University of California,
Irvine

Model-based Analysis of Event-driven Distributed Real-time Embedded Systems

Dissertation

submitted in partial satisfaction of the requirements
for the degree of

Doctor of Philosophy

in Computer Science

by

Gabor Madl

Dissertation Committee:
Chancellor's Professor Nikil D. Dutt, Chair
Professor Tony Givargis
Professor Ian G. Harris

2009

The dissertation of Gabor Madl
is approved and is acceptable in quality and form for
publication on microfilm and in digital formats:

_____

_____

_____

Committee Chair

University of California, Irvine
2009

"The higher we soar, the smaller we appear to those who cannot fly."

Friedrich Nietzsche

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ABS Anti-lock Braking System.

ACE Adaptive Communication Environment.

ADL Architecture Description Language.

AEDRE Asynchronous Event-driven Distributed Real-time Embedded.

AIRES Automatic Integration of Reusable Embedded Software.

ALDERIS Analysis Language for Distributed, Embedded, and Real-time Systems.

AMBA AHB ARM Advanced Microcontroller Bus Architecture Advanced High-speed Bus.

AMI Asynchronous Method Invocation.

ASIC Application-specific Integrated Circuit.

AVC Advanced Video Coding.

BCET Best Case Execution Time.

BDD Binary Decision Diagram.

BFS Breadth First Search.

BPC Bit Plane Coder.

CABAC Context-Adaptive Binary Arithmetic Coding.

CARTA Cross-abstraction Real-time Analysis.

CAVLC Context-Adaptive Variable Length Coding.

CCATB Cycle Count Accurate At Transaction Boundaries.

CCM CORBA Component Model.

CF Context Formatter.

CORBA Common Object Request Broker Architecture.

CoSMIC Component Synthesis using Model Integrated Computing.

COTS Commercial off-the-shelf.

CPS Cyber-physical Systems.

CPU Central Processing Unit.

CTL Computational Tree Logic.

DCT Discrete Cosine Transform.

DE Discrete Event.

DES Discrete Event Simulation.

DFS Depth First Search.

DMA Direct Memory Access.

DQ/IT Discrete Quantization/Inverse Transform.

DRE Distributed Real-time Embedded.

DREAM Distributed Real-time Embedded Analysis Method.

DSM Domain-specific Model.

DSML Domain-specific Modeling Language.

DVFS Dynamic Voltage Frequency Scaling.

DWT Discrete Wavelet Transform.

EBCOT Embedded Block Coding with Optimal Truncation.

ECU Engine Control Unit.

EDF Earliest Deadline First.

EJB Enterprise JavaBeans.

EOT Event Order Tree.

ESL Electronic System Level.

ESP Electronic Stability Programme.

FIFO First In First Out.

FSM Finite State Machine.

GME Generic Modeling Environment.

GPS Global Positioning System.

GREAT Graph Rewriting and Transformation.

HA Hybrid Automata.

ICT Irreversible Color Transform.

IDE Integrated Development Environment.

IP Intellectual Property.

LCD Liquid Crystal Display.

LET Logical Execution Time.

LTL Linear Time Logic.

MB Macroblock.

MCT Multi-Component Transform.

MDA Model-driven Architecture.

MIC Model-integrated Computing.

MoC Model of Computation.

MPSoC Multi-processor System-on-Chip.

MQ-Coder Arithmetic Coder.

OCL Object Constraint Language.

OMG Object Management Group.

ORB Object Request Broker.

OSATE Open Source AADL Tool Environment.

PCI Peripheral Component Interconnect.

PEARSE Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals.

POA Portable Object Adapter.

PTIDES Programming Temporally Integrated Distributed Embedded Systems.

QCIF Quarter Common Intermediate Format.

QoS Quality of Service.

RCPSP Resource Constrained Project Scheduling Problem.

RCT Reversible Color Transform.

RMA Rate Monotonic Analysis.

ROI Region Of Interest.

RTL Register-Transfer Level.

SA Stopwatch Automata.

SAE AADL Society of Automotive Engineers: Architecture Analysis & Design Language.

SCADA Supervisory Control and Data Acquisition.

TA Timed Automata.

TAO The ACE ORB.

TCTL Timed Computational Tree Logic.

TDMA Time Division Multiple Access.

TSA Task Stopwatch Automaton.

TTA Task Timed Automaton.

TTDRE Time-triggered Distributed Real-time Embedded.

UART Universal Asynchronous Receiver/Transmitter.

UML Unified Modeling Language.

VEST Virginia Embedded Systems Toolkit.

VIATRA VIsual Automated model TRAnsformations.

WCET Worst Case Execution Time.

XML Extensible Markup Language.

Acknowledgements

I would like to express my gratitude to all the people who supported me turning this work into reality. First and foremost, to my advisor, Professor Nikil D. Dutt. I thank him for the respect that he so greatfully had shown towards me. His guidance and support was crucial in developing this dissertation and myself. I am grateful for the freedom that he has granted me to pursue my interests. Professor Dutt had shown me the true meaning of mentoring, and I cannot thank him enough for the time and effort he spent on my education.

Second, I would like to express my gratitude to my past mentor, Professor Sherif Abdelwahed for showing me the beauty of science. I thank him for his endless patience and guidance developing this work that once seemed impossible. I thank him for his faith in me, and for standing by me when I needed it most.

The research in this work was sponsored by the National Science Foundation grants CCR-0225610, ACI-0204028, CNS-0615438, CNS-0613971, a grant by the Center for Pervasive Communications and Computation, and funding by Fujitsu Laboratories of America. Financial support is graciously acknowledged.

I would like to thank all the great researchers in the CECS lab at UC Irvine who have directly or indirectly contributed to my research. In particular, I thank Sudeep Pasricha, Luis A. D. Bathen, Kyoungwoo Lee, Minyoung Kim, Qiang Zhu, Ilya Issenin, Shireesh Verma, Radu Cornea, Jayram M. Nageshwaran, Arup Chakraborty, Jesse Dannenbring, Jeff Furlong for interesting discussions and their contributions. I am grateful to Professor Tony Givargis and Professor Ian G. Harris for their guidance in developing this dissertation, and to Professor Nalini Venkatasubramanian, Professor Fadi J. Kurdahi, and Professor Ahmed M. Eltawil for the interesting discussions and motivating my research. I am also thankful to Melanie Sanders, Melanie Kilian, and Grace Wu for their help in administrative matters.

I thank the researchers and engineers at Vanderbilt University who contributed to my research. In particular, I am grateful to Professor Douglas C. Schmidt and the DOC group for motivating my research interests in distributed real-time embedded systems, and for his support. Professor Schmidt has demonstrated to me that hard work and discipline are essential for success. I thank Professor Janos Sztipanovits for introducing me to the semantics of model-based design, and strengthening my confidence to stand on my own feet. Without his help, I would not have taken the leap ahead. I am grateful to Professor Jonathan Sprinkle, Professor Brandon Eames, Professor Miklós Maróti, Professor Ákos Lédeczi, Professor Gabor Karsai, Andrew D. Dixon, Matthew J. Emerson, Kai Chen, Ethan Jackson, Stoyan Paunov, János Sallai, György Balogh, Branislav Kusy and Sebestyén Dóra for shaping my research. I am also grateful to all my friends for all the great times spent together.

I am grateful to Professor András Pataricza and Professor István Majzik from the Budapest University of Technology and Economics for their mentoring and help during my Master's course work.

I thank the great engineers who influenced my research during my work and internships. In particular, I am grateful to Balázs Gábor Józsa from Loxon Solutions, Peter Grun and Chulho Shin from ARM, Praveen K. Murthy, Wei-Peng Chen and Sreeranga P. Rajan from Fujitsu Laboratories of America, and Zhenyu (Victor) Liu from Google, among others.

I am especially grateful to my family for their love and unconditional support. There are no words that would suffice to thank my parents for their help and support. I thank my fiancée, Andreia S. Dias for standing by me during the hard times, and enabling me to pursue my dreams. Finally, I thank the governator, Arnold Schwarzenegger, for the bragging rights for having his signature on my diploma.

Curriculum Vitae

Gabor Madl

Education

2009	Ph.D. in Computer Science, University of California, Irvine, USA
2005	M.Sc. in Computer Science, Vanderbilt University, USA
2002	M.Sc. in Computer Engineering, Budapest University of Technology and Economics, Hungary

Research & Industry Experience

2005 - 2009	Graduate Researcher	University of California, Irvine, USA
2008	Software Engineer Intern	Google, USA
2007	Intern	Fujitsu Laboratories of America, USA
2005	Intern	ARM, USA
2003 - 2005	Research Assistant	Vanderbilt University, USA
1999 - 2002	Software Engineer	Loxon Solutions, Hungary

Award

2008 The ACM SIGBED/SIGSOFT Frank Anger Memorial Award.

Selected Publications

1. *Gabor Madl, Sudeep Pasricha, Qiang Zhu, Luis Angel D. Bathen, Nikil Dutt*: Combining Transaction-level Simulations and Model Checking for MPSoC Verification and Performance Evaluation, submitted to *ACM Transactions on Design Automation of Electronic Systems*.

2. *Gabor Madl, Sudeep Pasricha, Nikil Dutt, Sherif Abdelwahed*: Cross-abstraction Functional Verification and Performance Analysis of Chip Multiprocessor Designs, submitted to *IEEE Transactions on Industrial Informatics*.

3. *Gabor Madl, Sherif Abdelwahed, Douglas C. Schmidt*: Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking, *Real-Time Systems, Special Issue: Invited Papers from the 25th IEEE International Real-Time Systems Symposium*, Volume 33, Numbers 1–3, Pages 77–100, July 2006.

4. *Gabor Madl, Nikil Dutt, Sherif Abdelwahed*: A Conservative Approximation Method for the Verification of Preemptive Scheduling using Timed Automata, In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Pages 255-264, 2009.

5. *Gabor Madl, Nikil Dutt*: Real-time Analysis of Resource-Constrained Distributed Systems by Simulation-Guided Model Checking, Ph.D. Forum, *the 28th IEEE International Real-Time Systems Symposium (RTSS)*, 2007.

6. *Gabor Madl, Nikil Dutt, Sherif Abdelwahed*: Performance Estimation of Distributed Real-time Embedded Systems by Discrete Event Simulations, In *Proceedings of EMSOFT*, Pages 183–192, 2007.

7. *Gabor Madl, Sudeep Pasricha, Qiang Zhu, Luis Angel D. Bathen, Nikil Dutt*: Formal Performance Evaluation of AMBA-based System-on-Chip Designs, In *Proceedings of EMSOFT*, Pages 311–320, 2006.

8. *Gabor Madl, Sherif Abdelwahed*: Model-based Analysis of Distributed Real-time Embedded System Composition, In *Proceedings of EMSOFT*, Pages 371–374, 2005.

9. *Gabor Madl, Sherif Abdelwahed, Gabor Karsai*: Automatic Verification of Component-Based Real-Time CORBA Applications, In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, Pages 231–240, 2004.

10. *Gabor Madl, Nikil Dutt*: Domain-specific Modeling of Power Aware Distributed Real-time Embedded Systems, *Proceedings of the 6th Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Pages 59–68, 2006.

11. *Chulho Shin, Peter Grun, Nizar Romdhane, Christopher Lennard, Gabor Madl, Sudeep Pasricha, Nikil Dutt, Mark Noll*: Enabling heterogeneous cycle-based and event-driven simulation in a design flow integrated using the SPIRIT consortium specifications, *Design Automation for Embedded Systems*, Volume 11, Numbers 2-3, September 2007.

12. *Dror G. Feitelson, Tokunbo O. S. Adeshiyan, Daniel Balasubramanian, Yoav Etsion, Gabor Madl, Esteban P. Osses, Sameer Singh, Karlkim Suwanmongkol, Charlie Xie, and Stephen R. Schach*: Fine-Grain Analysis of Common Coupling and its Application to a Linux Case Study, *Journal of Systems and Software*, Volume 80, Issue 8, Pages 1239–1255, 2007.

13. *Stephen R. Schach, Tokunbo O. S. Adeshiyan, Daniel Balasubramanian, Gabor Madl, Esteban P. Osses, Sameer Singh, Karlkim Suwanmongkol, Minhui Xie, and Dror G. Feitelson*: Common Coupling and Pointer Variables, with Application to a Linux Case Study, *Software Quality Journal*, Volume 15, Number 1, Pages 99–113, 2007.

14. *Peter Grun, Chulho Shin, Chris Baxter, Christopher Lennard, Mark Noll, Gabor Madl*: Integrating a multi-vendor ESL-to-silicon design flow using SPIRIT, *IP-SoC 2005*.

Abstract of the Dissertation

Model-based Analysis of Event-driven Distributed Real-time Embedded Systems

By

Gabor Madl

Doctor of Philosophy in Computer Science
University of California, Irvine, 2009
Chancellor's Professor Nikil D. Dutt, Chair

As embedded systems become increasingly networked, and interact with the physical world, *Distributed Real-time Embedded (DRE)* systems emerge. DRE systems range from small-scale *Multi-processor Systems-on-Chip (MPSoCs)* operating in resource constrained environments such as cell phone platforms, medical devices and sensor networks all the way to large-scale software-intensive systems of systems used in avionics, ship computing environments, and in supervisory control and data acquisition systems managing regional power grids.

This dissertation focuses on the model-based analysis of event-driven DRE systems. Event-driven DRE systems are based on a reactive communication paradigm, where the execution of tasks is triggered asynchronously, invoked by external events, interrupts, or by other tasks. Events can also express time, providing a common semantic domain for the compositional analysis of time- *and* event-driven DRE systems.

Key technical contributions of this dissertation are *(1)* the specification of a formal semantic domain for DRE systems, *(2)* a model checking method for the real-time verification of non-preemptive DRE systems by timed automata, *(3)* a performance

estimation method for DRE systems by discrete event simulations, *(4)* a conservative approximation method for the verification of preemptive event-driven asynchronous DRE systems by timed automata, *(5)* a method for the functional verification and performance estimation of MPSoCs built on an industry standard MPSoC interconnect protocol, and *(6)* a cross-abstraction real-time analysis method for MPSoC designs utilizing bus matrix interconnects.

The novelty of our approach lies in combining formal methods and symbolic simulations for the system-level evaluation of DRE designs early in the design flow, and utilizing multiple abstractions to trade off analysis accuracy in scalability.

We implemented the proposed analysis methods in the open-source *Distributed Real-time Embedded Analysis Method (*DREAM*)* framework for the model-based real-time verification and performance estimation of DRE systems. DREAM focuses on the practical application of formal analysis methods to automate the verification, development, configuration, and integration of event-driven DRE systems. DREAM is available for download at `http://dre.sourceforge.net`. We applied the proposed design flow to the domain of software-intensive mission-critical avionics DRE applications, and the domain of multimedia MPSoCs.

