for up to 256 tasks. On task creation, the list is searched in order to find a matching task, so repeated dynamic creations of a single task only generates a single entry. However, there was no "garbage collection" in the recorder task list, so tasks which are no longer in the trace history still occupy an entry. This issue is however solved in the latest recorder implementation, described in Section 7.4.5. Interrupt routines were recorded by adding two probes in every interrupt service routine (ISR). Task-switch events are stored in the beginning and in the end of the ISR, using the interrupt code to look up a "faked" task entry, specified in a static table containing all interrupts. Nested interrupts are supported using a special purpose stack, holding the identity of the preempted ISRs, as well as the currently executing task.

The CPU overhead of the recording was measured and found higher than in previous cases, although still acceptable. The event rate was found to be around 500 Hz, i.e., about ten times less than in the ABB system, but the slow, low-end CPU (16-bit, 20 MHz) caused relatively high probe execution times, around 60 μs. This is 75 times longer than the probe execution times in the ABB system (0.8 μs). With a 500 Hz event rate, this translates into a CPU overhead of 3 %, which is significant, but probably not a serious issue compared to the potential benefits of trace recording. However, this recorder was not optimized for CPU usage; it was rather a first prototype on this platform. Several optimizations/fixes are possible in order to reduce the CPU usage of this recorder solution, as discussed in Section 7.4.6.

In a first evaluation by developers at the company, the welding system recorder was used together with the Tracealyzer tool in order to pinpoint the cause of a transient error which they previously had not been able to find. By studying a recorded trace in the Tracealyzer tool they could find that the error was caused by a wrongly placed "interrupt disable" instruction, which allowed for interrupts occurring during a critical section where interrupts should have been disabled. The company did however not integrate the developed recorder solution on a permanent basis, but has used the solution later for similar purposes. On those occasions, they have created a custom build using the instrumented RTXC Quadros kernel. This can lead to probe effect [89] problems, i.e., that the activation (or deactivation) of recording changes the system behavior.

7.4.4 The TEL Project

This project was performed together with an anonymous company in the telecom industry, which develops products based on the operating system OSE from ENEA. The particular system studied used a high-end PowerPC CPU, running at 1 GHz and with 256 MB of RAM. This project had the goal of providing means for exact CPU load measurements. Previously they had used a tool which sampled the currently executing task at randomly selected times and in that way got an approximate picture of the CPU usage of the various tasks. This was however considered too inaccurate. A Master's thesis project was initiated in 2008 in order to develop a recorder for this system [96].

A recorder for the Tracealyzer tool was developed and evaluated using standard performance tests of the system. The recorder used the "kernel hooks" feature of OSE, which is similar to the callback features in VxWorks, and 16-bit STIDs for tasks (*processes* in OSE terminology), stored in the "user area"

of the process. The main problem was that OSE did not allow direct access to the kernel memory, for reading the process control block. It was thereby not possible to get the scheduling status of the tasks, which is necessary in order to identify task instances. A workaround was implemented, the Tracealyzer was modified for this case, so that priorities were used instead of status. This assumes that the priorities are static since the recorder cannot read them at the task-switch events, only at task creation. The resulting recorder was evaluated in the company lab using their normal test-cases for load testing. The CPU overhead of the recorder was found to be 1.1 % at an event rate of 18 KHz and a CPU load of 30 %. This result has to be considered as most acceptable, especially since the recorder was not optimized for CPU usage.

The company did however not use the resulting recorder since it was not mature enough for industrial deployment, which requires a very robust solution, and since there was no obvious receiver at the company who could take over the recorder development and verification.

7.4.5 The RTOS Project

In 2009 the thesis author was contacted by a representative of Quadros Systems, Inc. who expressed interest in a collaboration aiming at developing a new trace tool for their operating system. This resulted in the development of the second generation Tracealyzer, along with a special version for Quadros Systems named *RTXCview*. This project also included the development of a whole new recorder design, in close collaboration with the chief engineer at Quadros Systems.

This recorder has little in common with the previous four versions. A major difference is that this recorder is designed for logging of generic operating system services without any hard-coded information in the recorder design. The recorder contains no assumptions on the operating system services that should be logged, this is configured through kernel instrumentation and using a configuration file of the Tracealyzer/RTXCview. All information needed by the off-line tool is stored in a single block of data which is statically initialized during compile-time. This eliminates the need for calling a recorder initialization routine at system startup, which was necessary in the previous versions. This design reduces the startup time of the system and makes it easy to retrieve the trace recording, e.g., if the system has stopped on a breakpoint using a debugger. This recorder does not use any bit-wise manipulations, which should reduce its CPU usage significantly. To achieve this, a larger event size was necessary, using eight bytes per event instead of four or six bytes.

In this design, there is no explicit task-list, as in other earlier recorders, but instead there is a generic symbol table which contains the names of tasks, user events, semaphores, and other named objects. A string added to this symbol table returns a 16-bit reference, the byte index of the string in the symbol table. If an identical string already exists in the symbol table, a reference to the existing string is returned instead of creating a new entry. This is therefore memory efficient and solves the issue of repeatedly created dynamic tasks. The symbol table lookup is fast since all symbol names which share a 6-bit checksum are connected in a linked list, as depicted by Figure 7.4. This however requires two extra bytes per symbol name, for storing the index of the next symbol with the same checksum, and an array holding 64 16-bit values, the linked-list heads. If a longer checksum (i.e., more checksum bits) is used, the look-up time is reduced, but the amount of memory required for the array of linked-list heads doubles for every extra checksum bit. For systems with plenty of memory, an 8-bit checksum should however not be any problems, since it only requires 512 bytes.

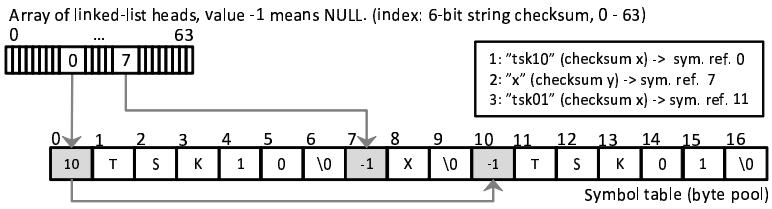

Figure 7.4: The symbol table

On task-switch events, the 8-bit RTXC task handles are stored without bothering about possible later reuse of the handle, which then might change the meaning of the currently stored handles. This is instead resolved off-line. The names of the currently active tasks are stored in a "dynamic object" table which is updated on task creation. When a task is terminated ("closed" in Quadros terminology), the name from the dynamic object table is stored in the symbol table and the resulting reference is stored, together with the RTXC task handle, in a special "close" event, which informs the off-line analysis tool that this mapping was valid up until this point. The off-line analysis can then find the correct task names of each execution fragment by reading the event trace backwards, starting at the trace end, and for each close event update the current mapping between RTXC task handle and name.

The described approach for handling reuse of dynamic task handles is used for all types of dynamically created kernel objects in RTXC Quadros, i.e., tasks, semaphores, mailboxes, alarms, etc. Time-stamps are stored in a relative manner, using 8, 16 or 32 bits per event, depending on the number of bytes available for each event type. Like in the other projects, XTS events are inserted when the normal time-stamp field is insufficient. The time unit of the time-stamps does not have to be microseconds as the time-stamp clock rate is specified in the recorder and provided to the off-line analysis tool, which converts into microseconds. It is thereby possible to use the hardware-provided resolution directly without run-time conversion into microseconds. Another time-related aspect is that absolute time-stamps are maintained also if the recording is stopped abruptly, e.g., due to a crash or breakpoint. The absolute time of the last stored event is kept updated in the recorder's main data structure and is thereby available for the off-line analysis. From this information and the relative time-stamps of the earlier events it is possible to recreate the absolute time-stamps of all events in the trace.

A prototype of this recorder has been implemented and delivered to Quadros Systems, who at the moment (Spring 2010) are working on integration of the recorder in their kernel. There are no big problems to solve; it is mainly a question of the limited development resources of Quadros Systems. No evaluation regarding the CPU overhead of this recorder has yet been performed. Developing and verifying a trace recorder for an operating system is much harder than for a specific embedded system, since an operating system recorder has to work for all hardware platforms supported by the operating system.

7.4.6 Summary of Recording Overhead Results

This section summarizes the measured recording overhead imposed by the recorders in the four cases where such measurements have been made, i.e., all cases except for the RTOS case (Section 7.4.5). The results are presented in Table 7.1. In all cases except RBT, each event requires 4 bytes. In the RBT case, 6 bytes per event is used.

Table 7.1: Measured recording overheads in four industrial cases

Case	OS	CPU	F (MHz)	ET (μs)	ER (KHz)	CPU OH (%)	RAM OH (KB/s)
RBT	VW	P. III	533	0.8	10.0	0.8	60.0
ECU	VW	PPC 603	80	2.0	0.8	0.2	3.1
WLD	RTXC	XC167	20	60.0	0.5	3.0	2.0
TEL	OSE	PPC 750	1000	0.6	18.0	1.1	72.0

The four right column of Table 7.1 have the following meanings.

- ET – average probe execution time
- ER – average event rate
- CPU OH – percentage of CPU used by recorder (overhead)
- RAM OH – number of event buffer bytes used per second

Note the relatively long probe execution time in the WLD case: 60 μs. The second longest probe execution time (for the ECU case) was 30 times shorter although the clock frequency was only four times higher. This is probably due to the difference in CPU type, the CPU in the WLD case is a 16-bit micro-controller, while more powerful 32-bit CPUs were used in the other cases.

The four evaluated recorders were optimized for low RAM usage, on the expense of higher CPU usage. It therefore possible to reduce the CPU overhead significantly by instead optimizing for CPU overhead, e.g., by increasing event size in order to avoid bit-wise encoding. Other possible optimizations are to move as much functionality as possible off-line (e.g., time-stamp conversion) and by using "inline" functions and macros instead of C functions. The latest recorder design, presented in Section 7.4.5, includes these improvements and should thereby give significantly lower CPU overhead, although not yet confirmed by experiments.

7.4.7 Measuring Probe Execution Time

Estimating the CPU overhead requires that the typical execution time of a single probe can be accurately measured. This is performed by executing two probes in direct sequence, and taking the difference in time-stamps. The second probe may execute faster than the first, due to caching and other hardware features, but the execution-time obtained would mainly correspond to the first probe, assuming that the time-stamp is obtained in the very beginning of the probe.

For the ABB case and the telecom case, the probe execution time was found to be shorter than 1 μs, the measurements gave a (truncated) difference of 0 μs. In such case, the time-stamp resolution has to be increased, e.g., by a factor 10. In the new design, developed for RTXC Quadros, this is easily accomplished since the time-stamp resolution is configurable. In earlier versions of the Tracealyzer tool, this was hard-coded and assumed to be 1 μs.

Another tested solution for measuring probe execution time was to measure over a sequence of adjacent probes and dividing the total execution time by the number of probes. Such a result will however be optimistic. Only the first probe executes under realistic conditions, while the following probes may execute faster due to caches and other hardware optimizations.

7.4.8 Lessons Learned

The five described projects have identified several issues and considerations related to trace recording, primarily with a technical focus but also a few "softer" organizational questions, related to technology transfer projects in general. The experiences have been condensed into a list of recommendations, considerations and general reflections.

An important consideration is choosing an appropriate level of detail for the trace recording, e.g., should the recording include events such as interrupts or semaphore operations? This is ultimately a trade-off between the value of the information, with respect to the purpose of the recording, compared to the consequences of the associated recording overhead, such as a reduction in system performance, or increased unit cost if compensating the overhead with better but more expensive hardware. Including too little information may however also lead to increased costs, if quality assurance becomes harder and more time consuming due to the missing information. Such costs are however very hard to measure, since there is no "control case" to compare with.

A related consideration is the trade-off between CPU usage and memory usage implied by using more advanced storage techniques, such a bit-wise encoding or data compression, which are more memory efficient but also more CPU demanding. It is however the author's belief that such techniques should generally be avoided in order to reduce the CPU overhead. The only exception would be low-end embedded systems with very limited RAM and where a long trace history is more important that system performance.

Another consideration is whether the recorder should be integrated in the system on a permanent basis, or only activated when necessary. A permanent integration means that the CPU and memory overhead of the trace recording becomes permanent, and thus may degrade the system performance for its customers. The author however recommends this approach for most systems, for several reasons:

- The risk for probe effects is eliminated, since the recording becomes an integrated, and tested part of the system.

- A trace is always available for diagnostic purposes, e.g., if the system fails in post-release use, and can simply be e-mailed by a user or service engineer to the development office.
- The automatic availability of a trace lowers the threshold for developers to begin using the trace recorder. If additional configuration is required for activating the recorder, the recorder might seldom be used.
- The recording cost, in terms of CPU and memory usage, is typically small, in many cases not noticeable, and therefore well motivated by the benefits.

An exception to this recommendation would be systems which are highly focused on average performance and where the unit cost is a major issue, such as low-end multimedia devices.

A good strategy is to store the information in a single static data structure, which is initiated in compile-time. By storing all events in a single buffer with fixed-size entries, the relative order of events is maintained. More advanced solutions, using multiple buffers and/or variable-sized events, may reduce memory usage, but leads to higher recorder complexity, higher risk of errors and higher CPU overhead.

The feasibility of using a custom trace recorder, i.e., not developed by the operating system vendor, mainly depends on the possibility for capturing task-switch events. This is possible on all real-time operating systems studied, either using built-in event callbacks, available in VxWorks and OSE, or by modifying the kernel, which is possible for open source platforms such as Linux, and also for proprietary platforms where the source code is provided for customers, such as RTXC Quadros.

The CPU overhead of trace recording can be expected to be below 1 % in average on high-end systems, and below 5 % in average on low-end systems, such as 16-bit micro controllers. The memory usage of the recorder can be expected to be between 4 – 8 bytes per event, depending on the recorder design, and the frequency of task-switch events seems to be in the range of range 5 – 20 KHz for high-end systems, and below 1000 Hz for low-end systems. Thereby, the memory required can be as low as 4 KB per second of event history, plus some 5 KB for additional meta-information, e.g., a symbol table. A five second event history would in this case require 25 KB, in total, and a 1 MB buffer gives over 4 minutes trace history. For a high-end system with a task-switch event rate of 20 KHz, e.g., the telecom system described in Section 7.4.4, the required amount of memory is 80 – 160 KB per second of event history, depending

on event size, which allows for 12 – 25 seconds of trace history if 1 MB is allocated for the purpose.

