Mälardalen University Press Dissertations
No. 43

REVERSE ENGINEERING OF LEGACY REAL-TIME SYSTEMS

AN AUTOMATED APPROACH BASED ON EXECUTION-TIME RECORDING

Joel Huselius

2007

MÄLARDALEN UNIVERSITY

Department of Computer Science and Electronics
Mälardalen University

ISSN 1651-4238
ISBN 978-91-85485-47-5
Printed by Arkitektkopia, Västerås, Sweden

Mälardalen University Press Dissertations

No. 43

REVERSE ENGINEERING OF LEGACY REAL-TIME SYSTEMS
AN AUTOMATED APPROACH BASED ON EXECUTION-TIME RECORDING

Joel Huselius

Akademisk avhandling

som för avläggande av Teknologie doktorsexamen i Datavetenskap vid Institutionen för datavetenskap och elektronik kommer att offentligen försvaras torsdagen, 14:e juni, 2007, 13.15 i Gamma, U, Högskoleplan 1, Västerås.

Fakultetsopponent: M.Sc. Antonia Bertolino, Istituto di Scienza e Tecnologie della Informazione A. Faedo, Italy.

MÄLARDALEN UNIVERSITY

Institutionen för datavetenskap och elektronik
Mälardalen University, Box 883, SE-72123 Västerås

Abstract

Many real-time systems have significant value in terms of legacy, since large efforts have been spent over many years to ensure their proper functionality. Examples can be found in, e.g., telecom and automation-industries. Maintenance consumes the major part of the budget for these systems. As each system is part of a dynamically changing larger whole, maintenance is required to modify the system to adapt to these changes. However, due to system complexity, engineers cannot be assumed to understand the system in every aspect, making the full range of effects of modifications on the system difficult to predict. Effect prediction would be useful, for instance in early discovery of unsuitable modifications. Accurate models would be useful for such prediction, but are generally non-existent.

With the introduction of a method for automated modeling, this thesis applies an industrial perspective to the problem of obtaining models of legacy real-time systems. The method generates a model of the system as it behaved during the executions. The recordings cover system level events such as context switches and communication, and may optionally cover data manipulations on task level, which allows modeling of causal relations. As means of abstraction, the models can contain probabilistic selections and execution time requirements. The method also includes automatic validation of the generated model, in which the model is compared to the system behavior. Our method has been implemented and has been evaluated in both an industrial case-study and in a controlled experiment. For the controlled experiment, we have developed a framework for automatic evaluation of (automated) modeling methods.

Using the models generated with our method, engineers can prototype designs of modifications, which allows for early rejection of unfeasible designs. The earlier such rejection is performed, the more time and resources are freed for other activities.

ISSN 1651-4238

ISBN 978-91-85485-47-5

Svensk sammanfattning

Området för denna avhandling är hållbar utveckling av befintliga industriella programvarusystem. I avhandlingen presenteras en nyutvecklad metod för att automatiskt skapa systemmodeller som bl. a. kan användas för att undersöka konsekvenser av tänkta systemutvecklingar och -förändringar. Genom tidig information om vilka förändringar som är genomförbara så ökar möjligheterna till återanvändning av existerande programvara vilket kan vara ett kostnadseffektivt alternativ till nyutveckling. För ett befintligt programvarusystem innebär detta hållbar utveckling, då det till en viss brytpunkt är billigare att underhålla och vidareutveckla det befintliga systemet jämfört med att bygga om systemet från grunden. Metoden som presenteras i avhandlingen kan användas till att skjuta denna brytpunkt framåt i tiden och på så sätt uppnås större lönsamhet över systemets hela livscykel.

Huvuddelen av underhålls- och vidareutvecklingskostnaden för industriella programvarusystem beror på att systemen är så komplexa att konsekvenserna av en planerad förändring i systemet inte går att överblicka innan förändringen är genomförd. Om förändringar inte kan utvärderas innan de är implementerade leder detta i värsta fall till att en förändring som inte uppfyller systemkraven implementeras, vilket kan medföra onödiga kostnader i såväl tid som resurser. Det är alltså önskvärt att så tidigt som möjligt kunna avgöra om en förändring är möjlig eller inte med avseende på systemkraven. Ett möjligt sätt att avgöra detta är att undersöka abstrakta prototyper (modeller) av systemet tillsammans med en tilltänkt förändring. Detta kräver dock en modell av systemet, vilken oftast inte är tillgänglig eftersom konstruktion och uppdatering av modeller normalt är för kostsamt att utföra manuellt och därför inte sker.

Metoden för automatisk modellering som presenteras i avhandlingen bygger på analys av inspelningar från programvarusystem under drift. Inspelningarna innehåller information om systemets funktion inklusive tidsbeteende och metoden extraherar systemets struktur. Metoden har implementerats och utvärderats i en fallstudie på ett styrsystem för en industriell robot och i ett kontrollerat experiment. Experimentet har genomförts i ett ramverk som har utvecklats för detta syfte. Tanken är att ramverket i framtiden även ska kunna användas för att jämföra olika metoder för automatisk modellering. Genom detta arbete visas att modellering (till viss grad) kan automatiseras och att de genererade modellerna kan användas för att undersöka konsekvenser av systemförändringar.

Till Rebecca

This is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

(Sir Winston S. Churchill, 1942.)

Acknowledgments

How can one man owe so much to so many? Now that this journey is coming to an end, this is a relevant question. Though, by definition, thesis writing is in large portions a one man job, it never ceases to amaze me how much help and support you need to do something on your own. There have been multiple points of failure in the conception of this thesis, meaning that there are a lot of people that I could not have done without, and I am glad that it seems as if all has worked out in the end. Thank you all!

The first specific thank yous go to my supervisors and my financiers: The work presented here has been skillfully supervised by Professor Hans Hansson and Professor Sasikumar Punnekkat and carried out within the SSF projects SAVE and PROGRESS. The great quality of the supervision and co-operation has been instrumental in producing this thesis. I would like to thank them both for starting to teach me the art of weighing my written words on a silver scale, and for all the other positive effects they have had on my writing. I am not there yet, but I hope that I have improved! I would like to thank Dr. Henrik Thane for introducing me to debugging of real-time systems and valuable supervision during my first three years as a Ph.D.-student, when this subject was my focus.

During the course of this work, there has been extensive co-operation with fellow Ph.D.-student Johan Kraft (formerly Andersson), resulting in co-authorships on both accounts. Thank you Professor Christer Nordström for support during my time at the department, and as an excellent teacher during my undergraduate studies. A number of people have helped in reading drafts of the thesis: Professor Hans Hansson, Professor Sasikumar Punnekkat, Dr. Thomas Nolte, Professor Paul Pettersson, Professor Bengt Jonsson at Uppsala University, Dr. Insik Shin, Dr. Cristina Seceleanu, Dr. Henrik Thane, Daniel Sundmark, Ylva Boive, Professor Christer Nordström, Professor Mikael Nolin, and Johan Kraft. I would also like to thank my previous co-workers Daniel Sundmark, Anders Pettersson, and Lars “Lalle” Albertsson (SICS). Daniel, Anders, and I have shared the same office space for almost my entire time at the department; thanks for putting up! I have had fruitful discussions with Professor Dmitrii Silvestrov and Dr. Anatoliy Malyarenko at IMa@MDH, and Olga Grinchtein and Therese Berg from Uppsala University. During the initial parts of my Ph.D.-studies, I was employed by the Swedish Institute of Computer Science (SICS). Thank you all at the CNA-lab at SICS!

Regarding help with specific technicalities and specifics of the thesis, I would like to acknowledge the fruitful co-operation we had with Stefan Bygde concerning the comparison measure for discrete distributions (see Section 8.3), and Johan Kraft and ABB Robotics for the case study in Chapter 7. Dr. Gustav Öquist and I sketched the first concepts of the evaluation framework over a couple of alcohol-infused malt-beverages (see Figure 8.1, Page 113). Professor Dmitrii Silvestrov and Dr. Anatoliy Malyarenko introduced me to Classification (see Section 9.4.6). I am also very grateful to Arkitektkopia in Västerås for swift and professional printing of this thesis.

From the department, I generally thank you all for creating a good environment, and especially the following: Johan Kraft, Hans Hansson, Sasikumar Punnekkat, Daniel Sundmark, Anders Pettersson, Thomas Nolte, Dag Nyström, Jonas Neander, Ewa Hansen, Harriet Ekwall (thanks for all the help and the talks), Else-Maj Silén, Monica Wasell, Ylva Boive, Johan Fredriksson, Larisa Rizvanovic, Frank Lüders, Mats Björkman, Marcus Lindgren, Marcus Nilsson; and, though he has moved on to Germany, Professor Gerhard Fohler who initiated this beginning which is, perhaps, now ending. Thank you all!

During this work, I have spent a lot of time at the department of course, but since moving back to Stockholm I have found two other creative environments: the study halls of Kungliga Biblioteket (eng. National Library of Sweden) at Humlegården and the Ritorno Café by Vasaparken. In fact, Ritorno also deserves honorable mention due to their porridge breakfast, their egg & anchovy sandwich, and last but not least their liberal views on coffee refills and use of wall sockets.

Thank you all who have provided me with distractions that prevented me from, like Jack Nicholson in The Shining, writing a book filled with the proverb “All work and no play makes Joel a bad boy”: Mikael Bendtsen, Maria Engvall, Johnnie Blom, Ruth McNamara, Fredrik Mannerstedt, Gustav Öquist, Malin Hjalmarsson, Peter Vouri, Stefan Lundgren, Camilla Urgell, Daniel Osser, Elin Malmberg, Jocke Ekström, Anu Lindquist, Ingvar Åkerblad, Kotten Öquist, Livia, Malin, and Niclas Ulltin, Ann-Sofie Berg, Johan and Birgitta Kraft, Jonas Neander, Ewa Hansen, Emanuel Sparring, and Anneke Söderquist.

I want to thank my large family: Morfar, Mamma, Pappa, Mia, David, Monica, Bianca, Hannes, Micha, Bengt, Jeff, Laban, Marre, Mimmi, Hasse, Kristian H, Sara, Tessan, Kimmen, Eva, Bernt, Gunnel, Mats, Emma, Mats, Ingrid, Erik, Dagmar, Ingrid, Anita, Petra, Magnus, Lotus, Kristian A, Katta, Emilia, Tore, Hanna. I want to thank Luis and Rosa for a lovely time in Ensenada, it was great seeing you again in Sweden! Thank you Bengt and Jeff for the great time you showed me and Johan in Sydney during RTCSA06!

Finally, I thank my wonderful wife and favorite travel companion Rebecca for being who she is!

Thank you all!

Joel Gustaf Huselius
Ritorno in May, with
spring finally here.

Publications

Mr. Huselius has authored or co-authored two theses, nine peer-reviewed papers, and two technical reports. A subset of these publications are directly related to this thesis.

Related publications

1. "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, pages 577-585. IEEE Computer Society, March 2007. Presented at the 5th Workshop and Session on Model-Based Development of Computer Based Systems.

   Synopsis: *We present a methodology for the empirical evaluation of dynamic models generated from execution recordings of real-time software. We also present a set of synthetic benchmarks with varying architectural styles, with which we evaluate the performance of dynamic model extraction.*

   Contribution by Mr. Huselius: Mr. Huselius wrote the paper under supervision of Professor Hansson and Professor Punnekkat. Mr. Kraft contributed in discussions.

2. "Automatic Generation and Validation of Models of Legacy Software", Joel Huselius, Johan Andersson, Hans Hansson, and Sasikumar Punnekkat. In Proceedings of the International Conference on Real-Time Systems and Applications, pages 342-349. IEEE Computer Society, August 2006.

   Synopsis: *In this paper, we present a method for automatic model validation and a method for using the validation method together with our method for automatic model generation to perform model extraction. We present a case study on a state-of-practice industrial robotics system, where we show the usefulness of model extraction.*

   Contribution by Mr. Huselius: Mr. Huselius was the main author of the paper. Together with Professor Hansson and Professor Punnekkat, Mr. Huselius developed the method for automatic model validation. He then implemented the method. The case study was performed together with Mr. Andersson.

3. "Presenting: An Automated Process for Model Synthesis", Joel Huselius, Hans Hansson, and Sasikumar Punnekkat. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-191/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, October 2005.

   Synopsis: *The report presents technical details of an early version of the method for automatic model validation presented in Paper 2 and a detailed process for dynamic model extraction.*

   Contribution by Mr. Huselius: Mr. Huselius was the main author of the report. Together with Professor Hansson and Professor Punnekkat, Mr. Huselius developed the method for automatic model validation.

4. "Model Synthesis for Real-Time Systems", Joel Huselius, Johan Andersson. In Proceedings of the 9th European Conference on Software Maintenance and Re-engineering, pages 52-60. IEEE Computer Society, March 2005.

   Synopsis: *In this paper, we introduce a method for dynamic model generation (here called model synthesis). We present initial experimental results suggesting the usefulness of the method.*

   Contribution by Mr. Huselius: Mr. Huselius was the initiator and the main author of the paper, he developed and implemented the method for dynamic model generation. Mr. Andersson contributed in discussions and performed and wrote about the experiment related in the paper.

Other publications

Apart from the above, Mr. Huselius has published a set of theses, papers and reports with the major focus on *debugging of real-time systems*.

Theses

1. Joel Huselius, "Preparing for Replay", Licentiate thesis no. 16, ISSN 1651-9256, ISBN 91-88834-15-8, Mälardalen University, November, 2003. Opponent: Professor Peter Fritzson (LiU), examiner: Professor Mats Björkman, supervisors: Professor Hansson and Dr. Henrik Thane.

2. Mikael Bendtsen and Joel Huselius, "Issues on the MidART Middleware for Mobile Devices on Wireless Networks", Master thesis, Mälardalen University, June, 2001. Supervisor and examiner: Professor Gerhard Fohler (presently at Technische Universität Kaiserslauten).

Peer-reviewed papers

1. "Extracting Simulation Models from Complex Industrial Real-Time Systems", Johan Andersson, Joel Huselius, Christer Norstöm, and Anders Wall. In Proceedings of the First International Conference on Software Engineering Advances, October 2006. Best paper award.

2. "Constant Execution Time Recording for Replay of Sporadic Real-Time Systems", Joel Huselius and Henrik Thane. In Proceedings of the 2nd Workshop on Compilers and Tools for Constrained Embedded Systems, pages 39-47, September 2004.

3. "Availability Guarantee for Deterministic Replay Starting Points in Real-Time Systems", Joel Huselius, Henrik Thane, and Daniel Sundmark. In Proceedings of the 5th International Workshop on Algorithmic and Automated Debugging, pages 261-264, September 2003.

4. "Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies", Daniel Sundmark, Henrik Thane, Joel Huselius, Anders Pettersson, Roger Mellander, Ingemar Reiyer and Mattias Kallvi. In Proceedings of the 5th International Workshop on Algorithmic and Automated Debugging, pages 211-222, September 2003.

5. "Starting Conditions for Post-Mortem Debugging using Deterministic Replay of Real-Time Systems", Joel Huselius, Henrik Thane, and Daniel Sundmark. In Proceedings of the 15th Euromicro Conference on Real-Time Systems, pages 177-184. IEEE Computer Society, July 2003.

6. "Replay Debugging of Real-Time Systems Using Time Machines", Henrik Thane, Daniel Sundmark, Joel Huselius, and Anders Pettersson. In Proceedings of the International Parallel and Distributed Processing Symposium, pages 288-295. IEEE Computer Society, April 2003. Presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging.

Technical report

1. "Debugging Parallel Systems: A State of the Art Report", Joel Huselius. MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-63/2002-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, September 2002.

Contents

Chapter 1

Introduction

According to the IEEE standard 610 [42], an *abstraction* is:

> *"A view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information."*

In other words, for a given purpose, abstraction reduces the available information to the relevant information. Abstractions are made and used all the time, as they are fundamental for efficient communication and cognition. In reality, it is often difficult to choose the most appropriate abstraction for a given situation. If the abstraction is too high, the value of the information is reduced. If the abstraction is too low, processing the information is tedious. Finding the most suitable abstraction is a balance between reducing the amount of information and preserving potentially relevant information. Unfortunately, the relevance of the information is not always clear at the time when the abstraction is made.

Furthermore, the process of making the abstraction can introduce errors, e.g., by failing to maintain the essence of all relevant information or by introducing erroneous interpretations. This illustrates the dangers of abstraction; if relevant information is disregarded, decisions based on the abstraction may be erroneous.

As an example of abstraction, as humans, we often require abstraction as a tool to manage the complex information flow that the implementation of a system represents. The same problem is seen when computers are used to, e.g., validate a design; the validation tools need more abstract views of the real implementation in order to avoid computational complexity in terms of state space explosion. In computer science, models can provide an abstraction of the often complex inner workings of computer systems [15]. The model, while being significantly less complicated than the actual system, should still be similar to the system; finding the appropriate balance to meet this requirement is a big issue in modeling.

In order to achieve the right level of abstraction, traditionally, modeling is an art rather than an algorithmic process; the wit and cunningness of the model designer is imperative to the accuracy, efficiency, and usefulness of the model. This traditional view is challenged by the research presented in this thesis in the sense that we are providing processes and methods that assist in modeling. Our long term goal is to eliminate the need for human expert interaction.

In our work, we target *the maintenance of legacy real-time software by automatic modeling*. In the following, we explain these terms informally:

In a *real-time system*, temporal and functional correctness are both important. A real-time system consists of a set of *tasks*[1] that may execute concurrently under the control of a scheduling algorithm. Each task is either triggered as a function of time or as a function of occurring events. At each triggering signal, the task spawns a job that executes for some time, possibly reading inputs, performing computations, or writing outputs. Jobs may also initiate events that trigger the execution of other tasks (e.g., by sending a message on a queue for inter-process communication). A scheduling algorithm is deployed to resolve the scheduling problem that occurs when more than one job is ready to execute at the same time. In this thesis, we will assume a fixed priority scheduling algorithm [68], but it is possible to extend the work also for other scheduling algorithms.

By the term *models*, we mean models of the whole system, which depict its behavior. Every task in the system is modeled according to its time consumption and environment interactions. *Modeling* denotes the activity of obtaining a simplified representation of the system to the extent that the produced model is of value to the engineers working on the system. *Automatic* modeling may require a set of input parameters, but apart from that it should be free from user interaction.

Maintenance is defined by the IEEE standard 610 [42] as:

> *"...modifying a software system or component after delivery to correct faults, improve performance or other attributes, or to adapt the product to a changed environment."*

[1]A task is, basically, a thread of execution.

In the context of this thesis, a *legacy* system has all or some of the following properties: it consists of millions of lines of code, it is maintained by a large team of engineers from several generations,[2] it contains code that originated several years ago, and it is expected to function for many more years to come. Real examples of these systems can easily be found within many domains such as automation, automotive, and telecom industries. In such systems, a large effort must be spent on keeping complexity at acceptable levels. If the complexity is allowed to increase without bound, the life expectancy of these systems will be reduced or/and costs will dramatically increase.

Maintaining legacy real-time software is a multi-faceted problem: In order to keep a long life expectancy, it is required that the software is carefully engineered to improve long term software quality and reduce the need for re-engineering. However, demands on short time-to-market limit the time budget available for careful engineering, and the complexity and lack of documentation/models of the software make efficient engineering difficult. Many products require a highly versatile software that can serve many purposes (e.g., product line architectures [116] or an industrial robot with multiple hardware configurations and operating environments), which increases complexity.

In his seventh *law of software evolution* [61, 62], Lehman proclaims that for an *E-type program*:[3]

> *"E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to changing operational environment."*

From this, it is evident that software maintenance is needed. In an uncontrolled quality decline software is made obsolete prematurely, which leads to even larger costs than required for maintenance. Lehman's second law (still concerning E-type systems) states that:

> *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*

Not only the software quality, but also the software complexity is jeopardized by poor maintenance. Intuitively, a less complex system is less difficult to maintain, and a more complex system is more difficult to maintain (this

[2]Several generations: The set of engineers that have contributed to the system is a superset of the set of engineers currently working on the system.

[3]E-type program: a program that solves a problem or implements a computer application in the real world, all industrial software applications are within this category.

has also been shown in [24]). Thus, maintenance should be a continuous effort throughout the life cycle of the system, and affordable efforts aimed to maintain the complexity within acceptable bounds will limit the cost of development. In this thesis, we assume that automatic modeling is affordable and that modeling is good for maintaining the level of complexity in the system.

We have good reasons for these two assumptions: As an automatic modeling process requires little input and guidance from the user, it is likely to be affordable, provided that the product (i.e., the model) is usable and that the computational overhead is low. Further, prototyping system modifications on models is a common technique used in many fields, including software engineering.

The technique has been presented as *model-based impact analysis* [4, 64, 116], where models are used to improve system quality. In model-based impact analysis, proposed changes are prototyped on a model, prior to implementation in the system. The idea is to allow early discovery of design flaws.

Late discovery of design flaws may induce large costs on software development, as correcting the flaw may require extensive re-design. Due to short term cost interests, late discovery could lead to the situation of that the implementation is being *patched* [42] rather than properly redesigned (i.e., design problems introduced early in the life cycle are not completely removed and remedied, but rather fixed via some "work around"). These patches will lead to an increase in the complexity of the implementation.

Despite the apparent benefits of using models, it seems that the industry is yet to be convinced [20]. In fact, out of all the industries that we have encountered during our work, none has used exhaustive modeling of the entire system. In our contacts with industry, we have felt that, part of the reason for this, is that modeling in general, and maintaining models in particular, is viewed as being a difficult manual task without clearly quantifiable benefits. We can only assume that the race for a short time-to-market yields an environment where visible short term goals are given undue importance over the long term objectives such as building and maintaining a reliable model of the system, probably due to the perceived fuzziness of the latter.

We list six identified problems with manual modeling:

1. **High learning threshold.** In order to be a good model designer, graduate level courses combined with extensive experience are often required. Many modeling languages have their very own tips and tricks that allow the user to get the most out of them. It is generally not sufficient to be able to use the language, one should actually master it in order to create

good and efficient models [14].

2. **Model validity.** A basic trade-off of modeling is that the model is supposed to look like the real implementation, but the motivation for using a model is to provide an abstraction. Thus, the model should be the same as the system, but at the same time different. This contradiction makes model validity subjective and difficult as there is no clear specification or measurement of the amount of difference.

3. **Model debugging.** Subtle errors in model design frequently have large implications on the accuracy of the model, but model-debugging tools are often crude or even non-existing.

4. **System knowledge.** Apart from knowledge of how to use the modeling language, an extensive knowledge of the system is also required. When a large and dynamic workforce is cooperating to construct and maintain a system, a consistent system view may be difficult to obtain.

5. **Continuous evolution.** The rapid evolution of systems compared to the slow and tedious work of modeling threatens to invalidate models before they are completed. Models that do not accurately describe the current version of the real implementation are not useful, but the process of implementing and maintaining the system cannot be halted to allow an accurate model to be constructed.

6. **Multiple modeling languages.** Many projects require the use of a variety of modeling languages. Among the many activities and processes that make use of models, we find design, verification, maintenance, etc. As these uses have differing requirements and call for different abstractions, there is a large variety of modeling languages available. As each language *provides* its own set of abstractions, and each use *requires* its own set of abstractions, the use of several modeling languages within the same project may be needed. Thus, the same organization often needs competence in a number of modeling disciplines.

We believe that tools that significantly ease modeling with respect to these problems will make the industry more inclined to adopt methods built on model-based analysis. In order to prepare for model-based analysis, tools are needed to make it possible to construct and maintain whole system models for legacy systems, where models are currently missing. These tools should be easy to use, allow model validation and debugging, and they should require little system knowledge. A broad range of tools is needed to support different modeling paradigms and languages. Finally, these tools should work fast and effectively so that models are kept up to date at the same pace with software evolution. In support of this vision, we have developed a method for automatic modeling, or *model extraction*, which uses recordings from executions of a legacy real-time system to generate and validate a model of the system.

At this stage, it is important to keep in mind that model extraction cannot be effectively performed in an arbitrary situation; the process lacks the ingenuity of an experienced engineer, and is only capable of reflecting a version of the reality that has already been conceived by engineers. Under this restriction however, model extraction can provide efficient aid in activities of development and maintenance.

We argue that model extraction can provide accurate up-to-date models, fast, without posing exotic requirements. This lets modeling take a more active place in the system development and maintenance. Further, it provides a quick way to model systems that have been constructed by a large and dynamic workforce in which no one is capable of maintaining a complete understanding of the entire system. We believe that model extraction will lead to reduced costs and to higher quality systems with a longer life expectancy.

Having our background in developing tools for debugging real-time systems by means of record/replay techniques [39, 40, 41, 104, 110], we are experienced in using recordings for extracting real-time information from existing systems during runtime. As we got acquainted with the problems of modeling, it was just natural to attempt an evaluation of the effectiveness of these techniques also in automatic modeling.

1.1 Problem definition

Assuming that the scenario described above holds, industry has a need for tools that support modeling of legacy real-time software at low cost and low overhead on the engineers and on the system.

This thesis investigates the possibility of supporting modeling using only recordings as input. We are doing this by developing methods that automate whole system modeling of legacy real-time software based on recordings. We have produced methods that are able to deduce models that can mimic recordings.

Generally, in real-time system research, the worst-case is the case that matters. However, one of the central issues of using recordings is the uncertainty of interpreting the results. Without performing a thorough analysis, an observer cannot determine if an execution has triggered the worst-case for the implementation. Thus, by necessity, the methods that we develop have a different aim. We aim for the models to reflect the general behavior of the system. Not the single most demanding instance, but some of the general-case behaviors that may occur. In addition, recordings are made based on a set of assumptions and a set of test-cases, and the act of making a recording is intrusive by nature. In return, basing models on recordings allows performance evaluation with respect to the general behavior of the system and implementation prototyping with respect to the performance of the system. These possibilities are not available to methods restricted to the worst-case behavior.

Assuming that a candidate for such an average-case method of automatic modeling is developed, this leads to the following main problems studied in this thesis:

1. Can the validity and accuracy of extracted models be quantified?
2. Are the overheads of model extraction acceptable?
3. Can the method of model extraction be evaluated?

At the end of this thesis, we will revisit these problems and assess whether they have been solved or not.

1.2 Research methodology

Research in the field of computer science can be performed within three paradigms [117]:

The mathematical method Here, abstraction of phenomena (e.g. computers, programming languages, algorithms), and reasoning about this abstraction are used to obtain information about the phenomena.

For example, in *deductive reasoning,* a set of assumed premises are logically (i.e., mathematically) proven to lead to a conclusion. One could say that deductive reasoning is a proof of the fact that a cause leads to an effect. The conclusion (or the effect) is then shown to be true iff[4] the premises (or the cause) are valid.

[4]iff: If and only if.

The empirical method Here, knowledge projected on a hypothesis is derived from experiments and other methods of data collection.

For example, experiments can show that a set of *independent* variables are controlling a set of *dependent* variables. The experiments should be *controlled*, i.e., they should be constructed such that there are no confounding variables (i.e., variables that should be in the set of independent variables, but are not included there). Also, it should be clear that the dependent variables are valid measures of the phenomenon targeted by the hypothesis. Finally, *external validity*, that several experiments indicate the same result, should be established. This form of triangulation is made to increase the confidence in the result.

The engineering method Here, the efficient fulfillment of a specific set of specifications and requirements is in focus. This is achieved by the conception of, e.g., a device, an algorithm, or a method. Essential in the engineering method is to by experiment or formal proof provide evidence of the fulfillment of the requirements.

Our motivation for developing the methods and performing the research presented presented in this thesis comes from the desires expressed by industrial partners, as well as observations stipulated in literature, which lead to the formulation of the problem definition as presented above. In this work, we have used all of the above methods to solve our set of problems:

Using the engineering method, we have developed a method for automatic modeling, and implemented a tool suite that supports this method and the evaluation thereof. Loosely speaking, the requirements were that a probabilistic model should be generated and validated from recorded execution traces. The method of validation for the generated model is based on mathematical methods applied on model checking timed automata [3]. Also, mathematical methods were used to formulate a comparison measure for quantifying the difference between two entities, which can be systems or models.

As for the empirical method, the tool suite has been evaluated in a state of practice industrial system, as well as in a controlled experimental study.

Intermediate results of this work have been disseminated and peer-reviewed by publications and presentations at international conferences, but also informal presentation for peers and industry.

1.3 Contribution

The following are the main contributions of the thesis:

- A method that uses recordings of real-time systems to extract probabilistic models of that system.
- A method to analyze the validity of the extracted model.
- Using these two results as the major components, we have developed a method of model extraction for the automatic modeling of real-time systems.
- The model extraction method has been implemented and tested in an industrial case-study.
- A classification of methods for automatic modeling.
- A measure for comparing two sampled time distributions as a means for comparing model against system behavior of real-time systems, including an algorithm that implements the comparison.
- A testbed for experimental evaluation of methods for automatic modeling. The testbed uses the method of comparing sampled time distributions to assess the accuracy of models.
- A controlled experiment using the introduced testbed. The collected evidence, including the above case study, provides evidence of the usefulness of the proposed method of model extraction.

In a wider perspective, the work presented in this thesis establishes the feasibility and supports the introduction of model-based analysis of legacy systems in industry. Model-based analysis aims to limit the uncontrolled complexity increase of legacy systems, a perspective which is generally overlooked in contemporary work. This thesis aims to highlight and address these important issues.

1.4 Organization

The remainder of this thesis is organized as follows:

Chapter 2 relates background and discusses topics of importance for the coming chapters. We discuss issues and problems related to recording, testing, maintenance, and model based development.

Chapter 3 introduces a simple classification of methods for automatic modeling and presents known work according to the classification.

Chapter 4 introduces the method for model extraction based on chapters 5 and 6. The model extraction is targeting online scheduled real-time systems, and is based on input from recordings of the executing system.

Chapter 5 presents details on the generation of probabilistic models based on recordings of a legacy real-time system. The output of the method is one model per task in the modeled system, the models use probabilistic constructs to specify non-deterministic relations (i.e., where the available information is insufficient for a complete model).

Chapter 6 presents details on the validation of generated probabilistic models based on a new set of recordings from the system. The output from the method is an assesment of the validity of the model; validation is answering the question: "Based on the available knowledge, is the probability that the model is representative of the system within a specified bound?"

Chapter 7 presents an industrial case study on the applicability of model extraction performed at ABB Robotics.

Chapter 8 presents a testbed for empirical evaluation of methods for automatic modeling, and results from the application of the testbed on model extraction. The testbed requires a comparison measure that can quantify the level of either likeness or difference between a model and a system. We introduce such a comparison measure based on the comparison of distributions of response times.

Chapter 9 concludes the thesis and presents future work.

Chapter 2

Background

As written text is communicated, it is important that the reader and the writer share understanding of the basic terminology of the material. Popularly: the writer should be on the same page as his/her readers. Due to the simplex nature of written communication, ensuring such understanding is difficult. This chapter is dedicated to relating the author's view on some fundamentals of the thesis. In the process, a multitude of references are made to related work.

For the understanding of the remaining chapters, the sections concerning real-time systems and recordings are probably the most important. For the motivation of the thesis, the section on maintenance is essential. The section on model based development discusses the long term goal for model extraction techniques.

2.1 Real-time systems

Common for all computer systems is that they are expected by their users to provide *functionally correct* results. For example, when considering a train about to leave from a station, it would probably be considered as functionally correct if the train departs from the station. If the train sets to sea, similarly to a ship, the result would probably be considered as functionally incorrect.

In *real-time systems* [101], results are required to be *functionally* and *temporally* correct. To exemplify, a temporally correct result is that a train departs on time from the platform. A temporally incorrect result is that the train departs from the platform, but the departure is too early or too late. As such behavior

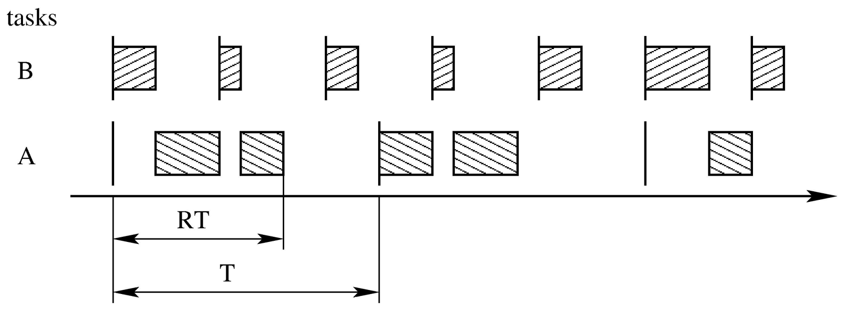


Figure 2.1: Task properties *periodicity* (T) and *response time* (RT) for one job of task A.

is not consistent with our expectations, that would be considered as temporally incorrect.

In computer science, real-time theories are typically applied to systems that interact with an environment governed by the laws of physics. Typical domains of application are telecom, aerospace, automotive, and automation-industries.

The workloads of real-time systems often consists of multiple threads of control that execute on one or more processing units. There are many names for the threads of control, whereas in this thesis we shall use the name *task*, which is the term generally used in the context of real-time systems. In other contexts, including some of our sources, the threads of control are called processes. In many cases, these terms are more or less interchangeable, although real-time tasks normally have less complex control-flows in the source code than many non-real-time systems, but they have more complex non-functional constraints.

Many real-time systems are of a periodic nature, for example the sample-actuate loops in control systems, where a task is to be performed with a certain frequency. Note that, two tasks in the same system may very well have different frequencies, and may be phase-shifted to each other.

Tasks may have a *periodicity* at which it emits *jobs*. A job is the execution of one instance of the code of the task. Non-functional constraints and attributes of tasks are often expressed using *deadlines*, *periodicity*, *response times*, *overhead*, etc. (see Figure 2.1).

The deadline of a task is the latest time at which the job of the task is required to complete. Should a job fail to complete before its specified deadline, its contribution to the computation cannot be considered usable. The severity of such a failure is grave for safety-critical *hard real-time systems*,[1] where functional and temporal correctness are equally important; complementary, *soft real-time systems*,[2] are designed to allow some amount of deadline-misses; i.e., functional correctness is more important than temporal correctness. The division of real-time systems into hard and soft is often insufficient for describing a given system; but it provides a framework that can be used for informal communication. In this thesis, we focus on a category of real-time systems that are hard in the respect that deadline misses may lead to failure of the system, but the system is not safety-critical, i.e., consequences of failures are not severe enough to motivate the rigorous development practice motivated in development of safety-critical hard real-time systems. An example of such a real-time system is provided in Chapter 7, where we present a case study performed on an industrial robot controller; in that system, a deadline miss may lead to a failure, and a failure can lead to significant economical loss in terms of reduced production rate, but the system is too complex to allow, e.g., traditional real-time schedulability analysis. Therefore, as a trade-off solution, the system is built using a real-time operating system, and instead of schedulability analysis, extensive testing and quality control is deployed to increase the confidence in the system.

We label the time measured from the point in time where the job is triggered (the *release time*) until the time when execution of a job is completed, as the *response time* of the job. To calculate the available resources for a task, the *overheads* of operating systems, communication protocols, etc., are important to account for as they interfere with the execution of the task.

Often, tasks have precedence orders that constitute dependency relations between events (e.g., in order to travel by train one must board before departing, thus, boarding has precedence over departure when traveling by train). Jitter, e.g., noice in the periodicity of task instances, is a consequence of the cooperative use of resources between tasks and properties of the environment. For example, as the processing power must be shared, and different tasks may have different periodicities, scheduling of tasks will differ between jobs.

Because of these complex constraints that may characterize the workload, it is a non-trivial task to schedule such systems. Real-time theorization has shown how various types of workloads could be scheduled to ensure off-line assertion of the schedulability of the specific workload. A scheduling algorithm operates using the available knowledge of the system. Depending on the composition and the complexity of the workloads, different scheduling algorithms can be

[1]Hard real-time systems: Traditionally, rockets, airplanes, etc.

[2]Soft real-time systems: Traditionally, multimedia streaming, toys, etc.

deployed to solve the scheduling problem. There are two fundamentally different approaches to solve the scheduling of a real-time system: by offline [54] or by online [68] scheduling. In the former, all issues are resolved offline by the creation of schedules whose fulfillment of the system requirements can be proved. In the latter, tasks are assigned priorities or other properties that are used as input to the scheduler in deciding online which task to execute at a given time.

To exemplify the new terminology, we use the human diurnal rhythm: the human real-time task of *sleeping* should emit a new job each night. Hence, the periodicity of sleeping is approximately 24 hours, the execution time of sleeping is perhaps averaging on eight hours, even if the worst-case execution time may be much longer. An alarm clock can be set to indicate the deadline of the task, a deadline miss would be oversleeping. Further, many people have arrival-jitter in their sleeping task as they do not fall asleep at the same time every night. The release time of the task could describe the activity of trying to fall asleep. Waking up in the middle of the night, having to go to the privy, would be a context-switch to another task. This multi-tasking will obviously increase overhead – it takes time to fall asleep again once you have awoken.

2.2 Model-based development

Model-based development (a.k.a. model-based engineering) aims to improve the efficiency and the quality of the software development process by making it more formal and more mechanical. As opposed to traditional code-oriented development, where the source code is the primary view of the developer, models are the primary view of the development and maintenance process [50]. They are used as media to formalize, convey, develop, and preserve the properties and requirements of the system. After that the model is completed and validated, the intention is that the push of a button (or the technical equivalent) should generate the application code. The hope is also that the extensive use of models that describe the functional and temporal properties of the implementation will facilitate reuse as well as automated validation and verification.

In order to use models as the primary view throughout the life cycle of the system and its parts, the models must present a uniform view to avoid divergence between models [94]. However, different activities in development and maintenance are focused on different aspects of the system [107]. As defined in Chapter 1, modeling is an abstraction for a given purpose, and different purposes have different requirements regarding what information is important and what can be abstracted. Thus, the modeling framework should allow for separation of concerns.

2.2.1 Introducing model-based development in maintenance of legacy systems

Research in model-based development focus on domain-specific modeling, meta-modeling for architectural descriptions, code generation from models, and analytic methods for supporting the development and maintenance of models [44]. Though much research has been conducted in these areas, it is probably safe to say that model-based development is not yet ready to make its mark in industry on a broad front.