Chapter 1

Introduction

Technological advances have had a major impact on our lives in the past few decades. We have witnessed the rise of the personal computer, have experienced the birth of the Internet, the emergence of online shopping, and have adopted e-mails and social networks to keep in touch with each other. There have been several crucial technology breakthroughs that allowed this *"information revolution"* that may not have been obvious to the casual observer. Advances in semiconductor technology provided ever faster and smaller microchips, satisfying the demand for growing computation power needed to build the necessary infrastructure to run the Internet. Optical cables, high-speed routers, and more recently, wireless communication were developed to keep up with the growing need for low latency, high bandwidth, mobile access points.

Innovation, however, does not stop at the Internet. Computers permeated not just communication, but nearly every aspect of our lives; an average car now comes equipped with more than a dozen processors, and high-level models may contain 5–6 times as many, managing not only the correct operation of the vehicle, but even correcting driver mistakes as electronic driving assist systems. Such processors and computation platforms are referred to as *embedded systems*. Embedded systems are often built with a special purpose or application area, and need to maintain an active interaction with their environment.

The use of embedded systems in avionics is even more widespread; fly-by-wire systems give total control to the pilots, where no mechanical link exists between the cockpit and the control mechanisms on the wings of the aircraft. Instead, electronic signals are sent to actuators that control the ailerons, elevators, and flaps with forces and precision that a human using mechanical links is incapable of.

Modern health care would be nearly unimaginable without the widespread use of embedded systems for diagnosis, monitoring, and even medicine delivery. Sensor networks are now beginning to be deployed to monitor health signals, production environments, buildings, traffic, and hundreds of other areas, providing a way for a more active interaction with the physical environment.

We increasingly rely on embedded technology even to store and prepare our food; "smart" refrigerators, high-end programmable fireplaces, microwave ovens, and even temperature monitoring devices are now widely used in a well-equipped kitchens. Grilling with friends in the backyard, we might not realize that we may have more computation power in our pockets – such as cell phones, watches, PDAs, MP3 players – than well-equipped computer labs just a few decades ago.

Cell phones have morphed into mobile computation platforms, that serve as calendars, digital cameras, platforms for e-mail and real-time chat with friends and colleagues, multimedia devices that store the whole library of our favorite songs, play our favorite video clips from the Internet, and can even prove to be formidable opponents in games, providing rich services that we occasionally interrupt with a phone call.

1.1 Distributed Real-time Embedded Systems

As embedded systems become increasingly networked, and interact with the physical world, *Distributed Real-time Embedded (DRE)* systems emerge. DRE systems range from small-scale *Multi-processor Systems-on-Chip (MPSoCs)* operating in resource-constrained environments such as cell phone platforms, medical devices and sensor networks all the way to large-scale software-intensive systems of systems used in avionics, ship computing environments, and in *Supervisory Control and Data Acquisition (SCADA)* systems managing regional power grids.

DRE systems provide the platform for the implementation of *Cyber-physical Systems (CPS)*, that increasingly run in open environments, in less predictable conditions than previous generations of real-time and embedded systems that are specialized for specific application domains. DRE systems provide a highly adaptive and flexible infrastructure for reusable resource management services, thereby providing a platform for the implementation of CPS.

Component-based design is an emerging paradigm for the engineering of complex high-availability DRE systems. Components represent reusable services, which can be configured and composed together to provide some functionality. Components provide an intuitive way to reuse proven designs and implementation in DRE systems, shifting the focus from development by construction to a *build-by-composition* methodology, where functionality is provided by the composition of verified and tested components.

Components have been successfully applied to both *Quality of Service (QoS)*-aware middleware [11] and hardware design [4]. In the software context, *component middleware* defines platform capabilities, and tools for specifying, implementing, deploying, and configuring *components* [78], and publish/subscribe services [41] that exchange messages between components. Components, in turn, are units of implementation, reuse, and composition that expose named interfaces called *ports*, which are connection points that components use to collaborate with each other. Component middleware helps simplify the development and validation of DRE systems by providing reusable services, and optimizations that support their functional, and QoS needs more effectively than conventional *ad hoc* software implementations.

In the hardware context, a component-based abstraction is a natural fit for MPSoC design, where *correct-by-construction* engineering is an economic necessity. In the MPSoC domain, bus interconnects and protocols are well-established [82], and services are implemented by synthesizing the communication subsystem [83] to integrate heterogeneous components on a common platform in a manner that respects key design

constraints, such as real-time performance, throughput, power consumption, area and temperature constraints.

We can group DRE systems in two major categories; *Time-triggered Distributed Real-time Embedded (TTDRE)* systems and *Asynchronous Event-driven Distributed Real-time Embedded (AEDRE)* systems. In the following subsections we describe the characteristics used for the categorization, and discuss key properties, application domain, and differences between the two categories.

1.1.1 Time-triggered Distributed Real-time Embedded Systems

TTDRE systems extend the concepts of the *time-triggered architecture* [58] to distributed and embedded systems. In TTDRE systems time in distributed components is synchronized to a global clock, and the execution of tasks is triggered by the clock. By separating the invocation of tasks from their activation, TTDRE systems achieve deterministic time behavior; by synchronizing the start of execution of tasks with the global clock designers achieve a high degree of *predictability*, and are able to express which tasks are allowed to execute at any point in time.

TTDRE systems provide an abstraction that allows designers to utilize the concept of *Logical Execution Time (LET)* [44], thereby creating an abstraction supporting deterministic time behavior. Logical execution time specifies a task's execution time as a constant $c \in \mathbb{R}_{\geq 0}$. The activation time of the task $t_a \in \mathbb{R}_{\geq 0}$ is tied to a specific valuation of the global clock, and the output of the task is available at $t_a + c$ time. By enforcing the constraint that none of the task's dependents may start their execution before $t_a + c$, we ensure that all the task's dependents have access to the correct output. Asynchronous activation is not allowed; should the task finish earlier than $t_a + c$, its dependents still have to wait until they are activated by the global clock at time $t_x \in \mathbb{R}_{\geq 0}, t_a + c \leq t_x$. While this approach has the undesired effect that it may result in tasks idling simply because they have not been activated yet, it also has the advantage

that it turns a task's execution time to constant for analysis purposes. Constant c is typically overestimated to avoid tasks from finishing after their deadline, making the idle time relatively long and the system inefficient. Constant c acts as the logical execution time; regardless of whether the task finished its execution faster or slower, designers are able to treat its execution as constant, resulting in deterministic time behavior, better predictability and greatly improved analysis performance. While time-triggered activation of tasks and logical execution time often appear together, they are different concepts, even though both aim to achieve deterministic timed behavior.

TTDRE systems perform very well regarding the real-time aspect of DRE systems, but have some disadvantages when considering distributed or resource-constrained embedded systems. Advantages of TTDRE systems include:

- *Deterministic timed behavior:* time-triggered tasks utilizing the concept of logical execution time provide a design and analysis abstraction where designers can specify with confidence which tasks are executing at what time. The execution order of tasks is fixed, and the execution of TTDRE systems is predictable, and therefore TTDRE systems can be efficiently analyzed by static analysis methods.

- *Improved analysis performance and scalability:* due to the deterministic behavior, analysis is greatly simplified. The execution order and activation times are fixed, therefore a single simulation of the TTDRE system can verify the satisfiability of all time-constraints.

Disadvantages of TTDRE systems include:

- *Global synchrony needs to be enforced on a distributed platform:* this problem becomes more significant as the scale of the system grows. The natural drift between clocks in distributed systems results in increasing loss of accuracy. To enforce global synchrony, TTDRE systems periodically need to perform global

time synchronization, that gets harder as distances between components increase. Practical TTDRE systems generally have limited size; such as *Anti-lock Braking System (ABS)* and *Electronic Stability Programme (ESP)* and *Engine Control Unit (ECU)* systems. Increasing the scale of TTDRE systems to physically more diverse platforms, such as web services, shipboard computing, SCADA systems, telecommunications is a very challenging task. In some cases the scale of TTDRE systems is improved by utilizing the *Global Positioning System (GPS)* for time synchronization.

- *Higher cost of implementation:* TTDRE systems are less widespread than AEDRE systems, and therefore suffer in comparison to AEDRE systems when one considers the degree of reusability. Open-source software provided us with *Commercial off-the-shelf (COTS)* real-time operating systems such as real-time Linux, middleware such as *Common Object Request Broker Architecture (CORBA)*, open-source web servers, software radio etc. There are very few results that TTDRE developers may reuse in their design. In the automotive domain there are some standards for time-triggered communication buses such as *Flexray* or *CAN bus*, but the implementation of TTDRE systems often resembles a build-from-scratch approach, leading to longer development times. Designers need to be familiar with the concepts of TTDRE implementations, leading to more expensive workforce. The extra costs can often be justified only in the domain of mission-critical computing.

- *Limited extensibility:* once a TTDRE system is implemented, it becomes challenging to extend it. Designers cannot take the freedom to influence the time behavior without considering its implications on the overall schedulability of the system. Adding a piece of extra code to a component may increase its execution time behind the logical execution time, that requires re-evaluating the whole

scheduling policy of the TTDRE implementation. Extending a TTDRE system will in most cases involve a complete redesign of the task activations to adhere to schedulability constraints.

- *Sub-optimal utilization due to resources "wasted" waiting for the global clock tick:* in TTDRE systems tasks are often forced to be idle. When a task finished earlier than its logical execution time, *slack time* occurs. Slack times are generally small in duration, but appear very frequently, leading to under-utilization of the resources. While low-priority tasks may execute in slack times, implementing such extensions to a mission-critical system is very challenging.

- *Decreased energy efficiency due to low utilization:* since the utilization of TTDRE systems is sub-optimal, energy consumption suffers significantly. Advanced power management techniques such as *Dynamic Voltage Frequency Scaling (DVFS)* have limited efficiency in TTDRE systems, as manipulating the performance of the underlying processor has implications on the timing behavior of the system.

1.1.2 Asynchronous Event-driven Distributed Real-time Embedded Systems

The vast majority of DRE systems fall in the category of AEDRE systems. AEDRE systems are based on a reactive, event-driven communication paradigm, where the execution of tasks is triggered asynchronously, depending on when they are invoked by external events, or other tasks. Event-driven systems provide a natural abstraction for DRE systems, as they closely resemble biological systems; whenever external events occur, the reaction follows as soon as possible.

The time behavior of AEDRE systems is inherently non-deterministic; the execution times of tasks depend on a myriad of factors, including utilization, caching, and even

the actual input data. For example, image compression performance may vary based on how dark or detailed the image is. Since AEDRE systems are built on asynchronous event-based triggering; whenever a task finishes its execution, its dependents are enabled for execution. Therefore, non-deterministic execution times of tasks result in non-deterministic triggering, and even execution order of the tasks. Although the increased degree of non-determinism results in worse analysis performance compared to TTDRE systems, the implementation of AEDRE systems is generally less costly. Advantages of AEDRE systems include:

- *Improved utilization:* unlike TTDRE systems, AEDRE systems generally do not require global synchronization, resulting in less complex implementation, and more efficient use of resources.

- *Lower cost of implementation:* most production DRE systems are predominantly AEDRE systems. Designers have a wide range of options to reuse existing COTS frameworks, including operating systems, middleware, databases, web servers etc. Most distributed software frameworks also follow an event-driven paradigm. AEDRE systems often have the advantage of cost due to the increased reuse of existing frameworks and paradigms.

- *Easier extensibility:* extending tasks may be as easy as extending the implementation of functions/classes. Component-based middleware and object-oriented software libraries can be readily integrated into AEDRE systems. However, the resulting code still needs to be checked for real-time properties.

- *Improved utilization of resources:* AEDRE systems do not face the problem of slack times, as tasks may start as soon as resources are available, potentially resulting in faster response times, as well as higher throughput.

- *Improved energy consumption:* the event-driven communication model provides a good match for advanced power management techniques such as DVFS. Nearly

all software written for cell phones, mobile platforms, or MPSoCs follows an event-driven communication model due to its improved energy efficiency.

Disadvantages of AEDRE systems include:

- *Non-deterministic timed behavior:* the execution time of tasks in AEDRE systems is non-deterministic in general, as a result of data-dependent processing, memory management, caching, congestions on communication buses etc. Non-deterministic execution times and the asynchronous event-driven activation of tasks results in the non-deterministic execution order of tasks, that is less predictable than TTDRE systems.

- *Decreased analysis performance and scalability:* analysis performance is significantly slower than in the case of TTDRE systems, as sources of non-determinism need to be enumerated for real-time verification. Verifying large-scale AEDRE systems is a challenging task, and requires significant effort and cost.

1.1.3 Composing Time- and Event-driven Distributed Real-time Embedded Systems

DRE systems are increasingly complex and diverse, and the question whether AEDRE or TTDRE systems should be used depends on the application domain, and key design constraints.

In most heterogeneous DRE systems, AEDRE and TTDRE systems are used simultaneously; critical functionality may be provided by time-triggered components, while non-critical functionality may be provided by event-driven components.

The *composition* of time- and event-driven systems is a significant challenge that may have significant impact on the design of modern DRE systems. Improved interaction between event-driven and time-triggered components may have many positive effects:

- *Improved fault-tolerance:* event-driven systems may take over some of the functionalities of mission-critical time-triggered systems in case of a failure. For example, microchips in a multi-media or navigation system may take over critical functionalities of a car in case of an emergency.

- *Dynamic adaptation based on new information:* ad-hoc mobile networks between pedestrians or cars may provide information that can be used for adaptation. For example, in case of an icy road or accident, information may be transmitted to cars that may be used to configure the engine, brakes, or suspension for the heightened risk.

The composition of time- and event-driven systems may provide designers with the option to design systems that have the advantages of both AEDRE and TTDRE systems, thus leading to greater design freedom and flexibility. Developing analysis methods for AEDRE systems, however, remains a key challenge.

This dissertation focuses on the *model-based analysis of AEDRE* systems. AEDRE systems are not only more widely used that TTDRE systems, but can also provide a common semantic domain for the analysis of TTDRE systems. Events can also express time, and can capture the composition of time- and event-driven DRE systems. By providing methods for the formal analysis of AEDRE systems, we bridge the gap between time- and event-driven DRE systems, enabling the use of compositional analysis methods.