A recommendation is to design trace recorders as simple and robust as possible and instead place the "intelligence" in the off-line tool. For instance, timestamps should not be converted during run-time, bit-wise encoding should be avoided, and complex startup initialization routines should be replaced by static initialization. A simplistic recorder design is also important if the recorder is to be maintained by the target system development organization, which may have limited time or interest in understanding a complex recorder design.

Don´t expect developers (i.e., the trace recorder users) to immediately realize the possibilities of all recorder features. They typically have little time available for pro-active, quality-oriented work, such as adding custom monitoring of application data ("user probes") in order to facilitate future diagnostics. In larger organizations, such activities often have to be performed as explicit "quality" projects, approved and budgeted for by management.

If developing a trace recorder for another organization, as an external expert in the area, make sure there is an explicit receiver of the solution, typically a developer or lower level manager, which have competence, interest and time available for taking over the responsibility for the developed solution. This was the main success factor in the projects which led to industrial use.

7.5 Recording of Simulation Timing Profiles

The RTSSim simulation framework presented in Chapter 3 requires timing data, execution times and inter-arrival times, which accurately describe the modeled system. In the current implementation this is however not explicit; the timing data is provided manually in the model code. The plan is however to keep all such data in a separate data file, loaded by RTSSim, which contains different data-sets to which the RTSSim model refer. This data file is referred to as a *timing profile*. The context of the timing profile in the overall analysis framework is presented in Figure 7.5. The timing profile may also contain task response time data, for use in model validation and impact analysis.

A straight-forward approach to constructing such timing profiles for existing systems is trace recording with additional code instrumentation for detailed execution time measurements between relevant program points. Typical execution-time information is thereby obtained from real executions, under realistic conditions. This data can also be complemented with WCET analysis results, in order to provide safe upper bounds.

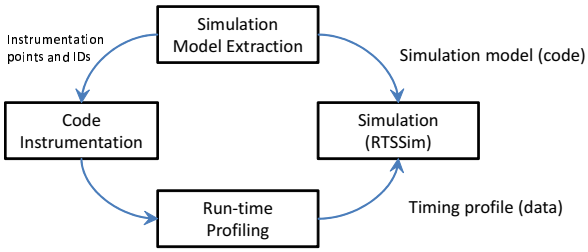

Figure 7.5: The context of the timing profile

This section presents a solution for this purpose, an component of the overall vision presented in Section 1.2. This solution has not yet been implemented, but a detailed design is here presented which is planned for implementation in future work.

In an automated analysis, the application code is instrumented with software probes, which when executed records a time-stamp and a numerical probe identifier associated to the code location. Two types of such probes are proposed: *execution-time probes* (*ETPs*) and *input event probes* (*IEPs*). The IEPs are discussed in Section 7.5.2.

7.5.1 Recording Execution Times

The recording of execution time data for the timing profile requires that an ETP is inserted at each point in the program where accurate time is of relevance in the simulation model, i.e., at the *time synchronization points*, or TSPs (as discussed in Section 6.1). The TSPs should include the program points corresponding to task inputs and outputs, including IPC and global variables shared between tasks, and calls of model focus functions. These points are identified by the model extraction tool, e.g., the MXTC tool presented in Chapter 6.

The sequence of ETP events resulting from executing the instrumented software can be viewed as a directed graph, named “Instrumentation Point Graph” by Betts and Bernat [9]. The ETPs corresponds to nodes in this graph and a graph edge represent all code paths which directly connects two consecutive ETPs, i.e., without other ETPs in between. In this work we refer to such edges as *probe graph edges*, or *PGEs*.

The approach is illustrated by Figure 7.6 using an example program, a simple task containing two ETPs with IDs 1 and 2. The illustration also shows a

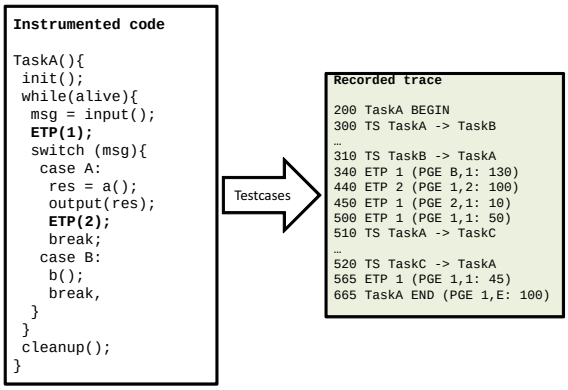

Figure 7.6: An instrumented program and resulting events

corresponding example trace[4] where the first column is the absolute time of the events, the second column is the type of event[5]. The annotations on the ETP events refer to the resulting PGE execution times. As an example, *PGE B, 1: 130* means that an execution time of 130 time units has been observed between the task beginning (the B event) and ETP 1. In this example, it is assumed that task begin events (B) and task end event (E) are detected without using explicit probes, but instead derived from the task-switch events.

The execution-time data of the timing profile corresponds to a set of PGE data sets, each representing the execution times for a specific PGE, i.e., the CPU time used between the ETP events corresponding to the PGE. The PGE data points are calculated by first taking the time-stamp difference between the two ETPs corresponding to the PGE and then subtracting the CPU time used by other tasks during this period.

Figure 7.7 shows the possible PGE edges, as a graph, annotated with the PGE execution times observed. Note that "PGE 2, E", i.e., from ETP 2 to the task end, has not been executed and execution time data is therefore not available in this case. This highlights a central issue of this approach, and with testing in general: getting sufficient test case coverage. The ability to assess the coverage is discussed in Section 7.5.6.

[4] In textual rather than binary form, for illustration purposes.
[5] TS stands for a task-switch event

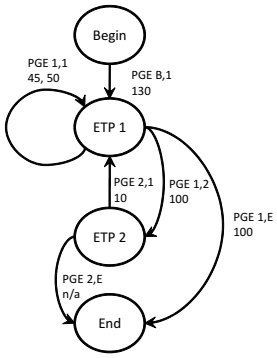

Figure 7.7: The probe graph of the example in Figure 7.6

7.5.2 Recording System Inputs

The inter-arrival time data describe the typical timing between input events, such as messages received from a TCP/IP socket or the triggering of an interrupt service routine by an external signal. Such events are often not periodic but occur in a seemingly random pattern.

Assuming that input event probes (IEPs) can be inserted at the code locations corresponding to the arrival of such events, e.g., interrupt service routines, recording such events is only a matter of logging *input events* containing an IEP identity and a time-stamp. This can be implemented as a user event for a trace recorder, as discussed in Section 7.2.4. The set of inter-arrival times for each input event probe can then be extracted from the sequence (trace) of input events. Since this is independent of the task scheduling, it is only a matter of measuring the distance in time between IEP events with the same identity.

If an input event provides data of relevance for the simulation model, e.g., a command code, there are two strategies, either modeling the inter-arrival time data and the input data separately, or together. The first option is easiest and the resulting model will be smaller storage-wise, but will have a larger search space, which may include infeasible behaviors. In the latter case, both inter-arrival time data and input data is modeled as a directed graph, where the nodes corresponds to specific input values and the edges corresponding to observed input data sequences. Each edge is associated with a set of inter-arrival time observations. This is the same technique as the one proposed for execution time, but with one graph per IEP probe, observed data values as

nodes (instead of probe IDs) and inter-arrival times in the data-sets (instead of execution times). This solution however assumes that the possible input values are relatively few, e.g., command codes or states. If an input has a large amount of possible values, e.g., a 16-bit sensor reading, this representation is not suitable. In such cases, it may be possible to use a model which predicts sensor readings by using a model of the physical, continuous system. Another solution is to use separate, independent models of input data and inter-arrival times.

7.5.3 Modeling Recorded Timing Data

The measured data, i.e., execution times, inter-arrival times and response times, will vary from time to time due to differences in software and hardware state between the executions and are therefore modeled using probability distributions. A simple, straight-forward approach is to use a uniform distribution between the lowest and highest observation (watermark) for a specific PGE execution time. This is suitable when using simulation optimization methods, as presented in Chapter 4. In this case, the focus is to find a scenario as extreme as possible with respect to the specified property, e.g., task response times. This is essentially a search problem, where the main concern regarding PGE execution times is the feasible intervals. A question in this regard is if to add a “margin”, since the watermarks probably does not represent the best-case and worst-case execution times. This is highly related to the RapiTime product of Rapita Systems, Ltd. [136], which predicts the worst-case execution time based on this type of measurements. This is however outside the scope of this thesis.

For performance analysis, i.e., of typical timing, uniform distributions are however not suitable since they do not preserve the mean values or shapes of the real execution-time distributions. In this case it is instead recommended to use empirical distributions, in which every observed PGE execution is represented as a individual data-point. During simulation, execution times for a specific PGE are selected by random (sampled) from the PGE data-set, using a uniform probability distribution. Thereby, each PGE execution time will be chosen with the same probability as observed in the measurements. The downside is that this requires a lot of memory and only the observed execution times will be used. In some cases, not every value in a specific PGE interval might be observed, even though very similar values have been recorded, both smaller and larger. Larger “gaps” should be taken into account, since such are most likely caused by control-flow conditions. However, small “gaps”, a single or a few individual missing values, should be regarded as missing data points, i.e.,

values which most likely are possible. It is desired to include also such values in the timing profile.

An interesting solution is to mix the two methods: starting with the empirical distribution, group (stratify) nearby data points into intervals (stratas), where the mean value of the strata bounds is as close as possible to the mean value of the included data points. Thus, the data points of each strata should be fairly evenly distributed within the strata interval. Each strata is given a probability corresponding to the number of data points within the group divided by the total amount of data points. This gives a PGE data set consisting of stratas, where each strata has an upper bound, a lower bound and a probability. When using such a timing model in RTSSim, the simulator first select what strata to sample from, based on their respective probabilities, and then select a value from the selected strata interval, using a uniform probability distribution. Note that the stratas (their intervals) may overlap without restrictions.

7.5.4 Using Timing Profiles in Simulation

The timing profile, a data file, is loaded by the simulator before starting simulation. It contains a set of data-sets of different types (execution-time, inter-arrival time or response-time), with identities. In this solution, the *Execute* function takes as parameter a ETP identifier. This is assigned by model extraction tool when the time synchronization point (TSP) is identified. Note that the TSP exists in the real system source code as well as in the model code, in the former as an execution time probe (ETP) and in the latter as an *Execute* call, both using the same identifier as argument.

The ETP ID is used to calculate a PGE ID, in the beginning of the *Execute* function, by merging the ETP ID with the recent history of ETPs, the *ETP history*. In Figure 7.7, a ETP history of 1 is used, which is the minimum, but a longer history can be kept using a fixed-size ring-buffer. The ETP history is updated in the end of the *Execute* function, where the current ETP ID is added to the ETP history.

Note that the PGE data-sets refers to the execution-time between *two* ETPs independent of ETP history length, but if using a longer history, of two (2) or more, there may be multiple data-sets for a PGE, depending on the execution path leading to the first ETP. A longer history reduces the risk of the simulation exploring infeasible scenarios, i.e., selecting a sequence of PGE execution times which cannot occur in practice, e.g., due to hardware-related dependencies between different code segments, such as cache conflicts. However, it also requires a larger amount of measurements in order to get complete coverage,

since the number of PGE data-sets grows with the history length. In a future simulation tool, the ETP history could be a parameter for the user to decide.

7.5.5 Systematic Data Collection

The instrumented software system needs to be exposed to large amounts of testing in order to collect a sufficient amount of data on each PGE, especially if using a longer ETP history (as discussed in Section 7.5.4). Large scale data collection for simulation timing profiles can be achieved by integrating the ETP and IEP instrumentation into the system on a permanent basis and relying on the existing system testing process to stimulate the system sufficiently. If a trace recorder exists in the system, and if the ETPs can be added automatically, e.g., by the model extraction tool, this is only a matter of updating the test case specifications with instructions to collect, label (system version and build configuration, etc.) and store the trace recording after a set of tests have been performed. This way, large amounts of trace data becomes available at very little extra effort, and the risk of a probe effect [89], i.e., that the activation (or deactivation) of recording changes the system behavior, is eliminated since the release version is identical to the analyzed system. This solution would however imply an additional performance overhead, which, depending on the amount of ETPs and IEPs, might be significantly larger than the overheads in the case studies presented earlier. An evaluation of this approach with respect to recording overhead is planned for future work.

A possible extension of this approach is to perform the extraction of PGE data periodically during run-time, e.g., by using a low-priority task which regularly analyze the last recorded events and perhaps even keeps a complete timing profile updated, i.e., describing the whole execution since system startup. In this way, the RAM buffer can be much smaller, compared to if doing the analysis off-line. However, this means keeping the timing profile in target system RAM, which imposes requirements on its size. Therefore, it might not be wise to store every observation separately, since the system may be running for days, weeks, or even months continuously when in post-release use. A more memory efficient solution is required, where the observations are grouped into intervals, as discussed in Section 7.5.3.

The low priority of the profiling task means that it will not affect system performance, but also a risk of losing data. If higher priority tasks delay the profiling task for a long time, and during this time create many probe events, there is a risk that unprocessed probe events are overwritten in circular event buffer. This risk can be reduced by increasing the rate of the profiling task

and/or increasing the event buffer size. However, in the worst case, a data loss does not invalidate the result, it only implies a temporary loss of data and the profiling can resume once the profiling task is allowed to execute.

Another possibility for the profiling is to use a hardware recorder, like the RTBx product from Rapita Systems, Ltd. [136], as presented in Section 2.5. The hardware support minimizes the CPU overhead of the analyzed system, as it only need to output the probe identifier on a digital I/O port, typically a matter of 1 – 2 instructions, while the time-stamping and data storage is performed by the recorder hardware (a separate computer). The downside of this approach is that it requires quite large and expensive hardware, which can be hard to use for post-release data collection since the equipment is generally too expensive (and large) to be included in products.