Systems maintained in industry are generally code-oriented, and as Littlejohn et al. [67] point out: wholesale redevelopment is cost prohibitive, and prior investments must be preserved. In order to take model-based development into industry, reverse engineering methods must be developed that either atomically transform code into models in a revolutionary manner, or find other means of making a more gradual or evolutionary shift of development paradigm. In code-oriented development, the existing code-base is a valuable asset of the company. It is the result of enormous investments and has taken many man-years using other development paradigms to perfect. Any technique for introducing model-based development must do so with such detail and quality that the existing code-base can be discarded. This poses very high demands on the techniques for introducing model-based development. Contrary to intuition, it is likely that the more gradual shift towards model-based development is associated with a higher stake. Due to the accumulated work performed to make the shift, and the reduction in work efficiency during the shift, the stake of a failed paradigm shift is higher than in the case of the “big-bang”. The risk however, is likely to be lower with a gradual approach that has the ability to adopt dynamically to issues that arise during the paradigm shift. Most sources seem to agree that a gradual paradigm shift is preferable [47, 67].

Thus, the conclusion is that model-based engineering provides features and capabilities needed in industry, but more research is needed to facilitate its introduction. The model-based technologies must mature, and introduction of model-based engineering must be directed. Nevertheless, this is probably where the world of embedded and real-time system development is going. As the area matures, engineering principles will evolve much like they have done in structural and civil engineering etc. In the beginning, we were building houses and bridges without blueprint – today, when adding a room or story to a completed house, construction is always preceded by the modification of blueprints. Model-based engineering represents one such engineering principle, time and economics will probably conquer short sightedness and the lazy nature of humans, and then we shall finally witness the much awaited demise of code-oriented development.

2.3 Maintaining and evolving legacy systems

Industries around the world are continuously evolving, developing, improving, maintaining, and adopting their respective systems. From an industry point of view, this is a major issue (and has been for some time [1, 65, 120]). However, from academia, little attention has been given to these matters [1]. For example, according to their web page, the IEEE organized 759 conferences in 2005. Six (6) out of these had a clear focus on maintenance.[3] To put this in perspective, twenty nine (29) of the conferences had the word "Wireless" in their title.

2.3.1 Reverse engineering

Reverse engineering [16] is the process of creating an abstract representation of a legacy system, its components and their interrelationships. The objective of reverse engineering is to gain an understanding of the software in order to perform maintenance (see Section 1) or re-engineering activities. As the workforce that maintains the legacy system is changing over time, it is likely that the engineers that originally constructed a given part of the system are detached from the development when the time comes to restructure or re-engineer it. In fact, even if the original engineers are still involved, they are likely to have forgotten many of the intricacies of the system and its implementation. To obtain an understanding of the system and its construction, reverse engineering must then precede any effort to restructure or re-engineer a system. Similarly to the study of any arbitrary topic, it cannot be expected that reverse engineering (i.e. the study of the legacy system) can be automated. Spending time with the system, and exploring its structure and implementation is a time consuming process for which there is no direct substitution. However, we can construct

[3](1.) The Working Conference on Reverse Engineering, (2.) the Annual Reliability and Maintainability Symposium - Product Quality & Integrity, (3.) the 9th European Conference on Software Maintenance and Reengineering, (4.) the IEEE International Conference on Software Maintenance, (5.) the IEEE Workshop on Source Code Analysis and Manipulation, and (6.) the IEEE 13th Workshop on Program Comprehension.

tools and methods to ease the process and to make it more predictable. For example, the process can be structured to prevent ad hoc browsing and allowing accounting and planning of the time required to perform the activity, and many of the tedious tasks of collecting information can be automated to save time and energy as well as to avoid misunderstanding.

There are contradictory views on the terminology. For example, Richner and Ducasse [88] differentiate between *reverse engineering* based on static information (e.g. code), and *program understanding* based on dynamic information (e.g. recordings). Our view however, as should be clear from the paragraph above, is that the intuitive meaning of program understanding is rather a subset of reverse engineering. The difference is that program understanding is based on the code as input [84], while reverse engineering can be applied to any form of system representation [16].

In [15], Byrne defines the concepts of *refinement*, *abstraction*, and *alteration*. His main contribution is a conceptual framework that allows discussion about software *re-engineering* [16], which has reverse engineering, or abstraction, as a key activity. Byrne starts by introducing a layered, refining, model of software development consisting of four abstract *stages*: Firstly, at the *conceptual stage*, general terms describe the system. Second, in the *requirements stage*, these general terms are transformed into requirements for the system. Third, in the *design stage*, requirements are mapped to architecture and data-structures. Fourth, in the *implementation stage*, the system is realized. The information content and the sheer amount of information increases for each step. Based on these four stages, Byrne formulates a set of properties for software development. For example, information in one stage influences information in lower stages, but never in higher stages. Also, characteristics of the system are created at a particular abstraction stage, and are then propagated to lower stages. Using this abstract model of software development, he then introduces the refinement, abstraction, and alteration concepts to describe how the flow of information can be reversed.

In Byrnes model, refinement is a process of replacing existing system information with more detailed information. Abstraction is a process of successively replacing existing system information with more abstract information, but also to emphasize certain system characteristics by suppressing others. This follows by the principle of creation of characteristics related in the above paragraph. Finally, alteration is an optional process that allow the introduction of changes within one stage. Alteration is optional, and could be seen as a shortcut for abstracting to a higher stage and then refining down to the original stage. Based on these three concepts, Byrne describes a set of strategies for re-engineering.

2.3.2 Model-based impact analysis

One important motivation for our work is that system designers can use models to prototype future design propositions [4, 6, 12, 64, 90, 100] in order to detect side effects (see Figure 2.2). First (not shown in the figure) a valid model of the existing system is generated. Second, a design proposition is constructed. Third, the generated model is manually modified to reflect the proposed modifications. Fourth, the properties of the modified model are analyzed. If the analysis show no evidence of problems with the design proposition (i.e. requirement violations), the design is deemed feasible to implement. Otherwise, a new design proposition is formed, and the process can start over. We label this *model-based impact analysis* [4], the purpose of which is to avoid bad designs without actually implementing them in the system. The assumption is that it is significantly easier to implement the change in the model than in the system. As model-based impact analysis can reduce the time spent and wasted on bad designs, early identification and rejection of infeasible design alternatives has the potential of improving quality and substantially reducing the cost of maintenance (i.e. the adaption of existing software to make it conform to changing requirements).

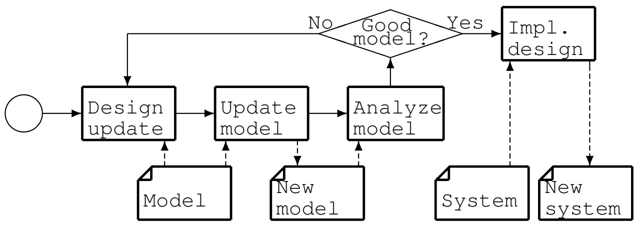


Figure 2.2: Model-based impact analysis.

We can identify the following set of properties that must be fulfilled in order to allow model-based impact analysis:

- the model must be a temporal and functional abstraction of the system,
- changing the model must be easier than changing the system,

- it must be possible to analyze whether the behavior of the changed model violates requirements posed on the system, and
- the changed model must be valid with respect to the changed system.

The last point above stresses an important issue: For correct model-based impact analysis, apart from that the original model must be valid with respect to the system, it is essential that the changed model is valid with respect to the changed system. *Changing the model should mimic changing the system.* Otherwise, the analysis of the changed model will have no bearing on the feasibility to introduce the proposed change in the system. Of course, since the objective of model-based impact analysis is to only implement an abstraction of proposed changes in the system, it may seem difficult to assess the general validity of the changed model with respect to the system; as the model should be an abstraction, one of the key properties of the model is that it should not be equivalent to the system.

We can formulate our requirement loosely: Under perfect conditions, a feasible change to the system is never deemed infeasible by model-based impact analysis of the model, and an infeasible change to the system is never deemed feasible by model-based impact analysis of the model.

In order to meet this requirement to as high degree as possible, we should formulate a set of properties with respect to which we intend to analyze the changed model. In the case of real-time systems, response times might seem like the natural choice. We should then strive to build confidence in that the response times of the changed model will correspond to those of the changed system. We do this by ensuring that our modeling capabilities are sound in the sense that the model is stable under change and that the modeling of the change is realistic. The evaluation framework presented in Chapter 8 can be used to assess the first of these two parts; that the model is stable under change. The second part is essentially left to the engineer performing the model-based impact analysis.

2.4 Recording the execution of real-time systems

Concerning recording, the following terminology is assumed:

By inserting *probes* into the system, we can *monitor* the *events* that occur during execution. The output of monitoring can be *logged* to facilitate offline analysis of the execution. Monitoring and logging are grouped into the activity *recording*. Our method for model extraction is dependent on recording to produce input to the process, but as recording costs resources, the amount of recording should be minimized.

Apart from being a part of dynamic model extraction as defined in this thesis, recording can also be part of tools for debugging [35], performance analysis, testing [109] etc.

2.4.1 Approaches to monitoring

In this section, we will discuss and compare three different basic approaches to monitoring: software, hardware, and hybrid monitoring.

Hardware

Hardware monitoring mechanisms are tailored devices that need to be adopted to the target system, which suggests that this is a rather expensive approach. On the other hand, they do not have to intrude at all on the device functionality of the monitored system [111].

Basic approaches to hardware monitoring include *bus snooping*, i.e. eavesdropping on information sent over buses or networks. The quantities of messages, and their size, result in large quantities of data that must be handled. Another problem [111] with hardware implementations is that they often look at very low level information. Hence, the data that is visible has low information content relative to the program execution. That is to say that a single bus message can not say much about the execution of a program, whereas (for example) the name of the current state can say a lot about the traversing of a state-machine. It is then up to off-line methods to interpret the collected information that is output from the recording process, correlate it to the system software and hardware, and translate the result into a format that is understandable to humans [53]. Needless to say, the amount of information may be quite extensive, but this problem is more or less inherent in the recording methodology as a whole. Also, implementations, and to some extent even solutions, are platform specific. Furthermore, advances in hardware technology make it more and more interesting to integrate solutions to a single chip, so called System-on-Chip (SoC) solutions [111]. SoC solutions are not observable as they limit the insight to the internals of the system, and it is therefore more difficult to construct hardware implementations for these systems provided that they are not incorporated on the chip [53]. A solution could be to move also the monitoring into the chip, but this is approach is of course only available to the designers of the device.

In their work on a “non-interference monitoring and replay mechanism”, Tsai et al. [112, 113] present a hardware solution for monitoring by bus snooping. In their solution, they use a duplicate processor that executes in parallel with the target. At certain points, the duplicate processor is frozen and its state is logged - that state can then be used during replay to start the replay from. Even though they claim in the title of their papers that their method provides these services without interference of the target environment, they do point out that they require the use of one occurrence of an interrupt to synchronize the two processors at the start of the monitoring session (which is not necessarily identical to the start of the system).

Boundary Scan IEEE Standard 1149.1 defines test logic [43]. The standard is a result from work by the Joint Test Action Group (JTAG).[4] The Boundary Scan method can be used to test Integrated Circuits (IC’s), interconnections between different assembled IC’s, and to observe and modify the operation of an IC. However, the Boundary Scan interface, through which data of all monitored events is to be fed, is a serial interface with a large shift register, a solution that incurs large temporal penalties.

In their article “Emerging On-Chip Debugging Techniques for Real-Time Embedded Systems” published in 2000 [72], MacNamee and Heffernan discusses the issue of On-Chip Debugging (OnCD) with a state of the practice point of view. OnCD has the capability of addressing the problem of monitoring the executions of complex processor architectures, especially those with on-chip caches, as it uses monitoring hardware that reside inside the components. However, solutions available today lack real-time capabilities in, e.g., memory monitoring (an example is the Motorola ColdFire). The lack of real-time monitoring of memory resources can be explained by the fact that real-time monitoring requires the monitoring mechanism to be prioritized over the application, thus leading to intrusive monitoring.

Logic Analysers are often used to monitor the behavior of hardware components. There are many devices available on the market. They have the capability to hook on to, and monitor, buses that transport data or instructions between physical modules of a system. On the positive side, logic analyzers are not necessarily intrusive on the target functionality, not even in the temporal domain. However, traces available are very low-level, and not all required information may be available. Systems that have integrated designs, perhaps with on-chip caches, or even multiple processors on a single chip, do not pass all required information on buses that are physically available for the logic an-

[4]The group has a homepage at www.jtag.com.

alyzer [53]. But the fact still remains that logic analyzers are used in many commercial projects, and even though they cannot solve all problems, or even provide good solutions to all of the problems that they can solve, they are among the better solutions commercially available.

Several of Motorola's MicroController Units (MCU's) support the Background Debug Mode (BDM) [33] interface. BDM is utilized in their EValuation Board (EVB) products that facilitate remote debugging of the MCU's. The BDM interface allows a user to control a remote target MCU and access both memory and I/O devices via a serial interface. BDM uses a small amount of on-chip support logic, some additional microcode in the CPU module, and a dedicated serial port. The BDM interface provides a set of instructions that can be issued in order to examine the state of the device. Instructions may be either hardware instructions, in which case they are not necessarily very intrusive on the functionality of the device, or they may be firmware instructions, which are intrusive. Hardware instructions allow reading or writing to all memory locations of the device, these operations are initially given the lowest priority, i.e. they are only executed if no other instructions are pending, but a fairness policy is used if the instructions are not issued within a predefined time. Firmware instructions must be issued in a special firmware-mode, and then the debugger can read and write registers on the device.

Motorola also provides an On-Chip Emulation (OnCE) interface with some models. This interface combines features of BDM and JTAG.

The Nexus 5001 standard [72, 45] describes a hardware solution that supports monitoring of embedded systems, it also supports super scalar and pipelined architectures.

Software

Software monitoring can either be performed at system or at task (process) level [111]. Monitoring at system level enables the monitor to see operating system specifics in the system. It is possible to view many of the data structures that affect system performance, such as Translate Look-aside Buffer (TLB) entries that describe the mappings between virtual and physical memory, also task control blocks, semaphore queues, and many other data structures are visible. Issues related to the control flow of the system that are visible on system level include interrupt occurrences, task switches and paths through code within system-calls. Monitoring at the task level will not allow monitoring of these, but other possibilities are open, such as events related to the specific task that is monitored. Concerning the data flow, we can monitor local and global variables, and of the control flow, we can monitor the execution flow through a program.

Thane [108] describes four architectural solutions for software monitoring: *kernel-probes*, *software-probes*, *probe-tasks*, and *probe-nodes*. Kernel-probes can monitor operating system events such as task-switches and interference due to interrupt occurrences. Software-probes are additions to the monitored task, they are auxiliary outputs from that task. Probe-tasks have as their only functional objective to monitor other tasks, either by cooperation from software-probes, or by snooping shared resources. Finally, probe-nodes are dedicated nodes that either snoop the communication medium used by other tasks, or receive input from either software-probes or probe-tasks.

Stewart and Gentleman [102] recommend the use of *data structure audits*, a construct which is also described by Leveson in [63] where it is also referred to as *independent monitoring*. An auditor could for example check whether a data structure is self-consistent, or simply monitor its changes. Auditing can be performed by a probe-task, also known as a spy task, and can be a more or less complex operation.

Hybrid

According to Tsai et al. [111] hybrid monitoring come in two flavors, *memory-mapped*, and *coprocessor* monitoring:

Memory-mapped monitoring uses a snooping device that listens to the bus, and reacts to operations on certain addresses. These addresses may either be snooping device registers that are memory-mapped into the address space of the task, or just a dedicated RAM area. Each event that should be monitored is forced to make a memory operation on the address that is associated with that event, which will allow the monitor to detect its occurrence.

Coprocessor monitoring uses a device that is a coprocessor to the processor that executes the application that is to be monitored. Events are forced to issue coprocessor instructions to the coprocessor as the events that are to be monitored will occur. The coprocessor monitoring approach requires, of course, that the architecture targeted allows the use of coprocessors.

From Applied Microsystems comes the CodeTEST Trace Analysis tool that provides hardware assisted software based tracing of program execution. An extra stage is inserted into the compile stage where unique tags are added to the program code according to some parameters (thereby leaving the original source code unchanged). A database is also created to relate the unique markers to specific lines of code.

Depending on where in the development stage the system is, different solutions are then used to collect information from the execution. Early in the design process a collection task that forwards the information to a remote host is run together with the normal task set; later in the process, tags are modified to only perform a memory write to a dedicated area, a hardware probe that can snoop the bus is then used to collect the information and send it to the remote host.

In "A Hardware and Software Monitor for High-Level System-on-Chip Verification" [96], El Shobaki and Lindh present a method for recording the execution of SoC's with a built in hardware component named MAMon (Multipurpose/Multiprocessor Application Monitor). The MAMon component is integrated with the design, and allows both hardware and hybrid monitoring. The MAMon component can be used both with software based and hardware based [66] real-time operating systems. In the case where the operating system is hardware based, task information can be extracted non-intrusively from the kernel. However, integration of MAMon into a SoC-system is only available to the hardware-designer.

2.4.2 The probe effect

It is important to note that recording of a running system has its limitations; it is not always possible to observe without interfering. If probes are added, removed, or altered over time, so that the level of perturbation that they cause varies between executions with otherwise identical premises, the system may react to the variation and thereby change its behavior. For example, this may lead to that the balance of race conditions is shifted so that one entity wins more often than before (imagine what this would do in, e.g., a bus protocol such as Ethernet – some nodes would get better service at the cost of service for the other nodes – similar consequences will result in the interactions between tasks). Thus, such a change may invalidate previous verification efforts.

The *probe effect* [23], which is another name for *Heisenbergs uncertainty principle* (a.k.a. Heisenbugs [91]) when applied to software engineering [60, 76, 99], can become visible when code is added or removed to a system, when breakpoints are used to debug the system, or when the system is modified in some other way that will affect execution times. Modifying the system in any way may alter the timing in the system. Extra code will require extra resources, the removal of code will free resources that can be used by tasks that would have been blocked, and modifications to data may change the program flow. Differences in the temporal behavior may in turn lead to that the modifications have a different result on the system performance than expected.

It is quite convenient to use real-time systems when exemplifying the probe effect. Imagine a system of two tasks that compete for execution resources, where some synchronization problem exists between the two tasks. Say that the two tasks control an external process, but that one of the tasks occasionally issues control commands too soon after that the previous task has issued a command, thus preventing the previous command from affecting the external process as intended. This would have lead to a failure, and a debugging-effort is launched.

In order to debug the system, we would like to probe into the state of the tasks so that we could determine the cause of the problem. However, if we implement this probe by inserting some *auxiliary code* (code that does not aid the progress of the system) that will monitor the system, that code will effect the system. If we are unlucky, it will do so in such a way that the time between the two control commands is lengthened, thus causing the bug to disappear during some executions which may very well be just that subset which we examine. If we then remove the probes, the bug will reappear. Also the opposite is possible, by adding probes to a system, we may cause errors to appear that where not previously present. Of course, also a combination of the two is possible, by adding probes to the system, we may remove one error, only to invoke another.

The last example is perhaps the most intriguing, we may then find ourselves identifying the wrong bug, and correcting that one instead of the real one. This problem should be detected by a regression testing procedure. The probe effect may, however, be ignored if the probes are allowed to remain inside the release version of the program [111]. Not all systems can afford this though.

Recording is not the only situation in which the probe effect may effect the system, it is also possible that modifications to old systems, or bug-fixes, cause the same problems. One may view it as that the removal of code is equivalent with removing a probe from the system, and that adding functionality can cause the same problems as adding a probe to the system. A general rule is that if the source code is modified, probe effect related problems may arise.

There are however two exceptions to this rule.

Schütz notes in “Fundamental Issues in Testing Distributed Real-Time Systems” [95] that it is possible to remove code if the only consequence of the removal is that the idle task of the system will receive more execution time. However, this is rather hard to ensure unless the system is time-triggered. Schütz states that, in a time-triggered system, provided that the scheduled execution slot of the task that is to be removed is not adjacent to the slot of any other task (except the idle task), the task is in a *temporal firewall*, and may be removed without consequence to the remaining system. This is provided of course that the task does not perform any work that is used by other entities in the system.

The second exception has been noted by Thane [108] concerning fixed priority scheduled systems. Thane starts with the same premise that Schütz did; that code can be removed if the only consequence of the removal is that the idle task of the operating system receives a larger percentage of the total system execution time. He then states that this requirement is satisfied if the task from where the code is removed has the lowest of priorities among the tasks in the system (apart from the idle task) and it is established that the task never blocks the execution of other tasks remaining in the system. Thus, the task from where the probes are removed cannot control mutual exclusion or communication primitives, such as semaphores or other, shared with tasks remaining in the system. The use of schemes such as direct inheritance or similar for deadlock avoidance will limit the use of such primitives even further.

Note that these solutions are only feasible under the assumption that the operation of the hardware (instruction pipelines, caches etc.) is not affected by the removal of the probes.

One approach to avoid the issue of the probe effect is to reduce the overhead of recording, this could be achieved by deploying one or several of the following techniques:

- Selecting the set of variables that are the least demanding to record.
- Using logging algorithms that optimize logging in space or time requirements (e.g. ECETES [40] or memory excluding checkpoints [86]).
- Choosing read/write phase in which to probe the state of the variables such that the workload is lower (e.g. only on read if writes are frequent to the variable).
- Choosing system phase in which to probe the state of the variables such that the workload is disguised (e.g. recording while the system is idle).

2.4.3 The correlation and the observability problems

In “Fundamentals of Distributed System Observation” published 1996 [21], Fidge describe the problem of obtaining a truthful view of the events in an observed system. For example, as a distributed system is being observed, if the observer cannot be tightly coupled with the system it is observing, problems related to the observers apprehension of the ordering of events on different nodes may occur. Depending on variations in the propagation time of observer notifications, the ordering of events may be confused. We shall refer to this as the *correlation problem*.

According to Fidge, we may divide the correlation problem into at least four sub-problems [21]: (1) multiple observers may see different event orderings, (2) observers may see incorrect orderings of events, (3) different executions may yield different event orderings, and (4) events may have arbitrary event orderings. All are more or less results of the absence of an exact global time-base, and/or the fact that network propagation times are not constant. Because of the lack of a exact global time, we cannot rely on any time-stamp taken at the node where the event occurred, if the observer is situated on another node.

1. In a system where many observers are used, different observers may see different event orderings, because the propagation of the event notification requires different time to different destinations.

2. As the propagation through a network may differ between two network packages, a package that is sent after another may arrive earlier. Thus, if two events occur on different nodes at different times, the notification of the last event may arrive at the observer before the first notification has arrived, thus erroneously implying that the last event occurred before the first.

3. Because the clock rate of each node will diverge slightly from the ideal clock and the other clocks in the system, and the rate of that deviation partly depends on environmental aspects, different invocations of a distributed system will differ.

4. Some of the events in the system are unrelated, and may therefore be allowed to occur in arbitrary orderings. The problem with this is that an observer must know and recognize that, as different tests are run, it is allowed to have differing orderings between some of the events.

Item number (4) in the list above is related to Polednas Ph.D. dissertation “Replica Determinism in Fault-Tolerant Real-Time Systems” from 1994 [87]. Poledna direct the problem of *replica determinism* when using redundancy as a mean to increase the fault-tolerance of a real-time system. In other words, he directs the problem of ensuring that two components, that are supposed to perform the same task, have the same behavior when they are operating correctly. This is related as (4) describe that we must be able to correlate executions that are temporally differentiated and Poledna does the same for spatially differentiated executions.

Schütz discusses a subject which he calls observability [95]. He states that a system must be *observable*, meaning that it must be possible to extract sufficient information from the system. What is "sufficient" is determined by the intended use of the observations. In this thesis, we shall refer to this as the *observability problem*.

2.4.4 Measuring execution time

To use recordings, it is often important to relate events to software execution. It must be possible to state how much execution resources a task has consumed between two records in the log. There are at least two ways of doing this, one is to use a hardware platform which supports instruction counting, cycle counting or similar, the other is to use a software implementation.

An example of a hardware solution is implemented in the Intel x86 architecture. A processor cycle counter is accessible through the use of the assembler instruction `RDTSC`. Note however that this implementation is not reliable in architectures such as Pentium II, Pentium Pro, and onwards. The reason therefore is that more advanced models in the x86 family use out-of-order execution which can lead to pessimistic or optimistic measurements.

In their article "Debugging Parallel Programs with Instant Replay" published in 1989 [77], Mellor-Crummey and LeBlanc present a method that can instrument assembler-code with counters, thus enabling the counting of executed instructions, the method is called Software Instruction Counter (SIC). The authors note that the code of a program consists of short chunks of sequential code, called basic blocks, and conditional, or unconditional, connections between some of the basic blocks (by branches, jumps, or function calls). These one-way connections can either connect a basic block with a later (with higher address-value than the present), a forward branch, or with a prior block, a backward branch. To uniquely mark each instruction instance that is executed, the authors state that a combination of the program counter value and the number of backward branches required for the execution to reach the instruction from a known starting point is sufficient. They can therefore construct a low-cost software-based instruction counter, which only resource requirements are a small computation overhead, and a reserved data-register that is used solely for performance reasons.

In distributed systems, clock synchronization [55] becomes a problem if nodes must have a global order of events [35]:

As events occur on concurrent nodes, some system architectures cannot produce a correct order between them. If this is a requirement, some measure must be taken. Tightly coupled parallel systems, and multitasking single-node systems, are able to produce a correct ordering because all system entities depend on the same real-time clock [111]. But, because of the correlation problem (See Section 2.4.3), distributed systems can only make weak assumptions about the ordering of events provided that they do not use an algorithm for global clock synchronization.

Ordering of events can be either *partial*, or *total* [111]. Where partial order describes the local sequence of events (in our context locally is on a specific node), and total order describes the global order of events. Thus, unsynchronized systems cannot determine the exact total order of events, but they may be able to find an estimation of the global order by using a method for clock synchronization or logic clocks [59]. Using any of these will inflict an additional overhead on the system.

2.4.5 Operating system support

Gathering of information is often a difficult task. In the research described in this thesis, we need to record occurrencies of context switches and system calls. We even need to record the occurrence of the start of a system call and the end of a system call. Monitoring context switches is significantly easier to solve if the operating system provides an interface for doing so. Also monitoring system calls can be helped by this as it reduces the possibility of introducing bugs in the recording if only one system level probe per system call is used as opposed to one task level probe per potential call. For example, in order to monitor system calls or context switches in Windows 2000/NT, one must know what functions to look for [71]. In view of this, it is interesting to survey the support for probes in some available operating systems.

The *Solaris 10* operating system provides *DTrace* [103], abbreviating Dynamic Tracing, which can be used to monitor a vast amount of events in the system. A programming language is provided that allow specification of: which events to probe, predicates for each probe that must be fulfilled in order for an event to be probed, and what to do when a probe is triggered. Thus, a piece of code can be made to execute at the occurence of a named event. At the time of occurrence, individual events can be recorded. Examples of events that can be probed are context switches and system call entry and exit points.

VxWorks by WindRiver [118] provides an interface to register *hooks* for selected events. These hooks can be used to monitor context switches etc. But the operating system does not facilitate the probing of system calls.

SMX (Simple Multitasking Executive) by Micro Digital [78] provides hooks for context switches. In the case of SMX, the hooks are task specific for entering and exiting a task switch.

OSEck by the Enea company OSE Systems [83] provides hooks for both context switches and selected system calls (send and receive).

During the work to compile this survey, we also investigated the Symbian and the QNX real-time operating systems, but failed to find a similar functionality there.

2.5 Testing

It is well known that testing has some fundamental limitations to the set of problems that can be solved by testing. For example, testing is performed under a set of premises and results of testing cannot easily be extrapolated to a different set of premises. Also, testing can only prove the existence of bugs, not their absence [19]. Nevertheless, regardless of these limitations, testing is a good technique for examining the operation of complex systems. The method of model extraction presented in this thesis use recordings obtained during testing to model the system.

2.5.1 The completeness problem

Kranzlmüller provides the following definition of a nondeterministic program in his Ph.D. thesis [57]:

> *"A program is nondeterministic, if - for a given input - there may be situations where an arbitrary programming statement is succeeded by one of two or more follow-up states. This freedom of choice may be determined by pure chance or unawareness of the complete state of the execution environment."*

Meaning that if one evaluation of a set of inputs may cause a task to, from one run to the next, behave differently, then the system is nondeterministic. Note that, according to this definition, a program is nondeterministic also if the irregularity of its products is completely depending on factors that are unknown but not necessarily unpredictable. Thus, a deterministic system can appear to be nondeterministic just because we lack the knowledge to understand it.

The opposite of a nondeterministic program or system, must clearly be a deterministic program. In the book “Communication and Concurrency” by Milner [79], the issue of determinism has been formally defined.

During testing, in order to ensure that a system complies to its specification, it is required that the testing procedure is performed under realistic conditions. Two properties that must be tested are that the system reacts as intended on different input data, and (in the case of real-time systems) that the temporal behavior of the system satisfies the requirements. As different invocations of a nondeterministic program, per definition, can behave differently even though all controllable inputs are identical in all invocations, it is difficult to determine the coverage of testing procedures. We state that a *completeness problem* exists in testing: it is difficult to determine the coverage of performed testing.

2.5.2 Test coverage

Testing the complete set of possible combinations of known input data and all execution orderings is normally referred to as *exhaustive testing*. Even in a very small system the number of test-cases is very large, and it increases drastically as the system grows. Therefore, exhaustive testing is normally not an option as it would require too long time[5] to perform. The alternative is to only test a subset of the input combinations, which leads to that only a certain level of *confidence* may be ascribed to the system's functionality or capability to fulfill its specification. The level of confidence relates directly to how well the system was tested, i.e. the *coverage* of testing. It is true that small parts of the system, that are considered as especially important, could be selected for exhaustive testing. This would of course increase the confidence in the system, but is directly comparable to testing only a small subset of the possible input combinations to the system.

In multitasking systems, the completeness problem implies that even if the system would be tested with all possible combinations of data inputs, bugs may still remain because different execution orderings in the system also affects the output and temporal behavior of the system. If the number of possible execution orderings are unknown, it may be difficult to determine the level of confidence that can be ascribed to the system's functionality. Thane et al. discuss this problem in [109] where they propose a method for testing real-time systems. The method describes how all possible orderings in a system can

[5]Consider a program that subtracts one 32-bit integer from another, it would require $(2^{32})^2$ test-cases. If one test-case can be run each nano-second, that would result in $(2^{64} \cdot 10^{-9})/(60 \cdot 60 \cdot 24)$, or approximately 200'000, days of testing.

be identified, how all sequences of interleaving due to interrupts, blocking by semaphores, or scheduling decisions can be listed. They can then group a particular monitored execution with an execution ordering. By running a sufficient number of tests and relating each test to its ordering, it is then possible to increase the confidence in the orderings that become subjected to testing.

However, this approach would either cause some of the less probable execution orderings to be insufficiently, or excessively, tested, due to the improbability, or probability, of experiencing those orderings. Therefore, reproducibility in the testing should be ensured by enforcing execution orderings during testing. By performing a sufficient amount of tests of a sufficient number of orderings, the confidence in the system can be calculated based on the confidence in each ordering. In their articles, Thane et al. states that the number of execution orderings, and therefore also the testability of the system, is directly proportional to the number of preemption points and the jitter present in the system. Note that the confidence in a system according to Thane et al. can be a 2-dimensional property, a confidence in each execution ordering, and a confidence in covered execution orderings.

However, it is difficult to obtain the exact number of orderings and the total number of unique executions for each respective ordering for a given system. Further, due to the observability problem (see Section 2.4.3), special constructs are required to sort performed executions based on ordering. Often, we are forced to measure the coverage of testing using other factors.

Zhu et al. [121] present an excellent survey of test techniques including some of the usual units of coverage measurement: statement coverage, branch coverage, path coverage, and mutation adequacy. Of these, the first two are intuitive, the third measures the percentage of possible paths between two points in the execution that where tested, and the fourth measures the percentage of inserted artificial bugs that are found during testing.

2.5.3 Test-case selection

To test, one must have a system to test, but also a setting in which to test it: A set of test-cases must be realized that can evaluate the ability of the system to comply to its expectations. Normally, a test-case consists of a combination of input to the system *and* the expected output. The later is needed in order to evaluate the outcome of the test. According to Zhu et al. [121], test-cases can either be based on the system specification (i.e. expectations or requirements in plain text, temporal logic etc.), the system implementation, or both.

Mandrioli et al. [73] provide a method for *automated test (case) generation* for behavioral models specified in a form of temporal logic. Hong et al. [32] show how test generation can be formulated as a model checking problem. Hessel and Pettersson [29] present an UPPAAL-based method for test generation using model checking. They assume the existence of timed automata models of the system and its environment, and can generate test-cases given a measure of coverage.

2.6 Discussion

In this chapter, we have introduced a number of different topics. These have all implicitly or explicitly influenced the technical presentations in later chapters. The current chapter can therefore be seen as a form of reference material for the coming chapters, as an orientation to the mindset of the author, and as required reading for those not familiar with recording etc.

Chapter 3

Related work

In this chapter we will discuss related work within the area of (semi-) automatic modeling. A number of methods for automatic modeling have been presented in the literature, but their scope and their assumptions vary significantly. In order to describe this difference, we introduce a classification of techniques for (semi-) automatic modeling. The classification is intended to aid in portraying the differences of the surveyed methods.

3.1 A classification of automatic modeling techniques

This section will introduce a classification scheme for model extraction and related activities. We have identified a matrix of four dimensions:

1. Model properties
2. Intended use of the model
3. Interactivity level
4. Type of model extraction

Each dimension has a set of feasible options, which are not necessarily mutually exclusive. For example, model properties can be both architecture and behavior descriptions.

Of course model extraction is closely related to the end product (the model) and the language in which that model is specified. Thus, the abstractions that the modeling language provides must be reflected in this classification scheme.

Note that the classification scheme is not intended to be complete to the extent that it becomes a taxonomy, but sufficient enough for our purpose of relating the work referred here to our contributions.

3.1.1 Model properties

The model properties describe what type of information is contained in the model. Note that the model properties are independent from the type of model extraction. Model properties may be one or several of the following:

- Architectural description
- Behavioral description
  - Operational model
    - Complete view
    - Partial view (e.g., not all modes of operation)
  - Non-operational model
- Temporal description
- Environment model

An *architectural description* defines entities in the system at some level of abstraction, and describes how these can potentially interact. A *behavioral description* defines the components of the system at some level of abstraction, and describes the abstract behavior of these during runtime. A *temporal description* describes the time needed to perform some amount of computation described by the model. An *environmental description* presents a model of the environment in which the system operates.

Regarding the behavioral model, this can be either an operational or a non-operational model. Non-operational models are defined as: models that are not *"sufficiently articulated in a form suitable for codification or automated processing"* [74].

3.1.2 Intended use of the model

Among the uses that we can see for extracted models are:

- Formal verification
- Simulation
- Debugging
- Testing
- Model-based impact analysis
- Study/clarification
- Performance evaluation
- Refinement

As in all modeling, the intended use of the model determines what type of information that must be contained in the model; thereby, the use poses a set of requirements on the modeling language.

In this particular classification however, we let the intended use of the model describe also properties such as the required input and the methods of model extraction that are feasible to use. For example, it is not possible to use a model for simulation if the model is not operational. Thus, the intended use says a lot about the mechanisms behind the immediately visible.

For each member in the set of uses that we have identified, we have formulated a set of properties that could facilitate that use:

There are many ways to perform *formal verification* on models, but in general the behavior expressed by the model, as well as the constraints posed by the model, must be formally defined. It must also be possible to express the requirements that are posed on the implementation. Imagine that we would like to prove that a model can never come to deadlock; this requires a model that can express the *limitations* of the implementation – there must be a notion of *absence* of action.

The use of a model for *simulation* requires that the model is operational and the environment must be specified or must be possible to specify.

Debugging is an activity that can take place on various levels of abstraction; after all, most things can go wrong and need debugging. Thus, the model can either just be used for debugging at a given abstraction, or must be organized for information hiding, since efficient debugging at a given level of abstraction requires hiding of information irrelevant for that level.

Testing requires the use of operational models with behavioral descriptions. Further, the models must be conservative in the behavior they describe; models allow for abstraction, which may lead to false positives, but efficient testing requires that we can determine if the test-case could actually lead to a failure in the implementation – the model must provide this correlation to the implementation.

In order to facilitate *model-based impact analysis* (see Section 2.3.2), the model must be modifiable. As the exact nature of the planned impact may be unclear, a notion of probability in the modeling language is useful.

Clarification is facilitated through documentation, which requires two properties of the model to be preserved: the structure of the implementation, and the naming of variables, functions etc.

To facilitate *performance evaluation*, the model must be able to express the resources that the computation uses in a given setting and there must be a way to correlate this amount to both the expected resource requirement, and the amount of available resources in the given setting. In this way, it is possible to control that the implementation performs as expected, or to find bottlenecks in the system.

Finally, *refinement* allows an existing model to be extended with additional information retrieved by model extraction; typically, this is performed as a hybrid model extraction that, e.g., adds information about observed execution time to an available model.

From this list of activities, more complex activities can be formulated, for example, *maintenance* may include testing as well as debugging.

3.1.3 Interactivity level

The method of model extraction can be:

- Interactive, or
- Non-interactive

Model extraction may, depending on the type of solution in relation to the model use, be a too complex task to complete. Thus, some solutions allow the user to influence the procedure with the intention to make the model more accurate. We refer to these methods of model extraction as *interactive*.

For example, if ambiguities are discovered, these may be resolved by human intervention. The user is presented with a set of possible interpretations that the model extraction has identified, and is required to pick one and only one of the interpretations.

Further, some methods require the user to specify translations for constructs identified in the implementation, or to specify what parts of the input that are of interest to the model extraction. These techniques are used to define the level of abstraction of the model.