1.2 Model-based Analysis of Distributed Real-time Embedded Systems

Developing a DRE system that satisfies multiple QoS properties is a complex constraint-satisfaction problem. To ensure optimal QoS support in practical applications, devel-

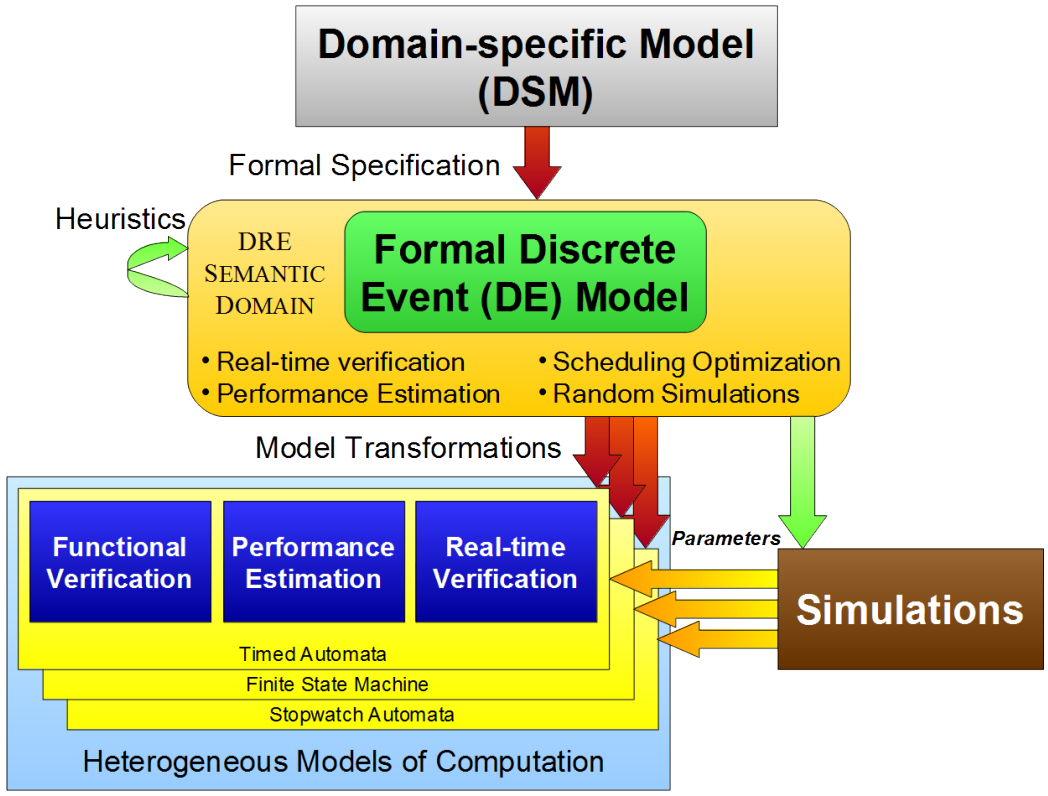

Figure 1.1: Model-based Analysis of Distributed Real-time Embedded Systems

opers often face hard or even undecidable problems. Despite recent advances in embedded systems' analysis and abstraction techniques the generic verification of production-scale DRE systems is largely unsolved.

We propose a model-based analysis method for the verification and dynamic performance estimation of DRE systems using the concept of platform-based design [10] and *Model-integrated Computing (MIC)* [94], as shown in Figure 1.1. *Domain-specific Modeling Languages (DSMLs)* play an essential role in the design and analysis process, and can also be used to synthesize executable code, simulations, or documentation.

Designing an application that satisfies multiple QoS properties is a multi-step process in which the *Domain-specific Model (DSM)* is continually evolved until the underlying analysis frameworks verifies that the QoS properties are satisfied. This evolution is performed by DRE system designers based on the feedback from the analysis frame-

work. The goal of the analysis is to aid DRE system development by choosing feasible design alternatives.

The design flow starts with the DSM, a high-level specification that captures key properties of the design, such as its structure, behavior, environment, and key constraints that it has to satisfy. The domain-specific model can be expressed in several ways, such as an *Architecture Description Language (ADL)*, textual specification, timing diagrams, meta-modeling, or other visual methods.

The DSM is then mapped to a formal *Discrete Event (DE)* semantic domain, that provides an executable formalism for the analysis. We use model transformations to specify the link between the DSM, and the formal semantic domain. The translation abstracts out the necessary details from the DSM; the formal models usually capture key properties of the system at higher-level abstractions than the DSM itself. As with simulations, the abstraction influences the complexity of the analysis as well as its precision. If the analysis model is too abstract, results may become inaccurate, if it is too complex the analysis will likely hit the state space explosion problem. Finding the right abstraction is the key for the successful model-based analysis of DRE designs.

When designing applications that require support for multiple QoS properties, the analysis often requires multiple tools for the analysis. For example, one tool can be used to verify real-time properties, while simulations can be used to predict the overall power consumption of the system. The DRE Semantic Domain provides a common semantic domain that can capture multiple QoS properties of a generic class of DRE systems. The DRE Semantic Domain provides the basis for the analysis of the DSM.

Simulators are often integrated into the analysis tools. When the model checking tool finds an unsatisfied property the simulator can be used to simulate the execution trace that yields this undesired behavior. For example, a model checker may find system deadlocks by checking the formal model of computation generated from the

domain-specific application model.

We use the profiling information from simulations to annotate the models used by formal methods. Common parameters obtained by simulations include execution times captured as intervals, priorities, scheduling, the size of messages sent between components etc. Since a single simulation usually consists of several hundreds/thousands transactions, parameters for components can be usually estimated with good accuracy, even when a small number of test cases are used. The major source of uncertainty arises from the concurrent processing and non-deterministic execution times, that the formal models inherently capture.

By utilizing an abstract formal representation of the system for simulations, significant simulation speedups can be achieved for dynamic analysis. As execution parameters are obtained by simulations, the accuracy of the formal analysis is comparable to simulation results, while providing better coverage. The formal executable model can be mapped to heterogeneous *Models of Computation (MoCs)* for formal analysis. We utilize the model checking [31] approach for the analysis of the formal semantic domain to address three key problems; *(1) functional verification*, *(2) performance estimation*, and *(3) real-time verification*.

This approach allows to find the formalism that provides the best scalability with the required precision. Moreover, unlike pure model checking methods, the proposed approach can provide partial simulation results in cases where exhaustive analysis is infeasible. Therefore, the combination of simulations and model checking improves the existing practice of random simulations and can provide partial results when model checking methods fail due to the state space explosion problem.

The three challenges addressed by our proposed analysis framework – *functional verification, performance estimation, real-time verification* – require different approaches, MoCs, abstractions, and tools for formal analysis. We pick the MoC and abstraction level for each analysis method that provides the most efficient analysis.

The proposed cross-abstraction real-time analysis framework provides a way to utilize the right level of abstraction for each analysis method.

1.3 Key Contributions of this Dissertation

This dissertation proposes a model-based design methodology to address three major challenges in the formal analysis of DRE systems: *(1) functional verification* – to ensure that the system will not be trapped in a deadlock or livelock state, *(2) performance estimation* – in order to obtain tight bounds on the worst case performance of the DRE design, and *(3) verification of real-time properties* – to prove whether individual deadlines for tasks and performance estimates hold for the DRE design. The novelty of our approach lies in *(1) combining formal methods and symbolic simulations* for the system-level evaluation of DRE designs early in the design flow, and *(2) utilizing multiple abstractions* to trade off analysis accuracy and scalability. The key technical contributions of this dissertation shown in Figure 1.2 are as follows:

- **Definition of a formal semantic domain for AEDRE systems:** we describe the DRE Semantic Domain – a formal executable domain for the analysis of DRE systems. We review common methods to specify semantics in Chapter 3, then describe our approach for the modeling of DRE systems by meta-modeling, and introduce the *Analysis Language for Distributed, Embedded, and Real-time Systems (*Alderis*)* DSML. This work is described in Chapter 4.

- **A model checking method for the real-time verification of non-preemptive AEDRE systems by** *Timed Automata (TA)***:** we specify TA models for the compositional analysis of DRE systems. We describe the refinement-based transformation process that allows the analysis of Alderis models by TA model checking methods in Chapter 5.

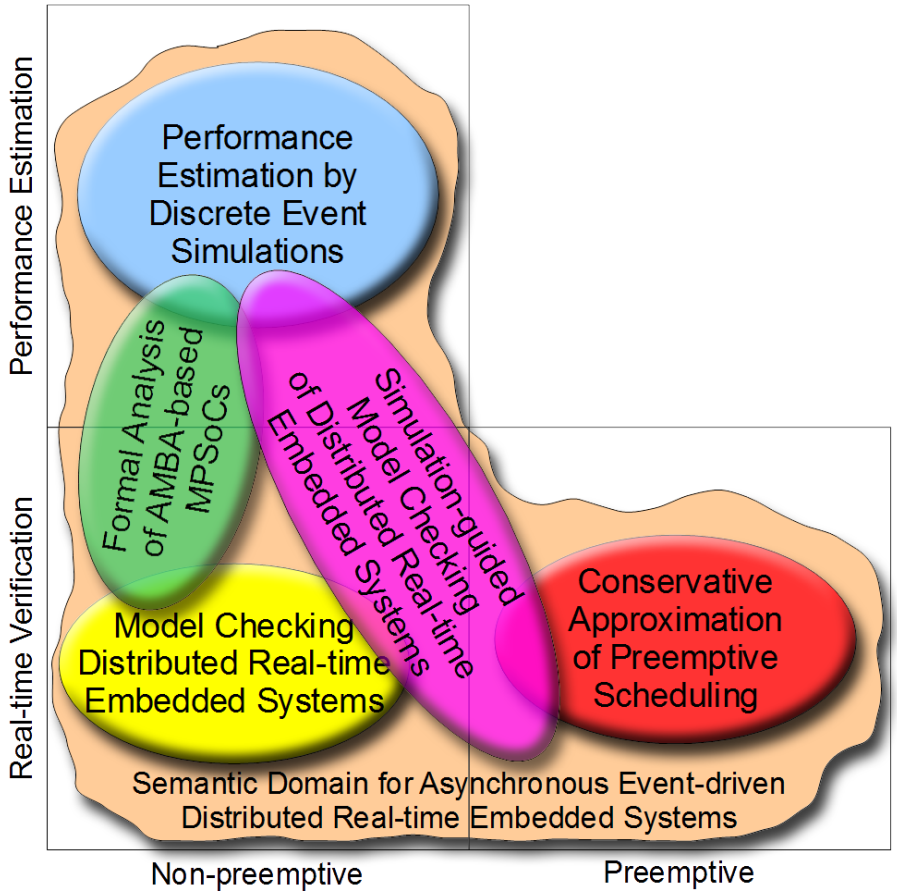

Figure 1.2: Key Technical Contributions of this Dissertation

- **A performance estimation method for AEDRE systems by *Discrete Event Simulations (DES)*:** we describe a novel DES-based performance estimation method for DRE systems. The DES-based method is applicable to large-scale DRE systems as it is based on repetitive simulations of the model, and therefore does not suffer from memory consumption limits. Moreover, it can provide partial results in case the models are too large for exhaustive analysis. This work is described in Chapter 6.

- **A conservative approximation method for the verification of preemptive AEDRE systems by TA:** In this dissertation, we refer to preemptive AEDRE systems as *Preemptive Event-driven Asynchronous Real-time Systems*

with Execution Intervals (PEARSE). Preemptable tasks can be expressed using *Stopwatch Automata (SA)* [77]. The reachability problem on the composition of SA as a task graph is undecidable in general, since it can be mapped to the halting problem [55]. The schedulability of preemptive multi-processor systems is undecidable using TA in the generic case [59], as *Timed Automata (TA)* cannot directly model stopwatches. Therefore, model checking PEARSE is a challenging problem that is undecidable in the generic case. Chapter 7 presents a novel conservative approximation method for the practical model checking of PEARSE. To the best of our knowledge, the proposed method is *the first decidable – and therefore practically applicable – method for the real-time verification of AEDRE systems designed as Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)*.

Utilizing the key technical contributions of this dissertation, we apply the analysis methods to the following problem domains:

- **Cross-abstraction verification and performance estimation of MPSoCs:** While MPSoC designs themselves can be viewed as DRE systems, the communication subsystem in MPSoC designs has a major impact on both design and analysis. Unlike software-intensive AEDRE systems that communicate over packet-switched networks, MPSoC designs often utilize complex bus matrix architectures, where access to the bus is managed by an *arbiter* (or several arbiters). Bus protocols and arbitration policies have a major impact on key design parameters such as throughput and delays, and present new challenges for functional verification. In particular, deadlock-freedom and livelock-freedom is not guaranteed by bus protocols, but is a key requirement for designers.

  This dissertation introduces an approach for the combination of transaction-level simulations and model checking for formal MPSoC performance estimation

and real-time analysis in Chapter 8. We then extend the real-time analysis to bus matrix MPSoC designs. Chapter 9 describes how methods for the analysis of AEDRE systems can be adapted to MPSoC designs utilizing fully connected bus matrix interconnects, and how point arbitration policies can be expressed by the non-preemptive scheduling of task graphs.

- **The open-source** *Distributed Real-time Embedded Analysis Method (*DREAM*)* **framework for the simulation-guided verification and performance estimation of AEDRE systems:** DREAM is an open-source tool and method for the model-based real-time verification and performance estimation of DRE systems, that implements the high-level design flow shown in Figure 1.1. It implements the key technical contributions of this dissertation; *(1) Timed Automata (TA)*-based real-time verification of non-preemptive AEDRE systems using the UPPAAL [26] and Verimag IF [15] model checkers, *(2)* DES-based method for performance estimation, *(3)* conservative approximation method for the verification of PEARSE. DREAM models may be specified using ALDERIS, a modeling language based on the DRE SEMANTIC DOMAIN. The DREAM project focuses on the practical application of formal analysis methods to automate the verification, development, configuration, and integration of AEDRE systems. Chapter 10 describes the implementation of the open-source DREAM tool. DREAM is available for download at `http://dre.sourceforge.net`.

Chapter 2

Related Work

A key contribution of this dissertation is the *integration* of various analysis methods, tools, and abstractions for the model-based verification and performance estimation of *Distributed Real-time Embedded (DRE)* systems. In this chapter we describe how the proposed model-based analysis framework improves on related work. As key technical contributions of this dissertation consider multiple aspects of DRE system design, we partition existing work into sections of relevant work.

2.1 Model-based Design and Analysis of Distributed Real-time Embedded Systems

This section compares the proposed model-based analysis framework for DRE systems with existing modeling approaches.