7.5.6 Coverage

The coverage of the timing profile, with respect to PGE execution times, can be determined in at least two ways. The safest approach is to compute all possible PGEs through static analysis of the source code and compare with the PGEs in the timing profile. The computational complexity of calculating the CFG can however be a scalability problem for analysis of complex embedded systems, especially since the calculation must be performed with a global scope, i.e., as an interprocedural analysis. Another approach is to rely on the simulation of the model for assessing the coverage. If the simulation requests a PGE which does not exist in the timing profile, an error is produced to highlight this issue. This approach is easier, but depends on the coverage of the simulations.

If PGEs are found to be missing, additional test-cases must be added in order to include the code path(s) corresponding to the PGE. Furthermore, if PGEs are found to have too few data points this must be addressed, especially if the PGEs represent a larger block of code, containing many paths. This thesis does however not elaborate on the necessary number of data-points per PGE, nor the process of selecting test-cases for maximum test-case PGE coverage.

The probabilistic modeling of execution time is associated to a risk of producing simulation results which are not feasible in practice, e.g., if there are hidden dependencies between different PGE due to hardware state. For instance, say that we have two PGEs, PGE 1 and PGE 2, which always execute in sequence. There is one task with higher priority which may preempt either in PGE 1 or PGE 2, but never in both during the same task instance, e.g., due to the task periods. The preempting task impacts the cache and causes a cache-miss when the preempted task resumes, which increases the execution time

significantly. Such cases correspond to the highest observed execution times for both PGE 1 and PGE 2. The simulation may however select the maximum values from both PGEs in the same instance and thereby produce an execution time which is not feasible in practice. Solving this issue however requires a more detailed timing profile which takes the previously selected execution times into account. This would quickly increase the size of the timing profile, which thereby requires more measurements for construction.

7.6 Conclusions

This chapter has presented uses, experiences and techniques for trace recording in the context of embedded systems in general, and in the context of the overall analysis framework presented in Section 1.2, where trace recording is a necessary and important component.

A central use of trace recording in the envisioned analysis framework is to collect timing data for the simulation models. For this purpose, this chapter has presented an approach for recording, modeling and integration of *timing profiles* in simulation frameworks like RTSSim.

Visualization, analysis and comparison of traces from simulations or real system recordings is necessary during impact analysis, model validation and regression analysis. Note that trace comparison is discussed in Chapter 8. Outside the scope of simulation-based analysis, trace visualization is of direct relevance for industry, e.g., for troubleshooting, optimization and overall system understanding. A trace visualization tool, the Tracealyzer, has been developed during this work. This is today used in all ABB robots and the second version of this tool is commercialization together with Quadros Systems, Inc., under the name RTXCview.

Since trace recording is a key component of the envisioned analysis framework, the general applicability of trace recording is naturally of high importance and is targeted by research question **Q3**, presented in Section 1.3. This has two aspects, implementation feasibility on common real-time operating systems, and the overhead with respect to CPU and RAM usage. This has been investigated in five industry collaboration projects where trace recorders have been implemented.

Evaluations of overhead has been performed in four cases, representing the domains of industrial robotics, vehicular systems, telecom systems, and automated welding systems. The system used three different operating systems and four different CPUs, ranging from a 20 MHz, 16-bit micro controller to a

1 GHz, 32-bit CPU. The CPU overhead from the trace recorders implemented on these systems was found to be between 0.2 – 1.1 % for 32-bit CPUs and 3 % for the 16-bit MCU. The RAM buffer usage was found to be 2 – 72 KB per second of trace history, depending on event rate (0.5 – 18 KHz) and event size (4 – 8 bytes). All three evaluated operating systems allow for third-party trace recorders, in two cases through callbacks/hooks and in one case since the kernel source-code was available and thereby could be modified.

No evaluation of overheads has yet been performed in the fifth and most recent recorder project, which is still ongoing. In this project, a generic trace recorder has been implemented and integrated in the commercial real-time operating system RTXC Quadros [137]. This is included in order to present the latest recorder design, based on the author's experiences of the four earlier projects combined with the expertise of the RTOS developers at Quadros Systems, Inc. [137].

Recorder implementation was possible on all five cases, although the operating system OSE, from ENEA AB [134], caused some problems due to security mechanisms preventing direct access to kernel data. While context-switches (task-switches) could be logged with respect to task identity and timestamp (the "what" and "when"), it was not possible to log the "why", i.e., the status of the suspended task (ready, blocked or terminated). This is needed for correct trace visualization. A workaround was found, which however assumes static process priorities in order to guarantee correct display. Note that this problem does not concern recording of timing profiles, which is straightforward in all five cases.

These results confirm the feasibility of trace recording in the context of the overall approach, presented in Section 1.2. The answer to research question **Q3** is thereby "yes" — custom trace recording is generally feasible, with low overhead. Note that the presented overhead figures do not include the instrumentation necessary for recording of timing profiles, i.e., ETPs and IEPs as proposed in Section 7.5. Adding such probes would increase overhead a lot, since the ETPs may be frequent. The evaluated recorders have however not been optimized for CPU overhead, so it is likely that the higher event rate can be somewhat compensated by shorter probe execution times. Moreover, the additional profiling (ETPs and IEPs) would also give improved troubleshooting support and thereby at least partially motivate the additional overhead, since these events could be presented visually, in the Tracealyzer or similar tool, which would help pinpointing errors and performance bottlenecks.

Chapter 8

Model Validity, Validation and Trace Comparison

This chapter discusses the issue of model validity and thereafter presents a five step process for validation of simulation models, through comparison of simulation traces with traces from the modeled system. The first four steps of this process is actually a general method for trace comparison, which can be used for impact analysis and regression analysis as well. These uses are described on a conceptual level in Section 1.2. A literature study on model validation is presented in Section 2.6.

Since a model is, by definition, an abstraction of the modeled system, a model cannot be expected to exactly predict the behavior of a complex system in all situations. In context of the simulation analysis framework presented in this thesis, the abstractions correspond to the probabilistic execution time modeling (Section 7.5) and use of explicit, manual modeling abstractions, as discussed in Section 5.7. However, if all details of the modeled system software and hardware was to be taken into account, e.g., in order to model the execution times in an exact manner, the result would be a very detailed simulator like Virtutech Simics [124], which would be several magnitudes slower and therefore not suitable for this approach.

A valid model does not have to be "perfect", as discussed, but should give predictions that are "good enough", with respect to accuracy and confidence. A major problem is, how accurate and confident does a result need to be, in order to be good enough, i.e., valid? This question cannot be answer for the general case. The validity of a model is investigated in an activity known as *model*

validation. Schlesinger et al. [41] defines model validation as the "*substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model*". This definition relies on 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 intended use of the model predictions.
- **The intended application of the model** considered is, as discussed in Chapter 1, impact analysis with respect to run-time properties, during software maintenance.

Thus, a model cannot be shown valid in general, only for a specific use, in a specific context. The required accuracy (and confidence) depends to a large extent on the purpose of the analysis model. 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 a 20 % margin of error in the predictions, since the consequences of a minor prediction error regarding user-perceived performance is not critical and since it is easy to verify this after implementation. However, if the model is used to predict properties critical for correct system operation, such as extreme-values in response time, a much higher accuracy is required since the consequences of not detecting such an error might be a system failure and, moreover, since the extreme-case scenarios found through simulations might be hard to reproduce and test on the real system.

This chapter proposes a five-step process for validation of simulations models for task-level timing analysis of embedded software systems, which also can be used for impact analysis and regression analysis. The model validation process utilizes the trace recording techniques and the Tracealyzer tool, presented in Chapter 7, and the simulation framework RTSSim, presented in Chapter 3. Section 8.1 provides a discussion of the potential threats against the validity of a model. Section 8.2 presents the proposed validation process, consisting of five tests of the model. Section 8.3 discuss the selection of comparison properties, a necessary and crucial step in this process. Section 8.4 gives an introduction to the two-sample Kolmogorov-Smirnoff test, a statistical test which is useful in several steps of this process. Section 8.5 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 8.6 concludes the chapter.

8.1 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 proposed analysis framework consists of several activities and tools and errors could be introduced in any of them. There are at least five potential error sources:

- Manual modeling abstractions,
- Execution-time modeling,
- Model extraction configuration,
- The probe effect, and
- Side-effects of black-box software.

Manual Modeling Abstractions The envisioned solution for automated model extraction includes support for manually specified *modeling abstractions*, where selected condition expressions are replaced with constants, or modeled in a probabilistic manner. It is believed a small amount of carefully selected modeling abstractions in many cases can reduce the model size significantly and thereby shorten the time required per simulation. It is however important that manually specified abstractions are valid with respect to the purpose of the model. For instance, if the purpose of the model is typical performance, conditions for error checking can probably be removed from the model, but if the purpose is extreme value analysis such abstractions might not be valid.

Execution-time Modeling When recording execution time data for the timing profile, as discussed in Section 7.5, it is important to get sufficient coverage. Insufficient coverage can however be detected during the simulation. A more serious problems is if there are hidden dependencies in the execution-time data, e.g., due to hardware state, which may cause the simulator to generate scenarios which are not feasible in practice, as discussed in Section 7.5.

Model Extraction Configuration The presented method for automated model extraction requires as input the set of *model focus functions*, i.e., the API functions of relevance for the run-time properties in focus of the model. It is important that this list is complete, since the model will otherwise fail to include relevant behavior, but including too many (irrelevant) model focus functions will make the simulation model larger and more complex than necessary, so it is important to select the model focus functions carefully. Another input of importance is to specify the right source code directories and preprocessor directives. In a large system, this is not always obvious as the build environment is complex and often heterogenous. Some files might be generated in compile-time. There are commercial tools which solve this by monitoring the build process in order to record what files and preprocessor directives that are used. This method is used by both CodeSurfer [123] and Coverity [142].

The Probe Effect If the code instrumentation used for constructing the timing profile is not permanent but added on demand, the behavior of the modeled system might not be the same as the behavior of the production version. The impact of adding or removing code instrumentation is commonly referred to as the probe effect [89]. In this thesis it is assumed that the probe effect can be avoided by allowing the probes to remain in the system. However, this might not be possible for some systems due to the cost of these probes, i.e., the additional CPU and memory usage. Another solution to avoid the probe effect is to use specialized hardware monitors that non-intrusively observe the system without affecting the temporal behavior of the system [109]. This is however not always an option, since custom hardware is required.

Side-Effects of Black-box Software Large software systems often contain third party software, e.g., databases, drivers, libraries, etc., for which the source code is not available. The approach to automated model extraction presented this thesis generally requires that the source code is available, but provides a mechanism for allowing black-box library functions. It is however assumed that most code is available, and that any such black-box library functions only perform simple operations with no side-effects of relevance for the simulation model.

However, larger black-box software components, like an SQL database, are likely to have such side-effects, e.g., spawning tasks or locking a semaphore. In order to model black-box software with such behaviors there are approaches based on dynamic analysis (trace recording), such as the works by Huselius [100]

and Jensen [49, 116]. A literature study on modeling methods using dynamic analysis is presented in Section 2.5. Such methods could be used as a complement to model extraction from source code.

8.2 A Process for Trace Comparison

Model validation is in the context of this thesis a matter of comparing simulation results with real world observations of modeled system, i.e., a matter of trace comparison. The term *trace data set* is used as a common label for all information contained in (or derived from) a trace recording. Comparison of trace data sets is necessary in all three scenarios in the vision presented in Section 1.2, i.e., model validation, impact analysis and regression analysis. In all cases, the issue in focus is whether or not there are significant differences between two trace data sets with respect to relevant aspects compared. The three uses for trace comparison are illustrated by Figure 8.1.

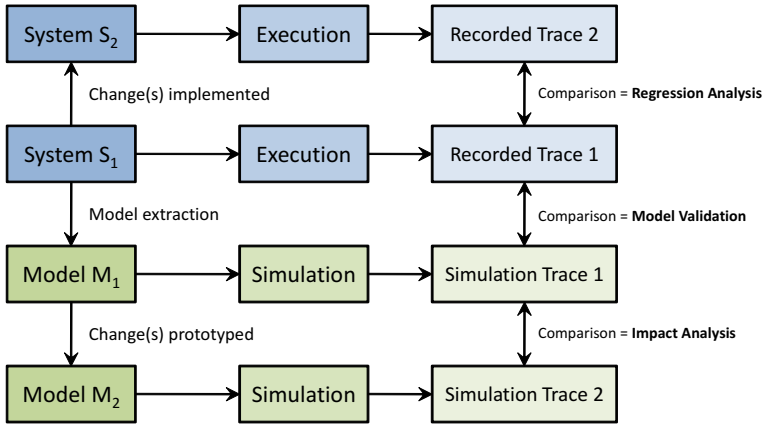

Figure 8.1: The uses for trace data comparison

This section presents a five-step process for finding differences between trace data sets. The process is presented in perspective of model validation, but the first four steps can also be used for impact analysis or regression analysis. The fifth step, sensitivity analysis, is however only for model validation.

The five steps are increasingly harder test of similarity between two trace data sets, one from Monte Carlo simulation of the model and one from moni-

toring during real system operation. Each test either fails the model (i.e., not valid), or allows the model to pass to the next test in the comparison process. The reason for starting with less accurate tests is that they are faster than the more accurate tests, but are still likely to detect any major errors in the model. The more accurate but time-consuming tests are only applied when the model has passed the previous, less accurate tests. Some of the tests have been previously proposed in research literature, by e.g., Law and McComas [60] and Sargent [62], but not in the context of validating simulation models of embedded software systems.

It is important to remember that this process cannot prove the validity of a model. This is not possible, in the same way as it is not possible to prove the absence of software errors using testing; there is usually an astronomic number of possible scenarios, too many to allow for a systematic, exhaustive comparison of each one. Model validation is therefore a matter of attempting to show that the model is incorrect, i.e., that there are significant differences between two trace data sets. The more tests performed that fail to show that the model is incorrect, the more confidence in the model.

Next follows an overview of the comparison process. Note that the first half of the process uses subjective methods, based on visualizations, while the later tests are based on statistical methods. Steps 1 – 4 are presented in detail in sections 8.2.1 – 8.2.4, respectively, while Step 5 is presented separately in Section 8.5.

1. **Subjective Trace Comparison** Task execution trace are visualized and compared subjectively, e.g., using the Tracealyzer tool presented in Section 7.3. The purpose of this first test is to quickly determine if there are major, obvious differences.