In any of these situations, two users may choose different translations, which would lead to non-determinism in the model extraction as it leads to two different models.

3.1.4 Type of model extraction

The mechanics of the model extraction is heavily dependent on the type of input that is used:

- Dynamic model extraction, based on:
  - execution trace with system-level information
  - execution trace with task-level information
- Static model extraction, based on:
  - source code
  - extended source code
  - compiled binary or bytecode
  - other models, which are:
    - operational
    - non-operational
- Hybrid model extraction (i.e., a combination of the above)

Generally, model extraction can take input from either runtime information (e.g., *dynamic model extraction*), or from information available even if the system has never been executed (e.g., *static model extraction*). We can also envision a hybrid of the two; for example, static model extraction can provide a skeleton that dynamic model extraction will refine by adding runtime information from a specific hardware platform.

Regarding static methods; due to the dependence on specification or source code as input, it seems reasonable to assume that static methods are dependent on the language used for the implementation of the run-time system, and may thus encounter difficulties when languages are mixed in the same system. We suspect that fundamental differences between languages (pointers vs. no pointers, object oriented vs. imperative, etc.) may limit the generality of such methods.

Dynamic methods, on the other hand, can only model the behavior that has actually been observed, which is likely to be only a subset of the valid system behavior (compare to the completeness problem at Page 30). Further, the extracted model is dependent on the observations of the running system – if no observations can be made, no model can be created (compare to the observability problem at Page 28). Finally, dynamic methods depend more heavily than their static counterparts on the interpretations of the observations made and the deductions made from these interpretations.

It should be clear from the above discussion that static and dynamic model extraction each has different properties that makes them complementary to each other. Some methods integrate these two in order to reap the benefits from both solutions and avoid some of the drawbacks inherent to a more pure breed approach.

We conclude that methods can be static, in which case the input may come from source code of the implementation (possibly extended with some extra annotations to support model extraction), the binary of the implementation, and/or other models. In the latter case, these can be operational or non-operational. As an alternative, methods may be dynamic, in which case the input comes from recorded traces which may include data known only at operating system-level, and/or at task-level. Finally, static and dynamic methods can be combined into a hybrid approach.

3.2 Applying the classification

In this section, we apply the classification to related work in the area of model extraction.

3.2.1 From UML to SDL

Bastos and Sanches [9] proposed a static method that, based on UML (Unified Modeling Language) models, produces SDL[1] models. The authors motivate their work by noting that development of object oriented, safety critical real-time systems often require both UML and formal models. The method is inherently object oriented, and does require some additional input from humans.

According to our classification, the approach is:

1. partial, operational, behavioral and temporal descriptions
2. for formal verification
3. interactive
4. a static model extraction from non-operational models enhanced with a special purpose notation.

3.2.2 Reverse engineering to UML sequence diagrams

Briand et al. [13] propose a dynamic method to synthesize UML sequence diagrams. Even though sequence diagrams can be helpful in the effort to understand the system and verify that a known functionality is performing as intended, it is uncertain if the overhead is justified. For example, the representation, while able to describe the act of making a selection, can only describe the one path of the selection that was actually performed (e.g. only one path of the selection can be modeled in a given sequence diagram). Further, the model cannot represent state, and can therefore not be used in simulations etc. In order to justify the overhead, it should be shown that the amount of recording performed is sufficient to produce also other types of UML models of the system.

According to our classification, the approach is:

1. behavioral, non-operational
2. for study/clarification
3. non-interactive
4. dynamic model extraction with task-level information.

[1]Specification and Description Language, see http://www.sdl-forum.org.

3.2.3 Using Angluin's algorithm on real-time systems

Grinchtein et al. [26, 27] present an dynamic method for model extraction to time deterministic event recording automata (i.e. transition guards of the automata are mutually exclusive). Their approach is to view model extraction as a learning problem.

In *machine learning* [7] a *Learner* is concerned with hypothesizing a model from a system by asking a *Teacher* if the system accepts a given behavior trace (i.e. asking *membership queries*). Which membership queries to ask is given by inconsistencies in the Learner's currently available data; if a set of behavior traces that the Learner thinks are equal yields different answers when the Teacher is asked, an inconsistency exists. By reformulating the hypothesized model and asking a set of more detailed queries, these inconsistencies are resolved. When a Learner has obtained sufficient confidence in the correctness of the hypothesis (i.e. when all inconsistencies have been resolved), the hypothesized model is checked with an *Oracle* to determine its correctness (i.e. asking *equivalence queries*). If the model is incorrect, the Oracle will present a counter example that can be used to extend the model. Implementation issues involve realizing the Teacher and the Oracle. Implementing a Teacher may be prevented due lack of *reproducibility* in the system [108]. Regarding complexity: Since membership queries need to be asked until all inconsistencies are resolved, the number of membership queries can potentially be large. Clark and Thollard [17] has presented learning of probabilistic automata.

Grinchtein et al. conform to the original timed automata [3], which does not include any notion of data state. In the setting of this thesis, this is a limitation. The absence of data state will lead to that either the model is unable to represent causal dependencies with longer span than one job, or the model will be unnecessarily big. The complexity of learning a timed automata is high (exponential): Given two behavior traces such that their sequence of input symbols are equal but their timing is different, if these yield different answers on membership queries, the possible set of membership queries is large (it includes the behavior traces with the same sequence of input symbols but different timing).

According to our classification, the approach is:

1. temporal and partial operational behavioral description
2. formal verification
3. non-interactive
4. dynamic with system-level information.

3.2.4 From C to Promela – as used in Spin

Holzmann and Smith [30, 31] introduced a method called *Modex* (acronym for model extractor) for static model extraction. They describe static model extraction as a hierarchical process, which makes it very comprehensible. As a proof of concept, they present a one shot experiment on a large telephone application.

The approach advocated by the authors takes the source code of the implementation as input to Modex, which is intended to make the model accurate to the implementation; optimizing compilers, however, may break the logical chain between source code and implementation, which is why the executable implementation may be preferable as input to static model extraction.

Apart from the source code of the implementation, four additional types of inputs are required; these can be seen as non-operational models that are supplied as input. As the implementation evolves, the additional inputs must be maintained by the user. It is claimed that the maintenance of these inputs is a simple task, but that some updates to the input can take in the order of hours to complete.

According to our classification, the approach is:

1. producing architectural and behavioral descriptions that are operational
2. for formal verification
3. interactive
4. a static model extraction from source code and non-operational models.

3.2.5 Test-based model extraction

Hungar et al. [34] describe a dynamic method for extracting models from system level recordings. They use methods of automata learning (specifically, an adaptation of Angluin's algorithm [7]) to extract behavioral models without timing information.

According to our classification, the approach is:

1. partial operational, behavioral description
2. formal verification
3. non-interactive
4. dynamic model extraction with system level information.

3.2.6 LQN-models for performance evaluation

Israr et al. [46] present a dynamic model extraction from traces to Layered Queuing Network models. The traces concern only message sending and receiving, and the models are made to represent message transactions rather than computational entities (e.g. tasks). Thus, the finalized models can only be used to analyze the message exchange patterns. Even though the model can express time as elapsed between sending and receiving of messages, they cannot express the execution load. Neither does the model have any concept of data state.

According to our classification, the approach is:

1. temporal, partial operational, and behavioral description
2. performance evaluation
3. non-interactive
4. dynamic model extraction based on task-level information.

3.2.7 Jensen's method for UPPAAL-models

In his Ph.D. thesis, Jensen [49] introduces a method for model extraction to the timed automata of the UPPAAL-tool [11]. The intended use is for testing properties such as response time and model checking against implementation requirements. For the verification, it is assumed that the requirements are available in the form of timed automata which are then parallel composed with the extracted model by the UPPAAL-tool to allow model checking. The thesis includes a schedulability test that (instead of WCET) uses a measure labeled Reliable Worst Case execution time (RWC). RWC is a statistical measure that is introduced in the thesis. As a proof of concept, Jensen includes a one shot experiment of the model extraction. In relation to our method for model extraction, the approach is similar, but more restrictive and provides less detail; missing task-level information leads to that only system calls and elapsed execution can be modeled.

Jensen poses restrictions on how selections are used in the model – they can only occur at the start of the job or after a performed `receive()` system call.

In addition to the architectural and behavioral model, Jensen's method outputs an environmental model. It is however comparable to a playback of observed behavior – no elaboration is performed on the collected data. Jensen assumes a normal distribution of task execution times – as a selection where one alternative takes between $10 - 20$ time units (tu) to execute, and the other between $100 - 200$ tu describes the inappropriateness of this assumption.

According to our classification, the approach is:

1. producing architectural, behavioral, and environmental descriptions that are operational and describes temporal properties
2. for formal verification
3. non-interactive
4. a dynamic model extraction from system-level information.

3.2.8 From sequence diagrams to state machines

Koskimies et al. [56] present dynamic model extraction of object oriented systems using machine learning such as Angluin's algorithm [7]. They use non-operational trace diagrams (a.k.a. sequence diagrams) as input to extract state machines that will be operational if the trace diagrams allow this. As described by Systä and Koskimies [105], the trace diagrams can also be obtained from recordings of the executing system – thus, the method is hybrid according to our classification.

The technology that the authors rely on is called *inductive inference*; basically, it is about collecting a large (in the optimal case and infinite) set of examples that are then used to guess a rule (a model) – if the examples do not contradict the rule, it is assumed to be correct.

The examples used for inductive inference are trace diagrams that describe interaction between objects and internal actions of individual objects in a temporally correct order.

A limitation of this method is that it is unable to represent a modifiable state in the system in any other way than as states in the state machine. There is no concept of variables etc., which leads to that the number of states could be quite large – as the number of states in the model is a parameter for the computational complexity of the method, this could be a problem.

According to our classification, the approach is:

1. producing an architectural and a behavioral description in the form of state machines that may be operational
2. for clarification

3. non-interactive
4. a hybrid model extraction from non-operational models or recordings of task-level information.

3.2.9 Extending Esterel

Logothetis et al. [69, 70] proposes an extension to Quartz, which is a version of the synchronous language Esterel. The purpose of the extension is to separate the parts of the implementation that are required in verification from the parts that are not – thus, as two different users could produce two different models, the method is interactive.

The method requires that the user augments the code of the real implementation with information about what parts that need not be expressed by the model. While this allows the user to have control over the level of abstraction provided by the model, it also poses a large overhead to the model extraction if the project is of industrial proportions.

According to our classification, the approach is:

1. producing operational architectural and behavioral descriptions
2. for formal verification
3. interactive
4. a static model extraction from extended source code.

3.2.10 Commercial soundness for CORBA-systems

Moe and Carr [80] present dynamic model extraction for CORBA-based systems. The work has been heavily influenced by the demand for a commercially sound result, meaning that (while the usefulness should be other than marginal) the impact must be low on both users and system performance; thus, recording must be efficient but transparent and the interface must provide a low learning threshold. The method is introduced in industry and used in an operation and maintenance system for cellular networks by Ericsson.

Using the CORBA notion of *interceptors* to record performed RPC-calls Moe and Carr obtains a transparent recording that can be modified dynamically. Note that the relevant probe effect (see Section 2.4.2) is not directed. The log of recorded RPC-calls is parsed offline, and call-response sequences and summary statistics that characterize the operation of the system are obtained. The published material lacks details on how this parse is performed.

The product of the method is not an operational model, but rather a visualization of occurred events. The view can be modified by the user to focus on the task at hand (e.g., during a given time interval, what fraction of calls to a given RPC-service that where terminated by an exception). This is reported to have helped discover and identify a number of bugs in real systems.

According to our classification, the approach is:

1. producing architectural description
2. for performance evaluation and debugging
3. non-interactive
4. a dynamic model extraction from system-level information.

3.2.11 Query based reverse engineering of Smalltalk implementations

Richner and Ducasse [88, 89] present a hybrid approach that use both statically and dynamically obtained data to extract models of object oriented non-real-time systems implemented in Smalltalk. This will provide for a more comprehensive picture than a strictly dynamic or static approach. However, their modeling language of choice cannot describe the nature of choices made in the execution of the system (e.g. variables and values that determine selections), which limits the applicability of the model.

The method is based on querying compiled static and dynamic information about the system using a logic programming approach. The static information concerns classes, inheritance-relationships, class attributes and methods, interclass uses, and interclass invocations. Dynamic information concerns performed transmissions between objects. A database of the information is collected and a Prolog middle-ware is used to extract component views of the system.

According to our classification, the approach is:

1. architectural description
2. for study/clarification
3. interactive

4. hybrid model extraction based on the source code and dynamic task level information.

3.2.12 From UML to timed automata

Shu et al. [97] provide an automated translation from their extended version of UML state machines and sequence diagrams to timed automata.

According to our classification, the approach is:

1. producing operational, temporal, behavioral description
2. for formal verification
3. non-interactive
4. static based on non-operational models.

3.2.13 Tagging Esterel code to make models

Sifakis et al. [98] proposed a static method that use tagging of real-time software with time constraints and environment behavior to facilitate automatic modeling based on the code as input. The method assumes that the system and the environment model is implemented in a version of Esterel. Allowing the specification of the environment in Esterel (including non-deterministic choices) enables engineers to specify the environment in the same language as they implement the system.

According to our classification, the approach is:

1. producing a partial, operational, temporal, description
2. for formal verification
3. interactive
4. static model extraction based on extended source code.

3.2.14 A workbench for extracting models from scenarios

Uchitel et al. [114, 115] introduce a method for extracting labeled transition systems using scenario-based specifications, specifically they use message sequence charts. In difference to most other work based on behavioral information such as recordings or scenario-based specifications, the models produced by Uchitel et al. are generalized with respect to the components of the system. That is, when other work can extract a model valid only in the observed case, Uchitel et al. use existing architectural descriptions to extract general models that can be instantiated into arbitrary architectures.

According to our classification, the approach is:

1. operational, behavioral description
2. simulation
3. interactive
4. dynamic with task-level information.

3.2.15 DiscoTect: retrieving architecture

The dynamic model extraction of Yan et al. [93, 119] is used to construct architectural models from object oriented Java implementations. These models can show the possible interactions between tasks and resources such as files, semaphores, and abstract data objects.

Using the proposed method, it is possible to obtain a high-level view of the functionality implemented in the system. This view can be compared to the intentions of the developers to verify that the implementation has realized the intended architecture. The authors note that static model extraction may not always be feasible as it requires that the entire setup of the system is defined – this may prevent use of dynamic libraries etc.

In relation to our contribution, this approach is inherently non-real-time – monitoring of the real implementation is performed by using a debugger to query the implementation during run-time to extract information about occurred events. The use of an ordinary debugger effectively prohibits the possibility to maintain real-time constraints [35].

According to our classification, the approach is:

1. producing an architectural description
2. for clarification
3. non-interactive
4. a dynamic model extraction from system-level recordings.

3.3 Discussion

According to our classification, our approach is:

1. temporal and partial operational behavioral description
2. simulation and model-based impact analysis
3. non-interactive
4. dynamic with system-level information.

Among the differences that we note from comparing our proposition to the work related above, we highlight the following:

Compared to the work by Grinchtein et al. [26], Koskimies et al. [56], and Hungar et al. [34], we do not use machine learning. In contrast to these, we cannot introduce loops on task level in the behavior of a job. On the other hand, for the types of systems that we target, such loops are often avoided due to predictability requirements. Also, we avoid the issue of having to construct Oracles and Learners. As we target a very specific problem, the complexity of our proposition is likely to be lower than that of machine learning, which target a more general problem domain: The complexity of our proposition is approximately $O(N + T \times S)$, where N is the total length of all recordings, T is the size of the model, and S is the total number of data states identified in the model. Typically, this yields an execution time in the order of seconds. For example, the complexity of Grinchtien et al. [26] is greater than $O(K^{4|\Sigma|})$, where K is the bound on constants in the edges of the automaton, and $|\Sigma|$ is the size of the automaton's language. There are no evident obstacles that prevent our proposition to be merged with methods for machine learning.

Our method is non-interactive (compare to Bastos and Sanches [9] and Holzmann and Smith [30, 31]). This is two sided: On one hand, a non-interactive method is easier to handle. On the other hand, an interactive method has the potential to produce a model which is more adapted to the needs of the engineers.

Compared to for example Israr et al. [46] and Jensen [49], our method allows modeling of the task data-state. Modeling of data state allows a compact model representation to express causality between jobs.

Our method does not assume the use of a specific programming language (compared to Logothetis et al. [69, 70]).

Chapter 4

Recording-based automatic modeling

In this chapter, we present a method for automatic modeling from recordings of real-time systems. The method includes model generation as well as model validation.

4.1 Notation

The set of positive integers is denoted $\mathbb{Z}$, and the set of non-negative integers by $\mathbb{Z}^*$. We use the letter z to denote a typeless undefined value. Given a set or tuple A we refer to a member $a \in A$ using a dotted notation, e.g. $A.a$. Given a vector v, we refer to the n^{th} element of the vector as v_n. Given a set S, $|S|$ denotes the cardinality of the set.

4.2 System model

This section is consistent with Section 2.1, where we discussed real-time systems. The definition is also compliant with state-of-practice real-time operating systems (e.g., VxWorks), which motivates that the system model is relevant to the current industry practice.

We assume that the system is a fixed-priority scheduled real-time system with a set of single threaded tasks that execute programs. Inter-process com-

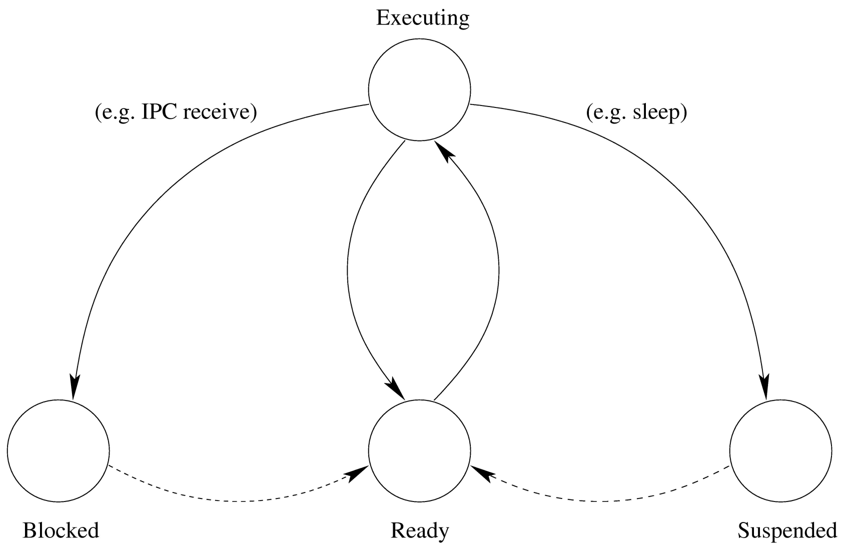


Figure 4.1: The operating system process states, dotted transitions are not possible to probe with commercially available operating systems.

munication (IPC) queues are used for explicit communication between tasks. Consecutive executions of a task (i.e. jobs of a task) are assumed not to overlap in time. The real-time operating system maintains an operating system state (OS-state) for each task, such that the later is in one of the following states: *executing*, *ready to run*, *suspended*, or *blocked*. Only one job at a time can have the OS-state *executing*, and that task is granted access to execute its program on the single shared computational resource (i.e. we assume a uniprocessor). Figure 4.1 shows the possible transitions between OS-states. Assuming that we have a probing support in the operating system as described in Section 2.4.5, the dotted OS-state transitions are not possible to probe.

Jobs are either *periodic* (i.e., triggered with a known and constant period time), or *event driven* (i.e., triggered by the receiving of new messages on a given queue). A periodic task is in OS-state *suspended* when it is between jobs. Tasks waiting for input on a FIFO-ordered[1] IPC queue are in OS-state *blocked*. Each task of the system can have a set of local variables (without loss

[1]FIFO: First In First Out.

of generality assumed to be integers) that can be manipulated and constitute the data-state of the task, which is modified by executing assignment statements. The data-state is persistent over jobs of the task, and controls the execution behavior of the task, i.e. determines the branching, outputs, and data-state modifications. Note that the data-state includes messages received on IPC queues, since we assume that messages are copied into local variables when they are received by a task. There are no global variables.

We show a conceptual view of this system model in Figure 4.2. Note that, in addition to what is stated above, the figure shows that tasks execute programs (P) and that the current location in the program is given by a program counter (pc).

We can summarize our system model by defining the system state as follows:

Definition 1 (system state)**.** The set of system states *SState* for a system conforming to our assumptions is given by:

- $\mathit{SState} \subseteq \mathit{TState}_0 \times \mathit{TState}_1 \times \ldots \times \mathit{TState}_n$.
- TState_i is the set of task-states for task i given by $\mathit{TState}_i \subseteq \mathit{DState}_i \times PC_i \times \mathit{IPC}_i$.
- DState_i is the data-state of task i given by $\mathit{DState}_i \subseteq \mathit{dom}(v^i_0) \times \mathit{dom}(v^i_1) \times \ldots \times \mathit{dom}(v^i_{|V_i|-1})$, where $\mathit{dom}(v)$ is the set of possible values (the value domain) of variable v.
- V_i is the set of data variables in task i, ranged over by v.
- PC_i is the set of possible program counter values in the program P_i executed by task i.
- IPC_i is the IPC state of task i given by $\mathit{IPC}_i \subseteq \mathit{list}(q_0) \times \mathit{list}(q_1) \times \ldots \times \mathit{list}(q_{|Q_i|-1})$, where $\mathit{list}(q)$ is a finite (possibly empty) list of messages each carrying (without loss of generality) a single integer value.
- Q_i is the set of input queues to task i, ranged over by q.

□

Note that it is outside the scope of this thesis to go into further detail of the system model, such as defining the semantics of an execution.

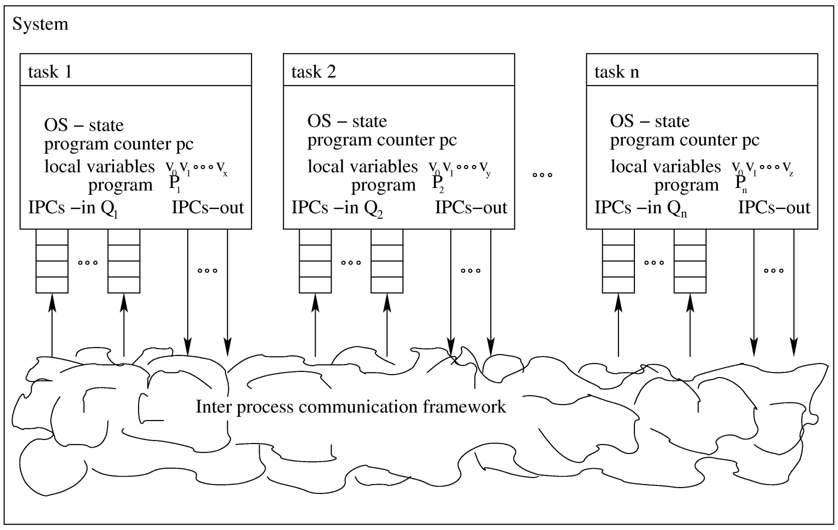


Figure 4.2: Conceptual system model.

4.3 Adding probes to the system model

The model extraction is based on recordings of the system. Probes are used to extract information on a set of *observables*, i.e. specific *events* and their *properties*. There is a set of obligatory probes to cover system-level events, and an optional set to cover modifications in data-state. A *probing configuration* or *probing setup* consists of the set of obligatory probes and a subset of the optional probes. Thus, there can be several possible probing configurations for a given system.

The system-level observables are *context switches* and *system calls*. The optional observables are *state-probes* that record the assignment of values to selected variables of those that represent the data-state in the system.

For the continued presentation, we shall assume that state-probes are used.

The state of the probed system is defined as follows:

Definition 2 (probed system state)**.** The set of system states for a probed system diverges from the states of the corresponding unprobed system (see Definition 1) in the following aspects:

- New local variables may be added such that $V_i \rightarrow V_i^{'}$.
- The program is extended with probes (i.e. code that collects data) to record observables such that $P_i \rightarrow P_i^{'}$.
- As a consequence, the set of possible program counter values is extended such that $PC_i \rightarrow PC_i^{'}$.
- Observables in recorded executions are stored in a global list *GR*, i.e., there is a global data structure *GR* in which the collected data is stored.

□

For model extraction as described here, the recorded executions (*GR*) represent the only available source of information on the execution of the system. However, the properties of probing do not ensure that the information in *GR* perfectly reflects the execution of the system. Hence, the relation between one recorded execution and one execution of the system is not bijective; several executions of the system can yield the same recording. Therefore, from our recorded viewpoint, the system may appear to be non-deterministic (see Section 2.5.1).

For example, only a subset of the data-state modifications to $V_i^{'}$ is included in *GR*, and the properties stored (e.g. describing an evaluation of $v \in V_i^{'}$) can be abstractions of the properties of the system event (e.g. a single value may be used to represent a range of values). Also, the recorded time of e.g. a variable update may be later than the actual update, since the probing (i.e. storing the update in *GR*) is distinct from the actual update.

The *model data-state* is built from recorded updates to the set of probed variables. Each model data-state is a projection of recorded values on the set of variables included in the optional probing. Thus, a model data-state corresponds to a unique evaluation of a subset of the variables in $V_i^{'}$, more specifically a unique evaluation of the probed variables.

4.4 Testing the probed system

A *test* is an execution of a system for a given *test-case* (i.e. a set of given inputs over the execution of the system). The test-case places the execution in a context, i.e., governs what is performed, what are the inputs, and what are the expected outputs.

Recordings with a given probing configuration are obtained by testing the system. We will assume that the test is performed by applying one or several test-cases defined by the user. The extracted model is strictly speaking valid only with respect to the test-cases used to generate the recordings. We will elaborate on model validity for other test-cases in Section 5.5.

The model is a projection of the system, characterized by the observables included in the recording and triggered by the test-case. Relative to test-cases, which are defined by the user, the model preserves the occurrence of observables and the approximate timing of these. Thus, if a task emits a job that performs a set of observable events, the model can recreate these and separate them in time as observed in the recording.

The model will always faithfully preserve the relative ordering of observables *within* jobs. In order to preserve the ordering of observables *between* jobs (e.g. mode changes), the update events on data-state that control this ordering must be included in the recording (i.e. the event of performing a mode change must be recorded). If this is not the case, the system may seem non-deterministic from the viewpoint of the recording. Such non-determinism will ultimately result in that the model will have probabilistic properties.

This thesis is not concerned with, and will not further elaborate on, how tests are selected. Test selection is a research field of its own, see e.g. [28].

4.5 Background: ART-ML

We have chosen ART-ML (Architecture and Real Time behavior Modelling Language) [4, 6, 116] as modeling language. The major reason for our choice is the probabilistic properties of the language that seem to rhyme well with the fact that observations may not be able to provide all details about the implementation.

ART-ML has been introduced to allow modeling of complex real-time systems. Each task is modeled individually, and a compiler and an FPS (fixed priority scheduling) simulator are provided. These tools allow a set of ART-ML task models to be co-simulated, tested, and evaluated. The current simulator mimics the VxWorks operating system version 5.5. In addition, the ART-ML framework includes tools that facilitate efficient analysis of recordings: the TraceAlyzer and the Property Evaluation Tool (PET). The intention is to facilitate a model of the system that can be altered in order to reflect the intrusion by future changes to the implementation. It is assumed that it is easier to modify the model than to modify the actual implementation. Thus, if the altered model

```
task AAA
 trigger period 1300
 priority 254
 deadline 1300
behaviour{
 chance(60){
  execute((20,100),(30,130),(50,200));
 }
 else{
  execute((50,200),(50,210));
  snd(MBOX0,0);
  chance(50){
   snd(MBOX0,1);
  }
 }
}
```

Figure 4.3: ART-ML example.

is successfully evaluated with respect to resource availability and temporal requirements specifications, the confidence in the proposed implementation can be increased. We believe that this will lead to less dead-ends and less ad-hoc alterations in order to make the implementation answer to its requirements.

As the ART-ML model provides a very high-level view of the system, the logic for selecting different behaviors might not be available in the model. ART-ML solves this by providing a notion of probability such that runtime selections can be resolved by chance. Relieving the model from much of the implementation details moves the focus from low-grained functional issues to architectural issues and temporal behavior.

4.5.1 Example

In Figure 4.3, we provide a small example of a task modeled in ART-ML. Among the set of reserved words in ART-ML, the following subset is important to the contents of this chapter (for a more detailed description, we refer to [116]):

`if, else`**:** works intuitively, as in e.g. C or Java. Variables can be defined and modified to provide input for the selection.

`chance, else`**:** a selection that takes a probability as input and makes a choice based on that value during run-time.

`execute`: taking a series of tuples (probability, execution time) as input, the execute-statement can represent computation with varying execution times.

`snd, recv`: provides means to perform inter-process communication.

4.5.2 The TraceAlyzer

The TraceAlyzer [6] provides a graphical interface to execution recordings that can originate either from the system or the simulator. The viewer shows a structured view of system execution including the task level probes. The tool can be used to observe the behavior and the task interactions in the system. The TraceAlyzer is successfully used at ABB Robotics today, where engineers are reportedly grateful to the insight that the tool provides into their complex system. Also, the tool is under evaluation at Bombardier Transportation.

4.5.3 The Property Evaluation Tool, PET

To compare differences between recordings, the PET tool [6] allows the specification of a set of queries and rules. Using PET, two recordings can be compared with respect to the properties of given tasks in the recordings. Rules are used to specify the requirements of the comparison.

For example, two recordings can be compared with respect to the response time of task T_A, which exists in both recordings. Rules can be set to require that the maximum response time of T_A does not exceed X time units for any of the recordings, and that the relative difference between the median of the two recordings' response time distributions for T_A is less than Y time units. The tool allows for a set of properties with accompanying rules to be tested quickly and effectively.

4.6 A process for automatic modeling

We introduce a process of *model extraction* that combines model generation and model validation as described in Figure 4.4. The process of model extraction consists of a set of activities. There are five inputs to the process (these are detailed with the respective step that uses them):

- The system *implementation* that is the subject of model extraction.

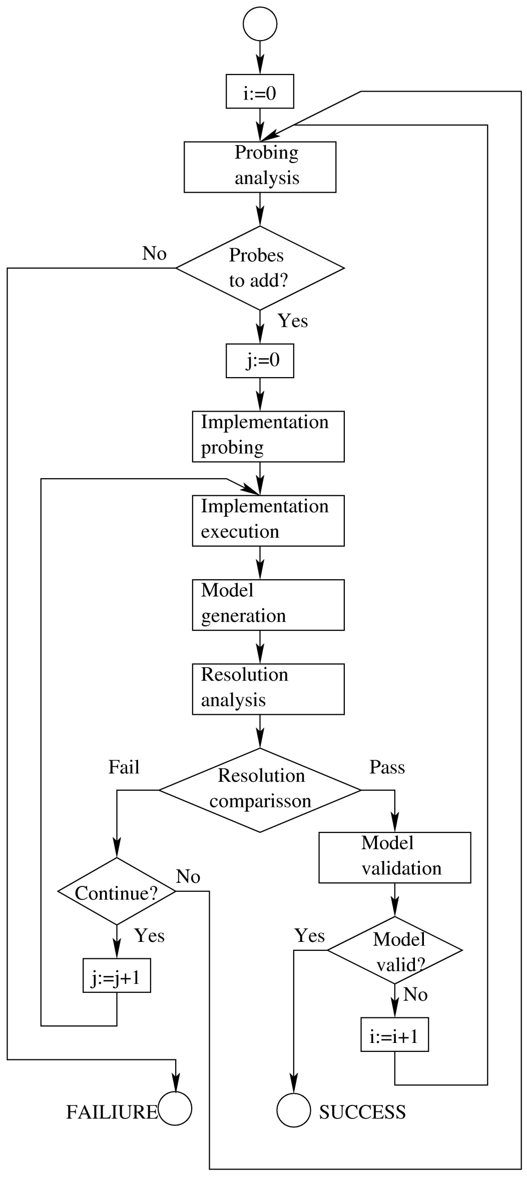


Figure 4.4: Our method of model extraction.

- The *test-case* that defines the context in which the system will be modeled.
- The *validity properties* which define the quality that the model should have in relation to the implementation.
- The *resolution threshold* that defines the resolution that is required for the probabilistic choices in the model.
- The *reiteration threshold* that defines the maximum number of reiterations to be performed in order to find a valid model with a given probing.

The output of the process is the *model*, which is described by the ART-ML modeling language [116].

The set of major activities and subprocesses in the process are described in the following *steps*:

- *Probing Analysis* checks if there are any untried probing setups (referred to as *probing configurations*), and evaluates the previous probing configuration (if any).
- *Implementation Probing* is responsible with choosing a probing configuration and preparing the implementation, by inserting the relevant probes so that required information can be obtained.
- *Implementation Execution* produces two set of recordings: one is used to produce the model, the other is used to validate the model.
- *Model Generation* generates a model based on recordings from implementation execution.
- *Resolution Analysis* examines auxiliary output from model generation to evaluate if the quality of the probabilistic choices in the recording is sufficient.
- *Resolution Comparison* evaluates the result of resolution analysis with respect to the resolution threshold.
- The *Continue*-step decides, based on the reiteration threshold, if the search for a usable recording is likely to terminate or not.
- *Model Validation* evaluates the level of similarity between the model and recordings from the implementation.

These steps are explained in more detail in the sub-sections that follow:

4.6.1 Probing analysis

The *probe setup* is the current configuration of probes in the implementation. There is a minimal set of system-level probes that is required for model generation, and an optional set of task-level probes that can be added to improve the produced model by recording the dynamic state of variables in the system. The process of determining the members of the set of optional task-level probes is a crucial part of model extraction; without these, the model obtained by the process described here is essentially unable to express causal relations between the functional behaviors of a task's jobs.

As the set of variables used in the implementation is finite, there is of course a finite set of possible probe setups, and these can be listed. Currently, this is a manual step, but it is possible to automate at least parts of this step. Once all probe settings have been tried unsuccessfully, it is concluded that model extraction cannot be performed under current circumstances. The process is then aborted, but it may be possible to adjust the parameters of the process and make a subsequent attempt. The failure to complete the process may be due to one or several of the following reasons:

- The validity properties are too restrictive for the non-determinism of dynamic inputs, etc., to the system.
- The probing required to capture the behavior of the implementation is too demanding. In this case, the overhead of probing has lead to *live lock* [35] in the system (i.e. as probing consumes too much resources, the amount of relevant work performed is low).
- The probing analysis failed to find all alternatives during the probe setup inventory in probing analysis.
- The implementation is not suitable for model extraction (e.g. the implementation does not conform to our system model).

The probe setup inventory performed by probing analysis is not (in any way) affecting the performance of model extraction – it is just a matter of whether there are more options or not. The analysis will consider the available variables in the implementation, and the history of used probe setups.

4.6.2 Probes to add

If the probing analysis has identified probe setups that have not been tried yet, the selection performed in this activity will lead to the fact that the probing is implemented in the “Implementation probing” activity. Otherwise, if there are no untried probe setups, this instance of model extraction will fail as described above.

4.6.3 Implementation probing

The subsequent model generation assumes that it is possible to probe both *system-* and *task-level* events in the system. Probing of system-level events (i.e. context-switches and system calls) are mandatory, and will allow the extraction of a relatively crude and undetailed model of the system. In order to refine the model, so that its behavior approaches that of the real implementation, task-level probes are added to extract information about the dynamic *state* (i.e. variable values) of the system.

Recording the state in the system by using task-level probes allows us to understand (and model) *causality* between jobs and events in the system – here, causality describes the fact that one event in one job affects the continued execution (i.e. the probabilities of future events and selections). Understanding and being aware of these form of causal dependencies between actions and reactions is fundamental to making good behavioral models; in our work we allow causal dependencies to be reflected in the model generation by probing variables. Other possibilities to achieve a similar result are by using static analysis of source code or evaluating advanced guesses about causal dependencies between events, using hypothesis testing etc.

The set of variables that are included in the task-level probing limits the quality of model extraction. However, excessive probing will lead to an unnecessarily high overhead in the system, and make the model more complex than necessary – thus, a trade-off has to be made: the set of variables that are probed should be sufficient, but the overhead of recording that set of variables should be kept as low as possible.

Deciding on the appropriate probe setup is a non-trivial task. Thus, as this activity will have such an impact on the final product, several iterations of the model generation in the model extraction process are performed, to search for an appropriate probe setup. Later in the process, a comparison between the implementation and the model is used as an evaluation to determine if the model from the current probe setup is sufficiently detailed. To reduce the number of iterations of model generation in the model extraction process, the search for an appropriate setup may use heuristic methods (such as those for memory exclusive checkpoints as introduced by Plank [86]) to make suggestions for variables to be included in – or removed from – the probing setup.

4.6.4 Implementation execution

The probed version of the implementation is executed twice, and two sets of recordings rec_i^g and rec_i^v are thereby obtained. Informally, a recording consists of a list of entries, where each entry has a type, a time stamp, and a set of type-specific parameters. The first set of recordings will be used for Model Generation (g), the second set for Model Validation (v).

The required length of the implementation execution is preset to some value and can then be renegotiated by the resolution analysis.

4.6.5 Model generation

Our method for model generation has been described in earlier work [36]. We provide a detailed explanation of the method in Chapter 5.

Model Generation operates on task basis and has three steps, the primary output is an ART-ML model for each task:

1. For each task, all jobs are extracted from the set of recordings.

2. According to a set of rules, all jobs (except the first job) from each recording are merged into a tree structure per task.

3. Each tree structure is transformed into an ART-ML model for the task.

The motivation for removing the first job of each recording in the second step is that the first instance of a task tends to have a rather different behavior compared to other jobs and it only occurs once in a recording. This provides too little data to make accurate predictions. To model a behavior that only occurs in the first instance of a task would require analyzing a multitude of recordings in order to get a sufficient amount of data on execution times and probabilities.

Model Generation is performed on both sets of recordings generated in the implementation execution step. Apart from the primary output, which is the ART-ML model based on rec_i^g, model validation uses the representation of jobs from rec_i^v and the tree of collected jobs from rec_i^g.