The *Society of Automotive Engineers: Architecture Analysis & Design Language (SAE AADL)* [34] is an international standard *Architecture Description Language (ADL)* originally developed for complex avionics applications. SAE AADL is a successor of the *Honeywell MetaH* toolset [99], a commercially available domain-specific ADL for developing reliable, real-time multi-processor avionics system architectures. More recently, the *Open Source AADL Tool Environment (OSATE)* tool was introduced, that is an open-source design framework implemented as a set of *Eclipse* plugins. *Eclipse* is a widely used *Integrated Development Environment (IDE)* by *IBM*. In the design framework presented in this dissertation, SAE AADL could be viewed as the domain-specific model shown in Figure 1.1. The *Analysis Language for Distributed, Embedded, and Real-time Systems (*Alderis*) Domain-specific Modeling*

Language (DSML) presented in Chapter 4 is less detailed than SAE AADL, but has formal semantics, allowing the real-time verification of the models.

Simulink and *Real-time Workshop* from *The Mathworks* provides a complex IDE for the design of real-time embedded systems. The tool can generate *ISO C/C++* code from models for multiple platforms. Single- and multi-rate, as well as asynchronous real-time systems are supported. Simulink is a de facto standard for the simulation-based evaluation of DRE systems. This dissertation considers the problem of combining simulations and formal methods for real-time analysis.

PTOLEMY II [63] is a complex design framework that composes heterogeneous *Models of Computation (MoCs)* for the evaluation of embedded systems, including continuous-time, discrete event, synchronous data flow, among others. PTOLEMY II focuses on the modeling, simulation, and design of concurrent, real-time, embedded systems, and can perform simulations of the composed models for evaluation. PTOLEMY II focuses on deterministic systems, and aims to provide an alternative to abstractions built on non-deterministic threads [62]. Recently, the *Programming Temporally Integrated Distributed Embedded Systems (PTIDES)* programming model [105] was introduced in PTOLEMY II for the analysis of DRE systems based on a *Discrete Event (DE)* model. PTIDES provides a programming model that inherently captures time as a key design constraint, and enforces the deterministic execution of concurrent real-time systems. Implementing deterministic distributed systems, however, is a challenging task that requires fundamental changes in hardware and software design. In contrast, this dissertation focuses on improving analysis methods in practical DRE systems, that are often non-deterministic. Communication delays, race conditions, resource congestions, caching, and even varying input data all contribute to non-deterministic execution times and delays.

GIOTTO [44] utilizes the notion of *Logical Execution Time (LET)* to enforce deterministic execution times in real-time embedded systems. GIOTTO is a time-

triggered language, and advocates the use of *Time-triggered Distributed Real-time Embedded (TTDRE)* systems. LET specifies a task's execution time as a constant $c \in \mathbb{R}_{\geq 0}$. The activation time of the task $t_a \in \mathbb{R}_{\geq 0}$ is tied to a specific valuation of the global clock, and the output of the task is available at $t_a + c$ time. By enforcing the constraint that none of the task's dependents may start their execution before $t_a + c$, we ensure that all the task's dependents have access to the correct output. Asynchronous activation is not allowed; should the task finish earlier than $t_a + c$, its dependents still have to wait until they are activated by the global clock at time $t_x \in \mathbb{R}_{\geq 0}$, $t_a + c \leq t_x$. While this approach has the undesired effect that it may result in tasks idling simply because they have not been activated yet, it also has the advantage that it turns a task's execution time to constant. Constant c acts as the logical execution time; regardless of whether the task finished its execution faster or slower, designers are able to treat its execution as constant, resulting in deterministic time behavior, better predictability and greatly improved analysis performance. In contrast, this dissertation focuses on the analysis of *Asynchronous Event-driven Distributed Real-time Embedded (AEDRE)* systems using multiple abstractions and analysis methods.

Model-integrated Computing (MIC) [94] is an approach that proposes the use of DSMLs for the specification, verification and integration of embedded systems using the concept of meta-modeling, and the *Generic Modeling Environment (GME)*. MIC advocates a model-driven development paradigm for the design of complex software systems [11]. The model-based design framework proposed in this dissertation is based on the concept of MIC, and the ALDERIS DSML.

The *Component Synthesis using Model Integrated Computing (CoSMIC)* [38] toolkit is an integrated collection of DSMLs that support the development, configuration, deployment, and evaluation of DRE systems, using the GME tool. The CoSMIC tools can be used to specify requirements, compose DRE systems and their supporting in-

frastructure from the appropriate set of middleware components. In contrast, this dissertation focuses on the analysis of AEDRE systems using multiple abstractions and analysis methods.

The *Virginia Embedded Systems Toolkit (VEST)* [92] is a framework designed for the reliable and configurable composition and analysis of component-based embedded systems from *Commercial off-the-shelf (COTS)* libraries. The modeling environment uses the GME tool. VEST applies key checks and analysis but does not support formal proof of correctness. In contrast, the model-based design framework described in this dissertation supports several formal analysis methods.

CALM and CADENA [24] provide an integrated environment for building and analyzing software-intensive DRE systems built on the *CORBA Component Model (CCM)*, or *Enterprise JavaBeans (EJB)*. Main functionality of CALM and CADENA include software modeling, dependency analysis, component integration, code generation, and real-time analysis. The emphasis of verification in CADENA is on software logical properties. In contrast, this dissertation focuses on the analysis of AEDRE systems using multiple abstractions and analysis methods.

SYSWEAVER [27] (previously referred to as TIME WEAVER and GEODESIC) is a component-based framework that supports the reusability of components across systems with different para-functional requirements. It supports code generation, as well as automated analysis. SYSWEAVER also builds a response chain model [57] of the system to verify timing properties. This model is used by real-time analysis tools, such as TIMEWIZ, to build a task set that can be analyzed using *Rate Monotonic Analysis (RMA)*. In contrast, this dissertation focuses on the analysis of AEDRE systems using multiple abstractions and analysis methods.

The *Automatic Integration of Reusable Embedded Software (AIRES)* [102, 39] tool extracts system-level dependency information from the application models, including event- and invocation-dependencies, and constructs port- and component-level

dependency graphs. Various polynomial-time analysis tasks are supported such as checking for dependency cycles as well as forward/backward slicing to isolate relevant components. It performs real-time analysis using RMA techniques. In contrast, this dissertation focuses on the analysis of AEDRE systems using multiple abstractions and analysis methods.

Graph transformations [89] were proposed for the implementation of model transformations in model-based analysis frameworks. In earlier work we considered the applicability of *Graph Rewriting and Transformation (*GREAT*)* [3] for model-based verification [67]. As the graph transformation rules get more complex, proving the property-preserving nature of graph transformations becomes a challenging academic exercise. This dissertation does not focus on graph transformations, as we define the semantic mapping manually by simple refinement rules in Chapter 4.

The *VIsual Automated model TRAnsformations (VIATRA)* model-based design framework was described in [97]. The authors utilize graph transformation to transform *Unified Modeling Language (UML)* models to a formal representation driving the verification of functional properties by model checking. In contrast, this dissertation targets the analysis of real-time properties in DRE systems specified as DSMLs.

2.2 Real-time Analysis of Distributed Real-time Embedded Systems

Real-time properties have been widely studied in the context of DRE systems. This section compares related work with the real-time analysis methods developed as part of this dissertation.

2.2.1 Classic Scheduling Theory

Time-triggered approaches [58] are becoming common in mission-critical systems. Classic schedulability analysis methods such as RMA and *Earliest Deadline First (EDF)* provide sufficient conditions for schedulability [64, 57, 20] in predictable periodic real-time systems such as TTDRE systems, but they do not address the dynamics of events, race conditions, and the non-deterministic execution order of tasks. These methods are typically overly conservative or unapplicable for the analysis of AEDRE systems.

A holistic method is proposed for distributed preemptive scheduling [96] that provides sufficient conditions for schedulability using a *Time Division Multiple Access (TDMA)* communication bus. The model is general enough to represent a handful of real-time systems but is not suitable for AEDRE systems with asynchronous event-driven communication.

Mathematical approaches that consider task dependencies to check real-time constraints are usually extensions of the job shop scheduling problem, which is NP-complete [14]. The *Resource Constrained Project Scheduling Problem (RCPSP)* is a prominent extension for which the fastest solutions use genetic algorithms and meta-heuristics [42]. The RCPSP, however, does not focus on providing guarantees on real-time properties.

2.2.2 Model Checking Non-preemptive Scheduling

Model checking provides alternative methods for the dynamic analysis of DRE systems. *Timed Automata (TA)* [6] was proposed as a semantic domain for real-time verification in dense (continuous) time. TA inherently captures the asynchronous event-based triggering of AEDRE systems.

A generic non-preemptive task scheduling model based on TA was proposed in [32]. The scheduling model is based on a ready queue, and schedulability is verified using the UPPAAL [26] tool. The analysis method described in Chapter 5 extends this

idea to AEDRE systems, capturing key components, including event channels, and distributed platforms. The open-source *Distributed Real-time Embedded Analysis Method (*DREAM*)* tool utilizes the UPPAAL model checker for real-time analysis, among others.

A promising way to address TA composition using priorities is presented in the Verimag IF toolset [15]. IF defines priorities between transitions in TA models to improve the composability of TA models. The number of priorities, however, is fixed, and therefore requires extra effort in modeling scheduling policies. The open-source DREAM tool utilizes the Verimag IF model checker for real-time analysis, among others.

A TA-based approach for the thread-level analysis of DRE systems is presented in [98]. The authors present a reusable library of formal models to capture timing and semantics in *The ACE ORB (TAO)* [90]. In contrast, this dissertation targets the modeling and analysis of DRE systems at the task-level.

We have presented a formal method for deciding the schedulability of non-preemptive real-time *Common Object Request Broker Architecture (CORBA)* applications using TA model checking methods in [66]. We extended this early work with event channels, and presented a comprehensive non-preemptive scheduling model for fixed-priority DRE systems in [67]. Chapter 5 is based on this research, with some minor improvements designed to increase scalability.

2.2.3 Model Checking Preemptive Scheduling

Stopwatch Automata (SA) [77] were proposed as a model of computation that can express preemptable tasks in asynchronous event-driven systems. The reachability problem on the composition of SA as a task graph is undecidable in general, since it can be mapped to the halting problem [55]. The schedulability of preemptive AEDRE systems is undecidable using TA in the generic case [59], as TA cannot directly model

SA. More specifically, the verification of preemptive scheduling using TA is undecidable if the following conditions are met: *(1)* tasks use event-based asynchronous triggering (*i.e.* a target task starts whenever its source finishes) on a distributed platform, *(2)* execution times are specified as continuous-time intervals, *(3)* preemptions may occur anytime within the continuous-time execution interval. In this dissertation, we refer to systems that satisfy these three conditions as *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)*.

Therefore, DRE system designers typically restrict the simultaneous occurrence of the three conditions used to define PEARSE systems to allow real-time analysis.

(1) Restricting asynchronous triggering: commonly used approach, resulting in TTDRE systems. Classic schedulability analysis methods described in Section 2.2.1 provide sufficient conditions for schedulability in predictable periodic real-time systems, but they do not address the dynamics of events, race conditions, and the nondeterministic execution order of tasks. These methods are typically overly conservative or unapplicable for PEARSE.

A method that combines scheduling theory with TA model checking for fixed-priority scheduling was proposed in [16]. Even though TA can express asynchronous event-driven communication, the authors focus on the analysis of TTDRE systems, as the approach is based on the periodic task model. The scheduling algorithm is not modeled, rather the authors assume that RMA analysis is sufficient, which is only true in TTDRE systems, as shown in Chapter 7.

A generic periodic task and scheduling model for preemptive systems based on TA was introduced in [35]. This work uses the idea of discretizing TA clocks for the modeling of preemptive systems, but restricts asynchronous triggering, and is therefore only applicable to periodic task models.

(2) Restricting continuous-time execution intervals: synchronous languages [13] propose a common mathematical model for synchronous systems with determinis-

tic concurrency, but do not address non-deterministic execution times, arrival times, and asynchronous event-triggering typical in PEARSE systems. In Giotto [44], deterministic execution times are enforced to facilitate formal analysis, based on the concept of logical time.

A method based on stopwatch automata [77] was proposed for the job-shop scheduling of preemptive systems [2]. This method is applicable for the real-time verification of PEARSE models, where execution times are given as constants. This restriction turns the reachability analysis decidable, as a single simulation trace can verify this model. However, this approach is too coarse for practical analysis, as execution times are often non-deterministic in AEDRE systems.

(3) Restricting preemptions: we discussed methods for the analysis of non-preemptive DRE systems in Section 2.2.2. Our earlier approach for modeling preemptive systems using timed automata [65] restricts preemptions to occur at discrete time steps.

The 3 restrictions described above involve cost, performance, and energy consumption overheads, as deterministic behavior has to be enforced on a distributed platform. These costs can often be justified only in the context of mission-critical systems. In most DRE systems, including consumer electronics, PEARSE models are commonly used. The industry practice for the analysis of PEARSE consists mostly of directed testing and random simulations. Although this approach may be helpful, it can only show the presence of timing violations, not their absence.

In Section 7 we propose a conservative approximation method that allows the simultaneous use of *(1)* asynchronous event-based triggering, *(2)* continuous-time execution intervals, and *(3)* preemptions that may happen anytime within a task's execution interval. To the best of our knowledge, the only alternative for the real-time analysis of systems that use the above three conditions (PEARSE systems) is stopwatch (hybrid) automata model checking. Reachability analysis is undecidable in general

on SA, therefore none of the methods described below are guaranteed to terminate on practical problems. Moreover, scalability issues constrain the applicability of the method to small-scale problems.

The problem of verifying real-time properties in PEARSE have not been studied in detail, and very few works target this problem directly. An approach for the response time analysis of DRE systems was presented in [17]. the authors present a language for the specification of distributed real-time systems, and describe an approach to transform the model to hybrid automata for reachability analysis using the HyTech [43] *Hybrid Automata (HA)* [45] model checker. Since reachability analysis on HA is undecidable in the generic case, this method is not guaranteed to terminate, and is not practical for the analysis of PEARSE.

Preliminary experiments for approximating stopwatch reachability analysis using timed automata were described in [22], but the method is not guaranteed to terminate, since reachability analysis is undecidable on stopwatch automata in the general case.

An approach for the verification of PEARSE using linear HA was presented in [100]. The author proposes a generic task model for preemptable tasks as utilized in the Honeywell MetaH toolset. The underlying analysis, however, is based on HA reachability analysis that is not guaranteed to terminate. Our work presented in [71], and described in Chapter 7 considers the same issues, but provides a conservative approximation method for deciding schedulability. This method is guaranteed to terminate, and has better scalability than existing methods, as shown in Chapter 7.

2.3 Performance Analysis

2.3.1 Static Performance Analysis Methods

An approach for the performance estimation of DRE systems was presented in [104]. The authors model DRE systems by periodic multi-rate tasks mapped to distributed

processing elements, and target the performance estimation problem by providing bounds on delays. The periodic task model, however, restricts the approach to TTDRE systems.