2. **Subjective Property Comparison** Specific properties of the trace data sets are selected (the *comparison properties*), visualized and compared subjectively. This test is more detailed than the trace visualization test, but still subjective.

3. **Variability Analysis** Several trace data sets from the same source are compared in order to study the amount of variability in the comparison properties. If trace data sets from the same origin show large variations, larger trace data sets are necessary. The variability should be investigated by using statistical methods; the two-sample Kolmogorov-Smirnoff test is suggested.

4. **Statistical Property Comparison** In this test, statistical methods are applied in order to determine the similarity between the trace data sets, with respect to the selected comparison properties. Like in the variability analysis, the two-sample Kolmogorov-Smirnoff test is recommended for statistical comparison.

5. **Sensitivity Analysis** This test is rather time consuming and is only recommended for initial model validation, and for verification of the overall simulation framework. This test checks if the simulation model is *robust* with respect to typical changes, meaning that the predictions from the model should correspond to the actual outcome when implementing the change in the real system. Model robustness and sensitivity analysis is discussed in a Section 8.5.

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 at least what test-cases that are used to stimulate the system (i.e., generate input), the hardware platform used (what timing profile to use for the simulations) and the software configuration.

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 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 number 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 extreme cases scenarios with respect to system stimuli, but it is also important to use validation environments corresponding to the normal use of the system.

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 investigate the cause of the discrepancy.

8.2.1 Step 1: Subjective Trace Comparison

In the first step in the process, traces are visualized using a suitable tool (e.g., the Tracealyzer, presented in Section 7.3), and subjectively compared. The purpose of this test is to quickly spot any major differences only.

When comparing the traces, it is important to note that the traces are samples of a very large set of possible behaviors due to seemingly random variations in execution-times and input event timing. Even though the validation environment has been specified, the model is still an abstraction of the real system, modeled in a probabilistic manner with respect to execution times and input events. Hence, an exact match cannot 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 fail the test.

An example is depicted in Figure 8.2, 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 *Ctrl* task, but in the model this is not the case. As depicted by Figure 8.2, the *Drive* task has a matching inter-arrival time (periodicity) and execution time, but it has a different offset with respect to the *Ctrl* task and the preemption pattern is therefore different. This is an example of a pattern which may also be used as a comparison property in the property visualization test, discussed in Section 8.2.2.

8.2.2 Step 2: Subjective Property Comparison

In the second test, specific properties of the real system behavior and the corresponding model simulation are visualized and compared subjectively. This test has been discussed by Sargent [62], where it was referred to as the *operational graphics* test. This test is stronger than the trace comparison (Step 1), as it apart from a set of validation environments also requires selecting a set of concrete properties to compare, the *comparison properties*.

The selection 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 8.3), and utilization of logical resources over time (Figure 8.4). Such properties are sensitive to a large set of possible differences between the model and the real system. The

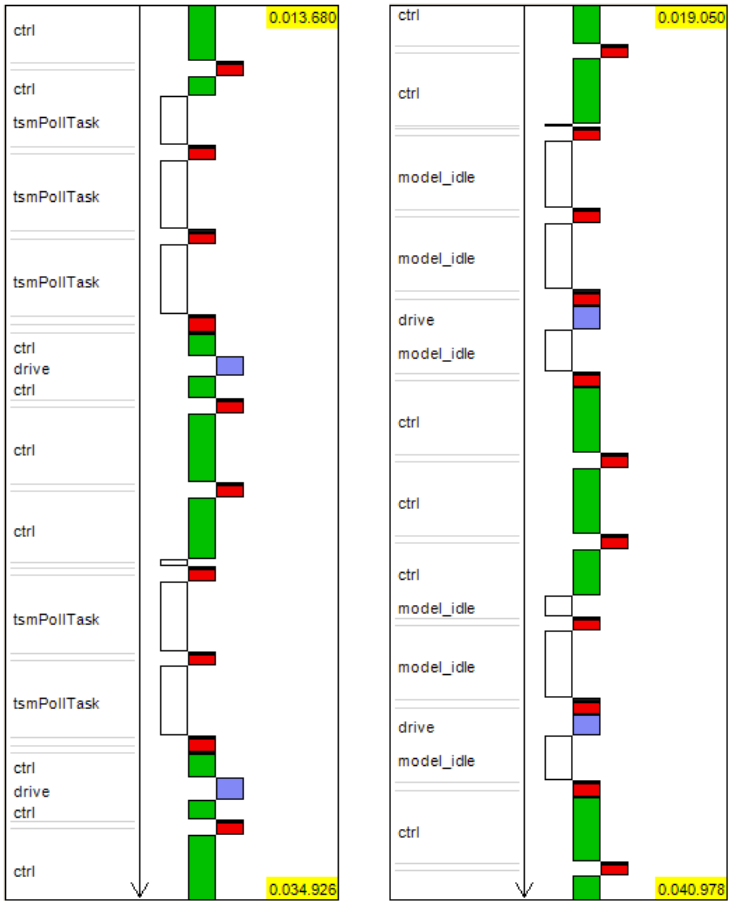

Figure 8.2: Trace comparison using the Tracealyzer tool

selection of comparison properties is discussed in Section 8.3.

The comparison properties may, e.g., 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 task instance or the utilization of a specific resource. Since execution traces should already be available from the previous step, the main effort of this test is the actual comparisons of property visualizations. 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, each data set will generate 100 visualizations to be compared. However, even if each comparison takes on average 1 minute, this takes less than 2 hours for a single person to perform.

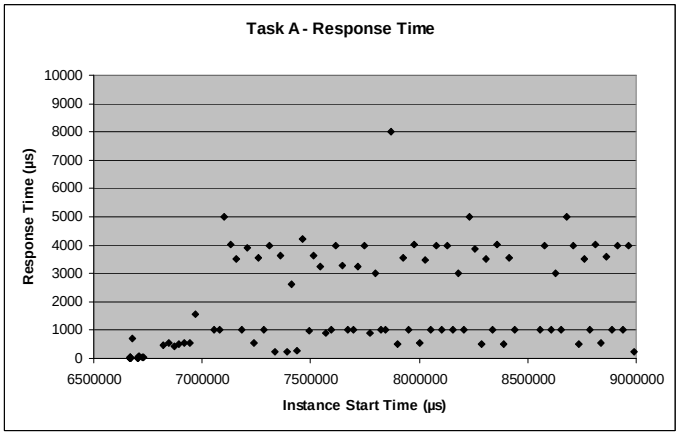

Figure 8.3: Visualization of the usage of a task response time

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 visualization is a fairly quick method of identifying the significant errors in specific properties, at an early stage in the validation process prior to more time-consuming tests.

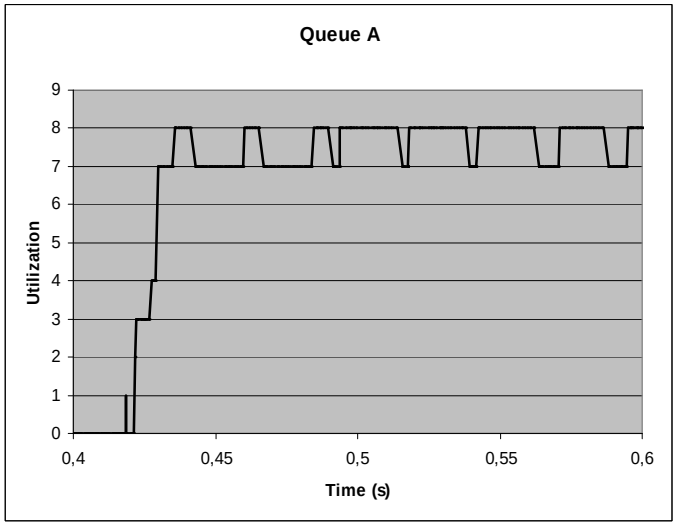

Figure 8.4: Visualization of the usage of a logical resource

8.2.3 Step 3: Variability Analysis

The third step in the validation process, the variability analysis, is important since seemingly random variations in execution time and input event timing will cause trace data sets from the same source to show variations between replications of the data sets. The output from Monte Carlo simulation will show variations due to the use of probabilistic modeling, while real system measurements will show variations due to variations in hardware state, such as cache memories, or due to variations in the timing of external input events. If the amount of variability in the analysis results is large, this implies that the predictions are based on an inadequate number of observations (simulation or real system recordings) and might therefore not be representative for the system behavior in general. The simulation results are not incorrect in the sense that the behavior predicted by the model may occur in the real system, but the results are of low confidence. This applies mainly to prediction of average-case behavior, using Monte Carlo simulation.

However, small variations between data sets from the same source are expected, and normal. Comparing data sets therefore requires a method where smaller variations can be tolerated, while larger (statically significant) differ-

ences are clearly identified. There are several established statistical methods for this purpose, i.e., to for determining if there are significant differences between two sets of data. The two-sample Kolmogorov-Smirnoff test (cf. Section 8.4 is recommended for this purpose, since it is non-parametric and makes no assumptions on the underlying distribution of the data. Another method is the one proposed by Huselius [100], a former department colleague of the author. Huselius dismissed the KS test for this purpose under the assumption that one of the data-sets compared needs to be modeled as a mathematical distribution. This is however not true for the two-sample KS test, which is discussed in Section 8.4.

8.2.4 Step 4: Statistical Property Comparison

In Step 2 of this process, comparison properties were selected, visualized and compared in a subjective manner, which can quickly show obvious differences. However, in order to test the model validity in more detailed, accurate and objective way, a statistical comparison is necessary.

However, it does not makes sense to compare the trace data sets directly. As an example consider Figure 8.5, which depicts the predicted and real response times of a task. Each data point represents the response time of a task instance. The data-points are plotted in chronological order according to the start times of the task instances.

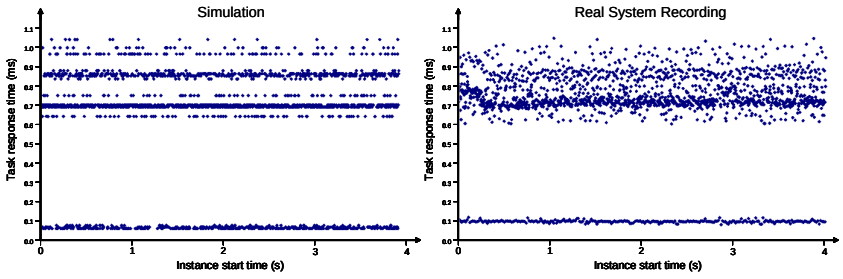

Figure 8.5: Response-time distribution – simulation vs. real system

The temporal behavior predicted by the model clearly resembles the response-time distribution from the real system measurements. 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 instance by instance, since the x^{th} instance of task T in a simulation trace

will probably not match the x^{th} instance of task T in a real system trace, even if the traces were synchronized, due to the probabilistic execution time modeling.

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. Like in the variability analysis (Step 3), the comparison properties should here be compared using the KS test (cf. Section 8.4), but instead of comparing data sets from the same source (i.e., two simulation results or two real system recordings), the data sets are in this case of different origin.

The result of this step shows if there are any significant differences between the two compared data sets with respect to the comparison properties. This is conceptually similar to a comparison of two physical objects from photographs taken from different perspectives; a photo of a cylinder-shaped object may appear very similar to a photo of a spherical object in a certain perspective, but not in others. It is therefore important to include a sufficient amount of comparison properties (perspectives), in order to detect any differences that exist.

8.3 Selecting Comparison Properties

Statistical property comparison naturally requires a selection of the properties to compare. If a sufficient number of comparison properties have been used and the comparison has been made with low tolerance, any model that passes this test should be highly accurate.

In the same way as when defining test cases for normal software testing, it is crucial to select the right test cases. As many comparison properties (test cases) as possible should be used, but at the same time it is only possible to use a limited amount for practical reasons.

The comparison properties typically include explicit system requirements and other system properties of high interest, but may also include system properties that are of less interest for the actual used of the model in order to increase the coverage of the comparison. Such extra properties are labeled *supporting properties*. They are affected by many aspects of the system and characterize the temporal behavior. Typical supporting properties are statistics on task inter-arrival times and utilization of logical resources, such as message queues.

As mentioned, as many relevant system properties as practically possible should be included in the set of comparison properties. However, the use of irrelevant comparison properties may result in the rejection of a valid model.

Sargent [62] denotes this 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. Sargent [62] denotes this 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, if they represent too few types of system properties, For instance, consider a case where only response-time properties are used as comparison properties. This would not detect if the inter-arrival time of a task is (slightly) different, but would have if the comparison properties included preemption patterns. Thus, the selected system properties should not only be relevant, but also represent a variety of aspects of the temporal behavior. Three general types of run-time properties have been identified as 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. An example is how often a certain task, Task A, is preempted by another specific task, 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).

8.4 The Two-Sample Kolmogorov-Smirnoff Test

The two-sample Kolmogorov-Smirnoff test [38], or KS test, is recommended for the statistical comparison of trace data in the model validation process proposed. A good overview of this test can be found at the U.S. NIST website [125]. The KS test is non-parametric and makes no assumptions on the underlying distribution of the data, which is important since response-times and execution-times are often not normal distributed but rather has a complex, multi-modal probability density distributions, as illustrated by Figure 8.5.

The KS test assumes that the data is of continuous nature. Even though time is discrete in RTSSim simulations, since modeled by an integer counter, the underlying concept of time is continuous and the execution- and response times are typically measured in hundreds or thousands of time-units. Such distributions can be regarded as continuous.

The KS test is based on the cumulative distribution function of the data set, which gives the ratio of data elements smaller than the specified element. For a uniform distribution, the cumulative distribution function is a linear function from $(0, 0)$ to $(1, 1)$, while for a data set containing only identical values, the cumulative distribution function will be a step function, i.e., the function value is zero (0) before the value and one (1) afterwards. These are the two extremes, but most cumulative distribution are somewhere in between, like in the example illustration provided in Figure 8.6, where the same data set is presented using a histogram (the left diagram) and the cumulative distribution function (the right diagram). The presented data distribution is clustered into intervals of 100 time units.