4.6.6 Resolution analysis

The objective of resolution analysis is to determine whether recorded data is sufficient to capture a model of the implementation in the current test-case. The analysis is made on parameters of the model to ensure that a longer recording time is not likely to give any additional detail.

We argue that it is reasonable to assume that the complexity of the implementation (and a given test-case), and therefore also the complexity of the model, will affect the required length of the recordings rec_i. Resolution analysis of the model is used to determine if the recording lengths used to obtain rec_i^g are sufficient.

Basically, resolution analysis is performed by observing the probabilistic assumptions of the model with respect to the *resolution threshold* criteria that is supplied as input to model extraction.

The *resolution threshold* consists of two parts: the *individual threshold*, which defines how many observations are required for each event observed, and *overall threshold*, which controls how large part of the observed events must fulfill the individual threshold.

If a sufficient number (defined by the resolution threshold) of parts of the model has sufficient observations (defined by the resolution threshold), so as to make it likely that all parts have been identified, the length used to obtain the recordings is deemed sufficient – otherwise, the length is increased by some factor C. Note that the value of C has no impact on the quality of the final model, but it will effect the time required to successfully perform model extraction, and the amount of memory required to house the recordings.

If the recording length is modified, the current iteration in model extraction is back-traced to the point of the implementation executions. This is performed with the new length, and the process is continued from that point on.

4.6.7 Resolution comparison

This selection acts on the output of the resolution analysis above. If the resolution was deemed sufficient, model validation will commence, otherwise, the future progress of model extraction is evaluated in the following selection.

4.6.8 Continue with increased recording time

As a consequence of the resolution analysis, it is possible to conclude a failure of this instance of model extraction with the current probe setting (see Figure 4.4). If the decision by the resolution analysis to increase the length of the recording does not help the resolution, we must abort this attempt, sooner or later. It is the task of the current step to determine at what point the process is no longer likely to succeed with its objective.

If the process seems unable to deem the resolution of the model fit under current conditions, it is likely that the test context supplied as input to the process is not restrictive enough. This will mean that the resolution analysis is performed for the same i more than a predefined number of times, such that the counter j exceeds a threshold value $\mathit{iteration}_{max}$. The optimal value of $\mathit{iteration}_{max}$ depends on the complexity of the implementation, the value of C, and the initial length of the recording.

4.6.9 Model validation

Our method for model validation has been described in earlier work [37]. We provide a detailed explanation of the method in Chapter 6.

We validate the fact that the recordings used to generate the model are sufficient to describe the system by answering the question "Would the model be drastically better if the length or number of recordings used during model generation are increased?". Automatic model validation uses a set of system execution recordings to answer this question. The model and the recordings are transformed into a set of communicating timed automata with integer variables [3, 10]. While the model-automaton is a graph structure that may contain more than one transitions from each label, the recording-automata are all sequential with one or zero transitions from each label. The *final state* of the recording-automata is the only state that hasn't got any exiting transitions. The validation is performed by reachability analysis of the final state in each recording-automaton when co-simulated with the model-automaton.

To allow the model to be an approximate abstraction of the system, the recording-automata are constructed using a *leeway*-parameter. The higher the leeway, the more forgiving the recording-automaton will be. The maximum allowed leeway can be supplied by the user as a parameter.

The stopping criteria of the validation is based on two factors: the *completeness measure*, i.e., the probability that the model can replicate any job that the system can exhibit, and the *accuracy measure*, i.e., the relation between the probability that the system exhibits a particular job and the probability that the model exhibits an equivalent job.

Validation can provide the maximum required leeway, the completeness measure, and the accuracy measure as auxiliary outputs. The primary output is the binary answer to the question posed at the top of the section.

4.6.10 Model valid

If model validation has succeeded, this will result in a successful termination of the process, otherwise, probing analysis will evaluate the possibility to modify the task probing.

4.7 Example

To explain the dynamics of model extraction, we present an abstract example:

For a system S, we are trying to extract a model M. During the course of the extraction, we will investigate several prospective models $M_{i,j}$, where i and j varies in the process as described by Figure 4.4. They are both zero at the start of model extraction. Initially, we use a recording length l_0 for all execution recordings. This can then change as $l_j = l_0 + C \times j$.

As a first step, probing analysis is performed to determine how many possible probing configurations there are in the system. If there are any, the selection Probes to Add will then lead to Implementation Probing, where a probe configuration is chosen and implemented in the system. Thereafter, Implementation Execution will yield two sets of recordings: rec_0^g and rec_0^v.

Using rec_0^g, Model Generation will yield $M_{0,0}$, but also rec_0^v is processed as intermediate output from model generation of these two sets are later used in Resolution Analysis and Model Validation.

After Model Generation, Resolution Analysis will examine intermediate output to ensure that all parts of the model are based on a sufficient number of observations (e.g. more than one).

Should Resolution Analysis conclude that the model is based on too few observations, j will be incremented and Implementation Execution will be re-performed with l_1, l_2, etc. This will yield models $M_{0,1}$, $M_{0,2}$, etc.

Eventually, else this probe configuration is deemed unusable after a total of *iteration$_{max}$* attempts have been made, Resolution Analysis will be satisfied and the selection Resolution Comparison will pass, leading to the commencing of Model Validation.

If Model Validation fails, Probing Analysis will determine if there are more probing configurations to try, and models $M_{1,j}$, $M_{2,j}$, etc. are generated from there.

4.8 Discussion

In this chapter, we have introduced our method for model extraction. The intent is to allow automatic modeling for model-based impact analysis in the context of legacy real-time systems.

Given this context, it is interesting to discuss the capability of a changed model to resemble a changed system, keeping in mind that the model is based on information obtained by some form of testing. Note that we have stated, on Page 30, that results of testing cannot easily be extrapolated to other premises than those under which testing was performed; the question that follows from this is: is it then possible to assume that the model can be valid after that it has been changed?

We argue that the question of validity applies only to the original model; clearly, a changed model is no longer valid with respect to the original system, but it may very well be valid with respect to the changed system (provided that the change applied to the original model is a valid abstraction of the change applied to the original system). In Chapter 8, we will examine the *stability* of model extraction, which will evaluate its possibility to extract a model that, when changed, is similar to the correspondingly changed system.

In the following two chapters, we describe in further detail two of the most essential steps in model extraction: model generation and model validation.

Chapter 5

Model generation

In this chapter, we present a method for model generation, first introduced under the name "model synthesis" [36]. Chapter 4 puts model generation, presented in this chapter, within the context of model extraction.

5.1 Synopsis

Based on recordings of a running system, a model that can describe the observed behavior is automatically generated. This allows for faster modeling of existing systems, as compared to manual modeling, and reduces the risk of introducing bugs in the model.

5.1.1 Assumptions

Apart from the implicit assumptions of the system model defined in Section 4.2, we assume that the following are known for each modeled task:

- the priority of the task,[1]
- the unique identification of the task,
- the method of triggering a new job of the task (i.e. periodic or event driven, see Section 4.2),

[1]It is possible to formulate an educated guess of task priorities out of recordings. Yet, for brevity, we have chosen not to do so.

Event	Abbreviation	Parameters
Context switch (T_A by T_B)	*csw*	time, operating system state of T_A, unique identifier (id) of T_A, id of T_B.
Send to IPC queue: initialize	*sndi*	time, queue identifier.
Send to IPC queue: finalize	*sndf*	time, message.
Read from IPC queue: initialize	*rcvi*	time, queue identifier, timeout.
Read from IPC queue: finalize	*rcvf*	time, message.
Variable assignment	*vas*	time, value, variable name.

Table 5.1: Events and their parameters in the recording.

- the IPC queue that triggers new jobs, in case the task is event driven,
- the operating system's task-state (OS-state) that signifies blocking of a periodically triggered task (i.e. the end of the current job), and
- the initial values of monitored variables.

For each event, a set of parameters, as described by Table 5.1, is logged.

5.1.2 From recordings to ART-ML models

The input to model generation is a set of recorded executions (recordings). Informally, each recording is a non-empty list of entries that originate from probes triggered by the execution of a system. Each entry has a time stamp that indicates the time elapsed from the start of the system execution, an event type, and a set of parameters that describe some attributes of the type.

Our model generation consists of the following three sequential steps:

1. **Extraction of task executions (jobs) from recordings.** Here, recordings of the system are separated into observed task executions, and jobs of the task are identified and described.

2. **Generation of a tree-representation of the task, from the jobs.** In this step, the jobs of each task are collected and, according to a set of rules, merged into a treelike-representation.

3. **Generation of ART-ML code from the tree-representation.** Here, the model code is generated from the tree-representation (the tree is folded into a timed and probabilistic automaton).

Each step in our model generation represents an increased abstraction of the system in the sense that the set of task behaviors (i.e. timed sequences of observables) represented in later steps are supersets of behaviors in earlier steps. The first step separates the cooperative execution of tasks into the jobs of individual tasks. The second step forms a compact tree-representation of the jobs observed; while the ordering and characteristics of observed observables is preserved, timing properties are abstracted by grouping similar execution demands. In the third step, as the model code is generated, the timing properties are further abstracted. Also, if the optional state-probes provide insufficient data for separation of the possible task behaviors, probabilistic selections are introduced and the ordering of jobs is abstracted.

These three steps are explained in detail in the following sections and algorithms are provided in Appendix A.

5.1.3 Example

We introduce a small example implemented in C that will guide the explanation of the three steps of model generation (see Figure 5.1). There are three tasks in the example: T_A is periodically triggered, and transmits IPC messages to trigger T_B, and T_C is the idle task, which executes with the lowest priority. IPC messages are sent by the function `IPCsend`, and received by the function `IPCreceive` (no timeout is assumed for this function, which is realized by setting the timeout parameter to infinity). Data-state probing is performed by calling the function `record_var` with an identifier for the variable and the evaluation of the variable. We assume that the idle-task T_C is executing at the start of the system.

Task T_A, medium priority, periodic

```
f(mailbox_t Q){
  int a=0,b=0;
  while(1){
    ...
    a=(a+1)%3;
    record_vas("a",a);
    if(a==0 || a==1){
      IPCsend(Q,0);
    }
    else if(a==2){
      if(b==0)
        IPCsend(Q,1);
      else
        IPCsend(Q,0);
      b=(b+1)%2;
    }
  }
}
```

Task T_C, low priority, periodic

```
h(void){
  while(1);
}
```

Task T_B,high priority, event driven

```
g(mailbox_t Q){
  int c=0;
  int d=0;
  message_t m;
  while(1) {
    m=IPCreceive(Q);
    ...
    if(c != 0){
      c=0;
      record_vas("c",c);
      d=0;
      record_vas("d",d);
    }
    else{
      c=0;
      record_vas("c",c);
      d=1;
      record_vas("d",d);
    }
    c=(m+d+1)%2;
    record_vas("c",c);
  }
}
```

Figure 5.1: Example with three tasks, T_A sends messages to T_B.

5.2 Extraction of task executions (jobs) from recordings

As indicated in Section 5.1.2, a recording is a non-empty ordered sequence of entries: $\langle entry_0,\ entry_1,\ entry_2 \ldots\ entry_n\rangle$, where each entry describes an observed event in the execution, together with the time when the event occurred. Each event is described by its type and parameters, as presented in Table 5.1.

The currently considered events in $EventType$ are given in Table 5.1. More system calls or other event types could be added.

As a first step in model generation, we analyze recordings collected from the system that is to be modeled. These are analyzed on a task-basis (using recordings of context switches to differentiate between tasks). The objective of the analysis is to identify all jobs and the events occurred in each job, for all tasks.

While doing this, we need to resolve the following:

- Find the recording entries signaling the starts and ends of jobs.
- Build and represent the model data-state.
- Identify events, actions and action parameters.

5.2.1 Example revisited – recording

A possible recording of our example is displayed in Figure 5.2. This is a typical example of input to the model generation.

At context switches, the OS-state of the preempted task is recorded, the first task identifier is the preempted task, and the second task identifier is the preempting task. Thus, at time 0, T_C is preempted by T_B.

5.2.2 Finding the starts and ends of jobs

Using knowledge of the triggering method of each task (see our assumptions in Section 5.1.1), we can determine the start and end of the jobs of each task in the recordings, in the following manner:

- If the triggering method is periodic, the logs of the recording are analyzed to determine the end and start of jobs based on the OS-state at *context switch* (*csw*) events. For a periodic task, transferring from *executing* to *suspended* signals the end of a job. Under our assumptions, the start of a periodic job cannot be exactly determined, but the start of a job is in the interval defined by the time of the end of the last job, and the time of the *csw* event that occurs when transferring from *ready* to *executing*.
- In case of event driven triggering, the end of a job is signaled by a *Read from IPC queue: initialize* (*rcvi*) event, such that the queue identifier parameter is the triggering IPC queue of the task. The start of the next job is then signaled by the first subsequent event, with action of type *Read from IPC queue: finalize* (*rcvf*) of that task.

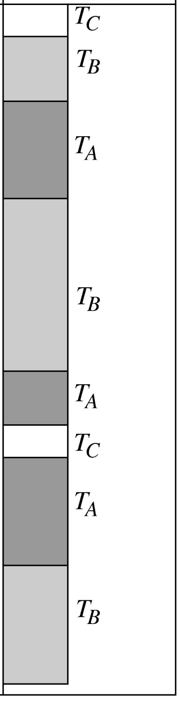
Time	Event	Parameters			Executing task
0	*csw*	ready	T_C	T_B	
1	*rcvi*	Q	infinity		
2	*csw*	blocked	T_B	T_A	
10	*vas*	"a"	1		
11	*sndi*	Q			
12	*csw*	ready	T_A	T_B	
13	*rcvf*	0			
20	*vas*	"c"	0		
22	*vas*	"d"	1		
24	*rcvi*	Q	infinity		
25	*csw*	blocked	T_B	T_A	
26	*sndf*	0			
27	*csw*	suspended	T_A	T_C	
127	*csw*	ready	T_C	T_A	
142	*vas*	"a"	2		
143	*sndi*	Q			
144	*csw*	ready	T_A	T_B	
145	*rcvf*	0			
159	*vas*	"c"	0		
⋮	⋮	⋮			

Figure 5.2: A possible recording from the example.

5.2.3 Distinguishing jobs with different initial data-state

In the model generation it will be important to distinguish jobs with different initial states (i.e., with different initial variable values). For obvious reasons, here, it is only possible to consider the observed variables.

From the definition of model data-state on Page 55 we can define the task model data-state for task i to be a corresponding projection of the observed variables of task i. To distinguish jobs with different initial task model data-state, we will label each event of the job with this state. Later on, these labels will be used as identifiers for classes of jobs that can be joined in the model generation, as will be explained in Section 5.3.

Action type	Abbreviation	Parameter
Send to IPC queue	*snd*	state, previous message, queue identifier, message
Read from IPC queue	*rcv*	state, previous message, queue identifier, timeout
Variable assignment	*upd*	state, previous message, value, variable name
Execute	*exe*	state, previous message, time consumption
End job	*end*	state, previous message, suspension time

Table 5.2: Members of the set $Actions$ and their action parameters in a recseq.

Definition 3 ($TaskModelDataStates$)**.** Given a task i and a set $\{v_0^i, v_1^i, \ldots, v_n^i\}$ of probed local variables a task model data state s is a unique evaluation of these variables, i.e. $s \in dom(v_0^i) \times dom(v_1^i) \times \ldots dom(v_n^i)$. We denote the set of possible such states $TaskModelDataStates$. □

5.2.4 Identifying remaining events, actions and action parameters

Between the start and end of jobs, we identify the events that constitute the observed behavior of the job. We define an *event* as follows:

Definition 4 (event)**.** An *event* $o \in Events$ is a tuple $\langle a, s \rangle$ where a is an action in the set $Actions$ with action parameters as described by Table 5.2, and $s \in TaskModelDataStates$. □

In order to compile a description of each job, the events of jobs are extracted from the recording; two events *sndi* and *sndf* as defined in Table 5.1 are joined and a single event with action type *snd* as defined in Table 5.2 is formed. A corresponding operation is performed to identify events with actions of type *rcv*. Events with actions of type *exe* are inserted as needed between other events, to account for the execution time spent.

Most of the parameters, except data-state, execution demands, previous message, and suspension time, are given from the recording via direct translation from event parameters. Exceptions are handled as follows:

- Handling of model data-state is consistent with Section 5.2.3. The initial state is derived from the initial values of variables (see our assumptions in Section 5.1.1) and the subsequently performed *Variable assignment* (*vas*) events, such that each variable in the model, which has *vas* events present in the recording, is represented.
- We identify the execution demands of each task in terms of accumulated execution time between non-context switch events. Measuring the time elapsed between recorded events in the tasks, and using the context switch information to subtract time spent on executing other tasks, we derive execution demands between each system level event such as *send*, *receive*, *variable assignment*, and *end*.
- Messages received in communication with the environment or with other tasks (i.e. by inter-process communication) are included in the *previous message* parameters of all event types. To introduce the knowledge of communication, we let the event immediately subsequent to the event with the receive action contain information on the content of the communication.
- The suspension time is only relevant for periodically triggered tasks. In this case, the suspension time is the time elapsed between the end of one job and the start of the next consecutive job.

5.2.5 A set of jobs of a task

The information extracted in the above is stored in an intermediate format called *recseq* (short for *recorded sequence*), see Definition 5. Intuitively, a recseq is a serial list of *jobs*, which are defined in Definition 6. The recseqs from a set of recordings can be concatenated into one recseq.

Definition 5 (recseq)**.** A recseq is a list of jobs $j \in ObservedJobs$ ranged over by R. □

Definition 6 ($ObservedJobs$)**.** A job $j \in ObservedJobs$ is a non-empty list of events $o \in Events$, for which each event with *exe* action must be succeeded by an event with another action, and for which an event with *end* action cannot have a successor, and all jobs end with an *end* action. □

Intuitively, a *job* is a list of $events$. At the end of each job, there would be an end-event. This event is inserted when generating the recseq.

5.2.6 Example revisited – recseq generation

Concerning the recseq for T_A based on the recording in Figure 5.2: Knowing that T_A is periodically triggered, we can determine that the first job of the task starts at time 2, as T_B is blocked on a receive on IPC queue `Q`. As the initial value of variable `a` is 0, all events in this job of the recseq will have the $TaskModelDataStates$ `a`=0. The task executes for 8 time units (tu), before variable `a` is assigned the value 1 at time 10. Thereafter, a send to IPC queue `Q` will trigger T_B to preempt the task until time 25, when the send is completed. After executing for a small amount of time, the job is ended as the OS-state goes to *suspend*. We measure the suspension time to be 100 time units (tu). In a more complicated example, the suspension time could be larger than this, but model extraction ensures that repeated measurements are made so that the minimum measured suspension time approaches the real value as expressed in the code of the task.

We find a series of $ObservedJobs$ for T_A as described in Figure 5.3.

The way in which we handle model data-state in the recseq is justified by the recseq generated for task T_B (see Figure 5.4 for a partial recseq): We display two identified behaviors, the only significant differences between them being the initial data-state of the jobs and the values assigned in events with action *upd*. Separation of these is not readily available for the events that differ, it is the data-state at the beginning of the job that determines which of the two behaviors is performed, but that data-state has been overwritten at the differing event. In the next step of model generation, we will generate a tree-representation based on the recseq. To maintain a small tree-size, recseq nodes with different data-states may be grouped in the same node of the tree. Therefore, we must be able to separate the two jobs even if a prefix of the jobs is located in the same path of the tree. By keeping the initial state all through the job, we ensure that the tree-representation cannot express the unobserved behavior, i.e., the beginning of the first job, and the ending of the second.

In general terms, we can describe this as follows: assume a recseq with two jobs $\{A, B\}$ for a task where state-probes cover at least one variable. Job A consists of a prefix X with data-state a, followed by an update-action transferring the data-state from a to c, and a postfix Y. Job B consists of a prefix X with data-state b ($b \neq a$), followed by an update-action transferring the data-state from b to c, and a postfix Z ($Z \neq Y$).

When generating the tree-representation, the prefixes will be in the same path of the tree. Our handling of model data-state ensures that only the prefix with data-state a can lead to the postfix Y, and only the prefix with data-state

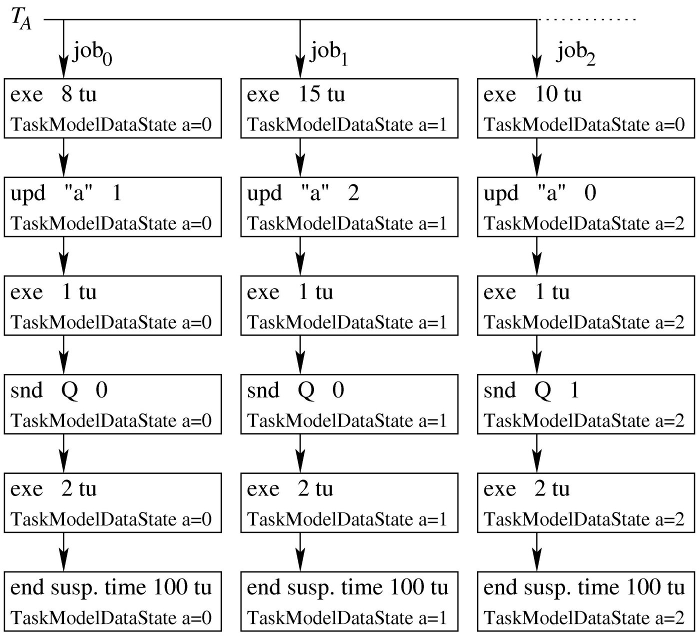
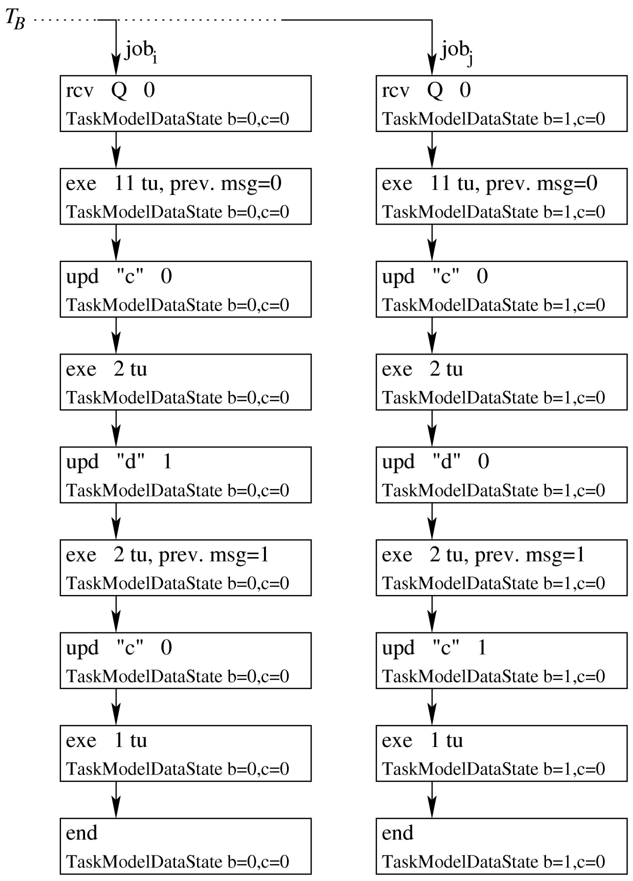


Figure 5.3: A recseq of T_A.



T_B
job_i
rcv Q 0
TaskModelDataState b=0,c=0
exe 11 tu, prev. msg=0
TaskModelDataState b=0,c=0
upd "c" 0
TaskModelDataState b=0,c=0
exe 2 tu
TaskModelDataState b=0,c=0
upd "d" 1
TaskModelDataState b=0,c=0
exe 2 tu, prev. msg=1
TaskModelDataState b=0,c=0
upd "c" 0
TaskModelDataState b=0,c=0
exe 1 tu
TaskModelDataState b=0,c=0
end
TaskModelDataState b=0,c=0
job_j
rcv Q 0
TaskModelDataState b=1,c=0
exe 11 tu, prev. msg=0
TaskModelDataState b=1,c=0
upd "c" 0
TaskModelDataState b=1,c=0
exe 2 tu
TaskModelDataState b=1,c=0
upd "d" 0
TaskModelDataState b=1,c=0
exe 2 tu, prev. msg=1
TaskModelDataState b=1,c=0
upd "c" 1
TaskModelDataState b=1,c=0
exe 1 tu
TaskModelDataState b=1,c=0
end
TaskModelDataState b=1,c=0

Figure 5.4: A recseq of T_B.

b can lead to the postfix Z.

5.3 Generation of a tree-representation of the task from the jobs

When all jobs have been compiled for a task, the next step is to compile a unified representation of the task that contains all information accumulated in the recseqs. These are combined to form a set of trees as follows:

We introduce the tree-structure *modset* to unify the collected recseqs of a task. Each modset is a set of modtrees as defined below:

Definition 7 (modtree)**.** $\mathbb{M}$ is a set of *modtrees* ranged over by T, whose elements are given by $\langle id, S, a, \mathbb{T}, c\rangle$, where:

- id is a unique identifier,
- $S \subseteq TaskModelDataStates$ is the set of valid observed data-states for the modtree $T \in \mathbb{M}$,
- $a \in Actions$ is the action of the modtree $T \in \mathbb{M}$,
- $\mathbb{T} \subseteq \mathbb{M}$, the set of successors, is the ordered list of the alternatives for subsequent execution after a has been performed, and
- c is a counter.

$\square$

Intuitively, a modtree performs an action, can have a set of successors, and is guarded by a condition on variables local to the task.

The use of modset in conjunction with modtrees allows a modset to have more than one first action of a job.

Definition 8 (modset)**.** A *modset* is a set of modtrees. $\square$

In the modset, nodes of the tree have actions describing execution and system calls (execute, send, receive, data-state update, or end of a job), and action parameters of these nodes describe the detail of the particular action (time, queue, etc.) as described by Table 5.3.

There are three differences between the different action parameters of the modtrees compared to the parameters of the events in recseqs: First, the data-states of all recseq-actions that comprise a modtree are added to a set of valid

Action type	Abbreviation	Parameters
Send to IPC queue	*snd*	states, previous message, queue identifier, message.
Read from IPC queue	*rcv*	states, previous message, queue identifier, timeout.
Variable assignment	*upd*	states, previous message, value, variable name.
Execute	*exe*	states, previous message, collection of time consumptions.
End job	*end*	states, previous message, minimum suspension time.

Table 5.3: Action types and their action parameters in a modset.

data-states for that branch. Also the number of occurrences of different data-states are recorded. Second, each modtree with an execute action represents execution times as a collection of values and associated data-states, rather than a single value. Third, the *minimum suspension time* parameter in modtrees with end actions represents the minimum suspension time found in the corresponding recseqs.

These differences are due to that each modtree in the modset is a compact representation of possibly several events from several recseqs.

5.3.1 Making the tree

To make a transition from a serial structure such as recseq, to a tree structure such as modset, rules to determine if two events in the recseq should be in the same node in the modset are required. The goal is to construct a minimal tree such that each path of the modset represents a job of the task. For two events to be placed in the same node of a tree, their predecessors in the job must be in the same path of the tree, and the two events must be equal. If the equality of events can be determined, constructing the smallest tree is straightforward.

Two recseq events are equal iff the *distinguishing parameters* of their respective action are syntactically equal.

Depending on their action types, different events have different distinguishing parameters as described in Table 5.4. We may then proceed to construct the smallest modset that can represent the set of job-sequences while respecting

Action type	Distinguishing parameters
snd	Action type, unique queue identifier, message, and the previous message.
rcv	Action type, unique queue identifier, timeout, and the previous message.
upd	Action type, unique variable identifier, value, and the previous message.
exe	Action type, type of the immediately subsequent action, and the previous message.
end	Action type.

Table 5.4: Distinguishing parameters of action types.

that only equal recseq events are located in the same node of the modset.

5.3.2 Example revisited – modset generation

In the recseqs of T_A, we find four different behaviors (see Figure 5.5). Three of these are dependent on the value of variable `a`, and one is not dependent on the data-state. Based on the distinguishing parameters, we can determine that all recseqs start with an event with action-type *exe*, followed by an event with action-type *upd*. Since this is all that is required for determining equality between these events, they are grouped into one node in the modset. The $TaskModelDataStates$ of the *upd*-node will contain all three possible data-states for variable `a`: `a=0`, `a=1`, and `a=2`. Thereafter, each recseq performs an update, but since the value of the update differs between three different behaviors, branches are introduced in the modset such that three behaviors are separated. One of these behaviors is subsequently branched into two behaviors. Since both behaviors of that branch has the same initial data state, the branch is not depending on the data-state and will give rise to a chance-selection in the ART-ML model. Note that the modtrees are supplied with an identifier ranging from *a* to *s*.

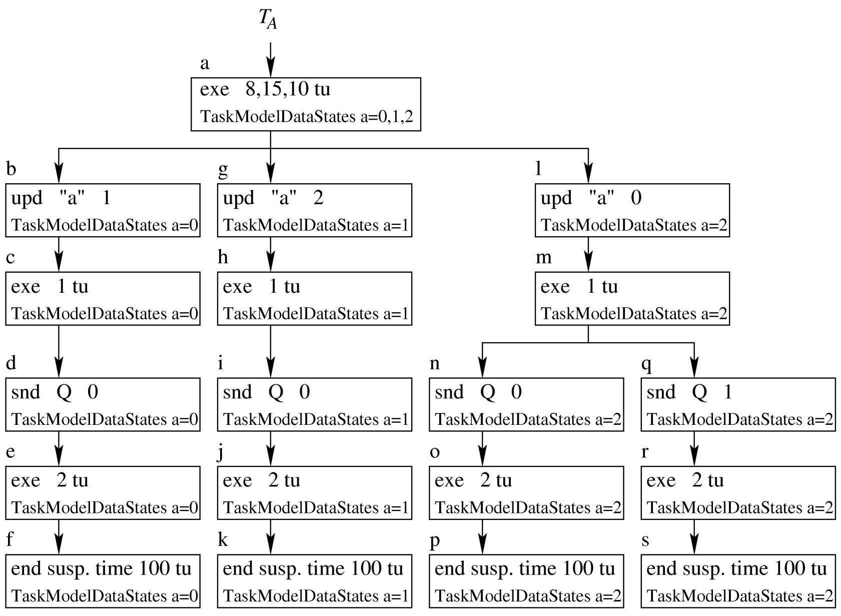


Figure 5.5: A modset of T_A.

5.4 Generation of ART-ML code from the tree-representation

Now that the modset has been formed from the collected set of recordings, we are finally ready to generate the model. This step includes representation of execution times and handling of selections based on data-state or probability as explained in the following.

5.4.1 Representing execution time

In comparing instances of different Execute-actions when constructing the final modset, the time spent on execution is not considered as a distinguishing parameter. Instead, as data is collected and many versions of the same execution-type modtree are discovered, these will be represented by a distribution of execution times. In this way, the size of the modset is reduced at the cost of precision.

When the ART-ML code is generated, the model should express this distribution in some descriptive manner that allows the model to behave temporally similar to the modeled system. This is indeed an important factor that has a large effect on the perceived resemblance between the behavior of the model and the behavior of the modeled system.

We could imagine several ways in which these distributions could be represented, and the final choice is inherently dependent on the capabilities of the modeling language.

In ART-ML, an execute statement represents execution time distributions as a set of pairs $\langle ET, P\rangle$, where ET is an execution time value and P is the probability of the execution time based on its occurrence ratio in the distribution. Given that limitation, we have chosen to limit our representation of execution time distributions to five execution time samples: The minimum value, the 25th percentile, the median, the 75th percentile, and the maximum value. Probabilities for these are used to describe the distribution over the sequence.

5.4.2 Introducing if- and chance-selections in the model

In ART-ML, there are two fundamentally different selections available: the *chance*- and the *if*-selection. The first is based on probability, and the second is based on model data-state. If a division of behaviors can be determined from the available model data-state and received IPC messages (i.e. the *previous message* parameter of events in the modset), ordinary if-selections will be inserted in the place of chance-selections.

The following deals with how to make the conversion from the modset to the ART-ML code:

From hereon, the model data-state of a modtree and the *previous message* parameter of the action of the same modtree are collectively referred to as *model-state*. We use a notion of *state-pairs* defined as follows:

Definition 9 (state-pair)**.** A *state-pair* is a pair $\langle S, B\rangle$, ranged over by *sp*, where S is a set of model-states and B is a set of modtrees.

To extract ART-ML code for a set of modtrees M ranged over by *m*, we follow the three steps below:

1. Initially, for each unique model-state s in M, there is a state-pair *sp* such that $sp.S = \{s\}$ and $sp.M = \{\forall m \in M : sp.S \subseteq m.S\}$. As modtrees that

may have several model-states and several modtrees can share model-state, each modtree may occur in several state-pairs.

This set of state-pairs is referred to as *SingleStateSP*.

2. Thereafter, all state-pairs $sp \in SingleStateSP$ with equal $sp.B$ with respect to distinguishing parameters are merged into a single state-pair.

   The set of new state-pairs is referred to as *SP*.

3. Finally, traversing all the modtrees in each member of *SP*, the ART-ML model is extracted. Between the state-pairs $sp \in SP$, if-selections are introduced. Note that the conditions of the if-selections are based on the model-states *and* the updates performed previously in the task. Within each state-pair $sp \in SP$, chance-selections are formed in between branches in $sp.B$. The probabilities of chance-selections are calculated based on the number of recseq events that the modtrees in the selection are based on (i.e. the c-parameter of the modtrees in the selection).

These three steps are performed recursively on all sets of modtrees in the modset. When branches with update actions are encountered, the details of these actions are passed on to the subsequent branches of the modtree.

5.4.3 Example revisited – ART-ML generation

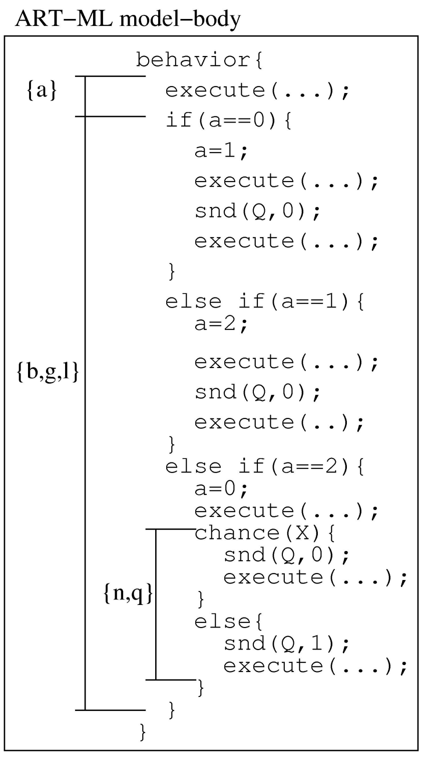
In Figure 5.6, we show both the *SingleStateSP*'s and the *SP*'s for all sets of modtrees in the modset. The first row is the root of the modset, which is $\{a\}$. The next row is $a.\mathbb{T} = \{b, g, l\}$.

The sets of state-pairs that are most interesting are those for the modsets $\{a\}$, $\{b, g, l\}$, and $\{n, q\}$: For $\{a\}$, the *SingleStateSP* contains three items, one for each model-state. Since all items *sp* in *SingleStateSP* has the same $sp.B$, this is then reduced so that *SP* only contains one item. Hence, this will not result in a branch in the ART-ML code. In $\{b, g, l\}$, such reduction cannot be performed, and since there are more than one element in the set *SP*, this will result in an if-else branch. Reduction is not possible in $\{n, q\}$ either, but since there is just one element in the set, we cannot introduce an if-else branch to separate the modtrees. Instead, the set of state-pair will result in a chance-else branch.

We display the finished ART-ML code for task T_A in Figure 5.7. We have marked the scope of the two branches in the model. The if-else branch is due to the difference between nodes $\{b, g, l\}$ in the modset. The chance-else

mdlset	SingleStateSP's:	SP's	Branch
{a}	{<{0},{a}>,<{1},{a}>,<{2}{a}>}	{<{0,1,2},{a}>}	No
{b,g,l}	{<{0},{b}>,<{1,{g}}>,<{2},{l}>}	{<{0},{b}>,<{1},{g}>,<{2},{l}>}	Yes
{c}	{<{0},{c}>}	{<{0},{c}>}	No
{d}	{<{0},{d}>}	{<{0},{d}>}	No
{e}	{<{0},{e}>}	{<{0},{e}>}	No
{f}	{<{0},{f}>}	{<{0},{f}>}	No
{g}	{<{1},{g}>}	{<{1},{g}>}	No
{h}	{<{1},{h}>}	{<{1},{h}>}	No
{i}	{<{1},{i}>}	{<{1},{i}>}	No
{j}	{<{1},{j}>}	{<{1},{j}>}	No
{k}	{<{1},{k}>}	{<{1},{k}>}	No
{l}	{<{2},{l}>}	{<{2},{l}>}	No
{m}	{<{2},{m}>}	{<{2},{m}>}	No
{n,q}	{<{2},{n,q}>}	{<{2},{n,q}>}	Yes
{o}	{<{2},{o}>}	{<{2},{o}>}	No
{p}	{<{2},{p}>}	{<{2},{p}>}	No
{r}	{<{2},{p}>}	{<{2},{p}>}	No
{s}	{<{2},{s}>}	{<{2},{s}>}	No

Figure 5.6: State-pairs for T_A.



```
behavior{
  execute(...);
  if(a==0){
    a=1;
    execute(...);
    snd(Q,0);
    execute(...);
  }
  else if(a==1){
    a=2;
    execute(...);
    snd(Q,0);
    execute(..);
  }
  else if(a==2){
    a=0;
    execute(...);
    chance(X){
      snd(Q,0);
      execute(...);
    }
    else{
      snd(Q,1);
      execute(...);
    }
  }
}
```

Figure 5.7: ART-ML model for T_A.

branch is due to the difference between nodes $\{n, q\}$. Observing the code of the task, we notice that the chance-else branch could be replaced for an if-else branch by including the task's variable `b` in the optional state-probing. In this example, we have omitted the details of execute statements and the probability calculation for the chance-else branch.