A generic, component-based formal framework for the scheduling analysis and formal performance evaluation of platform-based embedded systems was proposed in [86]. The authors utilize event streams to model communication characteristics of tasks.

SymTA/S [85] is a formal analysis tool that applies methods from scheduling theory and symbolic simulations for the performance analysis of complex heterogeneous *Multi-processor System-on-Chip (MPSoC)*. This approach can provide bounds on end-to-end latencies, bus and processor utilization, and worst-case scheduling scenarios.

Modular Performance Analysis [33] is an approach to characterize DRE systems merely by describing incoming and outgoing event rates, message sizes, and execution times. Resources and the distributed execution platform is defined in similar terms, and *Real-Time Calculus* is then used to compute upper and lower bounds of the system performance.

Although all static analysis methods provide scalable solutions for performance evaluation they cannot model dynamic effects, such as varying delays and race conditions, as they do not capture the flow of data, and are less accurate than dynamic estimation methods. Communication in embedded systems is often non-deterministic, data-dependent, and hard to model as well-formed event streams.

In contrast, the analysis methods in this dissertation explicitly capture dependencies, and the asynchronous event-based communication of AEDRE systems. This approach captures dynamic effects, such as varying delays and race conditions in distributed systems, and results in more accurate performance analysis at the price of being computationally more intensive.

2.3.2 Dynamic Performance Analysis Methods

Simulations are the preferred and widely accepted way to evaluate the performance of DRE system designs in the industry today. A simulation-based design space exploration method, however, has several disadvantages. Developing the models for a design alternative may take weeks or months therefore only a handful of alternatives may be practically analyzed given the short product development cycles. Moreover, designers typically notice performance issues late in the design cycle – after the simulation model is complete – therefore addressing changes can be rather time-consuming and costly.

Register-Transfer Level (RTL) languages such as *VHDL* [49] and *Verilog* [50] are classic hardware description languages that target hardware specification at low-level abstractions providing a high precision, synthesizable platform for hardware development. The low-level abstraction, however, results in slow simulation speeds unsuitable for the analysis of complex MPSoCs.

Due to the increase in MPSoC design complexity as well as the decrease in the time to market window, today's designers are turning to *transaction-level* modeling languages such as *SystemC* [80] and *SystemVerilog* [51] to perform early design exploration and hardware-software co-design in order to shorten the design cycle. Transaction-level modeling focuses on the *interactions* between systems components, such as bus transfers, interrupts or signals, rather than on gates or registers. Transaction-level languages employ higher-level abstractions than RTL languages and are often not synthesizable.

A semi-formal simulation-based performance evaluation method for MPSoCs was proposed in [60]. The authors represent execution traces as symbolic graphs for performance analysis, annotated with execution times obtained by simulating individual components of the system. Although the approaches described in [60] improves simulation speed by utilizing symbolic representations of execution traces, the quality of

results depends on the ad-hoc selection of test vectors.

This dissertation considers the problem of combining simulations and formal methods for real-time analysis. Simulations in our approach are used to obtain execution intervals, that we use to annotate the formal models for design space exploration. Our symbolic model captures all possible execution traces of the system, not just one execution trace. This is a more accurate model for AEDRE systems, where execution times are rarely constant. Moreover, we formalize our method for obtaining test vectors based on the DE model, that provide better coverage than random simulations.

2.3.3 Model checking methods

Although model checking methods described in Section 2.2.2 and Section 2.2.3 provide the means for the formal real-time verification of DRE systems, their practical applicability to the performance evaluation of large-scale DRE systems is limited. The exhaustive verification of large-scale DRE systems is often infeasible in practice, and is often unnecessary, as the performance can usually be estimated accurately with less than perfect coverage. Moreover, most model checkers are based on logics [75, 25, 18] tailored towards yes/no questions which makes formal performance evaluation a tedious process.

Chapter 6 proposes a method based on *Discrete Event Simulations (DES)* for the performance evaluation of large-scale DRE systems. Although the proposed method provides a way for the formal verification of real-time properties, fast symbolic simulations are its main advantage, and is directly applicable to large-scale systems, that cannot be analyzed using an exhaustive state space search. The DES-based analysis increases the coverage by gradually simulating the symbolic models, and can answer complex questions on the symbolic executable models. We show that this approach provides a way for fast design space exploration, and can achieve better coverage in some cases than alternative methods.

2.4 Functional Verification of MPSoCs

A generic method for protocol verification using synchronous protocol automata is presented in [30]. Synchronous protocol automata can be mapped to the *Finite State Machine (FSM)* MoC, and a main contribution of the paper is to show how protocols can be translated to a formal language for functional verification.

A method for the functional verification of the *Peripheral Component Interconnect (PCI)* protocol is described in [23]. The authors model the PCI protocol by the FSM MoC, and use the *Cadence SMV* [56] tool for functional verification. A similar approach is used to verify the *IBM CoreConnect* arbiter in [37].

An early work on applying model checking methods to the *ARM Advanced Microcontroller Bus Architecture Advanced High-speed Bus (AMBA AHB)* protocol was presented in [88], where the authors used FSM models and the SMV tool to uncover an unspecified condition in the AMBA AHB specification. The described case study is due to flawed implementation rather than the protocol itself.

A verification platform for AMBA-ARM7 is presented in [93]. The authors use the SMV tool to prove the functional correctness of the AMBA AHB protocol by checking various properties. The authors do not describe any ambiguities, rather they focus on properties that have turned out to be valid.

A verification platform for AMBA AHB using a combination of model checking and theorem proving is described in [7]. The author extends earlier approaches by considering *both* control and data properties, and describes properties that have proven to be true.

We have presented a method for the functional verification and formal performance evaluation of AMBA AHB-based MPSoC designs in [74, 73], and showed how model checking may improve the test coverage of transaction-level simulations for performance evaluation. We also presented an ambiguous case in the AMBA AHB specification that might lead to flawed implementations.

Our results do not imply that the AMBA AHB protocol is incorrect, and neither does it imply that the works described in [93] [7] are invalid. Rather, it shows that ambiguities in protocol specifications are often manually resolved on a case-by-case basis when implementations or formal models are created and only the correctness of such models can be shown rather the correctness of the specification itself. The main reason for this is the ambiguity of natural languages that should be resolved by future designers by providing a formal specification for their protocol.

We presented an approach for the cross-abstraction analysis of MPSoC designs based on AMBA AHB *bus matrix (crossbar switch)* interconnects in [72]. This work applies the DES-based performance estimation method to complex MPSoC designs, and is described in detail in Chapter 8.

Chapter 3

Specifying Semantics

This chapter reviews some of the key concepts utilized in this dissertation. Section 3.1 reviews the specification of *Domain-specific Modeling Languages (DSMLs)*, Section 3.2 gives an overview of semantics, and how it is used to assign meaning to DSMLs. Section 3.3 describes how meta-modeling can be used to specify DSMLs.

3.1 Domain-specific Modeling Language

DSMLs are modeling languages tailored towards a specific problem domain. DSMLs play an essential role in the design and analysis process, and can also be used to synthesize executable code, simulations, or documentation. This approach is rather different from mainstream modeling efforts that focus on specifying a large and generic modeling language to be used in a wide range of applications, such as *Unified Modeling Language (UML)* [54]. DSMLs in our approach are defined using *meta-modeling* [94] therefore the designer has the option of defining languages that have well defined semantics and are a good fit for a problem domain. Large-scale systems that involve several application domains are modeled as a *composition* of DSMLs. We believe that defining semantics to smaller modeling languages and their composition is more likely to succeed than to define it for a large generic modeling language. Figure 3.1 illustrates how a DSML is defined:

- **Concrete syntax (C):** the concrete syntax is concerned with representation; it specifies how the DSML is represented in terms if actual characters, letters, or other visual constructs. More specifically, the concrete syntax assigns a concrete representation to elements of a DSML. In the case of a natural language, the

concrete syntax specifies not the alphabet itself, but rather how all the characters in the alphabet are represented. In the case of a programming language, the concrete syntax specifies the representation of the source code.

- **Abstract syntax (A):** the abstract syntax is concerned with structure, and defines what the elements in the concrete syntax represent. The concrete syntax is mapped to the abstract syntax by the (M_C) concrete syntax mapping. The syntax is abstract in the sense that it does not capture every detail that the concrete syntax captures.

  In the case of a natural language, the abstract syntax specifies the alphabet, and how words can be formed by streams of letters. The concrete syntax mapping specifies which letter corresponds to a specific representation. For example, the 'A', '$\mathtt{A}$', 'A', '$\mathcal{A}$', characters all represent the letter 'A' in the alphabet, and we subconsciously perform the concrete syntax mapping as we read the text. The abstract syntax also specifies that 'fimews', 'cat', 'yawon' are character streams that represent words, but is not concerned about the meaning of the words themselves. In the case of a programming language, the abstract syntax represents the abstract syntactic representation of source code. An *abstract syntax tree* is commonly used, where nodes represent various constructs in the source code.

- **Semantics (S):** the semantics of a language refers to "meaning", and is concerned about the meaning of words/sentences/models constructed using the abstract syntax. The *semantic domain* refers to elements with meaning, and the semantics of a DSML is specified by the semantic mapping (M_S) from the abstract syntax to the semantic domain. Semantics plays an important role in model-based design and analysis, and we give a brief overview on it in Section 3.2.

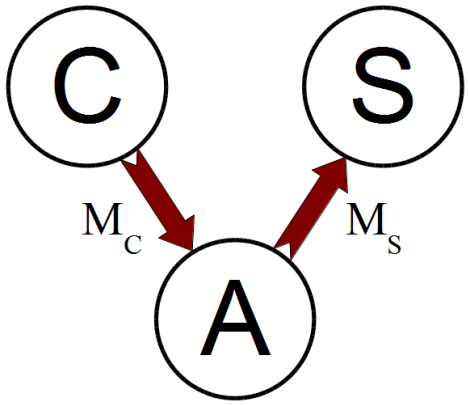

Figure 3.1: Domain-specific Modeling Language

3.2 The Semantics of "Semantics"

Semantics is a broad concept, that is hard to grasp at first. A good overview of what semantics is, and what it is not is presented in [40]. In this Section we give a short overview, and describe major approaches for defining formal semantics.

Semantics can be formal or informal. Most natural languages have informal semantics, that may lead to ambiguities. For example, a fast car on the street may not be a fast car on the race track. Asking for a big glass of beer in the US will likely be less than half liter, while in Europe it may be as big as 1 liter.

To enable rigorous analysis, ambiguities need to be avoided, which leads us to formal semantics. Formal semantics of a language aim to specify the mathematical meaning of syntactically well-formed sentences in the given language. Defining the formal semantics of a language is a prerequisite to any formal analysis. However, the majority of languages does not have well-defined formal semantics. Commonly used programming languages such as *C++* and *Java* also lack formal semantics, but that does not prevent them from being used to implement programs.

3.2.1 Semantic Domain

The semantic domain refers to rigorous mathematical concepts with unambiguous meaning. The semantics or "meaning" of a language is specified by the semantic mapping; a mapping from its abstract syntax to the semantic domain. By assigning a formal mathematical representation to elements in the abstract syntax of the language, we define what the syntax represents.

There is only one key requirement for the semantic domain: it has to define unambiguous meaning to the abstract syntax. Therefore, the choice of the semantic domain focuses on finding a mathematical representation that provides a good "fit" to represent the abstract syntax. Some widely used choices for a semantic domain include set theory, graph theory, linear algebra, probability theory, *Finite State Machines (FSMs)*, Petri-nets, *Timed Automata (TA)*, various classes of *Hybrid Automata (HA)*. By specifying the mapping from the abstract syntax to a formal semantic domain, we assign formal semantics to a language (such as a DSML). Note that it is not a requirement for the semantic domain to "look mathematical". Petri-nets, for example, are a visual modeling language, but have rigorous semantics.

3.2.2 Model of Computation

A *Model of Computation (MoC)* specifies a set of allowable operations on a specific domain. The basic components of MoCs are as follows:

- an *abstract machine*,
- the *state* of the abstract machine,
- a *transition function* that maps one state to another, expressing how the state of the machine evolves,
- an *initial*, and a final (accepting) state.

To define the semantics of a MoC, one has to define how its abstract syntax maps to a formal semantic domain. There are several MoCs with formally defined semantics. Some example include FSM, *Discrete Event (DE)*, TA, HA, Petri-nets etc.

If a MoC has formal semantics, then it can be used as a semantic domain itself. By mapping a language to a MoC with formal semantics, the semantic mapping is specified, and thus we define semantics for the language.

3.2.3 Structural Semantics

Structural semantics specifies the structure of the language, the structure between elements in the semantic domain. While abstract syntax also represents structure in the syntax, structural semantics goes one step farther, and specifies well-formedness rules and various constraints on valid phrases of the language. Structural semantics is incomplete in the sense that it does not specify the full semantics of a language, rather just the subset of semantics relevant to the structure and interrelation of elements in the semantic domain.

3.2.4 Operational Semantics

Operational semantics for a language specifies how program written using the language is interpreted through a sequence of operators. This approach generally utilizes a MoC, and defines the operation of the abstract machine through state transitions. However, operational semantics is not the MoC itself, as it needs to specify the semantic mapping from the abstract syntax to the semantic domain – which in this case is a MoC.

A common method to define operational semantics is to specify *trace semantics*; formalize meaningful sentences that can be constructed using the language, and specify the operation of the abstract machine on the set of all sentences (traces). The final state of the abstract machine provides the output, and defines the set of traces

that the abstract machine accepts.

3.2.5 Denotational Semantics

Denotational semantics is an approach to define the semantics of a language by associating phrases in the language with abstract mathematical objects, such as numbers, tuples, functions etc. The mathematical object is called the "denotation" of the phrase. The collection of the mathematical objects is the semantic domain, and semantics is defined by specifying the mapping from the abstract syntax to the semantic domain.

A key difference between the operational definition of semantics and denotational semantics is that denotational semantics is not concerned about representing the operation of an abstract machine. Due to this fact, it has traditionally been hard to use denotational semantics to define the execution of programs written in imperative programming languages (such as *C, C++, Java* etc.).

3.2.6 Axiomatic Semantics

Axiomatic semantics defines the semantics of a phrase by specifying logical assertions on values and variables, omitting details on how the computation is carried out. Computations are specified by assertions; the initial assertion must evaluate to true before the computation begins, and the final assertion must evaluate to true when the computation finishes. This approach may provide a good match to define the semantics of declarative languages.