The KS-test is an hypothesis test, where the null-hypothesis is that the two data sets are identical. The KS test is performed by calculating a statistic measure which describes the maximum difference between the cumulative distribution functions of the two data sets, as illustrated by Figure 8.7. The null-hypothesis is rejected (i.e., a significant difference is present) if

$$\sqrt{\frac{n_1 * n_2}{n_1 + n_2}} * D > K_\alpha$$

where n_1 and n_2 is the number of elements in the two data sets. The D statistic is, as illustrated by Figure 8.7, the maximum distance between the cumulative distribution functions. This is calculated by

$$D = sup|F_n(x) - F_{n'}(x)|$$

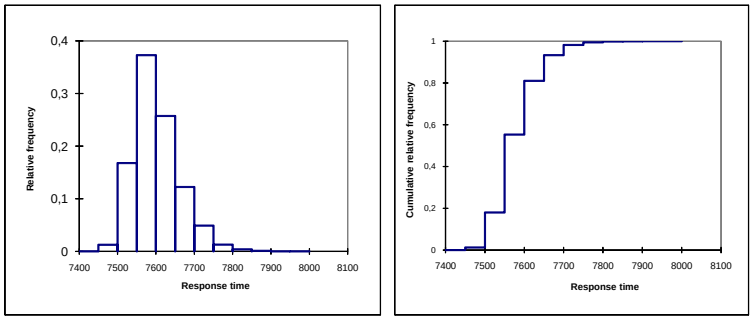

Figure 8.6: Probability density (left) and cumulative (right) distribution (clustered data)

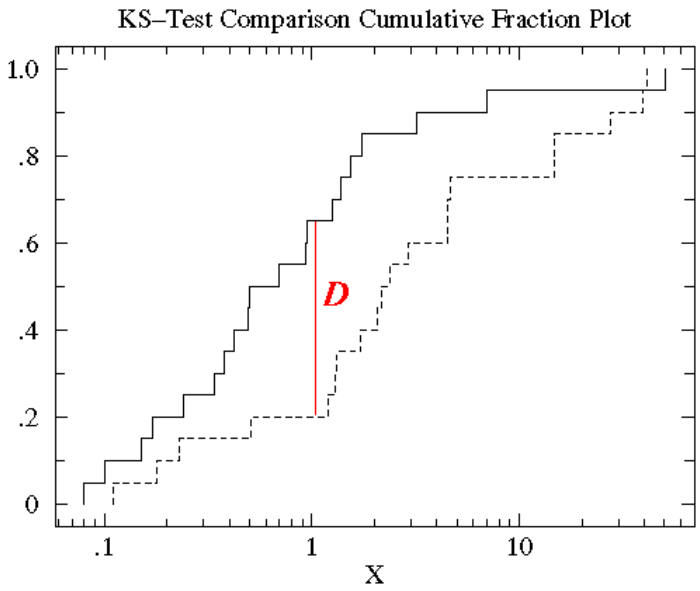

Figure 8.7: The Kolmogorov-Smirnoff statistic (D)

where $F_n(x)$ is the cumulative distribution function for data set with n elements and sup denotes the supernum of the resulting set, in practice the largest difference between the two cumulative distribution functions. K_α is given by the Kolmogorov distribution (K), such that

$$Pr(K \leq K_\alpha) = 1 - \alpha$$

where α is the desired level of significance (typically 0.95).

8.5 Model Robustness and Sensitivity Analysis

A simulation model is *robust* with respect to a certain type of changes if such changes, when applied to the simulation model, impacts the simulation result in the same way as the corresponding change impacts the modeled system. This section presents a method for determining the robustness of a behavior model of a complex embedded system. This activity is referred to as *sensitivity analysis*. This is the fifth and final step in the proposed process for model validation.

To demonstrate the importance of model robustness, consider a system containing a binary semaphore protecting a shared resource. A timeout occurs if a task has been waiting for the semaphore for a certain predefined time. If the timeout occurs, the task is activated and executes longer than normal due to error handling. In all previous versions of the system, this timeout has never occurred. If the timeout is left out when modeling the system (e.g., due to a manually specified modeling abstraction, or due a bug in the model extraction tool) the model will still seem accurate since the timeout never occurs. However, if a change to the system (e.g., a new feature) causes the timeout to occur in some situation, the simulation model will no longer be valid since it does not include this mechanism.

This approach to sensitivity analysis is influenced by *system identification*, a technique used in the domain of control theory [110]. 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, since it is possible to interpolate the behavior between the tested signals. However, computers are not continuous systems, they are discrete systems where the behavior may change dramatically as a result of small changes. 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.

The robustness of a model can be assessed through a *sensitivity analysis*. The basic idea is to perform impact analysis with respect to common types of changes and verify that they impact the behavior predicted by the model in the same way as they impact the 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 introduce a new task,
- to change priority or rate of an existing task,
- to modify existing functionality of a task and thereby change its execution time distribution,
- to add new dependencies between existing tasks, e.g., through new uses of semaphores or interprocess communication.

The selection of change scenarios requires experienced engineers that can describe typical types of changes to the system. 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 using the first four steps of the comparison process presented in this chapter. If no discrepancies can be found, the model is considered robust with respect to the change scenarios. 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 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 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 differences in this comparison, assuming that the model was sufficiently accurate prior to the sensitivity analysis. The general sensitivity analysis process is illustrated by Figure 8.8. This process is performed for each validation environment.

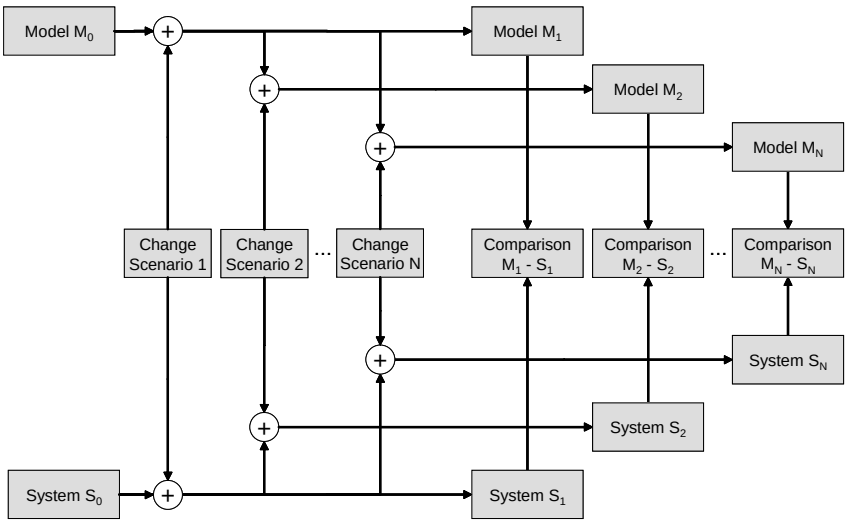

Figure 8.8: The sensitivity analysis

A sensitivity analysis can be regarded as an 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 "*increase 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 is updated. It is sufficient if a sensitivity analysis is performed on the initial model of system, after major changes of the system architecture or simulator framework, or if new change scenarios are identified.

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 $e \times c \times p$. The time consuming part of this analysis is to run the real-system measurements in order to obtain reference data sets for comparison. Complex embedded systems often takes considerable time to compile and start up. A full build may require hours, and even the smallest change, a single module, often gives a 20 minute cycle time for building, rebooting, running the test and finally collecting the data. Since $e \times c$ recordings are necessary, this can take a considerable amount of time. For instance, if three validation environments and five change scenarios have been defined, recording the execution traces takes at least 5 working hours.

8.6 Conclusions

This has proposed a process for comparison of trace data sets, from simulations or real system measurements, which can be used for model validation, but also for impact analysis and regression analysis. The proposed approach consists of a five-step process of increasingly demanding tests of trace data set similarity. The first four tests are can be used both for model validation and for impact analysis. The fifth test, the sensitivity analysis, is however strictly a test for use in model validation. Since this test is quite time consuming, it is not realistic (or necessary) to perform after every change of the model. It is sufficient if a sensitivity analysis is performed on the initial model of system, after major changes of the system architecture or simulator framework, or if new change scenarios are identified.

Some of these 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 should be suitable for validation of RTSSim simulation models. Evaluation of these methods for model validation however requires simulation models describing real (and sufficiently complex) software systems. However, since the MXTC tool is not yet able to generate such models, as discussed in Chapter 6, this evaluation has not yet been possible.

Chapter 9

Conclusions and Future Work

This thesis has proposed a framework for simulation-based timing analysis targeting complex embedded software systems. The motivation behind this work is large industrial systems with requirements on timing and/or performance, but where timing analysis has not been taken into account in the system design and evolution. As a result, these systems violate the assumptions of analytical methods for response time analysis. Using formal analysis methods such as model checking is typically not an option in the context of large industrial systems, since such methods does not scale sufficiently and often require expertise in formal modeling.

Simulation can be applied on virtually any system since the method does not impose any assumptions, at least not on a conceptual level. The RTSSim simulator framework is however limited to analysis of systems using a traditional single-core CPU. Simulation does not require manual modeling, since the simulation model can be generated automatically from the source code. Simulation can be used to study virtually any property of the run-time behavior. The drawback of simulation is the lower confidence compared to model checking or analytical methods for response time analysis; it is a best-effort analysis and can only show the presence of errors, not prove their absence. This is however still very valuable for industrial systems which today have insufficient means for timing analysis, i.e., systems where analytical or formal analysis methods are difficult to apply. On such systems, developers typically rely on system-level testing for finding timing problems. This is inefficient

since timing problems often only manifests in rare situations, dependant on event timing, which are difficult to identify and reproduce. Introducing the proposed analysis framework for such systems can improve quality assurance by enabling predictions of potential timing errors in early phases. This analysis support can help system designers avoid choosing unsuitable designs which otherwise might cause major additional costs and project delays.

This thesis has presented scientific contributions in three areas related to the proposed analysis framework:

- The simulation framework RTSSim and two techniques for simulation optimization: MABERA and HCRR.
- Automated extraction of simulation models based on a new approach to program slicing, which unlike existing methods scales to large software systems.
- Efficient trace recording techniques for embedded systems, for use in impact analysis, model validation and task profiling during model extraction.

These contributions include methods, implementations as well as evaluations of key components of the envisioned analysis framework. Additional framework components have also been presented in this thesis, but not claimed as scientific contributions: the trace visualization tool Tracealyzer, and a not yet evaluated process for comparison of trace data, proposed for use in impact analysis and model validation.

The Tracealyzer tool served as a "low hanging fruit" for industry collaboration, which allowed the author to perform five industry collaboration projects where trace recording was developed for different platforms. Three of these projects lead to industrial use of the Tracealyzer. In one case, at ABB Robotics, the recorder was integrated in their system permanently. Every ABB robot delivered since 2005 is monitored by a Tracealyzer recorder, at all times. This approach eliminates the "probe effect" issue and the availability of the trace facilitates troubleshooting greatly. The Tracealyzer is today a product of Percepio AB.

The two approaches to simulation optimization were evaluated on simulation models describing fictive but realistic systems, with similar analysis challenges as observed in industrial systems studied. The HCRR method gave promising results. It found 4 – 11 % higher response times than the previously proposed MABERA method (which in turn found higher results than Monte

Carlo simulation) and reached the final result 13 – 112 times faster than the time required for the MABERA results.

The scalability of HCRR on real models of industrial systems is however yet unknown, since the automated model extraction is still not fully operational. However, unlike model checking, simulation does not require an exhaustive search of the model state space and is therefore always applicable, although more complex models (with larger state space) gives less likelihood of finding extreme cases close to (or equal to) the worst case scenario. It is however not fair to compare this approach with formal verification methods, which can prove properties of a model. This approach should rather be regarded as a specialized type of testing, for problems related to timing and resource usage.

Future work in this area includes a simulator supporting multi-core processors and distributed systems, "smarter" heuristics for simulation optimization (as discussed in Section 4.7), as well as an evaluation of the approach to execution time modeling presented in Section 7.5.

The solution for automated model extraction has been evaluated on industrial code from ABB Robotics, with respect to model size and analysis runtime. The industrial code used is rather small, only 6 000 lines of code, but it is real industrial code of high complexity. The model extraction results were satisfying, although improvements are possible. The size of the resulting simulation models (the number of model-relevant executable statements) ranged from 3 – 59 % of the total amount of executable statements but the total runtime was only 3 minutes. It is believed possible to shorten the runtime significantly, perhaps with a factor 10, by porting the model extraction tool from Perl to C/C++. The resulting simulation models are relatively large. In the best case observed, the model extraction removed 97 % of the executable statements. However, even if these number were true for the system as a whole, 3 % of the original code is still an overwhelming amount of code for large industrial systems. For the ABB Robotics system, this corresponds to about 90 000 lines of code. In early work, we believed that the simulation models could serve as architecture documentation, but a simulation model of that size has little value as documentation or for program comprehension purposes. This type of model extraction is therefore mainly useful as a means for speeding up the simulations by reducing their size. However, as presented in Chapter 5, three possibilities have been found for reducing the model size further: (1) enabling control-flow sensitive model extraction, (2) eliminating unused task outputs, and (3) allowing for manual modeling abstractions. Implementing and evaluating these are important parts of future work.

Chapter 1 presented three research questions, which have been answered by the five formal contributions of this thesis:

- **Q1:** *Can simulation models be extracted automatically from C source code, with sufficient efficiency and accuracy for scaling to complex embedded systems?*
  Answer: Yes, using the Katana approach (contribution **C1**), which according to an evaluation on industrial code (contribution **C2**) is high scalable and sufficiently accurate.

- **Q2:** *Is simulation optimization an efficient approach for predicting extreme cases in the temporal behavior of complex embedded systems, compared to existing methods for timing analysis?*
  Answer: Yes. Two methods for this purpose, MABERA and HCRR, has been developed, evaluated and found to be significantly more efficient than traditional Monte Carlo simulation (contributions **C3** and **C4**). HCRR however found 4 – 11 % higher response times and reached the end result 13 to 112 times faster than MABERA. When comparing to traditional analytical methods (on simple system models, analyzable using such methods), HCRR found the theoretical worst case response time for a particular task every time, MABERA found it sometimes, while it was never found using Monte Carlo simulation.

- **Q3:** *Is software trace recording generally applicable on common commercial operating systems for embedded systems, for platform users (product developers), with respect to implementation feasibility, and run-time overhead?*
  Answer: Yes. This is based on experiences from five industry collaboration projects where such such monitoring support were developed (contribution **C5**). CPU and memory overhead is very small on 32-bit computer systems, negligible in practice, and acceptably low also for resource constrained 16-bit computer systems. Implementation was straight-forward in four out of five cases, and a sufficient functionality could be achieved also in the last case.