5.5 Discussion

In this chapter, we have presented automated model generation from recordings of real-time systems. There is a number of issues with the current version of the model generation, some possible to amend, some inherent in the approach.

The input to model generation is recording. Hence, model generation is subject to the problems described in Section 2.4. Further, we cannot ensure that the model generated describes the implementation in every aspect (compare to the completeness problem, Section 2.5.1). This could be partially amended by combining the tool with a static model generation as in [5], or by using a limited amount of manual modeling.

The test-cases determine the context of the model, and in the same way that a photographer cannot take a picture from a new perspective without seeing it, model generation cannot make a model for a new test-case without executing it. One model could however include a set of test-cases, in which case the optional state-probes should be used to distinguish disjunct behaviors.

Obtaining an accurate measure of the start of a job is often difficult. Regarding periodically triggered tasks, the quality of the measure is given by the probing technology used: an observability problem (see Page 28) is often present as the probes cannot see exactly when a task is placed in the ready-queue of the operating system, but see only when it receives its first time quanta to execute (see Section 2.4.5 on operating system probing support). Regarding event triggered tasks, depending on how the probing was realized, the first event of the task is either a preemption event or a *read from IPC queue, initiate* event (see Table 5.1).

The drawback with the optional state-probes is that their use normally requires access to the implementation code and the possibility to modify it – the mandatory probes on context-switches and system calls can be added in an operating system abstraction layer. Modifying the source, however, requires a white-box rather than a black-box view of the implementation. Additionally, there is also significant risk that the probing activity is flawed; if a subset of state-updates are missed, this can lead to bad models. It may be possible to avoid these problems if the application is using a data-base such as that described in [82]. In such a system, the data-base can be accessed by an observing probe transparently from the system and without treating the system as a white-box. However, without the ability to use the source code as guide, it could be difficult to find the most suitable set of variables to record. The options are to either perform an exhaustive search, or to use profiling to find variables that are modified in patterns that are representative of their importance (e.g. once per job).

Currently, we only support two system calls: send and receive over inter-process communication queues. This is indeed a limiting factor, but we expect no problems in extending our method to support other system calls such as semaphore operations etc.

Further, the probabilistic nature of the models may lead to that worst case execution times are over or under estimated, and that best case execution times are under estimated: Imagine a trace through the model of a task that passes two execute-statements in the same job of the task. In the real implementation, it may be that executing for example a low time-count in the first execute-statement will lead to that the second statement must execute a high time-count. This implicit knowledge is not necessarily incorporated in the model, which is why the distribution of modeled execution times may cover a larger interval than what is actually possible in the running system. This can of course be amended by incorporating the optional state-information into the recording effort, but requires relevant variables with respect to task control-flow to be identified and included in the probe configuration.

Chapter 6

Model validation

As the model generation phase of model extraction may have to be iterated in order to find an appropriate probe setting and a suitable recording length that result in an acceptable model, efficient automatic modeling requires automated model validation. This chapter shows how to perform validation by testing the generated model of each task (in the form of a modset) against a new set of recorded traces (a set of recseqs) obtained by recording the execution of the modeled system. The test serves to determine whether more, longer, or more detailed traces are needed for model generation to produce models of the required quality. To this end we need techniques to compare the modset with the set of recseqs.

The necessary techniques can be found in automata theory; model checking allows us to compare the two, since it is possible to formulate the inclusion checking as a reachability problem.

To achieve our goal, we can either translate recseq and modset into corresponding timed automata, or develop new tools suited for our existing datastructure. We choose the first option as we believe it is the least demanding. As a part of our solution, we will evaluate if pairs of intervals overlap. Timed automata allow a simple notation for specifying such evaluations. Therefore, we choose to develop a translation from recseq and from modset to timed automata.

We follow the definitions of timed automata by Bengtsson et al. [10], which extends Alur and Dill's original timed automata [3] by adding integer variables (see Section 6.3.1). Using integer variables allows for expressing causality between jobs (for example modeling mode changes) and for communicating action-properties between automata.

The two types of automata are constructed so that the automaton for the model of a task (a modset) is a tree-like structure, whereas the automaton for each recording (each recseq) is sequential. A proof that the recorded traces are contained in the model is constructed by using model-checking to verify the reachability of the final state of each trace automaton when composed with the model automaton. The objective is to show that the sequences of actions and action-properties in the recseqs do not contradict the modset. The modset is discarded if any of the tests fail. Otherwise, we conclude that the model is valid. Properties of the set of recseqs determine the *validity measure*: the level of confidence that can be placed in the final model.

Previously, model validation has been a manual process performed by people with knowledge in the system and/or in the modeling language. Balci [8] suggests a range of manual approaches for validation out of which this chapter automates one: *Predictive validation*, in which the model is provided with authentic inputs and its output is compared to that of the system. In this thesis, we implement this by extracting and analyzing intermediate data from model generation. Other manual methods for model validation have been presented by Sargent [92].

Similar to the work presented here, Szemethy and Karsai [106] present a method for translating their handmade SMOLES models of component based real-time systems into timed automata as a step in model-based development. Shu et al. [97] provide an automated translation from their extended version of UML to timed automata. Contrary to our work, the goal of both Szemethy and Karsai and Shu et al. is to perform system validation rather than model validation. Further, their work does not consider models with data state, and queries performed on the model have to be tailor made for the specific system.

6.1 Validating the selection of recordings

In order to achieve automated model validation in the way described here, we need to address the following 3 major issues:

A. Obtaining the automata. We need a translation from the modeling language to timed automata and from traces to timed automata.

B. Stopping criteria. It must be possible to determine when a sufficient confidence in model validity has been established.

C. Allowing leeway. Since the model is meant only to be an abstraction of the system that is modeled, the model should be allowed to differ slightly from the traces. The validation must be able to allow a user-defined leeway parameter for the model with respect to the system. A variable leeway will provide the user with the ability to decide the granularity of the solution taking into account the constraints on cost and effort as well as the precision necessary for the intended use of the model.

Since the solutions to C influence the solution to A, we present our solutions in the order in which they appear in our approach.

6.2 Allowing leeway as a precision parameter

A useful model should be an abstraction of the system it models. Since we are primarily interested in modeling real-time systems, this requires the model to be an abstraction of the system with respect to timing properties. Being an abstraction, the model cannot be a perfect reflection of the system; rather, it should provide a similar behavior while being significantly less complicated than the modeled system.

We choose to realize the abstraction, or leeway, by relaxing the timing requirements in the timed automata for recseq, thus providing the ability of passing validation even though the model does not exactly correspond to the modeled system. This could be implemented in several ways: According to definition, each execute statement in the recseq is based on a single observation in an execution recording of the system. Each such time observation could be manipulated with some function to describe a wider span than that single execution time. This would relax the requirements posed by guards on edges of the recseq-automaton that represent execution requirements. One example of such a function is to use percentages (i.e., each time observation t is replaced with the interval $(t - t \times p, t + t \times p)$, where p is a percentage). Another possibility is to allow a number of mismatches; e.g., for a recseq of X events, Y of these are allowed to lack correspondence in the modset. That would however require a more complex recseq-automaton than in the previous example.

In our implementation, we take an even simpler approach by replacing each time observation t in the recseq with the interval $(t - pp, t + pp)$, where pp is a user supplied *precision parameter*.

6.3 Obtaining the automata

Our method for model validation requires that a modset and a set of recseqs can be transformed into communicating timed automata. Once that is achieved, the modset-automaton can be paired with each recseq-automaton. Each pair is composed in parallel, and a reachability test is performed for the last location in each recseq-automaton.

To obtain the two automata required, we need algorithms to perform the transformation. We present such algorithms in Appendix B. Below, we describe these algorithms abstractly.

6.3.1 Timed automata

Timed automata was introduced by Alur and Dill [3]. Tools such as UPPAAL [11] and KRONOS [18] provide verification of systems of timed automata models through model checking of properties formulated in some temporal logic (e.g. TCTL [2]).

Being a kind of automata, a timed automaton is essentially a set of nodes and edges, where the edges connect the nodes in some pattern (each edge has exactly one source and exactly one destination). One of the nodes is the initial starting node from which traversal of the automaton can proceed to adjecent nodes, such that exactly one node is the current node at any given time. Associated with each edge is a *guard* that states the (temporal) conditions that must be fulfilled if the edge is to be taken and an action that can be used to synchronize with other automata.

We conform to a definition of timed automata as introduced by Alur and Dill [3], extended with integer variables by Bengtsson et al. [10]. However, our application of timed automata requires only a subset[1] of the original work:

Definition 10 (timed automata)**.** Let $\mathcal{C}$ be a finite set of real-valued variables ranged over by c and d. Let $\mathcal{W}$ be a finite set of variables with values in $\mathbb{Z}^* \cup \{z\}$, ranged over by x, y and z. Let Σ be is a finite alphabet of actions including the internal ε-action, ranged over by a and b.

Let $\mathcal{B}(\mathcal{C})$ be the set of clock constraints of the form: $c \leq n$, $c \geq n$, $c - d \leq n$, or $c - d \geq n$, where $c \in \mathcal{C}$, $d \in \mathcal{C}$ and $n \in \mathbb{Z}^*$.

Let $\mathcal{B}(\mathcal{W})$ be the set of variable constraints of the form: $x = n$, or $x - y = n$, where $x, y \in \mathcal{W}$ and $n \in \mathbb{Z}^*$.

[1]We do not use invariants or urgent locations, and the clock and variable constraints are simplified.

A timed automata $\mathcal{A}$ is a tuple $\langle Locations, l_0, Edges\rangle$ where:

- $Locations$ is a finite set of locations,
- $l_0 \in Locations$ is the initial location,
- $Edges \subseteq Locations \times Locations \times \mathcal{B}(\mathcal{C}) \cup \mathcal{B}(\mathcal{W}) \times \Sigma \times 2^{\mathcal{C}} \cup 2^{\mathcal{W}\times\mathbb{Z}^*}$ is the set of edges.

$\square$

Intuitively, a timed automaton with integer variables is a finite state machine with integer variables and real-valued clocks. Clock values are increasing over time, but individual clocks can be reset at any time. The value of integer variables can be changed instantaneously at any time. Associated with edges, guards are postulated and must be respected on transition over edges.

We write $l \xrightarrow{g,a,r} l'$ with $\langle l, l', g, a, r\rangle \in Edges$, where g is the constraint, or guard of the transition, a is the action that caused it, and r is the clock reset and variable update performed during the transition. Normally, ε is used to denote the internal action, which cannot synchronize with other automata. In many tools and representations, an action is associated with an exclamation mark to signal that it wants to perform a synchronization, or a question mark to signal that it offers a synchronization.

6.3.2 Example

To complement the presentation in this chapter, we continue the example from Chapter 5. In the current chapter, we focus on the validation of the periodically triggered task T_A, for which the recseq-automaton is displayed in Figure 6.1, and the modset-automaton is displayed in Figure 6.2.

Note that names of locations are noted next to each location (e.g. L5), and that the initial location has a circle inside it (e.g., L0 in Figure 6.1).

As a concrete example of possible transitions from one location to another, consider the edge from L2 to L3, in Figure 6.1. This edge defines that transitions from L2 to L3 are possible while the internal clock c_t is less than 6. If the transition is performed, the internal clock c_t is reset. For the transition to be performed, there must be a second automata that offers the action *snd?* (e.g., the edge between L1 and L4 in Figure 6.2).

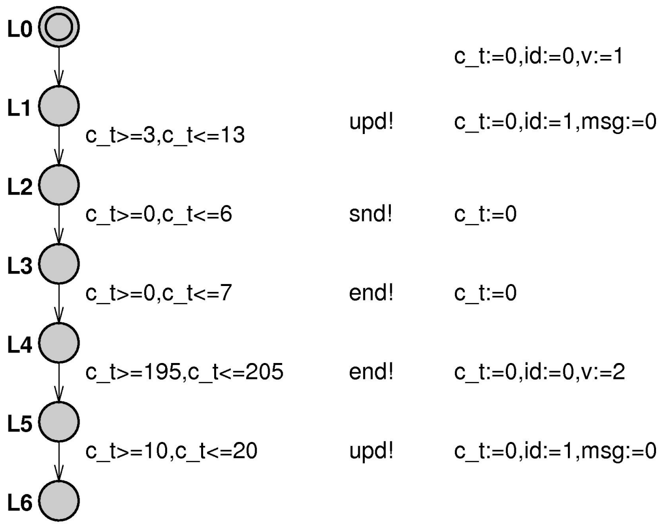


Figure 6.1: A timed automata representation of part of the recseq in Figure 5.3, using a precision parameter of 5.

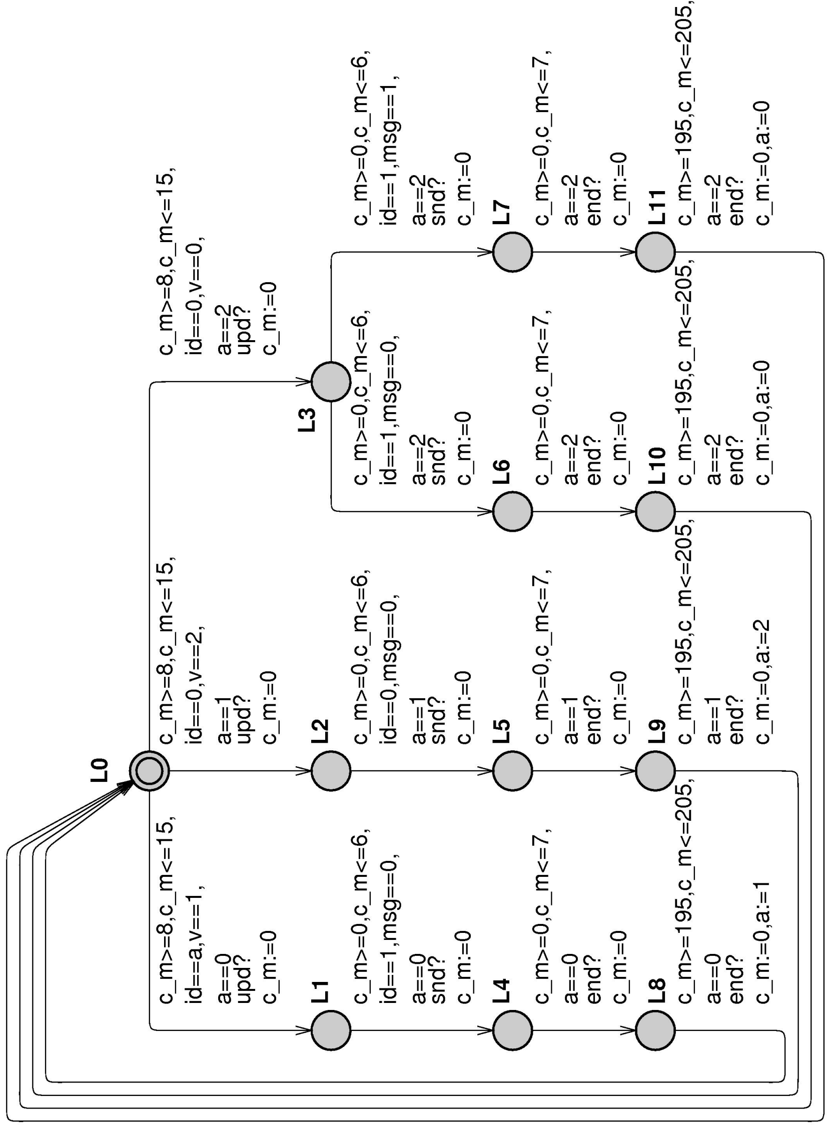


Figure 6.2: A timed automata representation of the modset in Figure 5.5.

6.3.3 General automata structure

A recseq-automaton consists of serial string of locations connected by edges: The initial location has no entering edge; i.e., there is no edge $l \xrightarrow{g,a,r} l'$ for the initial location l'. Each node has at most one outgoing edge; i.e., there is at most one edge $l \xrightarrow{g,a,r} l'$ for a given location l.

A modset-automaton is a simple graph: The initial location is the only location that can be reached from more than one location. Each path with an edge exiting the initial location can always reach the initial location.

As described in the previous chapter, there are five different action-types in recseqs and modsets: Send to IPC queue (*snd*), Receive from IPC queue (*rcv*), Variable assignment (*upd*), Execute (*exe*), and End job (*end*). The language of the automata is defined as $\Sigma = \{\varepsilon, \mathit{snd}, \mathit{rcv}, \mathit{upd}, \mathit{end}\}$. Hence, the action-type *exe* is not included (since each *exe* is translated to clock constraints, as explained below).

Out of the five action-types, *snd*, *rcv*, and *upd* will generate one edge in a recseq-automaton per occurrence. An occurrence of an *exe*-action in the recseq will not generate any edge in the automaton by itself, but the minimum and maximum of the execution-time distribution for the action is included in the guard on a local clock for the subsequent action (e.g., the edge between L4 and L8 in Figure 6.2). An occurrence of an *end*-action in the recseq will generate two edges in a recseq-automaton with the action *end*. The first of these edges checks possible execution time, and the second checks possible minimum suspension time as well as updates of the model data-state. Furthermore, the local clock is reset at every edge.

The modset-automata is constructed very similarly to the recseq-automata, but with two major differences: First, the modset-automaton does not add leeway to the execution-time guards on its local clock. Second, the modset-automaton maintains a data-state. Like the modset, the nodes of the automaton are labeled with the initial task model data-state of the job. This initial task model data-state is used in construction of guards on the automaton data-state on edges leading to each label. At the end of the job, the automaton data-state is updated according to the updates that where made during the job (e.g., see the edge between L8 and L0 in Figure 6.2). The initial task model data-state is added as guards to the edges of the modset-automaton (e.g see the edge between L0 and L1 in Figure 6.2). As a path through the modset may have a set of initial data-states, this yields a set of guards. Each separate guard is assigned to a separate edge, which may share the same destination node.

Next, we describe how action parameters are handled in the two automata:

The parameters must be checked to ensure that the action sequences in the recseqs and the modset have the same parameters. The general idea is that there is a set of global variables that are updated by the recseq-automaton according to its action parameters, the modset-automaton adds guards on its edges to control that the assignments from the recseq-automaton describe an event that is in the modset (e.g., see the edge between L1 and L2 in Figure 6.1 and the edge between L0 and L1 in Figure 6.2). In order for this to work, it is required that the recseq-automaton makes its assignments to the global variables *before* the synchronization of the action-types. Therefore, the recseq-automaton starts with an edge with an ε-action that performs the updates required for the parameters of the first event in the recseq (see the edge between L0 and L1 in Figure 6.1). These assignments will then remain phase-shifted to the event that they describe throughout the automaton.

6.4 Stopping criteria

We use two validity measures on the observed jobs to determine whether we have performed enough testing to make us sufficiently confident that the model is an adequate abstraction of the system. We also require that the validation fulfills certain user defined criteria based on these measures:

- *Completeness measure*: i.e., the probability that the model can replicate any job that the system can exhibit. It is desirable for the model to have a high completeness measure.
- *Accuracy measure*: i.e., the probability that the system exhibits a particular job. This measure must be sufficiently close to the probability that the model exhibits an equivalent job.

To this end, we need a new notion for estimating equivalence between observed jobs. Using such a notion, we will be able to group the jobs into classes and estimate the above measures based on the properties of the classes.

We say that two jobs of a task are syntactically equivalent iff their action sequences, variable updates, inputs, and outputs coincide.

Intuitively, syntactic equivalence can be tested using a time abstracted version of the timed automata test described above (i.e., a test that disregards timing).

The completeness measure is assessed by analyzing the frequency with which new Syntactic Equivalence Classes (SECs) are discovered in the recording of the system. This set is denoted $SoSECs$. We start out with the set of

SECs identified in the recordings used to generate the model (see Chapter 4), and we assume that the probability of discovering a job of an unknown SEC follows a binomial distribution.

There is a plentitude of ways to determine how many recseqs need to be tested to satisfy a given completeness requirement. Essentially, we want to know the number of tests (i.e., recseqs) needed to attain a certain confidence in the modset. For example, if we state the null hypothesis h_0 that there are more SECs than have been discovered so far, with probability $\pi_0 \leq 0.01$ of falsely rejecting h_0, with significance $\alpha = 0.01$ and margin of error $\varepsilon_0 = 0.01$, we can estimate that the required sample size is lower than $1,000$ according to Jeffrey's $100(1-\alpha)\%$ confidence limits [85].

The produced models are probabilistic – that is, the model can use probability to determine selections at behavioral level – which requires that the model has a valid estimate of the probabilities of different SECs. The *accuracy measure* should determine if these estimates are valid; i.e., within allowed tolerances.

Let G denote the set of jobs used to generate the model, and V the set of jobs used for validation. Internally, we group the jobs into SECs and analyze the probability distributions of equivalence classes in the two sets G and V. For the analysis, we let the function $sec\_cnt : 2^{job} \times SoSECs \rightarrow \mathbb{Z}^*$ denote the number of jobs of a SEC in a set of jobs. The function $h : SoSECs \times 2^{job} \times 2^{job} \rightarrow \mathbb{Z}^*$ compares the probabilities of a SEC n in G and V.

$$h(n, G, V) = \left| \frac{sec\_cnt(G, n)}{|G|} - \frac{sec\_cnt(V, n)}{|V|} \right| \tag{6.1}$$

The lower $h(n, G, V)$ is, the more accurate the model is. We specify an accuracy threshold h_a that is the maximum allowed difference between probability of equivalence classes in the two traces. The objective is to ensure that, for all identified equivalence classes n, h is below this threshold, i.e.:

$$\forall n \in SoSECs : \quad 0 \leq h(n, G, V) \leq h_a \tag{6.2}$$

Intuitively, if $h(n, G, V)$ is in the specified interval, the accuracy measure applied to both sets of system recordings is within the specified bound. This shows that the distribution of behaviors (i.e., distribution of SECs) described by the recordings is a representative sample of the behavior distribution of the system. Thus indicating that the probabilistic model is likely to emit the same distribution of behaviors as the system.

6.5 Analysis

The intention is that, after both the modset and the set of recseqs have been transformed into their timed automaton (given some precision parameter), then their co-simulation is possible. In fact, if model generation has succeeded, there should indeed be a way to co-simulate each possible pair of modset-recseq-automata so that the last location of the recseq-automaton is reached. This property can be checked via reachability analysis, using the same techniques as in a dedicated model checker for timed automata (e.g. UPPAAL [11]). Due to the generality of the model-checking algorithms and the number of physical locations for our models, this is unfortunately feasible only for small examples. For realistic examples, the number of locations in the automaton is too big for a general purpose model checker like UPPAAL. This is mostly due to the recseq-automaton, which typically has more than $5,000$ locations. However, our requirements are quite simple: we test only one property of two automata with one clock each, and out of which at least one automata (the recseq-automaton) has a maximum of only one outgoing edge per location and no edge out from the accepting state. For this scenario, it is quite straight forward to implement a lightweight model checker that can perform only this test (i.e. check for the inclusion of a finite timed trace in the model). We have implemented such a model checker [37]. The complexity of the test is linear to the number of edges in the modset-automaton multiplied by the length of the recseq – even for traces of up to $5,000$ events, the computation time for our tailor-made model checker is in the order of seconds on a Pentium 4.

The argument for why this test will validate the version of the model produced by model generation is as follows: As the last location in the recseq is reachable, the recseq is included in the model. As the recseq is included in the model, the model refines the recseq, and therefore the model is a representation of the recseq which in turn is a representation of the implementation. This chain links the behavior of the model to the behavior of the system. It is important that the set of recordings used to generate the recseq-automaton is disjunct from the set of recordings used to generate the modset and the modset-automaton, otherwise this would be a circular argument.

The measure of the precision parameter required for the test to succeed is a measure of the quality of the model. Viewing the test as a function of the precision parameter only (i.e. with the same recordings and the same model), yields a monotonic function of the precision parameter. The function starts at $false$, and subsequently, turns $true$, and then remains true. As the function is monotonic, we can use binary search [51] to find the smallest precision parameter for which the function is $true$. This value is then used to express the validity of the model.

6.6 Discussion

In this chapter, we have presented a method for validation of models that are automatically generated from execution recordings.

The method validates that the recordings that are used to generate the model are representative for the test-cases used in model generation. Validation is performed by testing to certify that the functional behavior of the model and the system do not differ, and that the temporal behavior of the model follows the temporal behavior of the system.

This type of validation is essential for model extraction. The validation answers two questions: Firstly, is the risk that the system can exhibit a behavior not captured in the model sufficiently low? Secondly, is the distribution of behaviors in the model similar to the distribution of behaviors in the system?

Admittedly, the result of the validation (whether new event sequences are discovered and the smallest achievable precision parameter) is highly dependent on the test-case selection, and in particular the relation between the test-cases used for model generation and those used for model validation. We are here implicitly assuming that the two sets are non-overlapping but similar; i.e., derived using the same strategy or method. A further investigation of the generality and validity of our model generation and validation is needed, yet it is outside the scope of this thesis.

Chapter 7

Case study: Automatic modeling of an industrial robot

In this chapter, we present a small case study performed at ABB Robotics to evaluate model extraction from an industrial perspective.

7.1 System overview and limitation

In the case study, we study the ABB Robotics state of practice industrial robot control platform IRC5.

The > 2500 KLOC[1] object-oriented C-code base of the robot runs under VxWorks 5.5 on Intel Pentium 3 industrial PC hardware. We study the main computer (MC), running almost 100 tasks with preemptive fixed priority scheduling. Many of the tasks are event triggered, typically executing one of several services requested by the triggering task.

Motion control is a critical subsystem of the MC, responsible for generating motor references and brake signals to a DSP that in turn controls the physical robot. The DSP issues requests to the MC with a fixed rate ($f > 200$ Hz). It is critical that the MC replies to each request within a given time. The motion control subsystem consists of tasks *A*, *B*, and *C* (see Figure 7.1). Task *A*, with

[1]KLOC: Kilo of Lines of Code, is an approximate measure of the system size.

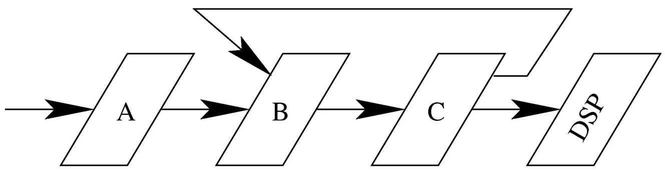


Figure 7.1: Robot motion control architecture.

low priority, calculates the motion control commands on a high level of abstraction and submits results to task *B*. Task *B*, with medium priority, communicates with *C* and *A* to reduce the abstraction of the motion control commands from *A*. Task *C*, with high priority, is responsible for maintaining the communication with the DSP.

In the experiment, we have focused on tasks *A*, *B* and *C*, whose implementation consists of more than 250 KLOC. The test-case that we analyzed was a complex, fast, moving pattern, where the robot moves short distances and halts its movement at designated coordinates. This pattern will result in high calculation intensity for the recorded tasks.

Probes are introduced to record context switches, explicit delays, selected variable updates, IPC-send, and IPC-receive events. Each probe generates 6 bytes, and takes about $0.8\mu s$ to execute, which is quite negligible w.r.t. the task execution times, which are often measured in milliseconds.

As we focused on only a small subset of the task set on the MC, simulating the behavior of the model required the use of an intricate manually produced environment model. As it turned out, this environment model affected the quality of the simulation.

7.2 Information extraction

In VxWorks, a concept of *hooks* allows insight in the dynamics of the operating system. Hooks are offered at special operating system events, and by hooking a piece of code to these events, the dynamic information about the events can be accessed. This setup facilitates performance monitoring and recording.

In the case study, we used hooks to access the context switches that occurred during execution. The hook-function receives pointers to the task control blocks (TCB's) of both the interrupted task and the interrupting task. Using these pointers, it is possible to access, e.g., the stack area of the task. However, we are only interested in the identity of the tasks involved in the context switch and the time at which the switch was triggered.

As the system platform is an Intel Pentium, the `RDTSC` assembler instruction can be used to read the processor cycle counter (see Section 2.4.4 for details about this instruction). This instruction provides us access to a reliable and accurate clock of fine granularity.

The system calls that we are interested in are mainly IPC-send and IPC-receive operations. Though the VxWorks hooks present a good method for accessing information about the system, only a limited number of events implement them. For example, IPC-queue operations do not provide a hook-interface. However, in the particular system that we consider, these operations are wrapped in a small middle-ware to support easy portation of the application, in particular to a simulator platform. The existence and extensive use of this wrapping middle-ware allowed us to insert probes at the sending and receiving on IPC-queues. As we knew that the portation of the software worked and was extensively used in development, we relied on that all operations of significance used the wrapping middle-ware.

Consequently, the only probes that had to be inserted into the actual application software where the probes that recorded the data state of the application. As the risk of introducing an error in probing is somewhat proportional to the number of probes that must be introduced, the relatively few probes we needed to insert increases our confidence in the correctness of the monitoring.

7.3 Model extraction

The search for the appropriate probing configuration, i.e., suitable variables to include in the data state monitoring, was difficult in that we are largely unfamiliar with the intricate functionality of the system as well as the use of data state in model generation. We had to make five iterations before finding a satisfactory probe configuration. In the final setting, we had introduced new variables to make behavior that we had observed explicit; implicit rules embedded in the construction of the system prevented certain behavior, this was made explicit and thus recordable by introducing new variables and variable updates. The TraceAlyzer (presented in Section 4.5.2) proved to be a handy tool in this work, and model validation effectively helped us to discard unsuitable probing configurations.

After having setup the equipment used, each probing configuration took approximately four hours to introduce and use, including building new exe-

	Execution time percentiles	Task A	Task B	Task C
Measured	Minimum	3	12	2
	25:th percentile	13	807	13
	50:th percentile	110	851	29
	75:th percentile	292	867	708
	Maximum	21281	2821	1117
Simulated	Minimum	4	20	6
	25:th percentile	13	735	29
	50:th percentile	121	852	115
	75:th percentile	356	926	833
	Maximum	23017	2723	1164

Table 7.1: Execution time distributions from system execution and model simulation.

cutables and recording of executions. This time could probably have been decreased significantly with better system knowledge and experience in the dynamics of the model extraction process.

7.4 Results

After a few unsuccessful attempts, when the model validation showed probing to be insufficient, we successfully generated models of the three tasks based on five recordings (the final probe setting generated approximately $11,000$ events per second). However, during validation using five other recordings, it was discovered that the two recorded variables in task *A* required a leeway in excess of 10 percent of the task's Measured Worst Case Execution Time (MWCET) to pass all tests (see Figure 7.2). After we had iterated the information extraction two times, we found a set of variables that when probed gave better data-state information. This allowed us to pass the test with less than 10 percent leeway. In the final information extraction setting, we recorded two variables in task *A*, one in task *B*, and one variable in task *C*.

As we analyzed the stopping criteria for the final set of recordings we found that, even though the completeness measure was very low (our sample size for validation was only five recordings), the accuracy measure was $h_a \geq 10\%$, as

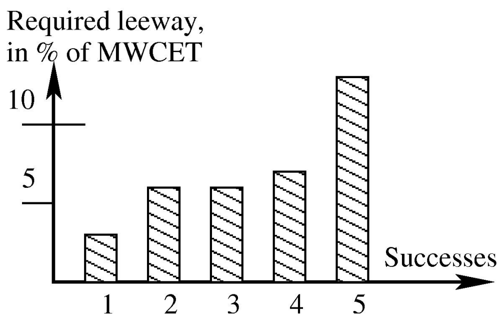


Figure 7.2: Leeway requirements for the first probe setting during the modeling of task A.

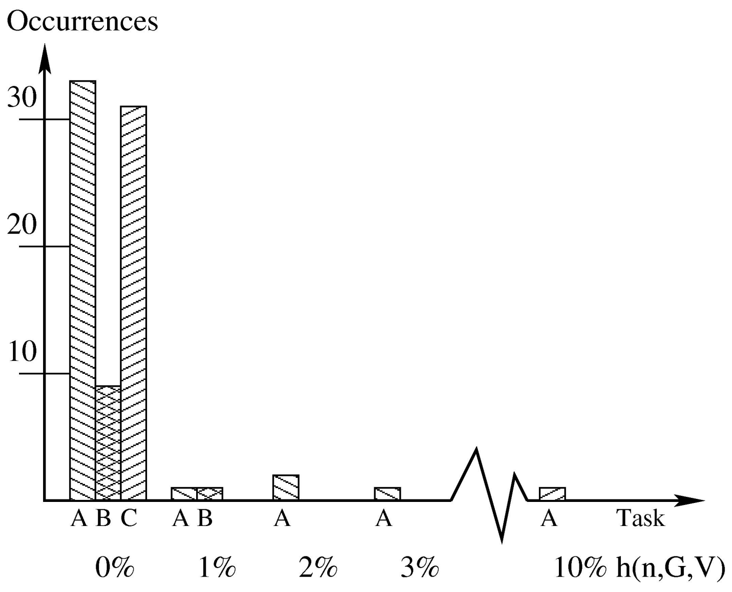


Figure 7.3: $h(n, G, V)$ distribution of all tasks.

can be seen in Figure 7.3. This is probably due to the length of the recordings: each recording was approximately 22 seconds long and spanned thousands of jobs.

When validated models had been obtained for all three tasks, we merged the models and co-simulated them together with a hand made environment model that partly emulated the remaining tasks on the MC. Thus allowing validation of the collected model from a system perspective. We performed a large series of simulations, using several different starting conditions. During these, we could observe a strong similarity in the behavior of the model and the legacy system. However, as the accumulated model does not represent the entire system, we cannot perform realistic measures of response times. We can however analyze the execution time distributions for the three tasks.

Comparing the execution times measured on the system with that of simulations (see Table 7.1), it is evident that there are deviations from the expected results: Due to limitations in the handmade environment simulation, the event triggered task *C* did not receive input as intended. This insufficiency in the environment model caused many short jobs never to be triggered in the simulation. This is seen in the 25^{th} and 50^{th} percentile of task *C*. With better environment modeling or with larger scope in the case study, this error could have be remedied.

The final model, including the environment model, occupied 4.0 KLOC without any optimizations, which is less than 1.6% of the size of the implementation.

7.5 Discussion

There are a few points that we learned regarding the practical work:

- The many probing configurations resulted in a large number of different recordings. As this number grew, it was difficult to remember the details of the configuration; i.e., which variables are probed and what are their functional meaning. Industrial application of our model generation would require an infrastructure to manage this information.
- Some functionality is implicit by the construction of the system. To model this it may be required to make the functionality explicit by introducing new variables.
- The use of a middle-ware to wraparound IPC system calls, and the open

nature of this that allowed us to modify the middle-ware, significantly eased the introduction of probes.

- More accurate methods than the one we used are needed to model the environment. We provide methods to model the system, but the reactive nature of the system often requires that also the environment is modeled; if the system is triggered by the environment, simulation requires that also the environment is modeled.

Chapter 8

Quality assessment

In this chapter, we present a framework for empirical testing and comparison of extracted models with respect to response time distributions. We present a set of system definitions with varying architectural styles to be used as benchmarks in this framework. We discuss the difficulties in comparing response time distributions, and present an intuitive and novel approach along with an associated algorithm for performing such a comparison. Using the framework, benchmarks, and the comparison algorithm, we present an empirical evaluation of the proposed model extraction.

In our evaluation, we consider the following qualities:

Generality The use of model extraction in an industrial setting requires that engineers are confident in that model extraction can produce models of their system. Therefore, the technique for model extraction should be evaluated with respect to generality by answering the questions: *"Which types of systems can be modeled? Which types of systems cannot be modeled?"*

Stability Model-based impact analysis as described in Section 2.3.2 requires that the model is *stable* with respect to the system and the change: i.e., when a change is applied to a system and an extracted model, the changed model should be valid with respect to the changed system. Stability should be considered only in those cases where generality applies, i.e., in the cases where both the original system and the changed system are in the subset of systems that model extraction can model.

Early versions of the evaluation framework and comparison algorithms are presented in [38].

8.1 Assessing the generality and stability of automatic modeling

The evaluation framework presented in this section serves to evaluate model extraction with respect to generality and stability. Generality is evaluated by extracting models from a diversity of system types. Assuming that any extracted model can also be obtained by manual modeling, we then proceed to extract models from variations of the system types. Variations to the system types simulate manual changes to the systems, and the models extracted from variations of the system types simulate manual change to models. Positive evidence of stability is provided if we can establish that significant variations in the system leads to significant variations in the model while generality persists.

8.1.1 Framework for empirical evaluation

In this section, we introduce the framework for empirical evaluation depicted in Figure 8.1. We use a notion of *archetypes* to describe architecturally different system designs. For example, a system consisting of a set of periodic tasks without inter-process communication (IPC) is a different archetype than a system consisting of event triggered tasks where the exchange of messages trigger execution.

To each system definition, which implements an archetype, we can add an increasing portion of *Population*, *Imperfectness*, or *Complexity* (PIC) for each test performed on the system.

Imperfectness regards the number of probes in the system that generate recordings. With low imperfection, all relevant information is recorded. Complexity regards the task complexity; e.g., the number of task instances and amount of environment stimuli. Finally, population regards the number of task-types in the system and the number of recordings used in model extraction.

For each system definition and PIC combination defined, we perform two sets of executions of the system: recordings from the first set are used to generate the model and recordings from the second set are used to validate the model (as described in Chapter 4).