Axiomatic semantics – like denotational semantics – do not capture how computations correspond to the sentences in a language, and are therefore hard to apply directly for the analysis of imperative programming languages.

3.3 Specifying Domain-specific Modeling Languages by Meta-modeling

This section describes how the concepts of *Model-integrated Computing (MIC)* [94] can be utilized to define DSMLs. MIC promotes an approach based on meta-modeling for powerful domain-specific abstractions that capture key concepts and concerns of DRE systems, such as their structure, behavior, and environment, as well as the *Quality of Service (QoS)* properties they must satisfy. Experience in developing mission-critical *Distributed Real-time Embedded (DRE)* systems [36] to date has shown that models are essential throughout the DRE system life cycle, including the design, configuration, integration, and analysis phases. MIC adopts the four-layered metamodeling architecture of the *Model-driven Architecture (MDA)* that has been standardized by the *Object Management Group (OMG)*. MIC can be viewed as an enhancement of MDA that is tailored towards system design via DSMLs. This approach provides a practical visual language that can be used by DRE domain experts that are not necessarily familiar with formal methods.

The *Generic Modeling Environment (GME)* [61] is an MIC tool suite that provides a visual interface to simplify the development of DSMLs. GME implements a meta-modeling environment that supports the definition of paradigms, which are type systems that describe the roles and relationships in particular domains. GME has a flexible object-oriented type system that supports inheritance and instantiation of elements of DSMLs. GME allows to specify a modeling language using meta-modeling.

Figure 3.2 illustrates the specification of the *Analysis Language for Distributed, Embedded, and Real-time Systems (*ALDERIS*)* language described in Chapter 4 using the GME meta-model, which is a variation of UML class diagrams. The figure shows a part of the ALDERIS meta-model with its corresponding visual representation in GME. The curvy arrows show how individual modeling elements and their relations

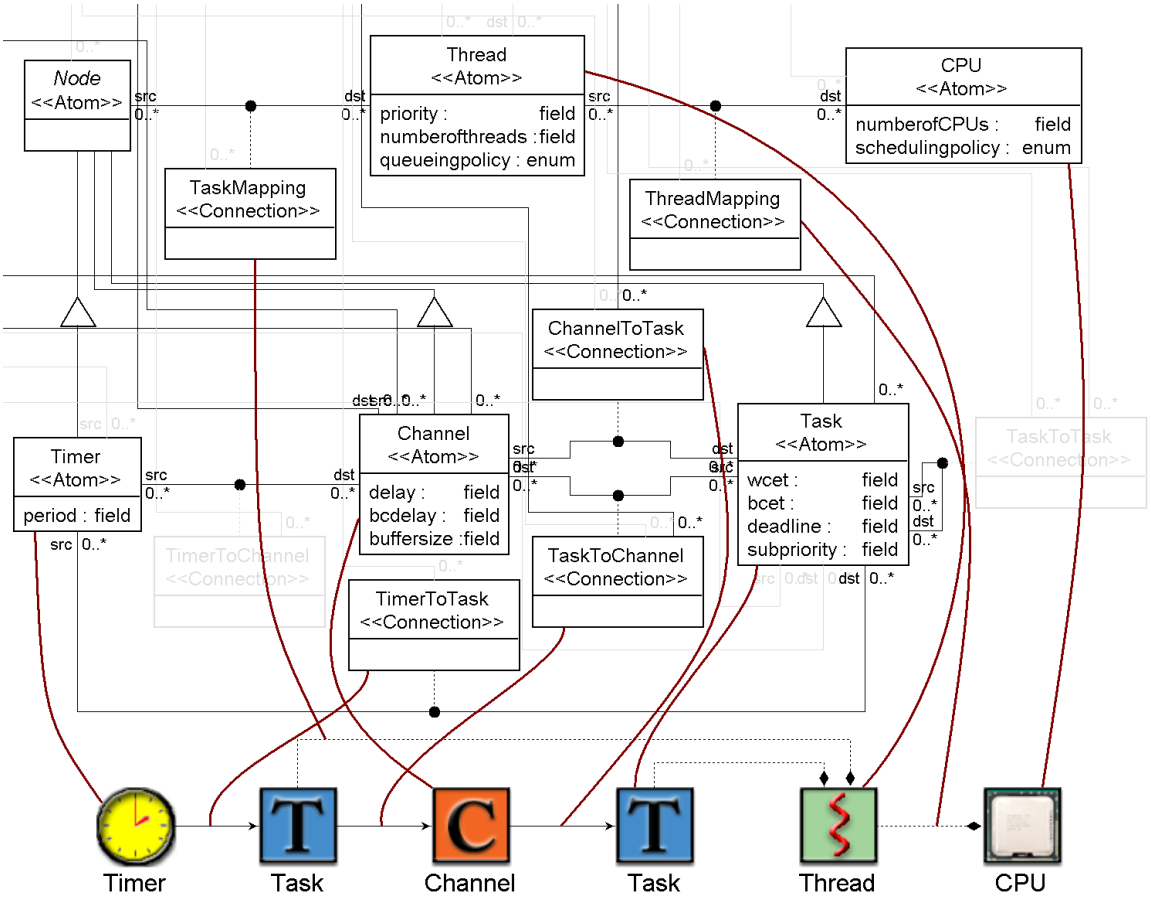

Figure 3.2: Specifying the ALDERIS DSML using Meta-modeling

are defined by different parts of the meta-model. The ALDERIS modeling language is automatically synthesized from the meta-model by the GME tool.

GME provides an integrated constraint definition and enforcement module based on OMG's *Object Constraint Language (OCL)*. OCL enables the definition of rules that must be adhered to by elements of models built using a particular DSML. The GME meta-model can also express various constraints such as cardinality, association attributes etc. The meta-modeling approach can successfully specify the structural semantics of DSMLs in some cases, but does not define semantics in general.

3.4 Stopwatch and Timed Automata

This section reviews the operational semantics of *Stopwatch Automata (SA)* and *Timed Automata (TA)*. Given a finite set of clocks C a *valuation* for the clocks is a function $v : C \rightarrow \mathbb{R}_{\geq 0}$ that assigns a value for each clock from the domain of non-negative real numbers. The valuation of clock $c_i \in C$ is denoted v_i. $\mathcal{B}(\mathcal{C})$ is the set of *clock guards* γ, that are of the form $c_i \diamond \mathbb{N}$, $c_i - c_j \diamond \mathbb{N}$, $\diamond \in \{=, <, >, \leq, \geq\}$. A valuation v satisfies clock guard γ, if for all expressions in γ $v_i \diamond \mathbb{N}$, $v_i - v_j \diamond \mathbb{N}$ is satisfied, respectively. Time progress is captured as clock c_t with a constant rate of 1 ($\dot{v}_t = 1$). The initial value of c_t is denoted v_{t_0}, and $v_{t_0} = 0$.

Definition 3.1 *A stopwatch automaton is a tuple* $SA = (L, l_0, C, E, Act, Inv)$ *consisting of the following components:*

- *a finite set* L *of vertices called* locations*;*
- *the initial location* $l_0 \in L$*;*
- *a finite set* C *of real-valued clocks;*
- *a finite set of edges* $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times Act \times 2^C \times L$ *called transitions, where* $\langle l, \gamma, \alpha, \lambda, l' \rangle \in E$*;*
- *a labeling function* $Act : L \times C \rightarrow \{0, 1\}$*, that defines the rates of clocks* ($c_i \in C$) *in locations as differential functions* $\dot{v}_i = k_l$*, where* k_l *is a constant 0 or 1 in each location;*
- *a labeling function* $Inv : L \rightarrow \mathcal{B}(\mathcal{C})$*, that is called the invariant of the location.*

Definition 3.2 *A state of the stopwatch automaton is defined as a pair* (l, v) *where* $l \in L$ *and* v *is the valuation of the clocks in* C*. The set of states is denoted* S*.*

Definition 3.3 *The* semantics *of a stopwatch automaton* $SA = (L, l_0, C, E,\ Act,\ Inv)$ *is given as a transition system* $T_{SA} = (S, s_0, \rightarrow)$ *where* S *is the set of states,* s_0 *is the initial state, and the* step relation $\rightarrow$ *is the union of the* jump *(discrete) transitions:*

- $(l, v) \xrightarrow{j} (l', v')$ *if* $\exists \langle l, \gamma, \alpha, \lambda, l' \rangle \in E$ *such that* γ *and* $Inv(l')$ *are satisfied and* $v' = \alpha$,

and flow *(continuous) transitions:*

- $(l, v) \xrightarrow{f} (l, v')$ *such that for each* $v'_i \in v'$, $v'_i = v_i + x \cdot Act(l, c_i)$, *where* $x \in \mathbb{R}_+$ *and* $Act(l, c_i)$ *is the labeling function that defines the rate of clock* c_i *as either 0 or 1 as introduced in Definition 3.1* ($\dot{v}_i \in \{0, 1\}$).

A *run* of the SA is a finite or infinite sequence of alternating discrete and continuous transitions of T_{SA}: $\rho : s_0 \xrightarrow{j} s_1 \xrightarrow{f} s_2 \xrightarrow{j} s_3 \ldots$ or $\rho : s_0 \xrightarrow{f} s_1 \xrightarrow{j} s_2 \xrightarrow{f} s_3 \ldots$

Definition 3.4 *A timed automaton is a subclass of stopwatch automaton, where the rates of clocks are set to constant 1* $(\forall l \in L)(\forall c \in C) Act(l, c) = 1$.

Chapter 4

A Formal Semantic Domain for Distributed Real-time Embedded Systems

This chapter describes the *Analysis Language for Distributed, Embedded, and Real-time Systems (*Alderis*) Domain-specific Modeling Language (*dsml*)*, that drives the proposed model-based design framework, and the underlying DRE Semantic Domain used for the analysis. We describe the syntax of the Alderis dsml in Section 4.1 based on set theory. Alderis is defined by meta-modeling using the *Generic Modeling Environment (*gme*)* tool as described in Section 4.2.

The semantics of Alderis is based on the DRE Semantic Domain, a formal framework that specifies elements commonly found in *Distributed Real-time Embedded (*dre*)* systems. We formally specify the DRE Semantic Domain in Section 4.3 based on the *Timed Automata (*ta*) Model of Computation (*MoC*)*. We then describe the DRE Semantic Domain as a *Discrete Event (*de*)* system in Section 4.4. Using the DRE Semantic Domain to specify the operational semantics of the Alderis dsml we define a formal MoC, that we refer to as dre MoC.

The advantage of having both a ta and de representation of Alderis models is the possibility to pick the right formalism for a given problem. For example, the real-time verification method described in Chapter 5 utilizes the ta formalism to show the schedulability of Alderis models, but the performance estimation method described in Chapter 6 builds on the de model for a *Discrete Event Simulation (*des*)*-based performance analysis.

4.1 The ALDERIS Domain-specific Modeling Language

This section describes the abstract syntax of the ALDERIS DSML. We propose a platform-based analysis of DRE systems consisting of two major aspects: *dependency*, which describes various relations and dependencies between tasks, and *platform*, which specifies the platform that executes the tasks. We capture both these aspects in ALDERIS by specifying the event flow between tasks and their mappings to machines.

4.1.1 Abstract Syntax

The ALDERIS DSML is a 6-tuple $A = \{T, C, TR, TH, M, D\}$ where:

- T is a set of *tasks*,
- C is a set of *communication channels*: $C \subseteq T$,
- TR is a set of *timers*: $TR \subseteq T$,
- TH is a set of *execution threads*,
- M is a set of *machines* that execute threads.
- D is the *task dependency relationship*: $D \subseteq T \times T$.

The ALDERIS DSML aims to model how a set of tasks $T = \{t_1, t_2, \ldots, t_n\}, n \in \mathbb{N}$ is assigned to execute on a set of machines $M = \{m_1, m_2, \ldots, m_q\}, q \in \mathbb{N}$ in a multi-threaded environment.

Tasks are assigned to execute on exactly one execution thread. Each thread is assigned to exactly one machine. Tasks are assigned to an execution thread by the mapping $\mathsf{thread}(t_k) : T \rightarrow TH$. Execution threads are assigned to machines by the mapping $\mathsf{machine}(th_l) : TH \rightarrow M$. Tasks are attributed by the following properties:

- wcet_k is the *Worst Case Execution Time (WCET)* of the task $t_k \in T$ specified by the mapping wcet: $T \rightarrow \mathbb{N}$,
- bcet_k is the *Best Case Execution Time (BCET)* of the task $t_k \in T$ specified by the mapping bcet: $T \rightarrow \mathbb{N}$,
- dl_k is the deadline of the task $t_k \in T$ specified by the mapping dl: $T \rightarrow \mathbb{N}$,
- p_k is the priority of the task $t_k \in T$ specified by the mapping p: $T \rightarrow \mathbb{N}$,
- sp_k is the sub-priority of the task $t_k \in T$ specified by the mapping sp: $T \rightarrow \mathbb{N}$.

The execution of tasks is modeled as an execution interval given by [bcet_k,wcet_k]. When the execution time is constant, bcet_k =wcet_k. bcet_k denotes the shortest execution time, wcet_k denotes the longest execution time of task t_k.

Channels in set C can be viewed as special tasks that *(1)* buffer events as *First In First Out (FIFO)* channels, and *(2)* represent communication delays. They are not required in the models as tasks can exchange events directly, but provide a mechanism to reduce *blocking waits*. For each channel $\forall c_d \in C$ we refer to bcet_d as the *best case delay*, wcet_d as the *worst case delay*.

We introduce a (hypothetical) execution thread $th_c \in TH$ that can execute an unbounded number of tasks at the same time. We express delays between tasks t_k and t_j, $\{t_k, t_j\} \in D$ by introducing a channel $c_d \in C$ that has an execution interval [bcet_d,wcet_d] as specified by the delay, and we assign c_d to the execution thread th_c using the mapping $\mathsf{thread}(c_d) = th_c$, and add the dependencies $\{t_k, c_d\}, \{c_d, t_j\}$ to the set D. Note that we only map channels to the hypothetical execution thread in order to express delays as non-preemptive executions in a formal setting, we do not restrict the actual implementation of event channels.

Note that when AND semantics are allowed, all parents need to finish their execution to trigger the execution of the (dependent) child task. This implies that events

from parents that send events at a higher rate need to be continuously discarded to avoid buffer overflows. Therefore, at this stage of development, we only allow OR semantics for task dependency; any parent task can trigger the execution of the (dependent) child task. This restriction prevents buffer overflows in tasks that depend on parents, that send events with different rates.

Timers in the set TR are special tasks that trigger the execution of tasks periodically. For each timer $\forall tr_k \in TR$, $\mathsf{wcet}_k = \mathsf{bcet}_k$, and we refer to the constant execution time as period.