To enable real use of the proposed analysis framework, a few framework components are however missing. The below "to do" list assumes a strict focus on impact analysis; the other two types of analysis proposed in Section 1.2 (explorative analysis and regression analysis) require additional results and have not yet been investigated.

- Three minor issues remain in the source code model extraction, as presented in Section 6.1.
  - Supporting function pointers.
  - Detecting unsupported pointer arithmetics.
  - Handling references to irrelevant symbols in model statements.
- The approach to execution time modeling, presented in Section 7.5, would need to be implemented.
- The proposed approach to model validation (Section 8.3) needs to be evaluated on reference cases before it can be used to verify the models produced using automated model extraction.

When these issues have been solved, the next step is a larger industrial evaluation of the integrated framework, evaluating simulation optimization methods on models from automated model extraction on industrial systems. This allows for verifying the simulation predictions with respect to traces recorded from the real system after the change has been applied.

Taking this a step further, an interesting but very demanding study would be to perform an "impact analysis of impact analysis", i.e., to study the impact of using this analysis framework on maintenance costs and software quality. This requires that the proposed analysis framework can be deployed for systematic industrial use; an empirical study on the economic impact of using the approach would then be possible, after some time, by studying the number of errors of the targeted types (timing-related errors) which have been discovered in late testing or post-release compared to before using the analysis framework. This study is however far from a trivial, to say the least, since it is real-world research with many influencing factors.

Finally, note that the Katana method presented in Chapter 5 is subject for a U.S. patent application – patent pending.

Appendix A

The Katana Algorithm

This appendix only serves to provide a more compact presentation of the Katana algorithm, presented in Section 5.3. The algorithm description relies on a set of supporting functions, presented in Section 5.4.

Formally, the Katana algorithm can be described as a function, *Katana*, which takes as input a set of model focus functions, and returns a set containing the relevant statements (i.e., the program slice). This main function depends on a set of functions with recursive dependencies, as depicted by Figure A.1.

Figure A.1 illustrates the relations between the Katana functions. In this graph, nodes correspond to the functions presented later in this section, and edges to call-by relations, i.e., the propagation and accumulation of analysis results. The edge from *FunctionSlice* to *Katana* means that *FunctionSlice* return results to *Katana*, and thereby implies that *Katana* calls *FunctionSlice*. Note that this description does not include all aspects of Katana. In order to simplify the conceptual understanding several details have been omitted, for instance the analysis cache, the handling of symbol reference filters and details regarding detection and considerations of LMR and GMR functions. These aspects are however described in Chapter 5.

Especially note the functions *OnEach* and *OnEach2*, which are commonly used in the later algorithm description. These are not visible in the illustration (Figure A.1) in order to make the algorithm illustration more readable. However, most edges in the illustration correspond to an OnEach operation.

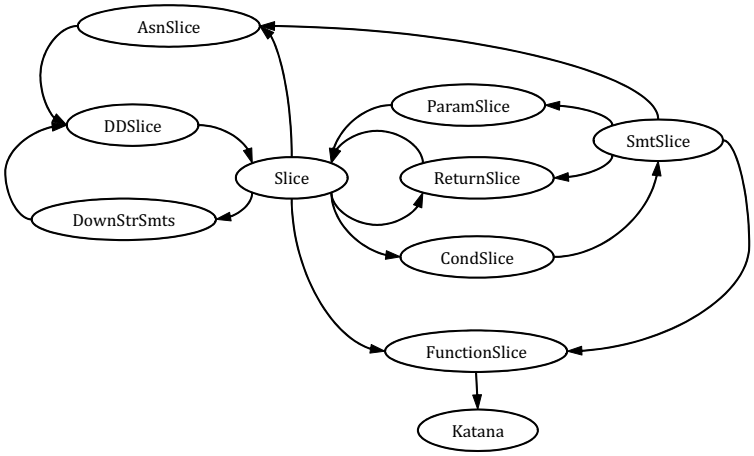

Figure A.1: The Katana algorithm illustrated

$$Katana(MFFs) = OnEach(FunctionSlice, MFFs)$$

$$FunctionSlice(Sym) = \\ OnEach(Slice, AllCallArgs(Sym)) \cup \\ OnEach(SmtSlice, AllCallers(Sym))$$

$$Slice(Sym) = \\ \begin{cases} DDSlice(Sym) & \text{if } \neg IsFunc(Sym) \wedge \\ & \neg IsParam(Sym) \\ DDSlice(Sym) \cup OnEach2(ParamSlice, & \\ CallerSmts(DefFunc(Sym)), Sym) & \text{if } IsParam(Sym) \\ DDSlice(Sym) \cup & \\ OnEach(ReturnSlice, ReturnSmts(Sym)) & \text{if } IsFunc(Sym) \end{cases}$$

$$DDSlice(Sym) = \\ OnEach(AsnSlice, AsnSmts(Sym)) \cup \\ OnEach(DownStrSmts, PtrUseSmts(Sym))$$

$$ParamSlice(Smt, Sym) = SmtSlice(Smt) \cup Slice(ArgOfParam(Smt, Sym))$$

$$SmtSlice(Smt) = \{Smt\} \cup OnEach(CondSlice, CondSmts(Smt))$$

$$CondSlice(Smt) = \\ \begin{cases} \{Smt\} \cup OnEach(Slice, Symbols(Smt)) & \text{if } \mid CondSmts(Smt) \mid > 0 \\ \{Smt\} & \text{if } \mid CondSmts(Smt) \mid = 0 \end{cases}$$

$$AsnSlice(Smt, Sym) = \\ \begin{cases} SmtSlice(Smt) \cup \\ OnEach(Slice, Symbols(Smt)) \cup \\ OnEach(SmtSlice, AllCallers(Smt)), & \text{if } IsGlobal(Sym) \\ SmtSlice(Smt) \cup \\ OnEach(Slice, Symbols(Smt)), & \text{if } \neg IsGlobal(Sym) \end{cases}$$

$$DownStrSmts(Smt, Sym) = \\ \begin{cases} OnEach(Slice, AsnTargets(Smt, Sym)), & \text{if } \neg IsCallArg(Smt, Sym) \vee \\ & \neg IsReturned(Smt, Sym) \\ OnEach(Slice, AsnTargets(Smt, Sym)) \cup \\ Slice(ParamOfArg(Smt, Sym)), & \text{if } IsCallArg(Smt, Sym) \\ OnEach(Slice, AsnTargets(Smt, Sym)) \cup \\ Slice(ContextFunc(Smt)), & \text{if } IsReturned(Smt, Sym) \end{cases}$$

$$\mathit{ReturnSlice}(\mathit{Smt}) \quad = \quad \mathit{SmtSlice}(\mathit{Smt}) \ \cup \ \mathit{OnEach}(\mathit{Slice}, \mathit{Symbols}(\mathit{Smt}))$$

$$\mathit{AllCallers}(F) \ = \ \mathit{CallerSmts}(F) \ \cup \ \mathit{OnEach}(\mathit{AllCallers}, \mathit{CallerSmts}(F))$$

$$OnEach(F, S) = \bigcup_{x \in S} F(x)$$

$$OnEach2(F, S, a) = \bigcup_{x \in S} F(x, a)$$

Appendix B

The RTSSim API

The RTSSim simulation framework, presented in Chapter 3, provides a "sandbox" environment with the core services and run-time mechanisms of most common real-time operating systems. It does not target any particular operating system, but the design is somewhat influenced by Wind River VxWorks [132], used at ABB Robotics, although the APIs and provided services are not identical.

The simulation framework expects the simulation model to contain a function named *model_init*, where tasks, mailboxes and semaphores are expected to be created. This function is called by RTSSim before the simulation starts. Tasks, mailboxes and semaphores may also be created dynamically, during the simulation, by calling the corresponding API functions from the task models. When creating tasks dynamically, the offset should be set to a point in the future, i.e., a value larger than *clk*.

The API of RTSSim contains the following functions:

- *CreateTask(name, priority, period, offset, jitter, func)*
  Creates a task with the specified task attributes and entry function. The scheduling attributes are described in greater depth in Section 3.2.4.

- *CreateMailbox(name, size)*
  Creates a mailbox, for communication between tasks. The messages are stored in a fixed size FIFO buffer, and the size parameter specifies the maximum number of buffered messages. The return value is a pointer to the created mailbox object.

- *CreateSemaphore(name)*
  Creates a binary semaphore, i.e., a mutex, which initially is unlocked. The return value is a pointer to the created semaphore object.

- *CreateUEChannel(name)*
  Creates a named user-event channel, i.e., a label for a particular type of user events. The return value is a user-event channel identifier.

- *SendMessage(mailbox, msg, timeout)*
  Attempts to send a message to the specified mailbox. If the mailbox is full, the SendMessage operation will block the task until the operation can complete or until the specified timeout expires. The return value is 0 on success, otherwise negative.

- *RecvMessage(mailbox, buf, timeout)*
  Reads the oldest message from a mailbox, or if the mailbox is empty, blocks the calling task until a message exists or a timeout occurs. If a message was successfully received, it is written to the specified buffer. The return value is 0 on success, otherwise negative.

- *SemWait(semaphore, timeout)*
  Attempts to lock the specified semaphore. If it already is locked, the calling task is blocked until the semaphore successfully locked by the calling task, or until a timeout occurs. The return value is 0 on success, otherwise negative.

- *SemPost(semaphore)*
  Releases the specified semaphore, if locked. No return value.

- *Delay(duration)*
  Suspends the calling task for the specified duration. No return value.

- *Execute(duration)*
  Consumes the specified amount of CPU time. No return value.

- *UserEvent(UEChannel)*
  Stores a time-stamped user event on specified user event channel. No return value.

- *UserEvent16(UEChannel, value)*
  Stores a time-stamped user event on specified user event channel, carrying a 16-bit value. No return value.

- *UserEvent32(UEChannel, value)*
  Stores a time-stamped user event on specified user event channel, carrying a 32-bit value. This probe is more costly to use compared to the 16-bit version, as it require two entries in the trace event buffer. No return value.

The timeout semantics is the same for all functions with a timeout parameter: -1 specifies no timeout, 0 specifies immediate timeout (without blocking) if the resource is not immediately available, and a positive non-zero value specifies a finite timeout duration, causing a timeout event to occur at $clk+timeout$, where clk is the current time at the call of the service. At the occurrence of a timeout event, RTSSim wakes up the blocked task by changing its status to "ready" and thereafter invoking the scheduler. Observe that this does not mean that the task will begin executing at this point; it depends on the scheduler. On successful completion of the service, the return value is 0. The return value on timeout is -1.

Appendix C

An Example RTSSim Model

This section gives an example of a fairly complex RTSSim model. The presented model has been used in the evaluation of the simulation optimization methods presented in Chapter 4; it was there labeled “Model 1". This model is hand made and describes a fictive system, with similar analysis challenges as ABB’s control system for industrial robots. The tasks of this model violate several assumptions of the traditional methods for analytical response-time analysis. The tasks in the model may:

- trigger the execution of other tasks through communication using message queues,
- be triggered both by timers and events, or a combination of both,
- have different temporal behaviors depending on the contents of received messages and the value of shared state variables,
- be blocked on sending and receiving of messages, and
- change the scheduling priority of tasks as a response to certain events.

The modeled fictive system controls a set of electric motors based on periodic sensor readings and aperiodic events. The calculations necessary for a real control system is not included in this model, the model mainly describes execution time, communication and other behavior that impact the temporal behavior. The model contains four periodic tasks:

An overview of the model is given in Figure C.1, where colors are used to indicate priority (red indicates top priority, yellow medium priority and green

Task	Priority	Period
PLAN_TASK	50	40 000 or 10 000
CTRL_TASK	40 or 20	10 000 or 20 000
IO_TASK	30	5 000
DRIVE_TASK	10	2 000

lowest priority. The illustration also shows the message queues (named XXQ) which the tasks use to communicate. The queue DDQ (in red) is critical in the application and is not allowed to become empty.

PLAN_TASK is responsible for high level planning of how to move the physical object connected to the motors. It periodically sends coordinates to CTRL_TASK through the queue CDQ (CTRL Data Queue). CTRL_TASK calculates control references for the motors with respect to input from CDQ and from IO_TASK, through the queue IOQ (I/O event Queue). The resulting motor control references from CTRL_TASK are sent to DRIVE_TASK, through DDQ (Drive Data Queue), which controls the motors. The purpose of IO_TASK is to collect buffered I/O events from the system's environment (from a low level buffer) and send this information to CTRL_TASK. Depending on the physical state of the controlled system, different numbers of I/O messages are received from the environment (e.g., sensors). The number of incoming messages for IO_TASK is modeled using the integer variable *nofEvents*, which is increased by the environment task IO_ENVTASK, by 0, 1 or 2, every 1 000 time units. IO_TASK, which has a period of 5 000, decreases this variable by 1 for each message that is sent to IOQ. The increments of nofEvents in IO_ENVTASK is a simulator input (i.e., determined by a random number).

As indicated by the table, both CTRL_TASK and PLAN_TASK may change priority and periodicity in response to specific events in the model. The period of CTRL_TASK is normally 20 000 time units, but when a movement is approaching the target, the period is decreased to 10 000 in order to improve control performance. The priority of CTRL_TASK is boosted if the input queue for DRIVE_TASK (DDQ) has decreased below a certain threshold, since this queue must never become empty. PLAN_TASK uses a shorter periodicity when idle, in order to faster detect a start event.

There are three types of events from the system environment: START, STOP and GETSTATUS. These events are sent to PLAN_TASK through the queue PCQ (PLAN Command Queue), which processes them accordingly; some are forwarded to CTRL_TASK and DRIVE_TASK, through their com-

mand queues CCQ and DCQ. The START event will cause the system to change state into active, which means that it powers up and controls the motors. The STOP event causes the system to power down the motors and go to idle state. The GETSTATUS event causes all tasks to send a status message to the user interface (an environment task). These events impact the execution time of the tasks. The events are generated by the environment tasks GETSTATUS_ENVTASK, START_ENVTASK and STOP_ENVTASK.