Varying the PIC will serve two purposes: It will further test generality, and it will test the stability of the model extraction with respect to changes in the

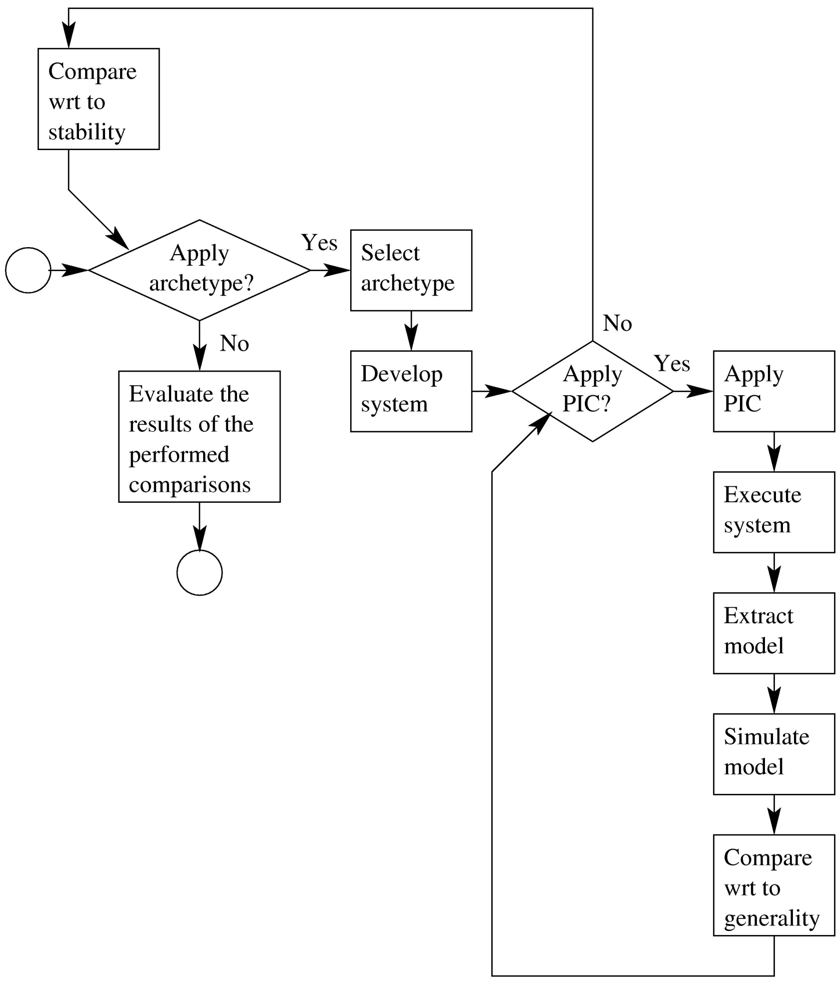


Figure 8.1: A process-view of the framework for empirical evaluation.

system.

We perform two sets of comparisons of response time distributions of system executions and extracted model simulations; the first comparison tests generality and the second one tests stability:

Generality is tested by comparing the simulations of the extracted model to new executions of the system. The collected set of such comparisons can then be analyzed to evaluate the generality of the method of model extraction. This test is performed in the step "Compare wrt to generality" in Figure 8.1.

Stability is tested by comparing executions of each system-PIC combination to simulations of all other systems within the same archetype but with different PICs. The corresponding comparisons are performed for each model. For stability to exist in a given archetype, the comparison between one model of PIC X and one model of PIC Y should have a relation similar relation to the comparison between the system of PIC X and the system of PIC Y (i.e., if the systems are different, the models should reflect this relation and vice versa). This test is performed in the step "Compare wrt to stability" in Figure 8.1.

As the set of archetype-PIC combinations is intended to be large, it is desirable to automate the comparisons. We present a novel automated measurement in Section 8.3 that can be used in such a comparison.

The decisions in the empirical evaluation framework are taken based on the availability of untested archetypes and PICs respectively. The manual implementation steps of the process are "Develop system" and "Apply PIC": The first of these is to develop a general implementation of a given archetype (given be the previous step "Select archetype"), this is performed only once per archetype. The second mutates a general archetype implementation with respect to some PIC (given by the previous step "Select PIC"), this is performed only once per archetype/PIC-combination. If any of these steps have already been performed for the given input in a previous evaluation, the old implementations can be reused. At the end of the evaluation, the manual step "Evaluate the results of the performed comparisons" is responsible for compiling and processing the obtained comparison results from the steps "Compare wrt to generality" and "Compare wrt to stability". This last step is inherently dependent on the method of comparison used to obtain the comparison results.

8.1.2 Archetypes and PIC

The following archetypes are used in our study:

1. **Client-server without reply.** This archetype describes a common design pattern in the industrial systems that we have encountered. A client sends varying service requests to a server that services the requests. Results of the computations may affect the environment or successive requests to a third or fourth task. PIC applied are priority ordering, frequency increase, execution time fluctuations, and accuracy in logging of data state.

   Specifically, this archetype is implemented with two tasks T_1 and T_2. With a fixed periodicity, Task T_1 sends a message to Task T_2 that reacts on the contents of the message. We distinguish between four different contents, representing four different commands, plus one default behavior in the case that the content is unrecognized. Task T_1 maintains a variable to keep track of the data sent, and task T_2 maintains a variable to keep track of the data received.

   We have defined four variants of PIC: At PIC 1, T_1 has lower priority than T_2. The relation is reversed in the other PICs. At PICs 3 and 4, the periodicity of T_1 is shortened and the execution time distribution is widened for both tasks. At PIC 4, the data state recording is inactive in both tasks.

2. **State machine.** Here, a task acts as a state machine which makes one transition per job. Transitions are triggered by messages from the environment or from another task. Task mode changes can be expressed by this archetype. In contrast to the client-server archetype, the same message can trigger different behavior at different points in time. PIC applied are reduced recording of variable assignments (simulating poor probing), environment stimuli, complexity of the state machine, priority ordering, frequency increase, and execution time increase.

   The archetype is implemented by two tasks T_1 and T_2. With a fixed periodicity, Task T_2 sends a message to Task T_1 with randomly selected contents 0 or 1. The event triggered Task T_1 consists of a finite state machine that can make one state transition per job. The target state of each transition is depending on the contents of the triggering message from T_2. A variable is maintained to keep track of the current state.

We have defined four variants of PIC: At PIC 1, T_1 has lower priority than T_2. The relation is reversed in the other PICs. At PICs 3 and 4, the periodicity of T_2 is shortened. At PIC 4, the distribution in execution time is widened in both tasks.

3. **Purely periodic without communication.** A task set of periodic tasks where execution times for any given task varies randomly between jobs within determined intervals. In this case, the PIC consists of increase of the task set size.

   In our experiments, the implementation consists of at most seven periodic tasks T_{1-7} that execute a bounded random interval in each job. For each task, the worst case execution time (WCET), the period (T), utilization (U), and analytical worst case response time (R) are described in Table 8.1.

   We have defined four variants of PIC: At PIC 1, tasks T_4, T_5, T_6, and T_7 are active. At PIC 2, tasks T_3, T_4, T_5, T_6, and T_7 are active. At PIC 3, tasks T_1, T_3, T_4, T_5, T_6, and T_7 are active. At PIC 4, all tasks are active.

	WCET	T	R	U
T_1	10	80	10	12.5%
T_2	30	120	40	25.0%
T_3	20	160	60	12.5%
T_4	15	180	75	8.3%
T_5	30	200	115	15.0%
T_6	40	300	300	13.3%
T_7	80	1000	960	8.0%
Σ				94.6%

Table 8.1: Maximum utilization for Archetype 3.

4. **Feedback loop.** Here, tasks exchange messages in a loop. Examples include client-server with reply, and a feedback control system. The PIC consists of priority ordering, message complexity, message reply frequency, and environment stimuli.

   The implementation consists of seven tasks, four of which (T_1 to T_4) are implementing the feedback loop, and the three remaining are concurrently executing a client-server without reply (T_5 and T_6) and a simple periodic task (T_7). Task T_2 maintains a variable that contains its internal data state. The architecture of the feedback loop is as follows: provided that it does not receive a command from T_4, T_1 periodically sends a message to T_2. Task T_2 forwards the message to T_3, and sometimes also to T_4. Task T_3 reacts on the message type by varying its execution time. When triggered, task T_4 sends a message to T_1.

We have defined four variants of PIC: At PIC 1, only the tasks implementing the feedback loop are active. At PICs 2, 3, and 4, T_5 and T_6 are also active. At PICs 3 and 4, T_7 is active. At PIC 4, the data state recording in task T_2 is inactive.

5. **State machine feedback loop** This archetype is a combination of the two archetypes 2 and 4, as is the PIC.

   The implementation consists of five tasks, T_1 to T_5. Tasks T_1 and T_2 implement the state machine feedback loop; both tasks are state machines, T_1 generates input to trigger T_2, T_2 generates input that, if available, will affect the execution of T_1. Task T_2 maintains a variable that keeps track of its data state. Tasks T_3 and T_4 implements a client-server without reply, and T_5 is a periodical task.

   We have defined four variants of PIC: At PIC1, the execution time requirements are lower and with less spread. At PICs 1 and 2, T_1 has a higher priority than T_2. The relation is reversed in the other PICs. At PICs 3 and 4, tasks T_3, T_4, and T_5 are active. At PIC 4, data state is not recorded.

In the test-bed setup, it is important that we are able to compare a model and a system and quantify the model's resemblance with the system (see Figure 8.1).

8.1.3 The system platform simulator

We have implemented an instance of the framework, where the system platform is a multitasking, fixed priority scheduled, instruction-set simulator that we have developed specifically for this purpose. Using our own simulator allowed complete control over execution and recording, and saved us from integration problems. The simulator is architected as follows:

The simulator can handle 16 types of instructions (Table 8.2), including absolute and relative sleep, branching, IPC, explicit logging, register manipulation and testing, and random number generation. Each instruction takes one clock cycle to complete. All instructions have two parameters.

TST_	<r>	< >	Test the contents of register r, updates the status register (SR) to SR_EQ_ if the contents is zero, to SR_NEG if the contents is less than zero, and to SR_POS if the contents is more than zero.
CMP_	<r>	<r>	Compare the difference between the contents of the first register and the contents of the second, updates SR to SR_EQ_ if the difference is zero, to SR_NEG if the difference is less than zero, and to SR_POS if the contents is more than zero..
BNEG	<l>	< >	Branch to label l if SR is SR_NEG.
BEQ_	<l>	< >	Branch to a line l if SR is SR_EQ_.
B___	<l>	< >	Branch to label l.
INC_	<r>	<r>	Increase contents of the first register with the value of the second.
LOAD	<r>	<d>	Set the contents of register r to an integer value d.
SND_	<q>	<r>	Send contents of register r on IPC-queue q (contents of r must be positive).
RCV_	<q>	<t>	Receive from IPC-queue q, store in register r1, time-out t clock ticks.
OUT_	<r>	< >	Print contents of register r to screen.
YREL	<t>	< >	Yield a time t relative to the starting time of the current job.
YABS	<t>	< >	Yield an absolute time t from now.
NOP_	< >	< >	Do nothing but consume one clock cycle.
RAND	< >	< >	Get a random number in register r1.
MULO	<r>	<d>	Modulo the contents of a register r with the value d - store result in register r1.
LOG_	<t>	<r>	Record an entry with event type t and data from register r.

Table 8.2: Instruction-set of the simulator.

```
{
  IPCS 3
  OUTPUT 0
  TICKS 65535
  TRACEFILE et.txt
}
At2{
  PRIO 11
  TRIG 2
  FILE At2.def
}
At1{
  PRIO 10
  TRIG 0
  FILE At1.def
}
idle{
  PRIO 255
  TRIG 0
  FILE idle.def
}
```

Figure 8.2: A system definition file.

```
LOAD   2 1
SND_   2 2
NOP_   0 0
YABS 300 0
LOAD   2 2
SND_   2 2
NOP_   0 0
YABS 300 0
LOAD   2 3
SND_   2 2
NOP_   0 0
YABS 300 0
NOP_   0 0
NOP_   0 0
NOP_   0 0
YABS 300 0
B___   0 0
```

Figure 8.3: A task definition file.

A system of tasks is defined by a system definition file (Figure 8.2) together with an assembler-file per task (Figure 8.3).

The system definition file defines a set of parameters for the system:

- The `IPCS` property defines the number of IPC queues in the system.
- The `OUTPUT` property enables (if set to `1`) a history of the simulation to be printed to screen. The printout is usable for debugging the system implementation.
- The `TICS` property defines the length of the simulation counted in clock cycles. As the system is executed, occurred task switches, performed IPC, and executed explicit log instructions are automatically recorded in a system specific log.
- The `TRACEFILE` property indicates the file where the system specific log is stored.

After the system parameters, each task is defined by three properties:

- `PRIO` is the priority of the task, 255 is the lowest and 0 is the highest.
- Each task is either defined by the `TRIG` property as triggered periodically or triggered as a result of input on an IPC queue. If `TRIG` is larger than zero, the value denotes the identifier of an IPC queue, otherwise the task is periodic and the periodicity is defined by the implementation in the task's assembler-file.
- The assembler-file is identified by the `FILE` property.

8.2 Response time

In real-time systems, the temporal and functional requirements are equally important. We assume that the embedded real-time systems we consider consist of a set of *tasks* that can either be periodic or event triggered. As a task is triggered, a *job* (a task invocation) is executed for some period of time, after which the task will await until further triggering. We label the time measured from the point in time where the job is triggered (the *release time*) until the completed execution of a job as the *response time* (RT) of the job.

8.3 Comparison of sampled time distributions

The framework proposed above assumes that it is possible to objectively compare two distributions of, e.g., response times. However, comparing two series of measured response times is a non-trivial task. There are several methods available, but many fall short of expressing 'similarity' in a useful and intuitive way for the problem at hand.

The χ^2 test of independence [22] and the Euclidean distance metric are both *categorical* [1] in the temporal dimension, which results in that they are not sensible to the difference that two samples have almost the same response time if they are in different categories (timing intervals). They are only sensible in the sample dimension, which means that they can acknowledge that *almost* the same number of samples in both distributions have response times in the same category. This leads to unintuitive results due to false negatives.

The Kolmogorov-Smirnov test [75] assumes that one of the distributions in a comparison is mathematically modeled [52, 81]. However, the execution time distribution of a program is often very complex. On the source code level in a system implementation, selections where one path has a significantly longer execution time than the other are common. Execution time distributions that cover both branches of such selections do not follow a simple pattern. Therefore, it cannot be assumed that a response time can be classified to a known distribution (e.g., a normal distribution). We know of no universal method of determining or estimating similarity between unclassified finite discrete distributions.

We wish to device a measure that can express a 'similarity' between distributions in both the 'time' and 'spread' dimensions. The measure should be sensible to similarities between samples from the two distributions (i.e., two samples can be similar even if they are not equal); this is the 'time' dimension. It should also be sensible to similarities in the spread of the two distributions (i.e., two distributions can be similar if the spread in their respective distributions are similar).

As an example, we observe Figure 8.4. These two distributions are not identical, but share a general topological structure with two peaks consisting of approximately the same number of samples at approximately the same re-

[1]Categorical variables are also called *nominal variables*; the distance between categories of nominal variables is binary (either 0 or 1). For example, consider the four categories dog, cat, giraffe, penguin; the distance between any pair of these categories is either 0 (if the categories are identical), or 1 (if the categories are different); the distance between two different categories is not depending on which categories that are compared.

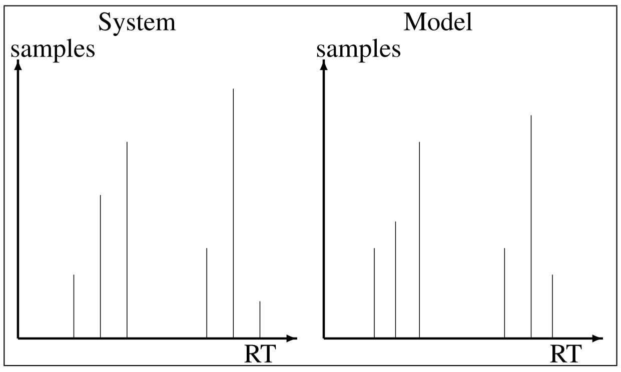


Figure 8.4: Two similar response time distributions, notice that the response times are not exactly the same and that the number of samples in the peaks differ slightly between the two distributions.

sponse time values. The method of comparison should be able to sense such similarities.

To implement a comparison that can express this abstract understanding of similarity, we introduce a novel objective measurement for sampled distributions based on the notion of *divergence* between two distributions, which we will introduce next.

8.3.1 The sum of divergence of two distributions

A distribution, or a series of sampled response times, *d* is represented as a binary tree where each node has the attributes *value* and *count*, these are referred to with a doted notation (e.g., *n.value* for node n of distribution *d*). The attribute *value* is the sampled response time and *count* is the number of samples of the same response time. All nodes in the binary tree have unique *value* attributes. The distribution has two operations *getLowest* and *remove*, these are called using a similar doted notation (e.g., *d.getLowest()*). The operation *getLowest* returns the node that has the smallest *value* in the distribution, it does not take any parameters. If the distribution is empty, 'invalid' is returned. The operation *remove* removes a number of samples of a certain value, the number and value are supplied as parameters to the operation.

To perform the comparison, the sizes of the two series of response times are first *normalized by size*: They are normalized by linearly adjusting the *count* of *value*'s such that the total sum of *count*'s in each of the two trees is equal to the Least Common Multiple (LCM) of the original total sum of *count*'s of both trees.

Definition 11 (sum of divergence)**.** Let U be the set of samples. There is a function $time : U \rightarrow \mathbb{Z}^*$. For a given sample $u \in U$, we use $time(u)$ to denote the value of that sample.

Let $A, B \in 2^U$ be two sets of samples from two sources (e.g., a model and a system) with equal cardinality. We define a *match* between these two as a *bijective* mapping between A and B, $\Delta_{AB} : A \rightarrow B$. Let C_{AB} be the full set of matches (i.e., the full set of bijective mappings) between A and B.

We are now able to define a measure of a match $\Delta_{AB} \in C_{AB}$ by the function $[\,] : C_{AB} \rightarrow \mathbb{Z}^*$ as:

$$[\Delta_{AB}] = \sum_{a \in A} |time(\Delta_{AB}(a)) - time(a)|$$

Then, the *sum of divergence* between two sets of samples is the measure of the most favorable match in the sense that the measure is minimized:

$$\text{sum\_divergence}(A, B) = \min_{\Delta_{AB} \in C_{AB}} \{[\Delta_{AB}]\}$$

□

Intuitively, for two equally sized sampled distributions A and B, Δ_{AB} describes a mapping such that each sample in A is paired with a unique sample in B (the uniqueness follows from that the mapping is bijective). The *sum of divergence* is then considering the best possible mapping in the sense that the sum of differences of response times that the matched samples represent is as small as possible.

Next, the measure must be normalized.[2] We let *distribution_size*(A, B) denote the normalized size of the two distributions A and B, and the function *max_divergence*(A, B) denote the maximum difference between samples of A and B. Then, normalization of the measure is performed as follows:

$$\frac{sum\_divergence(A, B)}{distributions\_size(A, B) \times max\_divergence(A, B)} \tag{8.1}$$

[2]In the same way the speed of a moving vehicle has low information content unless the current speed limit is known, the presented measure must be normalized in order to be useful.

That is, the mean of the measure is compared to the maximum of the terms of the measure. This yields a percentage in the interval $(0\%, 100\%)$.

Intuitively, a small value indicates that the divergence of the two distributions is small compared to the dispersement of the distributions, which should indicate that the two distributions have a close resemblance. Consequently, a high value indicates that the distributions have low resemblance.

8.3.2 An algorithm for measuring the sum of divergence of two distributions

In this section, we will introduce an algorithm for efficient calculation of the sum of divergence for two sampled distributions. The algorithm assumes that the distributions are normalized by size as described above.

Algorithm 1 $SumDivergence(i_1, i_2)$ calculates the divergence between two equally sized response time distributions i_1 and i_2.

```
Require: $DivSum \in \mathbb{Z}$
Require: $d \in \mathbb{Z}$
Require: $c \in \mathbb{Z}$
 1: $DivSum := 0$
 2: repeat
 3:   $e_1 := i_1.getLowest()$
 4:   $e_2 := i_2.getLowest()$
 5:   if $e_1$ is valid $\wedge$ $e_2$ is valid then
 6:     $d := |e_1.value - e_2.value|$
 7:     if $e_1.count < e_2.count$ then
 8:       $c := e_1.count$
 9:     else
10:       $c := e_2.count$
11:     end if
12:     $DivSum := DivSum + d \times c$
13:     $i_1.remove(e_1.value, c)$
14:     $i_2.remove(e_2.value, c)$
15:   end if
16: until $e_1$ is invalid $\vee$ $e_2$ is invalid
17: return $DivSum$
```

Intuitively, Algorithm 1 iteratively pairs and removes the smallest samples in both distributions, and terminates when both distributions are empty.

The theoretical complexity of the *SumDivergence* algorithm for two distributions A and B is $O(M_A \times M_B)$, which are normalized by size, where M_X is the number of unique sample values in distribution X. Typically, with M_X of approximately $1,000$, the algorithm takes in the order of seconds to execute.

8.3.3 Proof of that the algorithm implements the measure

Lemma 1. Algorithm 1 will eventually terminate.

Proof. Initially, the sets A and B are finite and normalized by size. Each iteration, $c > 0$ elements are removed from one distribution, and equally many are removed also from the other distribution. The algorithm is completed as both distributions are empty. □

Lemma 2. Algorithm 1 finds a match Δ_{AB}, a bijective mapping, between two distributions A and B.

Proof. The semantics of removing an element from a distribution is that the element has been matched with another element. In each iteration of the algorithm, equally many elements are removed from both distributions; the semantics of this is that a match has been established between the sets of elements removed; though there may be several possible matches between the elements of the removed sets, we assume that exactly one is chosen. That the algorithm finds a match is then given by the proof to Lemma 1. □

In the following, we prove by induction that the algorithm provides the minimal sum of divergence. We prove a general step:

Lemma 3. Given two distributions A and B of the same (unknown) size n, and assuming that $n-2$ samples are optimally paired in both cases (with respect to small sum of divergence): the optimal pairing of the remaining samples a, b, c, d where $a, b \in A$; $c, d \in B$; $a \le b$; and $c \le d$ is: $\langle a, c\rangle$ and $\langle b, d\rangle$.

Proof. With the pairing suggested by Lemma 3, $[\Delta_{A,B}]$ is:

$$[\Delta_{A,B}] = X + |a-c| + |b-d|$$

The only other possible pairing yields $[\Delta_{A,B}]$:

$$[\Delta_{A,B}] = Y + |a-d| + |b-c|$$

Thus, if Lemma 3 is not true, the following proposition should hold:

$$\text{P1:} \qquad Y + |a-d| + |b-c| < X + |a-c| + |b-d|$$

As the other samples are optimally paired, it follows that $X = Y$, which yields:

$$|a - d| + |b - c| < |a - c| + |b - d|$$

Dividing this into the possible cases yields three alternatives:

1. if $a < c < b < d$, then

$$-a + d + b - c < -a + c - b + d \rightarrow 2b < 2c, \text{ but } 2b \not< 2c$$

2. If $a < c < d < b$, then

$$-a + d + b - c < -a + c + b - d \rightarrow 2d < 2c, \text{ but } 2d \not< 2c$$

3. If $a \leq b \leq c \leq d$, then

$$-a + d - b + c < -a + c - b + d \rightarrow 0 < 0, \text{ but } 0 \not< 0$$

Thus, since all three alternatives are falsified, proposition P1 is false, and it follows that Lemma 3 holds. □

We then prove two base cases:

Lemma 4. For two distributions A and B where both distributions contain exactly one element each such that $a \in A$ and $b \in B$, the algorithm will find the match that yields the sum of divergence as per Definition 11.

Proof. The algorithm will find the only feasible match $\langle a, b \rangle$, which yields $[\Delta_{A,B}] = |a - b|$. □

Lemma 5. For two distributions A and B where each distribution contains exactly two elements such that $a, b \in A$; $c, d \in B$; $a \leq b$; and $c \leq d$, the algorithm will find the pairing that yields the sum of divergence as per Definition 11.

Proof. There are two possible matches:

1. $\langle a, c \rangle$ and $\langle b, d \rangle$, which yields $[\Delta_{A,B}] = |a - c| + |b - d|$, or
2. $\langle a, d \rangle$ and $\langle b, c \rangle$, which yields $[\Delta_{A,B}] = |a - d| + |b - c|$.

As given by the proof to Lemma 3, the first match will never yield a higher divergence than the second. As the algorithm will choose the first match, Lemma 5 holds. □

Using the general step and the two base steps, we then prove the correctness of the algorithm by induction:

Lemma 6. There is no other match such that the resulting sum of divergence is smaller than in the match found by Algorithm 1.

Proof. The algorithm iteratively pairs and removes the smallest samples in both distributions, and terminates when both distributions are empty. The proof of that the algorithm will provide the sum of divergence as per Definition 11 is then given by induction over the distribution size n with lemmas 4 and 5 as base cases for $n = 1$ and $n = 2$, for odd and even number of elements in the distribution respectively, and Lemma 3 as the general case. □

The intuition of Lemma 6 is that, for the smallest unpaired sample of A, if that is not paired with the smallest unpaired sample of B, some other sample of A will be. That case would then lead to a suboptimal pairing that contradicts Lemma 3.

Theorem 1. Algorithm 1 calculates the sum of divergence between two distributions.

Proof. Given by lemmas 1, 2, and 6, since Lemma 1 proves that the algorithm will terminate, Lemma 2 proves that the algorithm yields a match, and Lemma 6 proves that the match yields the sum of divergence. □

8.3.4 Discussion

In contrast to the Euclidean distance metric and the χ^2 test, the comparison measure introduced here is not categorical. Further, in difference to the Kolmogorov-Smirnov test, the comparison measure is not dependent on mathematical modeling of the compared distributions.

In the case of performing this measurement on response times, the observability problem [95] must be respected: Take the example of a system with a set of strictly periodic tasks. Here, especially if the system load is high, it is likely that jobs of tasks are ready to execute long before they receive their first quanta of processing time. According to our definition of response time (see Section 8.2), the time of the triggering of the task must be known. Thus, probes that can access the ready queue of the operating system must be used to obtain a truthful measure of the response time.

8.4 Evaluation

We have performed evaluation of all the archetypes and PIC combinations detailed in Section 8.1.2. Details of this evaluation and its resulting data are presented in this section.

We performed model extraction on all archetype-PIC combinations described above. For each combination, models were extracted using 2, 4, 8, 16, and 32 simulations for model generation and equally many for the model validation. The intention was to evaluate if there was a significant improvement in the quality of the generated model as the set of input grew.

8.4.1 Generality evaluation data

Data from the generality evaluation are in the following sets: From model validation (see Section 6), we observe the leeway (or precision parameter) and the accuracy measure in the form of $h(n, G, V)$. From comparing the simulation of the model with the recording of the system, we obtain the likeness measure defined in Section 8.3. These sets of data are obtained for each model extraction.

As models for Archetype 1 were extracted, the process of model validation reported accuracy measure $h(n, G, V) < 0.2\%$ for all tasks, the leeway was 0 for all PICs. The data of the comparison is presented in Table 8.3.

Tasks	T_1					T_2				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	2	2	2	2	2	0	0	0	0	0
PIC 2	6	6	6	6	6	0	0	0	0	0
PIC 3	6	5	5	5	5	1	1	1	1	1
PIC 4	22	21	22	21	22	1	1	1	1	1

Table 8.3: Sum of divergence between system and model response times in tests for tasks T_1 and T_2 of Archetype 1 [%].

As models for Archetype 2 were extracted, model validation reported accuracy measure $h(n, G, V) < 6.2\%$ for all tasks, the leeway was 0 for PICs 1,2 and 3, and 34 for PIC 4 . The data of the comparison is presented in Table 8.4.

Tasks	T_1					T_2				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	6	1	0	0	2	0	0	0	0	0
PIC 2	2	7	2	3	5	2	7	2	3	5
PIC 3	0	1	2	5	1	0	1	1	5	1
PIC 4	4	3	3	4	2	4	3	2	4	2

Table 8.4: Sum of divergence between system and model response times in tests for tasks T_1 and T_2 of Archetype 2 [%].

As models for Archetype 3 were extracted, model validation reported accuracy measure $h(n, G, V) < 0.0\%$ for all tasks, the leeway was 0 for all PICs. The data of the comparison is presented in Table 8.5.

Tasks	T_1					T_2				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	-	-	-	-	-	-	-	-	-	-
PIC 2	-	-	-	-	-	-	-	-	-	-
PIC 3	31	27	35	31	29	-	-	-	-	-
PIC 4	31	29	32	30	29	5	5	6	6	6
Tasks	T_3					T_4				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	-	-	-	-	-	24	22	24	22	27
PIC 2	21	21	21	21	17	3	4	4	4	5
PIC 3	14	43	19	42	19	5	4	7	5	6
PIC 4	21	21	22	21	21	14	15	16	14	16
Tasks	T_5					T_6				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	4	6	5	5	5	19	18	18	18	19
PIC 2	4	5	5	5	4	9	10	9	9	15
PIC 3	5	5	4	3	5	13	12	13	12	14
PIC 4	4	4	5	3	5	5	6	7	7	6
Tasks	T_7									
Files	2	4	8	16	32					
PIC 1	22	19	21	23	22					
PIC 2	16	12	14	12	15					

PIC 3	9	7	7	6	10					
PIC 4	30	28	31	26	27					

Table 8.5: Sum of divergence between system and model response times in tests for tasks T_1 to T_7 of Archetype 3 [%].

As models for Archetype 4 were extracted, model validation reported accuracy measure $h(n, G, V) < 0.3\%$ for all tasks, the leeway was 0 for all PICs. The data of the comparison is presented in Table 8.6.

Tasks	T_1					T_2				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	1	1	1	1	1	2	2	2	2	2
PIC 2	6	7	7	7	7	11	11	10	9	10
PIC 3	6	6	6	5	6	10	9	11	11	10
PIC 4	6	6	6	6	6	10	10	9	11	11
Tasks	T_3					T_4				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	2	0	0	1	0	0	0	0	0	0
PIC 2	1	0	1	0	0	10	11	9	8	9
PIC 3	1	0	0	0	1	9	9	9	9	9
PIC 4	4	2	2	2	3	9	10	10	9	9
Tasks	T_5					T_6				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	-	-	-	-	-	-	-	-	-	-
PIC 2	16	16	17	17	15	0	0	0	0	0
PIC 3	18	16	19	17	15	0	0	0	0	0
PIC 4	18	15	16	18	16	0	0	0	0	0
Tasks	T_7									
Files	-	-	-	-	-					
PIC 1	-	-	-	-	-					
PIC 2	-	-	-	-	-					
PIC 3	38	38	39	38	38					
PIC 4	38	38	38	38	38					

Table 8.6: Sum of divergence between system and model response times in tests for tasks T_1 to T_7 of Archetype 4 [%].

As models for Archetype 5 were extracted, model validation reported accuracy measure $h(n, G, V) < 0.6\%$ for all tasks, the leeway was 31 for PICs 1 and 2, 71 for PICs 3 and 4. The data of the comparison is presented in Table 8.7.

Tasks	T_1					T_2				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	5	12	8	7	7	5	5	2	8	7
PIC 2	10	7	4	4	13	7	2	8	6	10
PIC 3	7	7	9	7	9	2	1	2	1	1
PIC 4	6	6	6	6	6	1	2	2	1	1
Tasks	T_3					T_4				
Files	2	4	8	16	32	2	4	8	16	32
PIC 1	-	-	-	-	-	-	-	-	-	-
PIC 2	-	-	-	-	-	-	-	-	-	-
PIC 3	16	18	14	15	15	0	0	0	0	0
PIC 4	16	17	15	16	17	0	0	0	0	0
Tasks	T_5									
Files	2	4	8	16	32					
PIC 1	-	-	-	-	-					
PIC 2	2	3	20	18	2					
PIC 3	3	4	3	4	2					
PIC 4	5	3	4	4	3					

Table 8.7: Sum of divergence between system and model response times in tests for tasks T_1 to T_5 of Archetype 5 [%].

8.4.2 Visual inspection of the generality evaluation data

Since the comparison that we have used in the evaluations is previously untried and therefore not validated, we manually observed plots of the generated distributions that were input to the comparison. The intention was to confirm that the comparison measure performed according to our intuition.

According to our subjective comprehension, there were three categories of results; either, the comparison measure was in the interval $(0\%, 10\%)$, in the interval $[10\%, 20\%)$, or in the interval $[20\%, 100\%)$. These categories are labeled I, II, and III respectively. Intuitively, a measure in Category I signifies a high correlation between the compared distributions, Category II signifies a

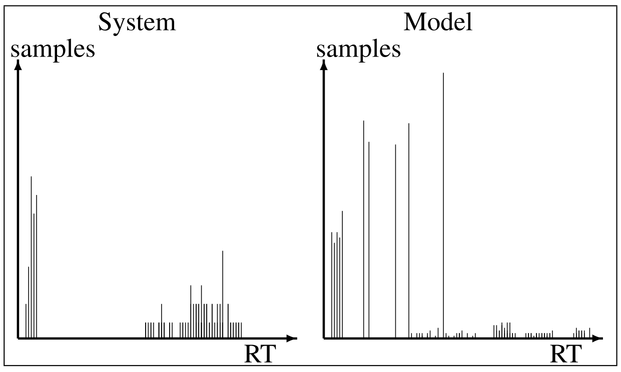


Figure 8.5: Response times for Archetype 1, T_1, 32 files, $sum\_divergence = 68,467,416$, $distributions\_size = 705,720$, and $max\_divergence = 440$, divergence sum normalization is 22%.

medium correlation, while Category III signifies a low correlation.

Regarding Archetype 1, most measures where in Category I. On visual inspection on the generated distributions, these measures where deemed to conform relatively well to the intention of the measure. Apart from these, measures for T_1 at PIC 4 where in Category III, these where indeed found to display significant differences. As an example, observe Figure 8.5.

For Archetype 2, all measures where in Category I. This was validated on visual inspection.

In Archetype 3, we find measures in all three categories, which may indicate varied sucess for model extraction. Here, only observing the measure, it would seem that the model extraction was not successful. However, for many tasks, the response times are in a relatively narrow interval. The response times of T_1 is in the interval $(3, 4)$, response times for T_3 is in the interval $(5, 46)$, and response times for T_4 is in the interval $(4, 67)$. As a result of the normalization, this yields a large percentage even for small divergences. As an example, observe Figure 8.6.

In Archetype 4, we also find measures in all three categories. Though we do find examples of the same problem as described above, the measure generally corresponds well to the intuition. Especially the poor correlation between model and system of T_7 is confirmed (see Figure 8.7).

The measures on distributions from Archetype 5 are in all three categories.

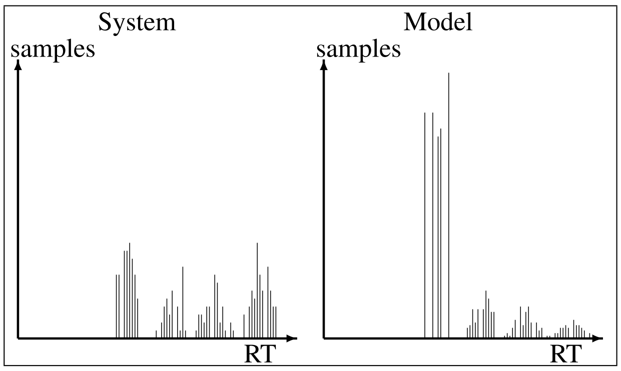


Figure 8.6: Response times for Archetype 3, T_6, 32 files, $sum\_divergence = 8,376,772$, $distributions\_size = 798,534$, and $max\_divergence = 54$, divergence sum normalization is 19%.

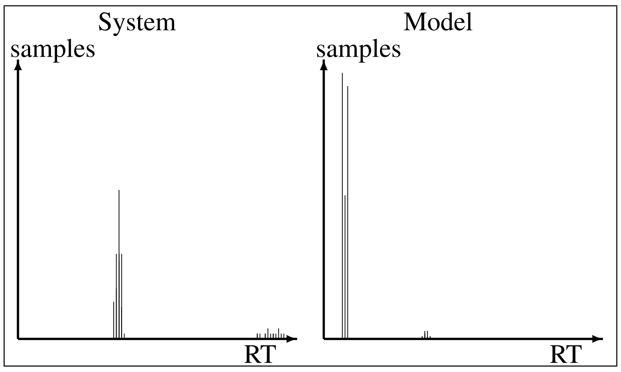


Figure 8.7: Response times for Archetype 4, T_7, 16 files, $sum\_divergence = 32,335,951$, $distributions\_size = 391,419$, and $max\_divergence = 214$, divergence sum normalization is 38%.

Most notably, T_3 has all measures in Category II. On examination however, the largest $max\_divergence$ was 8, which is very low in comparison with other tasks. It turned out that this too was a result of a narrow interval in the execution time distribution. The measures of T_1 in Category II and T_5 in

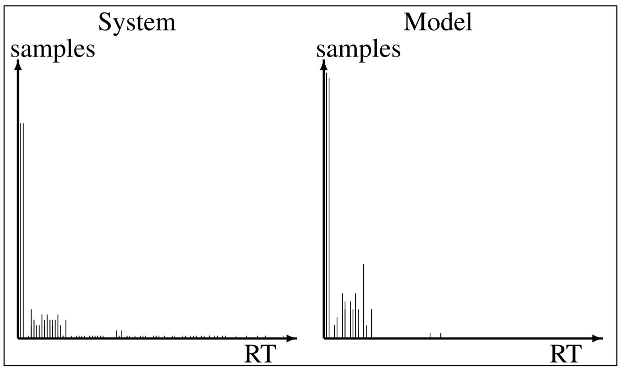


Figure 8.8: Response times for Archetype 5, T_2, 32 files, $sum\_divergence = 3,972,608$, $distributions\_size = 60,386$, and $max\_divergence = 623$, divergence sum normalization is 10%.

Category III however, where deemed to coincide with intuition. We find an interesting example in T_2 with PIC 2 and 32 files (see Figure 8.8). Here, the distributions clearly differ in small ways, but the measure is able to abstract from these and determine that the distributions are relatively similar.

8.4.3 Stability evaluation data

The stability evaluations are performed on the same recordings that were used in the generality evaluation. Thus, data from the model validation presented above applies to this evaluation too.

We present data of comparisons in the following tables: Archetype 1 in Table 8.8, Archetype 2 in Table 8.9, Archetype 3 in Table 8.10, Archetype 4 in Table 8.11, and Archetype 5 in Table 8.12.