We define the set of dependencies $D = \{(t_a, t_b), (t_c, t_d), \ldots, (t_m, t_n)\}$. Dependencies model partial ordering in the model, if the dependency (t_k, t_j) is part of the set D then task t_j has to execute after task t_k has finished. We say that task t_j *depends* on task t_k. A task may depend on several tasks and execute several times during the execution of the model. Moreover, we assume that there are no circular dependencies between tasks. Therefore, if task t_k depends on task t_j, then task t_j does not depend on task t_k. D is acyclic (a forest) with timers as root elements. We define the semantics for dependencies in Section 4.3 and Section 4.4.

Figure 4.1 shows an example DRE model created using the GME tool [61]. Timers are represented by the timer icon, tasks by the `T` icon, channels by `C`, threads by the thread icon, and machines by the processor icon. The solid arrows show the dependencies (set D) in the model. The mapping of tasks and timers to threads (the function $\mathsf{thread}(t_k) : T \rightarrow TH$) and the mapping of threads to machines (the function $\mathsf{machine}(t_k) : TH \rightarrow M$) are shown by the dashed arrows. Channels are not mapped to any threads as they are executed non-concurrently. Since there is only one thread assigned to each processor, the model utilizes non-preemptive scheduling.

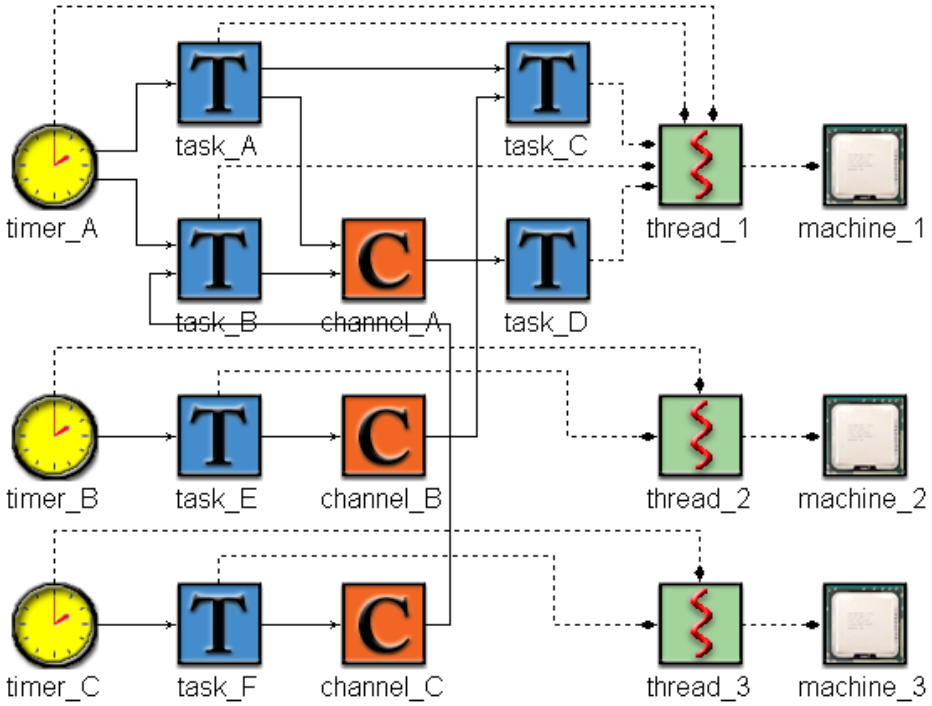

Figure 4.1: Example DRE Model

4.2 Specifying the ALDERIS Domain-specific Modeling Language by Meta-modeling

We build on the concept of *Model-integrated Computing (MIC)* to specify the ALDERIS DSML in the GME tool by meta-modeling. Meta-modeling allows to define the abstract syntax of a DSML, and in some cases even the structural semantics, as described in Section 3.3. We have defined the abstract syntax of the ALDERIS DSML in Section 4.1.1. This section describes how meta-modeling can provide a visual modeling language that adheres to the formal specification defined in Section 4.1.

Figure 4.2 shows the meta-model for the ALDERIS DSML using the GME meta-model, which is a variation of *Unified Modeling Language (UML)* class diagrams. Types for classes are specified as stereotypes for classes. *Atoms* are simple objects that have no internal structure, and cannot contain other objects. Models are abstract objects that represent entities that may have an internal structure. A model is a

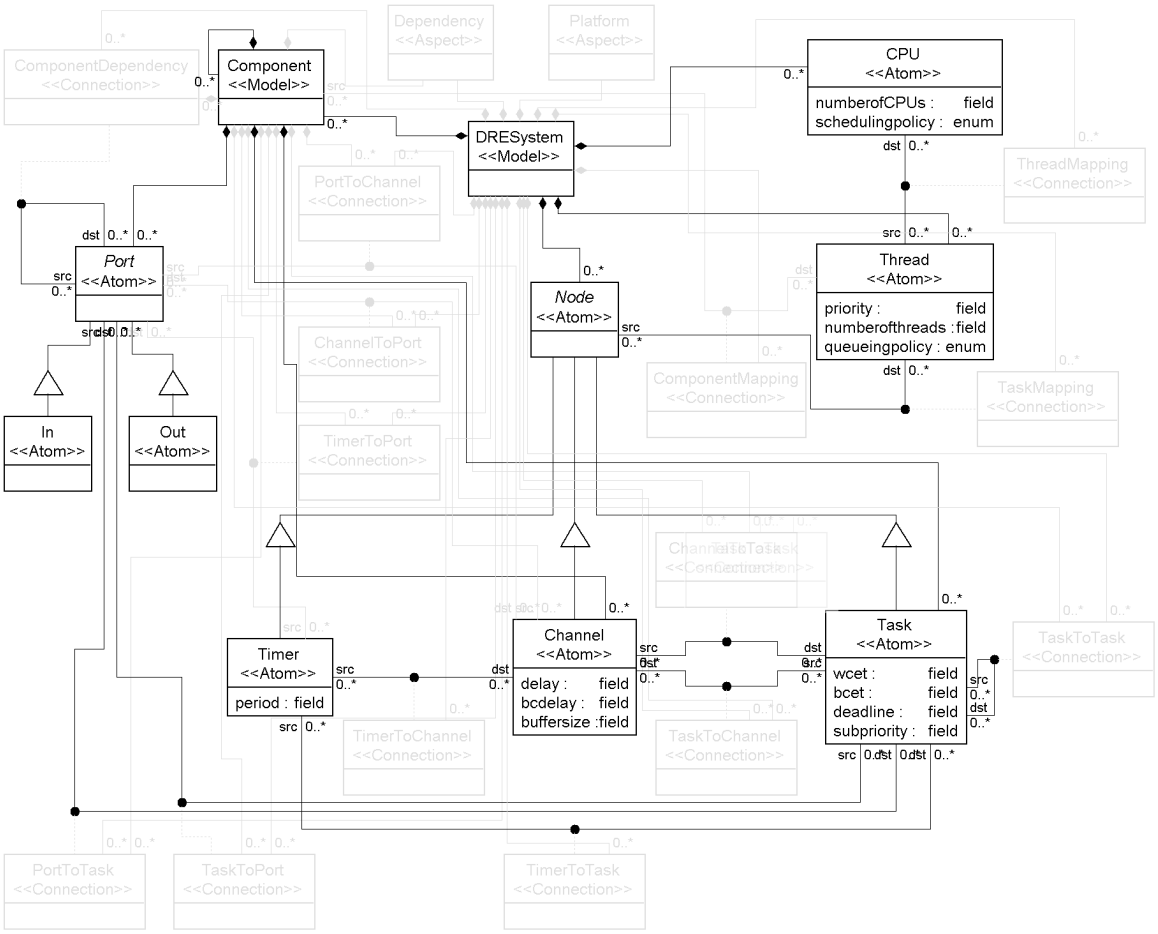

Figure 4.2: Meta-model for ALDERIS specified in GME

parent of all the objects it contains. Parts of a model may communicate through ports; which provides an abstraction to express hierarchy. Between atoms and models, *connections* may be defined, that are associations that may contain further attributes. For example, guards on a transition may be expressed as attributes for a connection. The GME meta-model can also express generalization, association and composition as well. Objects in GME are represented as a tree, where each model is a parent of its parts.

The `DRESystem` model shown in Figure 4.2 is the root model, that expresses the 6-tuple A defined in Section 4.1.1. `DRESystem` contains the `Node` abstract class, and the `Timer`, `Channel` and `Task` atoms inherit from the `Node` atom abstract class. The

`Task`, `Channel` and `Timer` atoms correspond to $T \setminus C \setminus TR$, C, TR as defined in Section 4.1.1. Inheritance – a concept commonly used in software engineering – has traditionally been hard to express using set theory. To alleviate this problem, we defined TR, $C \subseteq T$, and we refer to the $T \setminus C \setminus TR$ set as the `Task` class in the ALDERIS meta-model (*i.e.* `Channels` and `Timers` are not tasks in the ALDERIS GME model, but are tasks in the formal definition of the ALDERIS abstract syntax in Section 4.1.1).

The `Thread` and `CPU` atoms represent sets TH and M in Section 4.1.1. The task dependency relationship $D \subseteq T \times T$ is expressed as a set of connection using the GME meta-model (`TimerToChannel`, `TimerToTask`, `ChannelToTask`, `TaskToChannel`, `TaskToTask`, shown in gray in Figure 4.2). This approach allows GME to express constraints, such as that no connections are allowed from timers to timers, from channels to timers or channel, and from tasks to timers. These constraints not only express abstract syntax, but also structural semantics, as they provide rules for well-formed models. The $\mathsf{thread}(t_k) : T \rightarrow TH$ and $\mathsf{machine}(th_l) : TH \rightarrow M$ mappings are defined as the `ThreadMapping` and `TaskMapping` connections in the ALDERIS meta-model shown in Figure 4.2.

The ALDERIS GME model can also express hierarchy; the `Component` model can contain groups of `Tasks` and `Channel`, as well as connections between them. Components have input and output `Ports`, and communicate through their ports. Connections `TimerToPort`, `ChannelToPort`, `TaskToPort`, `PortToChannel`, `PortToTask` shown in gray in Figure 4.2 all extend the task dependency relationship D to hierarchical models. We did not formalize hierarchy in Section 4.1, as it is not required for formal analysis, rather it is just a convenience for designers. Therefore, `Components` and `Ports` are not part of the formalism, but can be used within the ALDERIS DSML. Hierarchical models can be flattened out for formal analysis if needed.

The `Dependency`, `Platform` aspects can be used to constrain the visibility of certain

modeling constructs. For example, the dependency aspect captures `Tasks`, `Channels`, and `Timers` together with the task dependency relationship. The platform aspect, on the other hand, expresses the `ThreadMapping` and `TaskMapping` connections in the ALDERIS meta-model. The `Dependency` and `Platform` aspects define the event-driven communication within *Asynchronous Event-driven Distributed Real-time Embedded (AEDRE)* systems, as well as the mapping of tasks to a distributed platform.

4.3 Specifying the DRE SEMANTIC DOMAIN by Timed Automata

This section formalizes a computational model that can express AEDRE systems for real-time analysis. The proposed MoC is *Timed Automata (TA)*, as defined in Section 3.4. We refer to the proposed semantic domain as the DRE SEMANTIC DOMAIN. We chose TA as the underlying MoC for our analysis since it has formally defined semantics [6], is supported by several automated model checking tools [26, 15], and is expressive enough to capture the dynamics of a wide class of DRE systems. Using the DRE SEMANTIC DOMAIN to specify the operational semantics of the ALDERIS DSML we define a formal MoC, that we refer to as DRE MoC.

The DRE SEMANTIC DOMAIN can be used to model basic components in DRE system such as timers, dynamic computation tasks, event channels, and schedulers. DRE system models can be built by the composition of these components.

The TA models introduced in this section are based on the UPPAAL model of TA [12]. UPPAAL extends the generic TA model [6] with various constructs such as constants, integers, `committed` and `urgent` constraints on locations, networks of TA and events broadcasted and unicasted between automata.

In the UPPAAL model, locations denoted by the C letter represent `committed` locations, and location denoted by the U letter represent `urgent` locations. Both

constraints imply that time cannot pass in that location; either an outgoing transitions must be taken as soon as the automaton enters the location, or the model deadlocks. The difference between `committed` and `urgent` locations is priority; if both `committed` and `urgent` locations are entered simultaneously in a model, outgoing transitions from the `committed` location(s) will be taken before outgoing transitions from the `urgent` location(s). Please see [12] for a discussion on the UPPAAL model of TA.

In this section we use the UPPAAL TA model as the extensions are very useful for modeling practical DRE systems. The Verimag IF toolset also introduces useful constructs such as integers, priorities between transitions, FIFOs for communication between automata, monitor automata etc. The reason we use UPPAAL in this section is due to UPPAAL's graphical syntax (IF uses a textual specification), that allows a compact representation of the TA models.

4.3.1 Timers

A timer in the DRE SEMANTIC DOMAIN is a simple periodic event generator that broadcasts events at a specified rate. Timers may represent sensors sampled at a predefined rate. In the generic setting we assume that clocks are synchronized. This restriction may be relaxed to clocks that may drift arbitrarily, as long as the drifts in timers are bounded. Timers can be represented by a generic timed automaton model shown in Figure 4.3.

The timer construct uses one clock, c_e to measure time. Whenever c_e reaches `bcperiod`, the self-transition becomes enabled, and can be taken non-deterministically (*i.e.* it may be taken, but is not required). The invariance constraint ($c_e \leq$ `period`) imposes a constraint on the location; the valuation of c_e cannot be higher than `period`. Therefore, the transition must be taken when c_e reaches `period`. The guard on the transition together with the invariance essentially imposes that the transition will be taken between [`bcperiod`, `period`] time. When the transition is eventually taken,

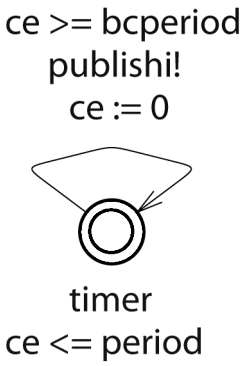

Figure 4.3: UPPAAL Timed Automaton Model for a Timer

the c_e clock is then reset to zero again, and a `publish`$_i$ event is broadcasted to other automata.

While this timer model can express drifting timers in theory, using the [`bcperiod`, `period`] interval to specify when the timer broadcasts events, drifting timers incur a significant performance hit on the analysis. In most practical cases, it is desirable to use synchronized clocks for the analysis. Note that synchronized timers do not imply the use of *Time-triggered Distributed Real-time Embedded (TTDRE)* systems, as the execution of tasks is not synchronized, only the timers that drive the computations.