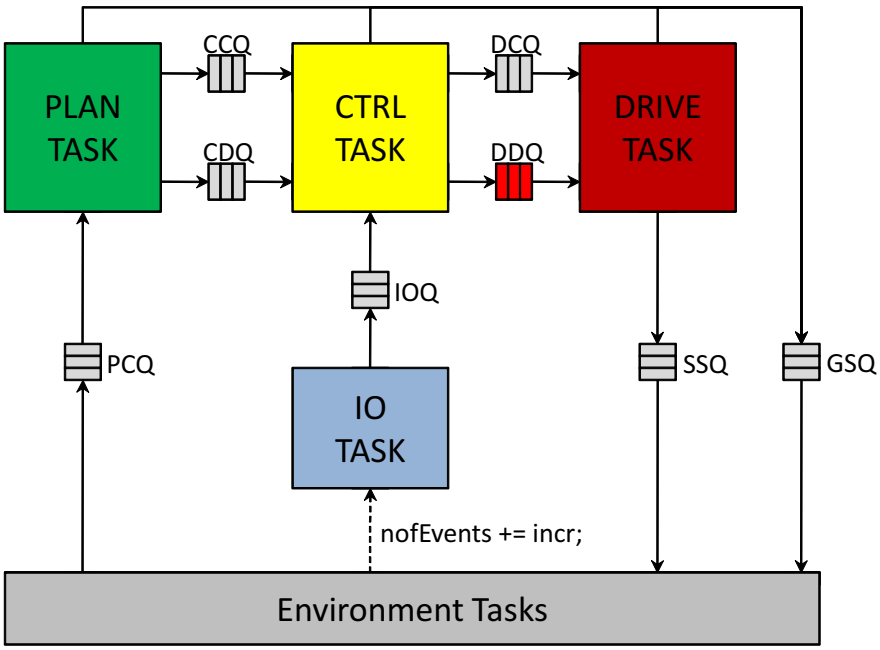

Figure C.1: Tasks and IPC in the example model

This model cannot be analyzed using traditional methods such as RTA, but an extreme scenario regarding the response time of CTRL_TASK has been identified using a simulation optimization method, HCRR. CTRL_TASK is the most complex task in the model and the case found is believed to be the worst case response time. The response time of CTRL_TASK is in the average case around 3 200, but can in a specific scenario be as long as 8 474 time units. The scenario depends on the following conditions:

- The number of messages in IOQ is 32 when the critical instance of CTRL_TASK begins to execute. This is very high, in fact the largest IOQ size observed in any experiment on this model.
- An instance of IO_TASK preempts the critical CTRL_TASK instance and refills IOQ with 10 messages during the CTRL_TASK's IOQ read loop, increasing the iterations of this loop from 32 to 42.
- A rare sporadic event (GETSTATUS) had just occurred, which results in messages for the following instance of CTRL_TASK and DRIVE_TASK, which increase their execution times.
- As a result of the long execution time of the critical instance of CTRL_TASK (6 224), it is preempted by five instances of DRIVE_TASK, of which one with an unusually long execution time due to a preceding GETSTATUS event, and two instances of IO_TASK.

The number of messages in IOQ has a major impact on the execution time of CTRL_TASK. The number of messages in IOQ is increased when IO_TASK executes, every 5 000 time units, and depends on the global variable nofEvents. Maximum 12 messages are sent to IOQ at each instance of IO_TASK. The nofEvents variable is in turn increased by an environment task, IO_ENVTASK, which executes every 1 000 time units and increases nofEvent by 0, 1 or 2 (according to simulator input data or random selection, depending on simulation mode). Reaching an IOQ size of 32 required an intricate sequence of input data, i.e. selections of the nofEvent increase, by 0, 1 or 2. It may seem natural that the worst case would occur if always increasing by the maximum number of events, i.e. 2. This gives a high response time as well, 8 324, but 150 less than the maximum found, 8 474, since the IOQ size only reach a maxiumum of 30 in this case, compared to 32 in the 8 474 case. The reason for this is in the relative timing between previous instances of CTRL_TASK and IO_TASK: In the worst-case scenario identified, i.e., the 8 474 case, the instance of CTRL_TASK preceding the critical instance had only 3 messages in IOQ to consume, which allowed it to finish the read loop before IO_TASK refilled it, which implied that these messages were instead processed by the next (the critical) CTRL_TASK instance. In the 8 324 case, i.e., where nofEvents is always increased by 2, the previous CTRL_TASK instance had more messages in IOQ to consume, compared to in the 8 474 case, which took longer time and caused the IO_TASK to preempt and refill IOQ during the read-loop. Thereby, in the 8 324 case also these messages were consumed by this previous

task instance, which caused the lower IOQ size (30) for the following, critical instance of CTRL_TASK, compared to the 8 474 case (32).

The large IOQ size the 8 474 case was partly caused by DRIVE_TASK; it increased the priority of CTRL_TASK momentarily, as the number of messages in DDQ has dropped below a specified threshold. This is a mechanism to prevent buffer-underrun situations on DDQ (it may not become empty) and implies that instances of IO_TASK are delayed, which changes the relative timing between IOQ's producer (IO_TASK) and consumer (CTRL_TASK).

Next follows the simulation model in detail. Note that declarations of global variables and `#include` directives have been omitted.

```
#define FOREVER -1

#define CDQSIZE 13

#define PLANSTATE_IDLE 0
#define PLANSTATE_BEGIN 1
#define PLANSTATE_WORKING 2

#define MSG_START 1
#define MSG_STOP 2

#define MSG_GETSTS 4
#define MSG_FLC 5
#define MSG_LAST 6
#define MSG_MOVING 7
#define MSG_NOTMOVING 8
#define MSG_STS_PLAN 9
#define MSG_STS_CTRL 10
#define MSG_STS_DRIVE 11

#define MSG_SLC 12
#define MSG_SLCD 13

#define cPLANstart 300
#define cPLANstop 300
#define cPLANgetsts 100
#define cPLANdecode 10
#define cPLANflc 2000
#define cPlanLast 100
#define cIOEvent 23
#define cCTRLdecode 18
#define cCTRLslc 398
#define cCTRLslcd 198
#define cCTRLgetsts 96
#define cCTRLioevent 48
#define cCTRLlast 18
#define cDRIVEdecode 18
#define cDRIVEslc 298
#define cDRIVEslcd 198
#define cDRIVEgetsts 98

#define MINDDQSIZE 5
```

```
void PLAN_TASK(TCB* tcb)
{
  int nFLCs, cmd, status;

  do   // process all requests in PCQ
  {
    status = RecvMessage(PCQ, &cmd, 0);
    Execute(cPLANdecode);
    if (status == 0)
    {
      switch(cmd)
      {
        case MSG_START:
          remainingFLC = 130;
          UserProbe16(probe_remaining_FLC, remainingFLC);
          planstate = PLANSTATE_BEGIN;
          UserProbe16(probe_plan_task_state, planstate);
          Execute(cPLANstart);
          break;
        case MSG_STOP:
          planstate = PLANSTATE_IDLE;
          UserProbe16(probe_plan_task_state, planstate);
          Execute(cPLANstop);
          break;
        case MSG_GETSTS:
          Execute(cPLANgetsts);
          SendMessage(GSQ, MSG_STS_PLAN, FOREVER);
          SendMessage(CCQ, MSG_GETSTS, FOREVER);
          break;
        default:
          sim_fail_int("Warning, got message: %d\n", cmd);
      }
    }
  }while (cmd != -1); // until no more messages

  // Execute periodic behavior, depending on state
  switch (planstate)
  {
    case PLANSTATE_BEGIN:
      planstate = PLANSTATE_WORKING;
      UserProbe16(probe_plan_task_state, planstate);
      closeToTarget = 0;

      if (remainingFLC < CDQSIZE)
      {
        nFLCs = remainingFLC;
      }else{
        nFLCs = CDQSIZE;
      }
      while (nFLCs > 0)
      {
        Execute(cPLANflc);
        SendMessage(CDQ, MSG_FLC, FOREVER);
        nFLCs--;
        remainingFLC--;
      }
      tcb->period = 40000;
      break;
```

```
    case PLANSTATE_WORKING:
      if (remainingFLC < 4)
      {
        nFLCs = remainingFLC;
      }else{
        nFLCs = 4;
      }
      while (nFLCs > 0)
      {
        Execute(cPLANflc);
        SendMessage(CDQ, MSG_FLC, FOREVER);
        nFLCs--;
        remainingFLC--;
      }
      tcb->period = 40000;
      break;
    case PLANSTATE_IDLE:
      tcb->period = 10000;
      break;
   }
   UserProbe16(probe_remaining_FLC, remainingFLC);

   if (((remainingFLC <= 0) &&
       (planstate != PLANSTATE_IDLE)) ||
       ((remainingFLC > 0) &&
       (planstate == PLANSTATE_IDLE)))
   {
     Execute(cPlanLast);
     planstate = PLANSTATE_IDLE;
     closeToTarget = 1;
     remainingFLC = 0;
     SendMessage(CDQ, MSG_LAST, FOREVER);
     UserProbe16(probe_plan_task_state, planstate);
   }
}


void CTRL_TASK(TCB* tcb)
{
  int msg, ioevent, status, i, nSLC = -1;

  msg = RecvMessage(CCQ, &msg, 0);
  Execute(cCTRLdecode);

  if (msg > -1)
  {
    if (msg == MSG_GETSTS)
    {
      SendMessage(GSQ, MSG_STS_CTRL, FOREVER);
      Execute(cCTRLgetsts);
      SendMessage(DCQ, MSG_GETSTS, FOREVER);
    }else{
      sim_fail_int("CTRL_TASK got message: %d\n", msg);
    }
  }
```

```
// read all pending messages in IO queue
i = 0;
do{
  if (RecvMessage(IOQ, &ioevent, 0) == 0)
  {
    i++;
    Execute(cCTRLioevent);
  }
}while (status == 0);
if (closeToTarget == 0)
{
  nSLC = 10;
  tcb->period = 20000;
}else{
  nSLC = 5;
  tcb->period = 10000;
}

// Process any FLC message from PLAN_TASK (maximum 1)
if (RecvMessage(CDQ, &msg, 0) == 0)
{
  switch(msg)
  {
    case MSG_FLC:
      if (idle == 1)
      {
        idle = 0;
        UserProbe16(probe_ctrl_idle, idle);
      }
      while (nSLC-- > 0)
      {
        // generate SLC data to DRIVE
        Execute(cCTRLslc);
        SendMessage(DDQ, MSG_SLC, FOREVER);
      }
      break;

    case MSG_LAST:
      idle = 1;
      closeToTarget = 0;
      Execute(cCTRLlast);
      UserProbe16(probe_ctrl_idle, idle);
      break;

    default:
      sim_fail_int("CTRL_TASK got message %d\n", msg);
      break;
  }
}
else // if no message
{
  if (idle == 0)
  {
    // if expecting message
    sim_fail("CTRL_TASK starvation!\n");
  }
}
```

```
  // if idle, generate default data (slcd)
  if (idle == 1)
  {
    while (nSLC-- > 0)
    {
      Execute(cCTRLslcd);
      SendMessage(DDQ, MSG_SLCD, FOREVER);
    }
  }
}


void DRIVE_TASK(TCB* tcb)
{
  int msg;
  if (RecvMessage(DDQ, &msg, 0) != 0)
  {
    sim_fail("DRIVE_TASK starvation!\n");
  }

  Execute(cDRIVEdecode);

  if (DDQ->current_size < MINDDQSIZE)
  {
    // boost priority of CTRL_TASK, above IO_TASK
    ctrl_task_tcb->prio = 20;
  }
  else
  {
    // normal priority of CTRL_TASK
    ctrl_task_tcb->prio = 40;
  }
  UserProbe16(probe_ctrl_prio, ctrl_task->prio);

  // process data message from CTRL_TASK
  switch(msg)
  {
    case MSG_SLC:
      Execute(cDRIVEslc);
      if (isMoving == 0)
      {
        isMoving = 1;
        UserProbe16(probe_drive_ismoving, ismoving);
        SendMessage(SSQ, MSG_MOVING, FOREVER);
      }
      break;
    case MSG_SLCD:
      Execute(cDRIVEslcd);
      if (ismoving == 1)
      {
        ismoving = 0;
        UserProbe16(probe_drive_ismoving, ismoving);
        SendMessage(SSQ, MSG_NOTMOVING, FOREVER);
      }
      break;
    default:
      sim_fail_int("Warning, got message: %d\n", msg);
```

```
      break;
  }
  // check for a getstatus request
  if (RecvMessage(DCQ, &msg, 0) == 0)
  {
    switch(msg)
    {
      case MSG_GETSTS:
        Execute(cDRIVEgetsts);
        SendMessage(GSQ, MSG_STS_DRIVE, FOREVER);
        break;
      default:
        sim_fail_int("Warning, got message %d\n", msg);
        break;
    }
  }
}


void IO_TASK(TCB* tcb)
{
  int status;
  int eventsToProcess = 0;
  if (nofEvents > 12)
  {
    // limit to 12, process remaining IO events later
    eventsToProcess = 12;
  }
  else
  {
    // normal case, process all IO events (<= 12)
    eventsToProcess = nofEvents;
  }
  while(eventsToProcess-- > 0)
  {
    Execute(cIOEvent);
    nofEvents--;

    // The value (42) of IOQ messages is not used...
    if (SendMessage(IOQ, 42, 0) != 0)
    {
      printf("IOQ overflow! clk: %d\n", clk);
    }
  }
}


void IO_ENVTASK(TCB* tcb)
{
  nofEvents += (int)(getRandomValue() % 3);
}
```

```
void GETSTATUS_ENVTASK(TCB* tcb)
{
  int reply;
  SendMessage(PCQ, MSG_GETSTS, FOREVER);
  RecvMessage(GSQ, &reply, FOREVER);
  if (reply != MSG_STS_PLAN)
  {
    printf("Warning, got unexpected message %d\n",reply);
  }

  RecvMessage(GSQ, &reply, FOREVER);
  if (reply != MSG_STS_CTRL)
  {
    printf("Warning, got unexpected message %d\n",reply);
  }

  RecvMessage(GSQ, &reply, FOREVER);
  if (reply != MSG_STS_DRIVE)
  {
    printf("Warning, unexpected message %d\n",reply);
  }
}



void START_ENVTASK(TCB* tcb)
{
  int reply;
  SendMessage(PCQ, MSG_START, FOREVER);
  RecvMessage(SSQ, &reply, FOREVER);
  if (reply != MSG_MOVING)
  {
    printf("Warning, unexpected message %d\n",reply);
  }
  // create the STOP task dynamically as a one-shot task
  createTask("STOP_ENVTASK",
             0,            // priority (highest)
             -1,           // period (-1 means one-shot)
             clk + 100000, // earliest start time of task
             100000,       // max additional delay (jitter)
             STOP_ENVTASK);
}



void STOP_ENVTASK(TCB* tcb)
{
  int reply;
  SendMessage(PCQ, MSG_STOP, FOREVER);
  RecvMessage(SSQ, &reply, FOREVER);
  if (reply != MSG_NOTMOVING)
  {
    printf("Warning, got unexpected message %d\n",reply);
  }
}
```

```
void model_init()
{
  PCQ = CreateMailbox("PLAN_CMD", 3);
  CCQ = CreateMailbox("CTRL_CMD", 6);
  CDQ = CreateMailbox("CTRL_DATA", CDQSIZE);
  DDQ = CreateMailbox("DRIVE_DATA", 9);
  DCQ = CreateMailbox("DRIVE_CMD", 6);
  SSQ = CreateMailbox("START_STOP_STATUS", 6);
  IOQ = CreateMailbox("IO_DATA", 40);

  // Create normal application tasks
  CreateTask("PLAN_TASK",
             50,// priority (lowest)
             40000,// period (can change to 10000)
             0,         // offset
             0,// max jitter
             PLAN_TASK // task entry function
            );

  // Keep the tcb handle for use in DRIVE_TASK
  ctrl_task_tcb = CreateTask("CTRL_TASK",
             40,// priority (can change to  20)
             10000,// period (can change to 20000)
             0,         // offset
             0,// max jitter
             CTRL_TASK  // task entry function
            );

  CreateTask("IO_TASK",
             30,   // priority
             5000,   // period
             500,      // offset
             0,   // max jitter
             IO_TASK   // task entry function
            );

  CreateTask("DRIVE_TASK",
             10, // priority
             2000, // period
             12001,      // offset
             0, // max jitter
             DRIVE_TASK  // task entry function
            );


  // Create the "invisible" environment tasks
  CreateTask("IO_ENVTASK",
             0,// priority (highest)
             2000,// period
             0,// offset
             0,// max jitter
             IO_ENVTASK // task entry function
            );

  CreateTask("START_ENVTASK",
             0,// priority (highest)
             -1,    // period (one-shot)
             0,// offset
```

```
            100000,// max jitter
            START_ENVTASK
           );

  CreateTask("GETSTATUS_ENVTASK",
            0,// priority (highest)
            90000,// period
            20000,// offset
            20000,// max jitter
            GETSTATUS_ENVTASK // task entry function
           );

  // Register probe channels for Tracealyzer output
  probe_remaining_FLC = CreateUEChannel("REMAINING_FLC");
  probe_plan_task_state = CreateUEChannel("PLAN_STATE");
  probe_ctrl_idle = CreateUEChannel("CTRL_IS_IDLE");
  probe_drive_ismoving = CreateUEChannel("DRIVE_ISMOVING");
  probe_ctrl_prio = CreateUEChannel("CTRL_PRIORITY_BOOST");

  // Clear all global state variables
  closeToTarget = 0;
  remainingFLC = 0;
  planstate = PLANSTATE_IDLE;
  idle = 1;
  nofEvents = 0;
  isMoving = 0;
}
```