Data is presented as two values for each task and each combination of PIC (e.g., 32-31 for Archetype 1, Task T_1, PICs 1 and 2 in Table 8.8). The first value concerns the comparison between the response times for two systems of Archetype 1, the first with PIC 1 and the second of PIC 2. The second value concerns the corresponding comparison for models of the same Archetype and the same PICs. If stability exists, the two values should be similar.

Note that, due to the abstractions introduced in the models, we cannot expect a perfect match between the values, but expect deviations which are less than 10. This limit seems intuitive since we used it in the above test of generality as the limit for Category I measures.

Task	PICs:					
	1 vs. 2	1 vs. 4	2 vs. 3	2 vs. 4	3 vs. 4	3 vs. 1
T_1	32-31	43-15	21-19	21-11	1-22	43-38
T_2	0-0	43-44	42-44	42-44	1-0	43-44

Table 8.8: Stability test results for tasks T_1 and T_2 of Archetype 1 [%].

Task	PICs:					
	1 vs. 2	1 vs. 4	2 vs. 3	2 vs. 4	3 vs. 4	3 vs. 1
T_1	1-1	20-20	3-2	19-22	21-20	2-1
T_2	31-25	43-42	3-2	18-22	21-20	28-28

Table 8.9: Stability test results for tasks T_1 and T_2 of Archetype 2 [%].

Task	PICs:					
	1 vs. 2	1 vs. 4	2 vs. 3	2 vs. 4	3 vs. 4	3 vs. 1
T_1	-	-	-	-	0-0	-
T_2	-	-	-	-	-	-
T_3	-	-	64-68	43-17	26-22	-
T_4	0-10	35-17	10-4	32-12	27-10	17-10
T_5	13-7	35-21	12-7	29-18	22-14	22-13
T_6	9-8	23-17	13-9	21-25	16-21	21-16
T_7	21-10	51-44	21-17	46-43	39-40	33-25

Table 8.10: Stability test results for tasks T_1 to T_7 of Archetype 3 [%].

Task	PICs:					
	1 vs. 2	1 vs. 4	2 vs. 3	2 vs. 4	3 vs. 4	3 vs. 1
T_1	11-8	11-9	0-1	0-1	0-0	11-9
T_2	0-11	0-11	0-0	0-0	0-0	0-10
T_3	0-0	0-2	0-0	0-2	0-1	0-0
T_4	0-9	0-9	0-0	0-0	0-0	0-9
T_5	-	-	1-0	1-0	1-0	-

T_6	-	-	0-0	0-0	0-0	-
T_7	-	-	-	-	0-0	-

Table 8.11: Stability test results for tasks T_1 to T_7 of Archetype 4 [%].

Task	PICs:					
	1 vs. 2	1 vs. 4	2 vs. 3	2 vs. 4	3 vs. 4	3 vs. 1
T_1	2-15	23-16	19-15	23-15	1-3	20-16
T_2	1-2	1-9	2-12	2-10	1-1	1-10
T_3	-	-	-	-	2-3	-
T_4	-	-	-	-	0-0	-
T_5	-	-	4-0	8-3	0-1	-

Table 8.12: Stability test results for tasks T_1 to T_5 of Archetype 5 [%].

8.4.4 Visual inspection of the stability evaluation data

For Archetype 1, the stability evaluation is reported in Table 8.8. Results for task T_1 between PICs 1 and 4, between PICs 2 and 4, and between PICs 3 and 4, suggest instability. On visual inspection, we could indeed tell differences in both cases. It seemed that the poor model extraction for task T_4 inflicted this loss in stability (see Table 8.3).

For Archetype 2, all model comparisons followed their respective system comparison.

For Archetype 3, a multitude of comparisons differed significantly. This follows from the performance of the normalized comparison measure in this archetype (see Section 8.4.2).

For archetypes 4 and 5, the indications of poor stability is related to problems discussed in Section 8.4.2.

A conclusion that we can draw from this is that model stability can only be evaluated in those cases where model extraction has been successful and where the comparison measure is able to assess similarity correctly.

8.5 Discussion

Based on the above evaluation, we can conclude that our method of model extraction is indeed general with respect to the archetypes that we have examined, and that stability exists in the cases where generality applies.

The following cases where found to contradict the generality of model extraction:

1. Archetype 1, PIC 4, Task T_1.
2. Archetype 4, PICs 3 and 4, Task T_7.
3. Archetype 5, PIC 2, Task T_5, PIC 1 and 2, tasks T_1.

We examine these three cases with the aid of the archetype and PIC definitions in Section 8.1.2:

In Case 1, the PIC applied leads to that the recording of data state is disabled. As the data state has a significant impact on the execution time in the system, the failure to perform model extraction is consistent with our understanding of the requirements on observability (see Section 2.4.3 on the observability problem).

In Case 2, Task T_7 has the lowest priority. Together with the high system load and smaller errors in other tasks due to their abstraction, this lead to the poor performance of the task. Thus, model extraction was not unsuccessful in the specific case of Task T_7, the error is rather a result of precision problems in other tasks. It seems that the more influence[3] a task has on its environment, the more detail is required in modeling of the task. This points to an important aspect in modeling using probabilistic modeling languages: influential tasks should only contain a minimal amount of probabilistic elements. This follows the intuition that important parts must be carefully investigated, while less important parts can be sketched.

In Case 3, problems in both tasks are due to the same issues as in Case 2. For Task T_5 however, these problems are only visible for some recording set sizes, which may indicate that more extensive recording is needed.

We found no cases that contradict the stability of model extraction.

This was the first application of the measure of comparison introduced in Section 8.3. We have found that the measure works as intended under ideal conditions, but when the spread of the distributions is narrow, the normalization of the measure is affected. It is likely that this is true also for distributions with wide spread. Thus, we can recommend the use of the measure, but please be advised to use the spread of distributions as a sanity check.

[3]Here, an *influential task* has high priority and/or relatively large amount of interaction through communication with other tasks.

The five archetypes used in this study are by no means a complete set, it is straightforward to specify and construct many more. We have let our understanding of industrial embedded systems stand as model for designing these archetypes and PIC; other individuals will most likely have other ideas for archetypes and PIC to include. Our aim has been to evaluate model extraction with respect to a set of general patterns that are commonly used in industrial applications. For each of these archetypes, we have then applied PIC as a form of noise to obtain instantiations of the general archetype. In this way, we were able to investigate a set of differing but still similar systems and obtain a higher degree of confidence that model extraction is indeed capable of modeling a given archetype.

Chapter 9

Conclusions

In this last chapter of the thesis, we will discuss contributions, limitations, validity, and future work of the thesis.

9.1 Results

The first few paragraphs in this thesis concerned *abstraction* and its role in software development and other fields. With this in mind, it feels appropriate to start this last chapter of the thesis with a form of abstraction of the previous chapters.

This thesis has seen the introduction of new methods for model extraction, a.k.a. automatic modeling. In the first chapter of this thesis, we formulated the following problems under the assumption that such methods could be developed:

Can the validity and accuracy of extracted models be quantified? This question has been addressed by the introduction of model validation in Chapter 6, which is integrated in the model extraction presented in Chapter 4. Our solution to model validity increases the confidence in the model by comparing the behavior of the system to that of the model. Several issues are investigated in this process: that the model can replicate any behavior that the system can exhibit, that the timing of the model is sufficiently similar to the timing of the system, and that the distribution of behaviors in the model resembles that of the system. As the solution is based on testing of the system, it is not able to determine whether the model can express behavior that the system cannot. Our conclusion is that it is indeed possible to quantify the validity and accuracy of extracted models.

Are the overheads of model extraction acceptable? The case study presented in Chapter 7 shows that the overheads of introducing model extraction in this specific case are indeed acceptable. In the study, a perturbation of approximately 0.88%, generating $66,000$ bytes of data each second, was observed. Regarding preparations for model extraction, we spent an approximate total of 20 hours to find and implement the final probing of the system (an engineer more accustom to the system would probably perform this task significantly faster). Whether these numbers are generally acceptable, or not, or whether the case-study is representable, is difficult to answer; a significant amount of data was generated, but the run-time overhead and the overhead for implementing probing is low. In the specific system that we observed, the amount of data generated did not present a problem. Also, because we can perform model generation on a *set* of recordings, we are not ultimately dependent on the lengths of individual recordings; the recording length can be reduced if more recordings are generated. Our conclusion is that there are evidence indicating that the overhead of model extraction could be acceptable, though a more general conclusion would require further studies.

Can the method of model extraction be evaluated? We have performed both a case study and an experiment to evaluate the appropriateness of our model extraction. With the framework for quality assessment presented in Chapter 8, we can evaluate individual methods of model extraction as well as compare several different methods to evaluate differences in the behavior of the final models. The comparison that we propose is a novel instrument for measuring the likeness between sampled time distributions, but the framework for quality assessment can use any other method of comparison. Our conclusion is that the method of model extraction can, as we have demonstrated, indeed be evaluated.

Thus, we have answered the questions postulated in the beginning of the thesis. Among the advantages of our method of model extraction are that it allows modeling without intruding on continuous software evolution, and that it allows modeling of a system without extensive system knowledge. That said, there are still issues to resolve as far as the presented method of model extraction is concerned. There is a set of disadvantages of model extraction as it is presented here (though some of these are probably solvable):

The fruits of effort. Some times, enduring a trialling experience is the most efficient way to gain enlightenment. As a model can be obtained without being understood, it is likely that the pedagogic value of the model is reduced. Therefore, it is possible that an interactive or completely manual method would prove more efficient in the sense that the engineer making the model actually gains insight into how the system is functioning.

The lost link. Use of recordings as the only significant input to model extraction may present a problem: If the recording does not contain information to what part of the code that an event originates from, the link from the model back to the source code is effectively broken. Depending on the details of the use of the model, this can be cumbersome (it may be difficult to associate a part of the model to a part of the implementation and vice versa). This problem could possibly be resolved by introducing more information into the recordings, but such a change also requires alterations to model generation.

Unutilized capacity. As stated previously, the method presented here cannot model task-level loops within a job. This is a general and crippling deficiency, but it is solvable by other methods (e.g. Angluin's algorithm [7]). Also, it may be possible to analyze the recordings even further, to find implicit causal dependencies not covered by the state-probes.

Context dependency. The set of test-cases used to extract the model limits the context in which the model is meaningful. Functionality not triggered by any of the test-cases cannot be observed, and will hence not be included in the generated model.

Apart from these disadvantages, under the assumptions stipulated in the thesis, our evaluations present evidence to suggest that model extraction as presented here can generate models whose temporal and functional behavior are comparable to that of the modeled system. Furthermore, the evidence suggest that these models can be used in model-based impact analysis. The argumentation for these conclusions are as follows: The industrial case-study indicate that our assumptions are viable from an industrial perspective, and the different overheads to perform manual probing and execution time increase due to probing are acceptable. Also, the case-study show no evidence or indication of fundamental flaws to the approach. Given the results of the controlled experiment, we were not able to find any evidence or indication to suggest that the generality or the stability of model extraction is questionable, provided that recording and probing of data state is thorough enough given the complexity of the modeled system.

9.2 Faithfulness of the generated models

For the discussion of faithfulness we will informally introduce the following concepts:

- A *behavior* of a system is an execution of the system (including both events and timing); note that behaviors are defined on some level of granularity, e.g., an event on one level could correspond to a sequence of events on another level; hence there is an element of abstraction in how executions are defined. We will not further discuss this aspect here, rather we assume models to be at the same level of granularity.
- A *faithful model* is a model whose executions are all valid executions of the modeled system.
- An *abstract model* is a model that includes at least all executions of the modeled system; hence, a faithful and abstract model (an *exact model*) includes exactly the behaviors of the modeled system.
- An *inventive model* is a model that is neither faithful, nor abstract, i.e. typically such models contain some (but not all) behaviors of the modeled system, together with some behaviors that are not included in the modeled system.

Figure 9.1 presents a Venn diagram illustrating the relations between the sets of behaviors of faithful models (II in the figure), abstract models (V), inventive models (III), and exact models (IV). The figure additionally shows how the set of observed behaviors (I) could be related to the introduced types of models.

The set of observed behaviors is the basis for our model generation. In fact, these behaviors are the only knowledge about the system that is available. With respect to these behaviors, model generation will produce an abstract model, or more specifically a model containing all observed behaviors and (due to the approximations made in the model generation) also behaviors that have not

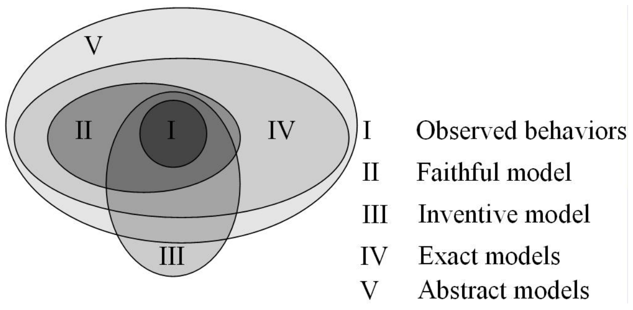


Figure 9.1: The behavior of the model compared to the behavior of the system and the observations on the system.

been observed. Unfortunately, since complete knowledge of the behaviors of the system is lacking, the generated model will (in the general case) be an inventive model relative to the system. However, by the introduced methods for model validation we gain confidence in that the majority of behaviors of the model are also valid behaviors of the system. Note also that, since the generated model remains abstract with respect to the observed behaviors when the number of observations are increased, the generated model will be an abstract model of the system if all system behaviors are observed. This is an important property, though it in most cases is impractical (and practically impossible) to observe all behaviors.

The differences between the observed behavior and the behavior of the model is due to abstractions in the model in relation to the system. During simulation of the model, these abstractions give rise to visible differences between the models and the system at three distinctly different levels:

- At system level, in terms of response times and preemption patterns (e.g., in the model, task A could preempt task B, though this will never happen in the system).

- At task level, in terms of ordering of the event sequences that are jobs of tasks (similarly to ordering of SECs as seen on Page 6.4).

- At job level, in terms of execution time distributions (e.g., the model has execution times that the system hasn't).

The abstractions in the model are mainly on the form that variables controlling the execution of a task are excluded from the model, leading to that causal rules depending on these variables are not expressed by the model. In more detail, relating to the three levels of differences, we note the following:

- At job level in the system, causality may dictate the execution time requirement for segments of code. If the data variables that control this causality are excluded from the recording, the model cannot capture the relation, and simulation of a job may yield shorter or longer execution times than seen in the recordings. For example, assume a job consisting of an execution E_1, followed by and event X, followed by an execution E_2; say that the system exhibits a long execution time for E_2, if E_1 is short, and vice versa; if that relation is not included in the model, the model may have a long E_1 execution time in the same job as a long E_2 execution time, in which case the total execution time of the job in the model will be longer than the execution time of any job in the system.

- At task level, for the same reasons as above, the model's event sequences, which are jobs of a task, may be performed in orders that are not seen in the recordings. Given three jobs A, B, and C; say that the system will iterate deterministically between these in the order $A, B, C, A, B, C, \ldots$; if that relation is not included in the model, the model may, e.g., execute in the order $B, A, C, C, B\ldots$

- At system level, the model's preemption patterns differ from the system due to the differences at the other levels, and/or due to abstractions in the environment modeling, or in the modeling of task properties (e.g., periodicity). For example, due to the above differences, the interactions between tasks in the system may be affected as execution times change, or as expected communication is postponed due to task level differences. If so, the faithfulness of the model may be in jeopardy - after all, what conclusions can be drawn from a model that does not behave like the system?

In order to determine the degree of faithfulness of an inventive model, we need methods to (automatically) validate the model at the different levels of difference. In this thesis, we introduce one such method in the form of the evaluation framework in Chapter 8, which evaluates the model at system level. Additional methods of model validation are however needed at all three levels.

9.3 Reproducibility

Regarding repeatability/reproducibility of the research, we present detailed algorithms for model extraction in appendices of the thesis, and the controlled experiment is performed in a simple and understandable framework. Model extraction follows a well-defined, traceable, and structured algorithm where the elements of subjectiveness have been restricted to a few sub-steps in the probing of the system and parameterization of model validation. Regarding the case-study, the implementation of the studied system is proprietary; hence, the exact reproducibility is limited, though the information provided could serve as a basis for similar studies.

9.4 Future work

During this work, a list of topics for future work has been identified:

9.4.1 Improving model generation

The method of comparing two sampled time distributions presented in Section 8.3 (or any other method that solves the same problem) can be used to make model generation produce more efficient models.

In the model produced by model generation, execution statements may be distinguished by a branch. The implicit assumption has been that if the subsequent behavior differs, the execution time represented by the statements is likely to differ too. A branch has therefore been introduced to separate the behaviors even before they have been shown to differ. By comparing the sampled time distributions of the execution statements in the branch, it is possible to test if this assumption is correct or not. If not, the execution statements can indeed be joined, and the branch introduced later in the model.

This method is also applicable for analyzing the model with the intent to introduce loops in the model. Such a method should check jobs to find any repeating patterns within the job; if so, there may be incentive to introduce a loop in the model. In this case, a hypothesis is formed: the loop is assumed to be feasible, and is introduced in the model, the hypothesis is then tested by model validation. In this context, comparing sampled time distributions is used in searching for repeating patterns within each job.

In addition to these improvements to model generation, we would also like to construct methods for environmental modeling; that is, for the method to reach its full potential as a tool for active use in industry, also environmental models need to be constructed. We believe that a method similar to model generation could be used to achieve this. The main obstacles are: identifying the equivalent of tasks in the environment (i.e. identifying the separate physical processes that form the environment), finding repeating patterns in these environmental processes corresponding to that of identifying the start and end of jobs, and using other primitives than execution time distributions to model the passing of time in the environment.

9.4.2 Improving model validation

As explained above in this chapter, we need more methods for automated model validation.

9.4.3 Improving probing analysis

We need tools to support (semi-)automatic probing analysis and probing implementation.

As described earlier (see Section 4.6.3), one possibility is to use for example techniques of memory excluding checkpoints [86] to determine the variables that are likely to yield good models; the search for appropriate probe settings should then focus on this set of variables.

9.4.4 Continued evaluation

Though evaluation is a significant part of this thesis, there is still much work to do in this area. We need to continue and deepen the evaluation as presented here even further. In addition, we would also like to perform more industrial case-studies to validate the method further, and ultimately evaluate the possibility of using model extraction as an active tool in system development.

9.4.5 Comparing automatic modeling techniques

Using the evaluation framework presented in Section 8.1, it is possible to compare different methods of automatic modeling. We are primarily interested in comparing the work presented here to the work of Andersson et al. presented in [5], but also other work could be included in a future comparison.

9.4.6 Using classification to compare sampled time distributions

In Section 8.3, we introduced a method for comparing two sampled time distributions. In the following evaluation using the method, we found that the normalization of the measure in the method could lead to unintuitive results (see e.g., Page 133). Therefore, we are interested in seeking alternative methods.

Another option, orthogonal to the presented comparison, is to use techniques of *classification* [25, 48, 58] (i.e., techniques to summarize large populations of objects in terms of a small number of classes of objects). Alternative approaches to classification include *cluster analysis* [48] and *pattern recognition* [58]. We do not know of any applied results of classification that solve our specific problem, but believe that such an applied method could be designed.

If samples are the objects that are classified, solving a particular classification problem provides an abstraction of the sampled time distribution. Comparing the abstractions of two or more distributions expresses the similarity of the distributions. We may optionally introduce differential weighting of classes of objects in order to increase the abstraction from the complexity of the sampled time distributions (for example, we could remove classes with few objects from the abstraction to reduce the noise in the comparison).

One drawback of this type of approach is that it may require extensive parameterization, which requires that the user is familiar with the classification technique and with the application domain of the comparison.

9.4.7 Using automatic modeling to achieve a design paradigm shift

As the methods of automatic modeling become more sophisticated, we would like to investigate the possibility of using automatic modeling in order to transform a code-oriented project into a model-based development project. Apart from significant advances in automatic modeling, a new set of methodologies are needed to make the transformation and to validate that the transformation has been completed.

9.4.8 Avoiding the probe effect

In many cases, the probe effect [23] hinders removal of probes from the implementation because it (in the general case) cannot be determined that the presence of the probes, and the perturbation that this inflicts on the system, does not effect the behavior of the implementation. This is of interest as, if the behavior is affected, the performed testing of the implementation is invalidated by adjusting the perturbation of the probes (i.e., modifying the probe setup). In the case of model extraction however, an exception could be motivated.

We present the following argumentation for the existence of this exception: As the probes are auxiliary to the system (i.e., they are not explicitly cooperating with the implementation) the architecture of the implementation does not change with the addition or removal of probes. As long as the temporal behavior of the model that has been extracted from recorded executions is modified accordingly (i.e., the execution time of the removed probes is subtracted from the measured execution time), some probes can be removed as the architecture of the implementation will not change based solely on the number of probes.

This remains true, and can be performed, as long as we can determine that the presence of the probes did not disturb the functionality of the system such that it triggers some *extraordinary behavior* (e.g., the probes are the source of some fault). If the recording was affected by extraordinary behavior, the model created from that recording is erroneous with respect to the implementation without probes. In that case, the probe setup must be kept until the system can be fully re-tested.

Assuming an implementation with two different probe setups, where one of the setups is a subset of the other, the following method can be used to determine whether the presence of the additional probes changed the behavior of the system.

1. First, the records from probes not available in both the probe configurations are removed from the recordings. This step includes modifying time-stamps by subtracting the time spent to execute the probes that are not available in both recordings. As motivated in other work, the execution time of probes should (for reasons of testability etc.) be constant [35].

2. Second, models are generated from each of the recordings.

3. Third, the two models are compared; if they are deemed to be equal, the probe effect had no impact.

Thus, if it is concluded that probes could be removed, the smaller of the two probe configurations must be kept, but the additional probes can be removed. In this way, we can use the probe setups used in previous iterations of the model extraction to remove subsequent additions to the probe setup. In the optimal case, the only probing that need to remain in the implementation is the minimal requirement of the model generation. The minimal probing is to probe the events task preemptions and performed system calls.

If the probe effect has no impact on the system, it has been avoided successfully in this case even if the probe setup is altered. The observation that is important to emphasize here is that if the overhead of task-level recording is low, the chance of avoiding the probe effect increases.

Appendix A

Algorithms for model generation

In this appendix, we present algorithms for model generation as described in Chapter 5. We make use of a set of functions, described by Table A.1, to add recseq events with the observed actions, add branches to a modset, etc.

Function	Description
addExecute	Adds a recseq event with action Execute.
addEnd	Adds a recseq event with action End.
addStateAssignment	Adds a recseq event with action Variable assignment.
addWriteToIPC	Adds a recseq event with action Write to IPC.
addReadFromIPC	Adds a recseq event with action Read from IPC.
makeMdlBranch	Makes a modtree from a given recseq event.
distinguishingParameters	Returns the distinguishing parameters for a modtree.
adjustStateToStateAssignments	Applies a set of Variable assignments to an existing state.

updateStateAssignments	Adds one Variable assignment to a set of Variable assignments, replacing any existing updates to that variable.
makeBeginIfSelection	Starts an ART-ML if-selection for a given set of $TaskModelDataStates$.
makeEndIfSelection	Ends an ART-ML if-selection.
makeBeginChanceSelection	Starts an ART-ML chance-selection for a given probability.
makeEndChanceSelection	Ends an ART-ML chance-selection.
makeActionStatement	Produces an ART-ML statement from a given modtree.
makeTaskHead	Produces the preamble of an ART-ML model
makeTaskFoot	Produces the postamble of an ART-ML model

Table A.1: Functions used in the algorithms for model generation.

A.1 Extraction of task executions (jobs) from recordings

Algorithm 2 The algorithm *prepareStateOfTask* finds the initial data state *state* $\in TaskModelDataStates$ of the first job to base the recseq on.

```
Require: Input: a recording $E$
Require: Input: a unique task identifier $t$
Require: There is a vector $state \in TaskModelDataStates$
Require: The vector $found$ has domain $\{defined, undefined\}$
Require: The variable $active$ has domain $\{true, false\}$
Require: There is a set of integer variables $i, j, v, id$
 1: $active := false, i := 0, v := 0$
 2: all elements of the vector $found$ are set to $undefined$
 3: for $i < |E|$ do
 4:   if $E_i.e$ is a Context switch event such that $t$ is now executing then
 5:     $active := true$
 6:   else if $E_i.e$ is a Context switch event such that $t$ is preempted then
 7:     $active := false$
 8:   end if
 9:   if $active = true \wedge E_i.e$ is a Variable assignment event then
```

```
10:     $id$ := a unique identifier for the parameter “variable name” of $E_i.e$
11:     if $found_{id}$ = $undefined$ then
12:       $found_{id}$ := $defined$
13:       $v := v + 1$
14:     end if
15:   end if
16:   $i := i + 1$
17: end for
18: $i := 0, j := 0$
19: for $j < v$ do
20:   if $E_i.e$ is a Context switch event such that $t$ is now executing then
21:     $active := true$
22:   else if $E_i.e$ is a Context switch event such that $t$ is preempted then
23:     $active := false$
24:   end if
25:   if $active = true \wedge E_i.e$ is a Variable assignment event then
26:     $id$ := a unique identifier for the parameter “variable name” of $E_i.e$
27:     $state_{id}$ := the parameter “value” of $E_i.e$
28:     $j := j + 1$
29:   end if
30:   $i := i + 1$
31: end for
32: return $\{i, state\}$
```

Algorithm 3 The algorithm *recording2recseq* extracts a recseq for one task from one recording.

```
Require: Input: a recording $E$
Require: Input: a unique task identifier $t$
Require: There is a recseq labeled $R$
Require: The variables $active,start$ have domain $\{true,false\}$
Require: There are two vectors $state, nextState \in TaskModelDataStates$
Require: There is a set of integer variables $i,j,e,id,save,time,last$
 1: $active := true, state := false, j := 0, e := 0, time := 0, last := 0$
 2: $offsetAndState := prepareStateOfTask(E, t)$
 3: $\exists offset : offset \in offsetAndState \wedge offset \in \mathbb{Z}^*$
 4: $\exists state : state \in offsetAndState \wedge state \in TaskModelDataStates$
 5: $nextState := state$
 6: for $offset < |E|$ do
 7:   if $E_i.e$ is a Context switch event such that $t$ is now executing then
 8:     $last := e$
 9:     $active := true$
10:   **else if** $E_i.e$ is a Context switch event such that $t$ is preempted **then**
11:     $time := time + E_{offset}.t - E_{last}.t$
12:     $active := false$
13:     **if** $t$ is triggered periodically & the OS state is “suspended” **then**
14:       **if** $start := true$ **then**
15:         **if** $time > 0$ **then**
16:           $R_{j,e} := addExecute(E_{offset}, t, time, state)$
17:           $time := 0, e := e + 1$
18:         **end if**
19:         $R_{j,e} := addEnd(E_{offset}, t, state)$
20:         $j := j + 1, e := 0$
21:         $state := nextState$
22:       **end if**
23:       $start := true$
24:     **end if**
25:   **else if** $t$ is event triggered & $E_i.e$ is receive on the triggering queue **then**
26:     **if** $start := true$ **then**
27:       **if** $time > 0$ **then**
28:         $R_{j,e} := addExecute(E_{offset}, t, time, state)$
29:         $time := 0, e := e + 1$
30:       **end if**
31:       $R_{j,e} := addEnd(E_{offset}, t, state)$
32:       $j := j + 1, e := 0$
33:       $state := nextState$
34:     **end if**
35:     $start := true$
36:   **end if**
37:   **if** $active = true \wedge start = true$ **then**
38:     $time := time + E_{offset}.t - E_{last}.t$
39:     $last := e$
40:     **if** $time > 0$ **then**
41:       $R_{j,e} := addExecute(E_{offset}, t, time, state)$
42:       $time := 0, e := e + 1$
43:     **end if**
44:     **if** $E_i.e$ is a Variable assignment event **then**
45:       $id :=$ a unique identifier for the parameter “variable name” of $E_i.e$
46:       $nextState_{id} :=$ the parameter “value” of $E_i.e$
```

```
47:         $R_{j,e} := addStateAssignment(E_{offset}, t, state)$
48:         $e := e + 1$
49:       else if $E_i.e$ is a Send to IPC queue: initialize event then
50:         $save := i$
51:       else if $E_i.e$ is a Send to IPC queue: finalize event then
52:         $R_{j,e} := addWriteToIPC(E_{offset}, E_{save}, t, state)$
53:         $e := e + 1$
54:       else if $E_i.e$ is a Read from IPC queue: initialize event then
55:         $save := i$
56:       else if $E_i.e$ is a Read from IPC queue: finalize event then
57:         $R_{j,e} := addReadFromIPC(E_{offset}, E_{save}, t, state)$
58:         $e := e + 1$
59:       end if
60:     end if
61:     $offset := offset + 1$
62:   end for
63:   return $R$
```

A.2 Generation of a tree-representation of the task from the jobs

Algorithm 4 The algorithm *recseq2modset* uses distinguishing parameters to make a modset from a recseq.

```
Require: Input: a recseq labeled $R$
Require: Input: an integer variable $j$ that denotes the current job in $R$
Require: Input: an integer variable $e$ that denotes the current event in $R_j$
Require: Input: a set of modtrees labeled $M$
Require: There are two modtree variables labeled $mb,m$
 1: $m := makeMdlBranch(R_{j,e})$
 2: if $j \geq |R| \vee e \geq |R_j|$ then
 3:   return $M$
 4: end if
 5: $mb \in M : distinguishingParameters(mb) = distinguishingParameters(m)$
 6: if $mb$ is defined then
 7:   $M = M \setminus \{mb\}$
 8:   $mb.c = mb.c + 1$
 9:   $mb.S = mb.S \cup m.S$
10:   if $mb.a$ is an execute action then
```

```
11:         Add the execution time distribution of $m.a$ to that of $mb.a$
12:     end if
13:     if $mb.a$ is an end action then
14:         Add the suspension time distribution of $m.a$ to that of $mb.a$
15:     end if
16:     $m = mb$
17: end if
18: if $m.a$ is an end action then
19:     $m.\mathbb{T} = recseq2modset(R, j+1, 0, m.\mathbb{T})$
20: else
21:     $m.\mathbb{T} = recseq2modset(R, j, e+1, m.\mathbb{T})$
22: end if
23: return $M \cup \{m\}$
```

A.3 Generation of ART-ML code from the tree-representation

Definition 12 (*StateAssignment*)**.** An assignment to a variable is represented by tuple labeled *StateAssignment*: $\langle$*unique identifier*, *value*$\rangle$. The semantics is that the value is assigned to the variable with the unique identifier.

Algorithm 5 The algorithm *modset2ART-ML* produces the body of ART-ML code from a modset.

```
Require: Input: a set of modtrees labeled $M$
Require: Input: a set of $StateAssignments$ labeled $SA$
Require: A set of $StateAssignments$ labeled $SA'$
Require: The variables $S'$,$SameBranchS$ are sets of $TaskModelDataStates$
Require: The variables $SP$,$SingleStateSP$,$SameBranchSP$ are sets of state-pairs
Require: The variables $a,b$ are state-pairs
Require: The variable $m$ is a modtree
1: $SingleStateSP := \{\forall sp \in 2^{TaskModelDataStates \times M} : \forall m \in sp.M \rightarrow sp.S \subseteq m.S\}$
2: $SP := \emptyset$
3: for $\forall a \in SingleStateSP$ do
4:     $SameBranchSP := \{\forall b \in SingleStateSP : b.M \equiv a.M\}$
5:     $SameBranchS := \emptyset$
6:     for $\forall b \in SameBranchSP$ do
7:         $SameBranchS := SameBranchS \cup b.S$
```

```
8:     end for
9:     $SP := SP \cup \{\langle SameBranchS, a.M \rangle\}$
10: end for
11: for $\forall a \in SP$ do
12:     $S' := adjustStateToStateAssignments(a.S, SA)$
13:     $makeBeginIfSelection(S')$
14:     for $\forall m \in a.M$ do
15:         if $|a.M| > 1$ then
16:             $makeBeginChanceSelection(m, a.M)$
17:         end if
18:         $makeActionStatement(m)$
19:         if $m.a$ is a Variable assignment then
20:             $SA' := updateStateAssignments(SA, m.a)$
21:         end if
22:         $modset2ART\text{-}ML(m.\mathbb{T}, SA')$
23:         if $|a.M| > 1$ then
24:             $makeEndChanceSelection()$
25:         end if
26:     end for
27:     $makeEndIfSelection()$
28: end for
```

A.4 Producing models for a set of tasks

Algorithm 6 The algorithm *modelGeneration* makes an ART-ML model from a set of recordings.

```
Require: Input: a set of recordings $REC$
Require: Input: a set of task identifiers $Tasks$
Require: The variables $R, R'$ are recseqs
Require: The variable $M$ is a modset
1: for All $t \in Tasks$ do
2:     $R := \emptyset$
3:     for All $E \in REC$ do
4:         $R' := recording2recseq(E, t)$
5:         $R := R + R'$
6:     end for
7:     $Mrecseq2modset(R, 0, 0, \emptyset)$
8:     $makeTaskHead()$
```

```
 9:     $modset2ART\text{-}ML(M, \emptyset)$
10:     $makeTaskFoot()$
11: end for
```

Appendix B

Algorithms for model validation

In this chapter, we present algorithms for transforming recseqs and modsets into timed automata.

B.1 Architecture for automata translation

In Figure B.1, we show the architecture of the solution presented here. There are, all in all, ten functions in the solution to obtain automata for the modset and recseq. Some of these (Model Get Edges, Model Event Count, Trace Get Edges, Trace Event Count, Trace Get Next) are recursive.

The calling of functions is conditional and there are seven different kinds of conditions, which are also indicated in the figure:

1. Always.
2. The type of the action is Execute.
3. The type of the action is Variable assignment.
4. The type of the action is either Send to IPC queue, Read from IPC queue, Variable assignment.
5. The type of the action is either Send to IPC queue, Read from IPC queue, Variable assignment, or End of job.

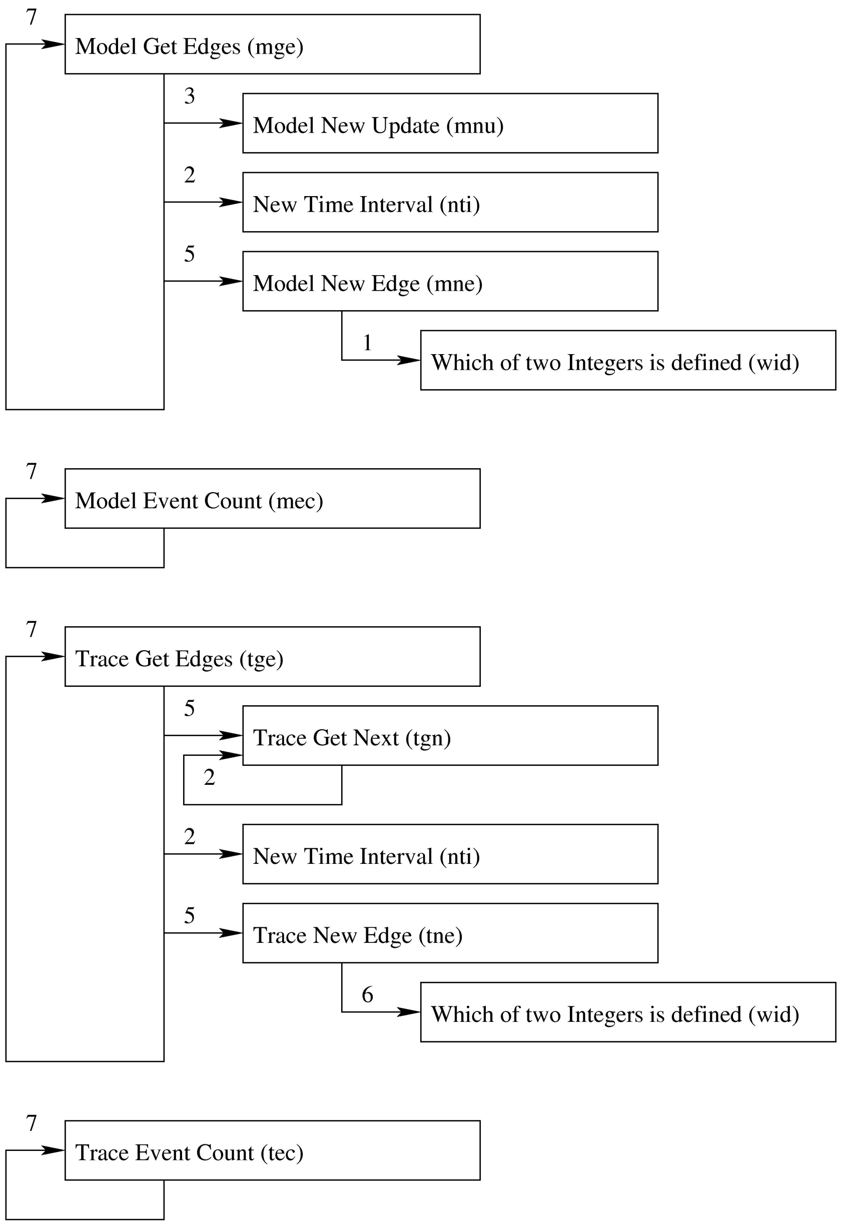


Figure B.1: Call graphs for the functions to obtain automata.

6. The type of the edge is either Send to IPC queue, Read from IPC queue, Variable assignment, or End of job.
7. For each action in the modset/recseq.