Bibliography

[1] Mikael Åsberg, Thomas Nolte, Clara M. Otero Perez, and Shinpei Kato. Execution Time Monitoring in Linux. In *Proceedings of the Work-In-Progress session of 14th IEEE International Conference on Emerging Techonologies and Factory*, September 2009.

[2] Mikael Åsberg, Johan Kraft, Thomas Nolte, and Shinpei Kato. A loadable task execution recorder for Linux. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, July 2010.

[3] J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. Alf - a language for wcet flow analysis. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[4] 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.

[5] J.T. Alander, T. Mantere, G. Moghadampour, and J. Matila. Searching Protection Relay Response Time Extremes Using Genetic Algorithm — Software Quality by Optimization. In *Proceedings of the International Conference on Advances in Power System Control, Operation and Management (APSCOM-97)*, volume 1, pages 95–99, 1997.

[6] R. Nossal and T. M. Galla. Solving NP-Complete Problems in Real-Time System Design by Multichromosome Genetic Algorithms. In *Proceedings of the SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Sys.*, pages 68–76, 1997.

[7] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, Jan. 1989.

[8] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[9] A. Betts and G. Bernat. Tree-Based WCET Analysis on Instrumentation Point Graphs. In *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 558–565, Washington, DC, USA, 2006. IEEE Computer Society.

[10] B. A. Wichmann and I. D. Hill. Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):188–190, 1982.

[11] R. Racu and R. Ernst. Scheduling Anomaly Detection and Optimisation for Distributed Systems with Preemptive Task-Sets. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 325–334. IEEE, Apr. 2006.

[12] S. Samii, S. Rafiliu, P. Eles, and Z. Peng. A Simulation Methodology for Worst-Case Response Time Estimation of Distributed Real-Time Systems. In *Proceedings of Design, Automation, and Test in Europe (DATE'08)*, pages 556–561, 2008.

[13] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In *FORMATS*, pages 60–72, 2003.

[14] M. Bohlin, Y. Lu, J. Kraft, P. Kreuger, and T. Nolte. Simulation-Based Timing Analysis of Complex Real-Time Systems. In *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, pages 321–328, August 2009.

[15] J. Kraft, Y. Lu, C. Norström, and A. Wall. A Metaheuristic Approach for Best Effort Timing Analysis targeting Complex Legacy Real-Time Systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, Apr. 2008.

[16] M. Bohlin, Y. Lu, J. Kraft, P. Kreuger, and T. Nolte. Best-Effort Simulation-Based Timing Analysis using Hill-Climbing with Random Restarts. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-236/2009-1-SE, Mälardalen University, June 2009.

[17] M. Harman, D. Binkley, K. B. Gallagher, N. Gold, and J. Krinke. Dependence clusters in source code. *ACM Transactions on Programming Languages and Systems*, 32, 2009.

[18] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[19] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[20] M. Weiser. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis. Technical report, University of Michigan, Ann Arbor, 1979.

[21] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up Slicing. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20. ACM Press, 1994.

[22] T. Reps, S. Horwitz, and D. Binkley. U.S. Patent Number 5,161,216, Interprocedural slicing of computer programs using dependence graphs. Technical report, 1992.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI'88)*, pages 35–46, New York, NY, USA, 1988. ACM.

[24] H. K. N. Leung and H. K. Reghbati. Comments on program slicing. *IEEE Transactions on Software Engineering*, 13(12):1370–1371, 1987.

[25] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[26] M. Weiser. Program Slicing. In *Proceedings of the International Conference on Software Engineering (ICSE'81)*, pages 439–449. IEEE Press, 1981.

[27] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper. Faster wcet flow analysis by program slicing. In *ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'06)*. ACM, June 2006.

[28] 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.

[29] M. Harman, M. Munro, D. Binkley, S. Danicic, M. Aoudi, and L. Ouarbya. Syntax-directed amorphous slicing. *Journal of Automated Software Engineering*, 11(1):27–61, 2004.

[30] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology special issue on Program Slicing*, 40(11-12):595–607, 1998.

[31] S. Danicic, C. Fox, M. Harman, and R. Hierons. ConSIT: A Conditioned Program Slicer. In *IEEE International Conference on Software Maintenance (ICSM'00*, pages 216–226. IEEE Computer Society Press, 2000.

[32] M. Harman, D. Binkley, and S. Danicic. Amorphous Program Slicing. In *Software Focus*, pages 70–79. IEEE Computer Society Press, 1997.

[33] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[34] 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.

[35] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, 1984.

[36] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

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

[38] I. M. Chakravarti, J. Roy, and R. G. Laha. *Handbook of methods of applied statistics*. Wiley, New York, 1967.

[39] 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.

[40] 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.

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

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

[43] 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.

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

[45] R. Kollman, P. Selonen, E. Stroulia, T. Systä, 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.

[46] 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.

[47] 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.

[48] 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.

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

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] J. G. Huselius and J. Andersson. Model Synthesis for Real-Time Systems. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 52–60, 2005.

[57] C. Norström, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proceedings of RTCSA'99*. IEEE Computer Society, December 1999.

[58] J. Andersson, J. G. Huselius, C. Norström, and A. Wall. Extracting simulation models from complex embedded real-time systems. In *Proceedings of the 2006 International Conference on Software Engineering Advances, ICSEA'06*. IEEE, October 2006.

[59] 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.

[60] 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, USA., 2001.

[61] O. Balci. Guidelines 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, USA., 1990.

[62] 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, USA., 1999.

[63] A. M. Law and W. D. Kelton. *Simulation, Modeling and Analysis*. ISBN: 0-07-116537-1. McGraw-Hill, 1993.

[64] J. G. Huselius, J. Andersson, H. Hansson, and S. Punnekkat. Automatic generation and validation of models of legacy software. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 342–349, August 2006.

[65] S. Danicic and M. Harman. Espresso: a slicer generator. In *SAC'00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 831–839. ACM, 2000.

[66] T. Reps and G. Rosay. Precise interprocedural chopping. In *SIGSOFT'95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 41–52. ACM, 1995.

[67] D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *SIGSOFT'94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 2–10, New York, NY, USA, 1994. ACM.

[68] M. J. Sifalakis. Finding Dependencies Using Program Slicing, MSc thesis report, University of Edinburgh, 2001.

[69] L. Bent, D. Atkinson, and W. Griswold. A Qualitative Study of Two Whole-Program Slicers for C, UCSD Technical Report CS2000-0643.

[70] D. C. Atkinson and W. G. Griswold. Effective Whole-Program Analysis in the Presence of Pointers. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE '98)*, pages 46–55, New York, NY, USA, 1998. ACM.

[71] 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.

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

[73] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 32–41, 1996.

[74] 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.

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

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

[77] 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.

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

[79] G. Behrmann, A. David, and K. G. Larsen. A tutorial on uppaal. 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.

[80] 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.

[81] 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.

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

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

[84] A Strategic Research Agenda for the Swedish Software Intensive Industry, Swedsoft, January 2010.

[85] Mjukvara dold jätteindustri inom svensk it, Computer Sweden, March 30th, 2010.

[86] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[87] 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.

[88] D. Decotigny and I. Puaut. Artisst: An extensible and modular simulation tool for real-time systems. In *Proceedings of the fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington D.C.*, 2001.

[89] 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.

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

[91] 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.

[92] 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.

[93] 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.

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

[95] J. Andersson. Modeling The Temporal Behavior of Complex Embedded Systems – A Reverse Engineering Approach, Licentiate thesis. Technical report, 2005.

[96] M. I. Mughal and R. Javed. Recording of Scheduling and Communication Events on Telecom Systems. Master's thesis, Mälardalen University, Västerås, Sweden, 2008.

[97] M. Johansson and M. Saegebrecht. Lastmätning av CPU i realtidsoperativsystem. Master's thesis, Mälardalen University, Västerås, Sweden, 2007.

[98] 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.

[99] 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.

[100] Joel Huselius. *Reverse Engineering of Legacy Real-Time Systems: An Automated Approach Based on Execution-Time Recording*. PhD thesis, Mälardalen University, June 2007.

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

[102] N. C. Audsley, 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.

[103] 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 Computer Science, Illinois Univ., Urbana, IL, USA, 1996.

[104] 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.

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

[106] 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.

[107] M M. Lehman, J F. Ramil, P D. Wernick, D E. Perry, and W M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[108] 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.

[109] 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.

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

[111] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[112] 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.

[113] 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.

[114] N. C. 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.

[115] 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.

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

Web References

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

[118] The ABB Group. http://www.abb.com/.

[119] Mongoose website at Google Code. http://code.google.com/p/mongoose.

[120] Arcticus Systems AB. http://www.arcticus-systems.com.

[121] Wisconsin Program-Slicing Project. http://www.cs.wisc.edu/wpis/html.

[122] TimesTool. http://www.timestool.com.

[123] GrammaTech, Inc. http://www.grammatech.com.

[124] Virtutech Simic. http://www.virtutech.com.

[125] U.S. National Institute of Standards and Technology (NIST) e-Handbook of Statistical Methods, Section 1.3.5.16: Kolmogorov-Smirnov Goodness-of-Fit. http://www.itl.nist.gov/div898/handbook.

[126] Graphviz. http://www.graphviz.org.

[127] Embedded Systems by the Numbers, J. Turley. http://www.embedded.com/1999/9905/9905turley.htm.

[128] Embedded C++. http://www.caravan.net/ec2plus.

[129] UppAal. http://www.uppaal.com.

[130] ComFoRT Reasoning Framework. http://www.sei.cmu.edu/predictability/tools/comfort.

[131] Kronos Home Page. http://www-verimag.imag.fr/TEMPORISE/kronos.

[132] Wind River, Inc. http://www.windriver.com.

[133] The SMV System. http://www-2.cs.cmu.edu/~modelcheck/smv.html.

[134] ENEA AB. http://www.enea.com.

[135] Imagix Corporation. http://www.imagix.com.

[136] Rapita Systems, Ltd. http://www.rapitasystems.com.

[137] Quadros Systems, Inc. http://www.quadros.com.

[138] Scientific Toolworks, Inc. http://www.scitools.com.

[139] AbsInt Angewandte Informatik GmbH. http://www.absint.de.

[140] SWEET WCET project Website. http://www.mrtc.mdh.se/projects/wcet.

[141] Bound-T time and stack analyser. http://www.tidorum.fi/bound-t.

[142] Coverity, Inc. http://www.coverity.com/.

[143] C++ vs. Perl vs. Python vs. PHP performance benchmark. http://blog.famzah.net/2010/07/01/cpp-vs-python-vs-perl-vs-php-performance-benchmark/.

[144] Python vs. Perl. vs. C++. http://tenser.typepad.com/tenser_said_the_tensor/2006/08/python_vs_perl_.html/.

All web references were verified on the 19th of May, 2010.