B.2 General definitions and functions

In the following sections, we introduce the translation of recseqs and modsets into timed automata. The translation requires the following definitions:

Definition 13 (Actions)**.** To represent both functional and temporal behavior, there is a set of actions $Actions \equiv \{snd, rcv, upd, exe, end\}$ ranged over by a. Each action in the set has a set of associated attributes as follows:

- Each inter task communication send (*snd*) has an associated $p_{id} \in \mathbb{Z}^*$ identifying the queue upon which a value was sent, a $p \in \mathbb{Z}^*$ identifying the value sent, and $msg \in \mathbb{Z}^* \cup \{z\}$ identifying a value received by an immediately preceding action of type *rcv*.
- Each inter task communication receive (*rcv*) has an associated $p_{id} \in \mathbb{Z}^*$, a $to \in \mathbb{Z}^* \cup \{\infty\}$ identifying the timeout, and $msg \in \mathbb{Z}^* \cup \{z\}$.
- Each variable update (*upd*) has an associated $p_{id} \in \mathbb{Z}^*$ identifying the variable, a value $p \in \mathbb{Z}^*$ assigned to the variable, and $msg \in \mathbb{Z}^* \cup \{z\}$.
- Each execute action (*exe*) has an associated time interval tt represented as a set of two integers that describe the duration of the execution, and $msg \in \mathbb{Z}^* \cup \{z\}$.
- Each end action (*end*) has an associated $msg \in \mathbb{Z}^* \cup \{z\}$ and a time interval tt represented as a set of two integers that describe the periodicity of the task should it be periodically triggered, otherwise, both integers are zero.

□

We use the definition of timed automata from Section 6.3.1. There are two subsets of clocks in the pair of automata: the modset-automaton has a set of local clocks $\mathcal{C}_T \equiv \{c_T\}$ and the recseq-automaton has a set of local clocks $\mathcal{C}_R \equiv \{c_R\}$. Thus, the set of clocks in a pair of modset-recseq-automata is $\mathcal{C} \equiv \mathcal{C}_T \cup \mathcal{C}_R$.

There are three sets of variables in the system of automata, where n is the size of the set of state variables: the modset-automaton has a set of local variables $\mathcal{W}_\mathcal{T} \equiv \{s_1...s_n\}$, the recseq-automaton has a set of local variables $\mathcal{W}_\mathcal{R} \equiv \emptyset$, and there is a set of global variables $\mathcal{W}_\mathcal{G} \equiv \{msg, id, value, to\}$. The variables in $\mathcal{W}_\mathcal{G}$ will be used to communicate properties of actions between the pair of automata that are co-simulated (see Section 4.5). Thus, the set of variables in a pair of modset-recseq-automata is $\mathcal{W} \equiv \mathcal{W}_\mathcal{T} \cup \mathcal{W}_\mathcal{R} \cup \mathcal{W}_\mathcal{G}$.

Definition 14 (Updates)**.** A set $Updates$ of pairs $\langle type, value\rangle$ ranged over by u denotes the set of updates in an automaton, where $type$ is a variable or clock that the update concerns, and $value$ is the new value with which the variable is updated. □

We use the notation $u.type$ to denote the variable or clock that the update concerns. Thus, $u.type \in \mathcal{W} \cup \mathcal{C}$.

The function in Definition 15 adds two time intervals and adjusts them according to the precision parameter *pp*. This function is used to add the precision parameter to edges in an automaton, thus implementing the leeway introduced in Section 6.2. The function is called by algorithms 7 and 10 when reaching execute-actions.

Definition 15 (New Time Interval)**.** $nti : 2^{\mathbb{Z}^* \times \mathbb{Z}^*} \times 2^{\mathbb{Z}^* \times \mathbb{Z}^*} \times \mathbb{Z}^* \to 2^{\mathbb{Z}^* \times \mathbb{Z}^*}$

$$\begin{aligned} nti(tt_1, tt_2, pp) = \{&min(tt_1) + min(tt_2 - pp), \\ &max(tt_1) + max(tt_2 + pp)\} \end{aligned}$$

□

The function in Definition 16 is called by the function in Definition 19 and by Algorithm 8 to determine which of two possible msg values that will be used in an edge (i.e. the value received by an immediately preceding action of type *rcv*). There are two mutually exclusive options: either a value originating from a previous execute event immediately prior to the current event, or from the current event. At least one of these is undefined, and the function will ensure that a defined value is used, if it exists.

Definition 16 (Which of two Integers is Defined)**.** $wid : \mathbb{Z}^* \cup \{z\} \times \mathbb{Z}^* \cup \{z\} \to \mathbb{Z}^* \cup \{z\}$

$$wid(msg_1, msg_2) = \begin{cases} msg_2 & when\; msg_1 = z \\ msg_1 & otherwise \end{cases}$$

□

B.3 The modset-automaton transformation functions and automaton definition

In this section, we present the transformation of a modset to a timed automaton.

The guards of edges in the modset-automaton are composed in a given structure: There is a time-span in which the guard is valid if the automaton is in the correct state, the msg-property has a given value, and the properties id and $value$ of the action have been updated as required by the recseq-automaton.

Definition 17 (Guard)**.** A guard $g \in Guards$ for an edge in an automaton can be represented as a tuple of values $\langle t_l, t_m, v_1, v_2, \ldots, v_n, v_{msg}, v_{id}, v_{value}, v_{to} \rangle$, where:

- t_l is the minimum time for the local clock of the automaton for which the guard evaluates to true,
- t_m is the maximum time for the local clock of the automaton for which the guard evaluates to true,
- v_x is the value of local state variable s_x, $x \in \{1, \ldots, n\}$ when the state is built of n variables,
- v_{msg} is the value of a received message $msg \in \mathcal{W}_\mathcal{G}$ in an immediately preceding receive action,
- v_{id} is an evaluation of an attribute $id \in \mathcal{W}_\mathcal{G}$ of the current action, which is the unique identifier of, for example, an IPC-queue or a variable,
- v_{value} is an evaluation of an attribute $value \in \mathcal{W}_\mathcal{G}$ of the current action, and
- v_{to} is an evaluation of an attribute $to \in \mathcal{W}_\mathcal{G}$ of the current action, which is a timeout of a *rcv*-action.

□

Thus, in an automaton with a state of size 2, we could represent a guard as the tuple $\langle 100, 200, 5, 3, z, z, z \rangle$, which is equivalent to the representation $\mathcal{B}(\mathcal{C}) \cup \mathcal{B}(\mathcal{W}) \equiv \{c_T > 100, c_T < 200, s_1 = 5, s_2 = 3\}$, when $\mathcal{C} \equiv \{c_T\}$ and $\mathcal{W} \equiv \{s_1, s_2, msg, id, value\}$. This particular guard would allow the time-span $(100, 200)$ if the first state variable has the value 5, the second state variable has the value 3, and msg, id, and $value$ are all undefined.

We recall the definitions of modtrees and of modset from Section 5.3.

Intuitively, by finding the size of the modset, the function in Definition 18, counts the number of locations required to construct the automaton for the modset by calculating the collected size of the tree.

Definition 18 (Model Event Count)**.** $mec : modset \times \mathbb{Z}^* \to \mathbb{Z}^*$

$$mec(T) = \begin{cases} 0 & when\, T \equiv \emptyset \\ \max\limits_{p=0}^{|T.\mathbb{T}|}(mec(T.\mathbb{T}_p), T.id)) & otherwise \end{cases}$$

□

The function in Definition 19 produces a new edge to add in the automaton. The function is called with information about the source and destination labels, the action, and the guard for the edge. The guard (see Definition 17) is comprised from a time interval, a data state, and four properties. The time interval specifies the minimum and maximum clock values between which the edge is valid. The data state is the tasks data state accumulated from a series of previous update actions. The properties are used to distinguish between different instances of the same action. For example, the variable identifier and the variable value are distinguishing properties of the update action. Independently of the type of the current action, if the previous action was a receive action, the msg-property specifies the value that was received. As execute actions are represented as a time interval rather than an edge, one edge may find a msg-property inherited from a previous execute action. In that case, msg_1 will be defined. If msg_1 is undefined, the current action may hold a defined msg-property. By construction, the two alternatives are mutually exclusive, that is, they cannot both be defined at the same time.

Definition 19 (Model New Edge)**.** $mne : 2^{\mathbb{Z}^* \times \mathbb{Z}^*} \times TaskModelDataStates \times \mathbb{Z}^* \cup \{z\} \times \mathbb{Z}^* \cup \{z\} \times \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \times Locations \times Actions \times Locations \times 2^{Updates} \to Edges$

$$\begin{aligned} &mne(tt, s, msg_1, msg_2, p_{id}, p, to, l, a, l', U) = \\ &\quad \langle l, \langle min(tt), max(tt), s_1, s_2 \dots s_{|s|}, wid(msg_1, msg_2), p_{id}, p, to\rangle, a, U, l'\rangle \end{aligned}$$

□

To perform updates to the data-state, a set of updates is maintained throughout the traversal of the modset. The collected updates for a path in the modset will be added to an edge that represents the end of the job in that path. If an update-action is encountered during the traversal, the function in Definition 20 is called by Algorithm 7 to add an update to the set. A special construct is employed to avoid the set to contain more than one update to the same state variable. The intention is to replace any old update with the new, thus the state that the updates represent will respect the precedence order of the updates.

Definition 20 (Model New Update)**.** $mnu : modset \times 2^{Updates} \times 2^{\mathcal{W}} \rightarrow 2^{Updates}$

$$\begin{aligned} mnu(T, U, \mathcal{W}) = U \backslash \{ & \forall_{s \in \mathcal{W}} \forall_{v \in \mathbb{Z}^*} \langle s, v \rangle | s = s_{T.a.p_{id}} \} \cup \\ & \{ \forall_{s \in \mathcal{W}} \forall_{v \in \mathbb{Z}^*} \langle s, v \rangle | s = s_{T.a.p_{id}} \wedge v = T.a.p \} \end{aligned}$$

□

Algorithm 7 is the main function for producing the modset-automaton, it creates a set of edges for the automaton. A modtree is iterated and the actions encountered determine the action taken.

Model Get Edges; $mge : modset \times Locations \times TaskModelDataStates \times 2^{Updates} \times 2^{\mathbb{Z}^* \times \mathbb{Z}^*} \times \mathbb{Z}^* \cup \{z\} \times 2^{\mathcal{W}} \times 2^{\mathcal{C}_T} \rightarrow 2^{Edges}$.

We are now able to define the modset-automata as Definition 21 shows, using the definitions and algorithms formulated above.

Definition 21 (Timed automata for modset)**.** Assume $\Sigma \equiv \{\mathit{snd}, \mathit{rcv}, \mathit{upd}, \mathit{end}\}$, $\mathcal{C} \equiv \{c_m\}$, and $\mathcal{W} \equiv \{msg, id, value, to, s_0, s_1, \ldots, s_{|guard|-1}\}$. Then, the automata $\mathcal{A}^T$ for modset T, is defined as follows:

- $L \subseteq \{l_0, l_1, \ldots, l_n : n = \max\limits_{T \in modset} mec(T)\}$,
- l_0
- $E \equiv \bigcup\limits_{T \in modset} \bigcup\limits_{s \in T.S} mge(T, L, s, \emptyset, \{0, 0\}, 0, \mathcal{W}, \mathcal{C}_T)$.

□

B.4 The recseq-automaton transformation functions and automaton definition

In this section, we present the translation of a recseq in to a timed automata. We recall the definition of recseq from the previous chapter.

Algorithm 7 $mge(T, l, s, U, tt, msg, \mathcal{W}, \mathcal{C}_T)$, where: T is the current modtree, l is the current location in the automaton, s is the state with respect to which the modset is traversed, U is the set of pending updates found in the traversal, tt is the accumulated execution time from a previous *exe*-action (if applicable), msg is the message received from a previous *rcv*-action (if applicable), $\mathcal{W}$ is the set of variables in the automaton, and $\mathcal{C}_T$ is the set of clocks in the automaton.

```
1:  $nE := \emptyset$
2:  $cU := \{\langle c_T \in \mathcal{C}_T, 0\rangle\}$
3:  $ntt := \{0, 0\}$
4:  if $s \in T.G$ then
5:      if $T.a = snd$ then
6:          $nE := mne(tt, s, msg, T.msg, T.a.p_{id}, T.a.p, z, l, snd, l_{T.id}, cU)$
7:          $nE := \cup_{V \in T.\mathbb{T}} mge(V, l_{T.id}, s, U, ntt, z, \mathcal{W}, \mathcal{C}_T) \cup nE$
8:      else if $T.a = rcv$ then
9:          $nE := mne(tt, s, msg, T.msg, T.a.p_{id}, z, T.a.to, l, rcv, l_{T.id}, cU)$
10:         $nE := \cup_{V \in T.\mathbb{T}} mge(V, l_{T.id}, s, U, ntt, z, \mathcal{W}, \mathcal{C}_T) \cup nE$
11:     else if $T.a = upd$ then
12:         $nE := mne(tt, s, msg, T.msg, T.a.p_{id}, T.a.p, z, l, upd, l_{T.id}, cU)$
13:         $nE := \cup_{V \in T.\mathbb{T}} mge(V, l_{T.id}, s, mnu(T, U, \mathcal{W}), ntt, z, \mathcal{W}, \mathcal{C}_T) \cup nE$
14:     else if $T.a = exe$ then
15:         $nE := \cup_{V \in T.\mathbb{T}} mge(V, l_{T.id}, s, U, nti(tt, T.a.tt, 0), T.msg, \mathcal{W}, \mathcal{C}_T)$
16:     else if $T.a = end$ then
17:         $nE := mne(tt, s, msg, T.msg, z, z, z, l, end, l_{T.id}, cU)$
18:         $nE := mne(T.a.tt, s, msg, T.msg, z, z, z, l_{T.id}, end, l_0, U \cup cU) \cup nE$
19:     end if
20: end if
21: return $nE$
```

We use the following notations: R_p refers to job p in recording R, $R_{p,q}$ refers to event q in job p in recording R, and $o.a$ refers to the action of event $o \in Events$.

A new edge in the recseq-automaton is created by a call to Algorithm 8. The possible actions on the edges are send (*snd*), receive (*rcv*), update (*upd*), end (*end*), and epsilon (ε). Updates in the edge are configured depending on the type of action on the closest subsequent edge that is not an execute-action. For example, checking a subsequent receive will require that the queue identifier and the timeout are updated in this edge.

Trace New Edge; $tne : 2^{\mathbb{Z}^* \times \mathbb{Z}^*} \times Locations \times Actions \cup \{\varepsilon\} \times Locations \times Events \times \mathbb{Z}^* \cup \{z\} \times 2^{\mathcal{W}_\mathcal{G}} \times 2^{\mathcal{C}_R} \rightarrow Edges.$

Algorithm 8 $tne(tt, l, a, l', o, msg, \mathcal{W}_\mathcal{G}, \mathcal{C}_R)$, where: tt is the accumulated execution time from a previous *exe*-action (if applicable), l is the source location for the edge, a is the current action, l' is the destination location for the edge, o is the closest subsequent action which is not an *exe*-action, msg is the message received from a previous *rcv*-action (if applicable), $\mathcal{W}_\mathcal{G}$ is the set of variables manipulated in the automaton, and $\mathcal{C}_R$ is the set of clocks in the automaton.

```
1: $updates := \{\langle c_R \in \mathcal{C}_R, 0\rangle\}$
2: **if** $o.a = snd \vee o.a = upd$ **then**
3:     $updates := \{\langle msg \in \mathcal{W}_\mathcal{G}, wid(msg, o.msg)\rangle\} \cup updates$
4:     $updates := \{\langle id \in \mathcal{W}_\mathcal{G}, o.p_{id}\rangle\} \cup updates$
5:     $updates := \{\langle value \in \mathcal{W}_\mathcal{G}, o.p\rangle\} \cup updates$
6:     **return** $\{\langle l, l', \langle min(tt), max(tt)\rangle, a, updates\rangle\}$
7: **else if** $o.a = rcv$ **then**
8:     $updates := \{\langle msg \in \mathcal{W}_\mathcal{G}, wid(msg, o.msg)\rangle\} \cup updates$
9:     $updates := \{\langle id \in \mathcal{W}_\mathcal{G}, o.p_{id}\rangle\} \cup updates$
10:    $updates := \{\langle value \in \mathcal{W}_\mathcal{G}, o.to\rangle\} \cup updates$
11:    **return** $\{\langle l, l', \langle min(tt), max(tt)\rangle, a, updates\rangle\}$
12: **else if** $o.a = end$ **then**
13:    $updates := \{\langle msg \in \mathcal{W}_\mathcal{G}, wid(msg, o.msg)\rangle\} \cup updates$
14:    **return** $\{\langle l, l', \langle min(tt), max(tt)\rangle, a, updates\rangle\}$
15: **else if** $o.a = \varepsilon$ **then**
16:    **return** $\{\langle l, l', \langle min(tt), max(tt)\rangle, a, updates\rangle\}$
17: **end if**
18: **return** $\emptyset$
```

In order to find the closest subsequent event that is not an execute-action, Algorithm 9 will traverse the trace until the current job ends or an event with another action than execute is encountered.

Trace Get Next; tgn : recseq $\times\ \mathbb{Z}^* \times \mathbb{Z}^* \to Events \cup \{z\}$

Algorithm 9 $tgn(R, p, q)$, where: R is the recseq, p is the index for the current job, and q is the index for the current event in the job.

```
1: if $p \geq |R| \vee p < |R| \wedge q \geq |R_p|$ then
2:     return $z$
3: else if $R_{p,q}.a = exe$ then
4:     return $tgn(R, p, q+1)$
5: else
6:     return $R_{p,q}$
7: end if
```

Algorithm 10 constructs edges for the recseq-automaton. The function iterates the collected trace and calls Algorithm 8 on all encountered events except those with execute-actions. On encountering events with *end*-actions, two edges with end-actions are created. A precision parameter *pp* is set by the initiator of the automata translation, its thoroughly described in Section 6.2.

Trace Get Edges; tge : recseq $\times\ \mathbb{Z}^* \times \mathbb{Z}^* \times \mathbb{Z}^* \times 2^{\mathbb{Z}^* \times \mathbb{Z}^*} \times \mathbb{Z}^* \times \mathbb{Z}^* \times 2^{\mathcal{W}_\mathcal{G}} \times 2^{\mathcal{C}_R} \to 2^{Edges}$

Intuitively, by counting the events of the recseq save all *exe*-actions, the function in Definition 22 (Trace Event Count) calculates the number of locations required to construct the recseq-automaton.

Definition 22 (Trace Event Count, tec : recseq $\times\ \mathbb{Z}^* \times \mathbb{Z}^* \to \mathbb{Z}^*$)**.**

$$tec(R,p,q) = \begin{cases} 0 & when\, p \geq |R| \\ tec(R, p+1, 0) & when\, q \geq |R_p| \\ tec(R, p, q+1) & when\, R_{p,q}.a = exe \\ tec(R, p, q+1) + 1 & otherwise \end{cases}$$

□

We are able to define the recseq-automata as in Definition 23, using the functions formulated above.

Definition 23 (timed automata for recseq)**.** Assume $\Sigma \equiv \{\varepsilon, snd, rcv, upd, end\}$, $\mathcal{C} \equiv \{c_t\}$, and $\mathcal{W}_\mathcal{G} \equiv \{msg, id, value\}$. Let *pp* denote the validity property as specified by the user. Then, the automata $\mathcal{A}^{\text{recseq}}$, representing recseq, is then defined as:

Algorithm 10 $tge(R, p, q, i, tt, pp, msg, \mathcal{W}_{\mathcal{G}}, \mathcal{C}_R)$ where: R is the recseq, p is the index for the current job, q is the index for the current event in the job, i is the current location counter, tt is the accumulated execution time from a previous *exe*-action (if applicable), pp is the precision parameter, msg is the message received from a previous *rcv*-action (if applicable), $\mathcal{W}_{\mathcal{G}}$ is the set of variables manipulated in the automaton, and $\mathcal{C}_R$ is the set of clocks in the automaton.

```
1:  **if** $p \geq |R|$ **then**
2:      **return** $\emptyset$
3:  **else if** $p < |R| \wedge q \geq |R_p|$ **then**
4:      **return** $tge(R, p+1, 0, i+1, tt, pp, z, \mathcal{W}_{\mathcal{G}}, \mathcal{C}_R)$
5:  **else if** $R_{p,q}.a = \mathit{snd} \vee R_{p,q}.a = \mathit{rcv} \vee R_{p,q}.a = \mathit{upd}$ **then**
6:      $nE := tne(tt, l_i, R_{p,q}.a, l_{i+1}, tgn(R, p, q+1), msg, \mathcal{W}_{\mathcal{G}})$
7:      **return** $tge(R, p, q+1, i+1, \{0,0\}, pp, z, \mathcal{W}_{\mathcal{G}}, \mathcal{C}_R) \cup nE$
8:  **else if** $R_{p,q}.a = \mathit{exe}$ **then**
9:      **return** $tge(R, p, q+1, i+1, nti(tt, R_{p,q}.a.t, pp), pp, msg, \mathcal{W}_{\mathcal{G}}, \mathcal{C}_R)$
10: **else if** $R_{p,q}.a = \mathit{end}$ **then**
11:     $nE := tne(tt, l_i, \mathit{end}, l_{i+1}, z, msg, \mathcal{W}_{\mathcal{G}})$
12:     $nE := tne(nti(R_{p,q}.a.tt, \{0,0\}, pp), l_{i+1}, \mathit{end}, l_{i+2}, tgn(R, p+1, 0), msg, \mathcal{W}_{\mathcal{G}}) \cup nE$
13:     **return** $tge(R, p, q+1, i+2, \{0,0\}, pp, z, \mathcal{W}_{\mathcal{G}}, \mathcal{C}_R) \cup nE$
14: **else**
15:     **return** $\emptyset$
16: **end if**
```

- $L \equiv \{l_0, l_1, \ldots l_{tec(\text{recseq},0,0)+1}\}$
- l_0
- $E \equiv tge(\text{recseq}, 0, 0, 0, \{0, 0\}, pp, z, \mathcal{W}_\mathcal{G}, \mathcal{C}_R)$

We let the state $l_{tec(\text{recseq},0,0)+1}$ be labeled “last”. □

Bibliography

[1] Gregory Abowd, Ashok Goel, Dean Jerding, Michael McCracken, Melody Moore, William Murdock, Colin Potts, Spencer Rugaber, and Linda Wills. MORALE. mission oriented architectural legacy evolution. In *Proceedings of International Conference on Software Maintenance*, pages 150–159, October 1997.

[2] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[4] Johan Andersson. Modelling the temporal behavior of complex embedded systems: A reverse engineering approach. Licentiate Thesis, Mälardalen University, Sweden, June 2005. ISSN 1651-9256, ISBN 91-88834-71-9.

[5] Johan Andersson, Joel Huselius, Christer Norström, and Anders Wall. Extracting simulation models from complex industrial real-time systems. In *Proceedings of the International Conference on Software Engineering Advances*, October 2006.

[6] Johan Andersson, Anders Wall, and Christer Norström. Decreasing maintenance costs by introducing formal analysis of real-time behavior in industrial settings. In *Proceedings of the 1^{st} International Symposium on Leveraging Applications of Formal Methods*, October 2004.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.

[8] Osman Balci. Guidelines for successful simulation studies. In *Proceedings of the Winter Simulation Conference*, pages 25–32, December 1990.

[9] Silvino José Silva Bastos and Maria Luiza D´Almeida Sanchez. Modelling real-time systems from object oriented methods. In *Real-Time Embedded System Workshop*. IEEE, December 2001.

[10] Johan Bengtsson, Willem Otto David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated analysis of

an audio control protocol using uppaal. *Journal of Logic and Algebraic Programming*, 52-53:163–181, July-August 2002.

[11] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang 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 III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[12] Per-Olof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifyability analysis (ALMA). *The Journal of Systems and Software*, 69:129–147, 2004.

[13] Lionel C. Briand, Yvan Labiche, and Yucong Miao. Towards the reverse engineering of uml sequence diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 57–66, November 2003.

[14] Ed Brinksma and Angelika Mader. On verification modelling of embedded systems. Technical Report TR-CTIT-04-03, Centre for Telematics and Information Technology, University of Twente, the Netherlands, January 2004. ISSN 1381-3625.

[15] Eric J. Byrne. A conceptual foundation for software re-engineering. In *Proceedings of the Conference on Software Maintenance*, pages 226–235. IEEE Computer Society Press, November 1992.

[16] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[17] Alexander Clark and Franck Thollard. Pac-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, 5:473–497, December 2004.

[18] Conrado Daws and Sergio Yovine. Two examples of verification of multirate timed automata with kronos. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 66–75. IEEE Computer Society, December 1995.

[19] Edsger Wybe Dijkstra. Notes on structured programming. EWD249, circulated privately, April 1970.

[20] Ivonne Erfurth and Wilhelm R. Rossak. A look at typical difficulties in practical software development from the developer perspective–a field study and a first solution proposal with upex. In *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 241–248, March 2007.

[21] Colin Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996.

[22] David Freedman, Robert Pisani, and Roger Purves. *Statistics*. W. W. Norton & Company, 3rd edition, 1998.

[23] Jason Gait. A probe effect in concurrent programs. *Software - Practice and Experience*, 16(3):225–233, March 1986.

[24] Virginia R. Gibson and James A. Senn. System structure and software maintenance. *Communications of the ACM*, 32(3):347–358, 1989.

[25] Allan D. Gordon. *Classification*, volume 82 of *Monographs on statistics and applied probability*. CRC Press, 1999.

[26] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. In *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systmes*, pages 379–396, September 2004.

[27] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. *Lecture Notes in Computer Science*, 4137:435–449, 2006. In Proceedings of the 17th International Conference on Concurrency Theory.

[28] Anders Hessel. Model-based test case selection and generation for real-time systems. Licentiate Thesis, Uppsala University, Sweden, March 2006. ISSN 1404-5117.

[29] Anders Hessel and Paul Pettersson. A test case generation algorithm for real-time systems. In *Proceedings of the 4th International Conference on Quality Software*, pages 268–273, September 2004.

[30] Gerard Johan Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification and Reliability*, 11(2):65–79, May 2001. Special Issue: The First International Workshop on Automated Program Analysis, Testing and Verification.

[31] Gerard Johan Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *Transactions on Software Engineering*, 28(4):364–377, April 2002.

[32] Hyong Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer, April 2002.

[33] Scott Howard. A background debugging mode driver package for modular microcontrollers. Technical Report Motorola Semiconductor Application Note AN1230/D, Motorola Inc., 1996. Available at http://www.freescale.com/files/microcontrollers/doc/app_note/AN1230.pdf, November 2006.

[34] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. In *Proceedings of the International Test Conference*, pages 971–980, September 2003.

[35] Joel Huselius. Preparing for replay. Licentiate Thesis, Mälardalen University, Sweden, November 2003. ISSN 1651-9256, ISBN 91-88834-15-8.

[36] Joel Huselius and Johan Andersson. Model synthesis for real-time systems. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, pages 52–60. IEEE Computer Society, March 2005.

[37] Joel Huselius, Johan Andersson, Hans Hansson, and Sasikumar Punnekkat. Automatic generation and validation of models of legacy software. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 342–349. IEEE Computer Society, August 2006.

[38] Joel Huselius, Johan Kraft, Hans Hansson, and Sasikumar Punnekkat. Evaluating the quality of models extracted from embedded real-time software. In *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 577–585. IEEE Computer Society, March 2007. Presented at the 5th Workshop and Session on Model-Based Development of Computer Based Systems.

[39] Joel Huselius, Daniel Sundmark, and Henrik Thane. Starting conditions for post-mortem debugging using deterministic replay of real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 177–184. IEEE Computer Society, July 2003.

[40] Joel Huselius and Henrik Thane. Constant execution time recording for replay of sporadic real-time systems. In *Proceedings of the 2nd Workshop on Compilers and Tools for Constrained Embedded Systems*, pages 39–47, September 2004.

[41] Joel Huselius, Henrik Thane, and Daniel Sundmark. Availability guarantee for deterministic replay starting points in real-time systems. In *Proceedings of the 5th International Workshop on Automated Debugging*, pages 261–264, September 2003.

[42] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, December 1990. IEEE Std. 610.12-1990.

[43] IEEE. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE, 2001. IEEE Std. 1149.1-2001.

[44] IEEE Computer Society. *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, March 2006.

[45] IEEE-ISTO. *The Nexus 5001 Forum Standard for a Global Embedded Debug Interface*. IEEE, 1999. IEEE-ISTO 5001 1999.

[46] Tauseef A. Israr, Danny H. Lau, Greg Franks, and Murray Woodside. Automatic generation of layered queuing software performance models from com-

monly available traces. In *Proceedings of the 5th International Workshop on Software and Performance*, pages 147–158, July 2005.

[47] Ivar Jacobson and Fredrik Lindström. Re-engineering of an old system to an object-oriented architecture. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pages 340–350, October 1992.

[48] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data.* Prentice Hall, 1988.

[49] Peter Krogsgaard Jensen. *Reliable Real-Time Applications. And How to Use Tests to Model and Understand.* PhD thesis, Aalborg University, Denmark, February 2001.

[50] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Theodore Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, January 2003.

[51] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Sorting and searching. Addison-Wesley, 1st edition, 1973.

[52] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Seminumerical algorithms. Addison-Wesley, 2nd edition, 1981.

[53] Harry Koehnemann and Timothy Lindquist. Towards target-level testing and debugging tools for embedded software. In *Conference Proceedings on TRI-Ada*, pages 288–298. ACM, September 1993.

[54] Hermann Kopetz. The time-triggered model of computation. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 168–177, December 1998.

[55] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *Transactions on Computers*, 36(8):933–940, August 1987.

[56] Kai Koskimies and Erkki Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software - Practice and Experience*, 24(7):643–658, July 1994.

[57] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs.* PhD thesis, Johannes Kepler University of Linz, Austria, September 2000.

[58] Ludmila I. Kuncheva. *Combining pattern classifiers: Methods and algorithms.* Wiley, 2004.

[59] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[60] Carol LeDoux and Stott Parker. Saving traces for ada debugging. In *Proceedings of the Ada International Conference on Ada in Use*, pages 97–108. ACM, May 1985.

[61] Meir Manny Lehman. Programs, life cycles and the laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980.

[62] Meir Manny Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, October 1996.

[63] Nancy Leveson. *Safeware - System, Safety and Computers.* Addison Wesley, 1995.

[64] Jun Li and Peter Hermann Feiler. Impact analysis in real-time control systems. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 443–452, August 1999.

[65] Bennet P. Lientz, E. Burton Swanson, and Gerry Edward Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.

[66] Lennart Lindh, Johan Stärner, Johan Furunäs, Joakim Adomat, and Mohammed El Shobaki. Hardware accelerator for single and multiprocessor real-time operating systems. In *the Seventh Swedish Workshop on Computer Systems Architecture*, June 1998.

[67] Kenneth Littlejohn, Michael V. DelPrincipe, and Jonathan D. Preston. Embedded information system re-engineering. *IEEE Aerospace and Electronic Systems Magazine*, 15(11):3–7, November 2000.

[68] Chung L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[69] George Logothetis and Klaus Schneider. Extending synchronous languages for generating abstract real-time models. In *Proceedings of the 2002 Conference and Exhibition on Design, Automation and Test in Europe*, pages 795–802, March 2002.

[70] George Logothetis, Klaus Schneider, and Christian Metzler. Generating formal models for real-time verification by exact low-level analysis of synchronous programs. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 256–265. IEEE, December 2003.

[71] Jacob Rubin Lorch. *Operating Systems Techniques for Reducing Processor Energy Consumption.* PhD thesis, University of California at Berkeley, USA, 2001.

[72] Ciaran MacNamee and Donal Heffernan. Emerging on-chip debugging techniques for real-time embedded systems. *Computing & Control Engineering Journal*, 11(6):295–303, December 2000.

[73] Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, November 1995.

[74] John Marciniak, editor. *Encyclopedia of Software Engineering.* Wiley-Interscience, 2nd edition, December 2001.

[75] Frank J. Massey, Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, March 1951.

[76] Charles McDowell and David Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[77] John Mellor-Crummey and Thomas LeBlanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86. ACM, April 1989.

[78] Micro Digital. *SMX User's Guide, 3.7*, 2005.

[79] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[80] Johan Moe and David Carr. Using execution trace data to improve distributed systems. *Software - Practice and Experience*, 32(9):889–906, July 2002.

[81] NIST/SEMATECH. e-handbook of statistical methods. Internet URL http://www.itl.nist.gov/div898/handbook/, November 2006.

[82] Dag Nyström. *Data Management in Vehicle Control-Systems.* PhD thesis, Mälardalen University, Sweden, October 2005.

[83] OSE Systems. *OSEck Kernel Reference Manual*, 2006.

[84] Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw. Theories and techniques of program understanding. In *Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 37–53. IBM Press, 1991.

[85] Walter Piegorsch. Sample sizes for improved binomial confidence intervals. *Computational Statistics & Data Analysis*, 46(2):309–316, June 2004.

[86] James Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software - Practice and Experience*, 29(2):125–142, February 1999.

[87] Stefan Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems.* PhD thesis, Technishe Universität Wien, Austria, April 1994.

[88] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the International Conference on Software Maintenance*, pages 13–22, August 1999.

[89] Tamar Richner-Hanna. *Recovering Behavioral Design Views: a Query-Based Approach.* PhD thesis, Universität Bern, Switzerland, May 2002.

[90] Matthias Riebisch and Sven Wohlfarth. Introducing impact analysis for architectural decisions. In *Proceedings of the* 14^{th} *Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 381–390, March 2007.

[91] Michiel Ronsse, Mark Christiaens, and Koenraad De Bosschere. Cyclic debugging using execution replay. In *International Conference on Computational Science*, volume 2074 of *LNCS*, pages 851–860, May 2001.

[92] Robert G. Sargent. Verification and validation of simulation models. In *Proceedings of the Winter Simulation Conference*, pages 130–143, December 2005.

[93] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, July 2006.

[94] Bernhard Schätz. Model-based engineering of embedded control software. In *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 53–62, March 2006.

[95] Werner Schütz. Fundamental issues in testing distributed real-time systems. *Real-Time Systems*, 7(2):129–157, September 1994.

[96] Mohammed El Shobaki and Lennart Lindh. A hardware and software monitor for high-level system-on-chip verification. In *Proceedings of the IEEE International Symposium on Quality Electronic Design*, pages 56–61, March 2001.

[97] Guoqiang Shu, Chao Li, Qing Wang, and Mingshu Li. Validating objected-oriented prototype of real-time systems with timed automata. In *Proceedings of the* 13^{th} *IEEE International Workshop on Rapid System Prototyping*, pages 99–106, July 2002.

[98] Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. Building models of real-time systems from application software. *Proceedings of the IEEE*, 91(1):100–111, January 2003.

[99] David Snelling and Geerd-R. Hoffmann. A comparative study of libraries for parallel processing. *Parallel Computing*, 8(1-3):255–266, 1988.

[100] Joseph L. Sowers and Paul E. Rubin. Reversing course: Issues in modeling legacy systems. *Telecommunication Systems*, 16(1,2):147–157, 2001.

[101] Jack Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, 1988.

[102] Darlene Stewart and Morven Gentleman. Non-stop monitoring and debugging on shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 263–269. IEEE Computer Society, May 1997.

[103] Sun Microsystems. *Solaris Dynamic Tracing Guide*, 2005.

[104] Daniel Sundmark, Henrik Thane, Joel Huselius, Anders Pettersson, Roger Mellander, Ingemar Reiyer, and Mattias Kallvi. Replay debugging of complex real-time systems: Experiences from two industrial case studies. In *Proceedings of the 5th International Workshop on Automated Debugging*, pages 211–222, September 2003.

[105] Tarja Systä and Kai Koskimies. Extracting state diagrams from legacy systems. In *Proceedings of the Workshops on Object-Oriented Technology*, pages 272–273, 1997. LNCS 1357.

[106] Tivadar Szemethy and Gabor Karsai. Platform modeling and model transformations for analysis. *Journal of Universal Computer Science*, 10(10):1383–1407, October 2004.

[107] Tivadar Szemethy, Gabor Karsai, and Daniel Balasubramanian. Model transformations in the model-based development of real-time systems. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 177–186, March 2006.

[108] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Kungliga Tekniska Högskolan, Sweden, May 2000.

[109] Henrik Thane and Hans Hansson. Testing distributed real-time systems. *Journal of Microprocessors and Microsystems, Elsevier*, 24(9):463–478, February 2001.

[110] Henrik Thane, Daniel Sundmark, Joel Huselius, and Anders Pettersson. Replay debugging of real-time systems using time machines. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 288–295. IEEE Computer Society, April 2003. Presented at the First International Workshop on Parallel and Distributed Systems: Testing and Debugging.

[111] Jeffrey Tsai, Yaodong Bi, Steve Yang, and Ross Smith. *Distributed Real-Time Systems: Monitoring Visualization and Debugging and Analysis*. Wiley-Interscience, 1996.

[112] Jeffrey Tsai, Kwang-Ya Fang, and Horng-Yuan Chen. A replay mechanism for non-interference real-time software testing and debugging. In *Proceedings of the Conference on Software Maintenance*, pages 209–218, October 1989.

[113] Jeffrey Tsai, Kwang-Ya Fang, Horng-Yuan Chen, and Yao-Dong Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[114] Sebastian Uchitel and Jeff Kramer. A workbench for synthesising behaviour models from scenarios. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 188–197, May 2001.

[115] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *Transactions on Software Engineering*, 29(2):99–115, February 2003.

[116] Anders Wall. *Architectural Modeling and Analysis of Complex Real-Time Systems*. Phd thesis no. 5, Mälardalen University, Sweden, September 2003. Available at: www.mrtc.mdh.se.

[117] Peter Wegner. Research paradigms in computer science. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 322–330. IEEE Computer Society Press, October 1976.

[118] Wind River Systems, Inc. *VxWorks Programmer's Guide, 5.5*, 2003.

[119] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *Proceedings of the 2004 International Conference on Software Engineering*, pages 470–479, May 2004.

[120] Stephen W. L. Yip, Tom Lam, and Stephen K. M. Chan. A software maintenance survey. In *Proceedings of the First Asia-Pacific Software Engineering Conference*, pages 70–79, December 1994.

[121] Hong Zhu, Patrik A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Jag spillde ut en kopp hett te över tangentbordet på min första laptop. Det förorsakade ett fullständigt elektroniskt haveri. Skärmen såg ut som ett tropiskt åskväder, och sedan gick det inte att få kontakt med hårddisken. [...] Egendomligt nog blev jag mycket upprymd och greps av den löjeväckande illusionen att jag aldrig skulle behöva skriva en rad mer.

(Horace Engdahl, Dagens Teknik, Nr 1 2007)