4.3.2 Non-preemptable Tasks

Tasks are the main components in DRE systems that carry out computations and interact with the physical environment. Tasks are enabled by `release` events that may be received from other tasks, event channels, or at regular intervals from a timer. Tasks in the DRE SEMANTIC DOMAIN do not model the actual computations carried out by tasks, just the time required for computations and their respective deadlines.

Figure 4.4 shows the UPPAAL TA model of a non-preemptable task. The task consists of 5 locations; the idle location is the initial location, wait represents when the task is enabled, but is waiting to be scheduled for execution, run represents the executing task, send is a committed location, and finally the error location represents

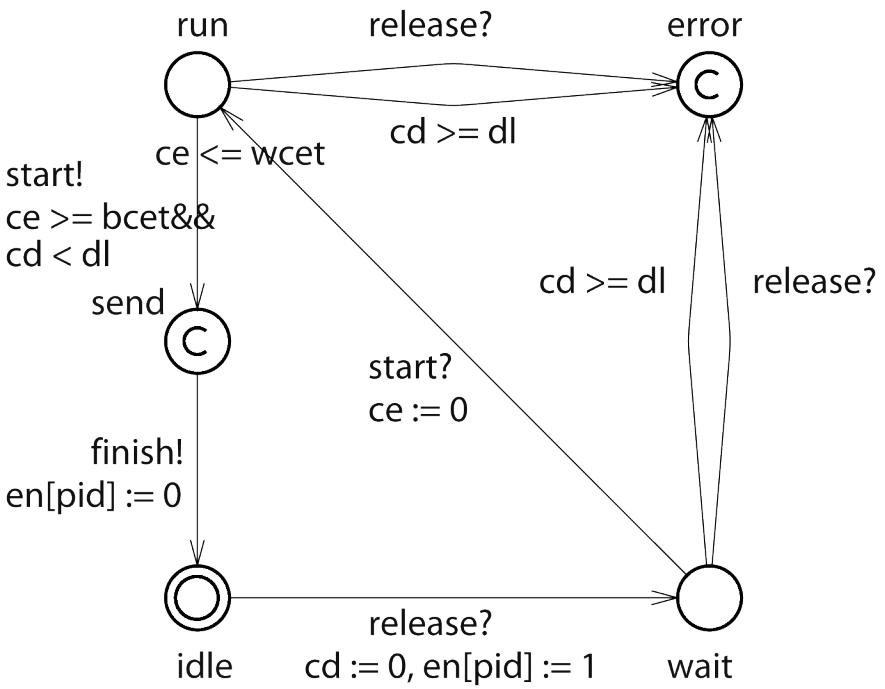

Figure 4.4: UPPAAL Timed Automaton Model for a Non-preemptable Task

deadline misses.

The automaton utilizes two clocks; c_d measures the time from the enabling event to the completion of the execution. c_d is referred to as the *deadline clock*, as it is used to measure whether the deadline is exceeded at any point during the task's execution. c_e is referred to as the *execution clock*, and measures the time spent in the run location, thus modeling the time that the task spends executing.

We now discuss transitions in the automaton to explain how Figure 4.4 models the execution of a non-preemptive task. The idle → wait transition is triggered when the task receives the task release event, which enables the task's execution. The transition resets c_d to zero, and signals that the task is enabled for execution by setting the flag in the `en[]` array.

The wait → run transition is triggered by the `start` event broadcasted by the scheduler, and resets the c_e clock to zero. The run → send transition gets enabled when the c_e clock is equal to or higher than bcet. The $c_e \leq$ wcet invariance forces

the transition to be taken, and the guard and invariance together enforce that the run → send transition will be taken in the [bcet, wcet] interval on the c_e clock. This is how the TA models execution intervals. The run → send transition broadcasts a `start` event, that triggers the idle → wait transitions of dependent tasks through the `release` event. The send location is `committed`, and therefore the send → idle transition will be taken immediately, a `finish` event is sent to the scheduler to notify it that the task finished its execution, and the flag in the `en[]` array is reset. The send location is necessary since the automaton broadcasts two events, `start` and `finish`, and therefore we break up the transition to two transition, connected by the `committed` send location.

The transitions to the error location become enabled whenever c_d reaches the dl deadline, or if a task receives another `release` event while processing another event. The channel and buffer constructs described in Section 4.3.4 and Section 4.3.5 must be used to buffer events if there is a possibility that the task may receive a `release` event while processing another request.

To reduce the state space, we set the `error` location to be `committed`. Time cannot pass in `committed` locations in UPPAAL; either a transition must be taken, or the model deadlocks. By using this constraint we introduce a useful side effect; whenever a timed automaton reaches the `error` location time cannot advance in that automaton. Since we cannot leave that location, the TA model will deadlock. This approach turns the real-time schedulability problem to reachability analysis; the task is schedulable *iff* the TA model does not deadlock, which implies that the error location is not reachable.

4.3.3 Preemptable Tasks

Clocks in TA cannot be stopped or resumed, only reset. Therefore, TA cannot model preemptable tasks inherently. *Stopwatch Automata (SA)* [77] were proposed as a

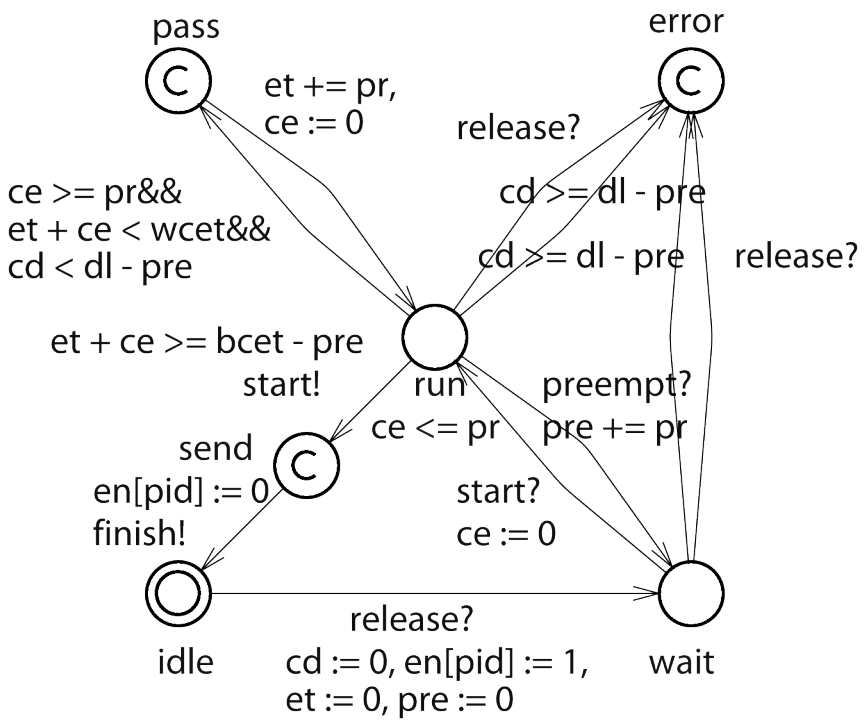

Figure 4.5: UPPAAL Timed Automaton Model for a Preemptable Task

MoC that can express preemptable tasks in asynchronous event-driven systems. SA can express linear *Hybrid Automata (HA)* [22], and the reachability analysis on SA is undecidable in general, since it can be mapped to the halting problem [55].

Chapter 7 describes a novel method for the model-checking of schedulability in *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)*. We approximate the *Task Stopwatch Automaton (TSA)* shown in Figure 7.1 with the *Task Timed Automaton (TTA)* shown in Figure 7.4.

The UPPAAL preemptable task model shown in Figure 4.5 is a syntactically more compact representation of the TTA model shown in Figure 7.4, but expresses the same idea for approximation, and is a direct result of our work on the approximation of SA by TA described in Chapter 7.

The UPPAAL model for a preemptable task consists of 6 locations. The idle, wait and error locations are no different than in the non-preemptive task model shown in

Figure 4.4.

The et variable represents the discrete portion of the execution clock, denoted as x is the index of locations $\mathsf{run}_{\mathsf{x,y}}$ in Definition 7.2 and in Figure 7.4. The pre variable represents the number of preemptions encountered, denoted as y is the index of locations $\mathsf{run}_{\mathsf{x,y}}$ in Definition 7.2 and in Figure 7.4. Thus, we encode the $\mathsf{run}_{\mathsf{x,y}}$ locations in Figure 7.4 as two integer variables. The pr variable defines the precision of the approximation, and is denoted as time unit t_u in Figure 7.4.

The approximation is implemented as follows. The run location represents the executing task. Whenever c_e reaches the time unit pr, the automaton moves to the pass committed location, and thus right back to the run location. The pass location models as the automaton saves the "checkpoint" time unit in the et integer. et is incremented by the time unit pr, the real-valued clock c_e is reset, and the execution continues. Whenever the automaton receives a preempt event in the run location, it moves back to the wait location, modeling preemption. The run $\rightarrow$ send transition is enabled in the [bcet - pre, wcet] interval on the et + c_e clock (denoted as $v_x = v_{ta} + x$) in Definition 7.2), to compensate for the BCET imprecision as described in Proposition 7.3. The error location is reachable if c_d reaches dl - pre. pre is used to compensate for the WCET and dl imprecision of the approximation, as defined in Proposition 7.2 and Equation 7.6.

4.3.4 Event Channels

Event propagations between tasks follow the UPPAAL TA non-blocking broadcast semantics, *i.e.*, events that are not received (and therefore lost) are not regenerated since received events are not acknowledged. To provide a reliable event service, therefore, event passing must be synchronized between the publisher and consumer tasks. To alleviate these restrictions, communication between tasks are coordinated through event channels that provide the mechanisms to allow reliable asynchronous

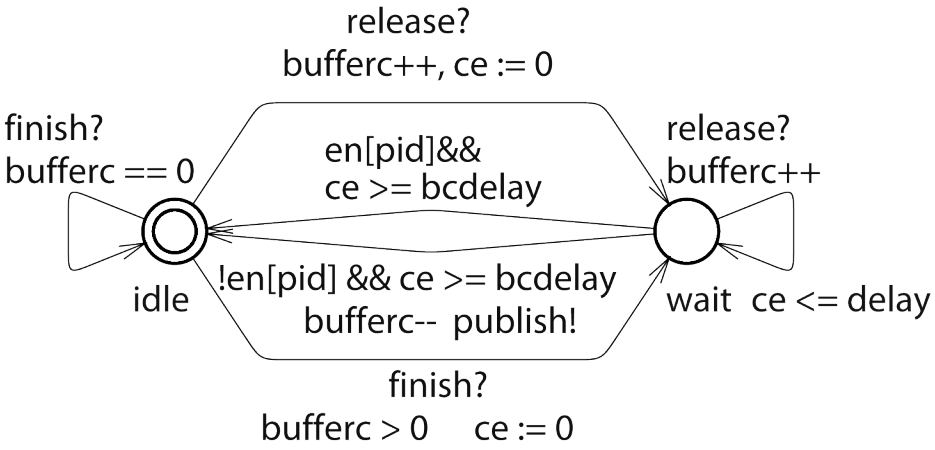

Figure 4.6: UPPAAL Timed Automaton Model for a Channel

communication, where publishers do not block until the consumer receives the event. Instead, published events are queued in the event channel until the consumer is ready to receive them.

Figure 4.6 shows the generic UPPAAL model of the event channel, and consists of only two locations, idle and wait. The event channel can express delays as intervals, and acts as an agent between two tasks, or a timer and a task. The event channel has only one clock, c_e that is used to model the delay.

The source timer or task triggers the idle $\rightarrow$ wait transition by broadcasting the `release` event. The event channel buffers the received event in the $\texttt{buffer}_c$ integer variable. The wait location represents the delay of the channel. If more `release` events are received in the wait location, the channel stores them in $\texttt{buffer}_c$.

One of the wait $\rightarrow$ idle transitions becomes enabled in the [bcdelay, delay interval. If the dependent task of the channel is in the idle location and ready to process the event from the channel, then then corresponding `en[]` flag is false, and the channel will broadcast the event ($\texttt{publish}_i$, and remove the event from $\texttt{buffer}_c$. If the dependent task is not in the idle location, then the channel moves back to idle, and the event remains stored in $\texttt{buffer}_c$.

When the dependent task finishes its execution, it notifies the channel through the

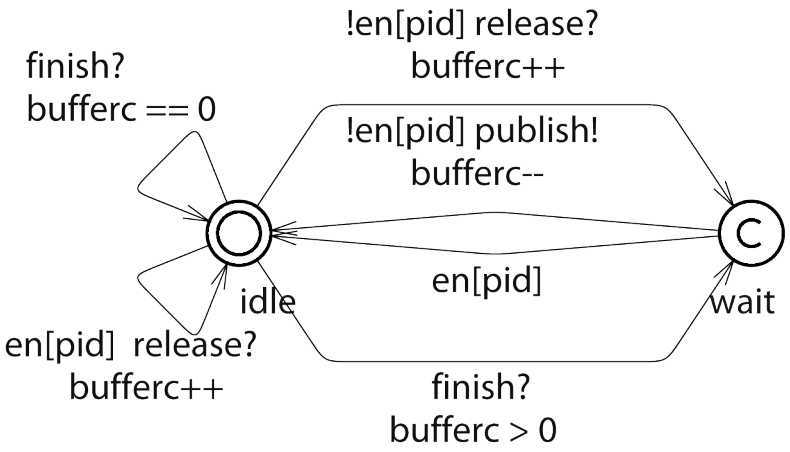

Figure 4.7: UPPAAL Timed Automaton Model for a Buffer

$\texttt{finish}_i$ event. If $\texttt{buffer}_c$ is not empty, the channel moves to wait as it tries to resend the event.

4.3.5 Buffers

The buffer modeling construct shown in Figure 4.7 is a special event channel that has no delay. Buffers represent events passed between tasks within the same thread (*i.e.* interprocess communication). The buffer ensures that the event will be delivered before the scheduler is invoked, and leaves no possibility for a race condition.

Buffers do not have a clock, as they do not model time. Transitions in the buffer are instantaneous. Similar to the channel, a buffer can store events in $\texttt{buffer}_c$ if the dependent task is busy serving another task, and can buffer multiple events.

4.3.6 The Scheduler

The scheduler models the mapping of tasks to a distributed execution platform. In the approach proposed in this dissertation, the scheduler is automatically synthesized from the priority (p: $T \rightarrow \mathbb{N}$) and subpriority (sp: $T \rightarrow \mathbb{N}$) mappings defined for ALDERIS in Section 4.1.

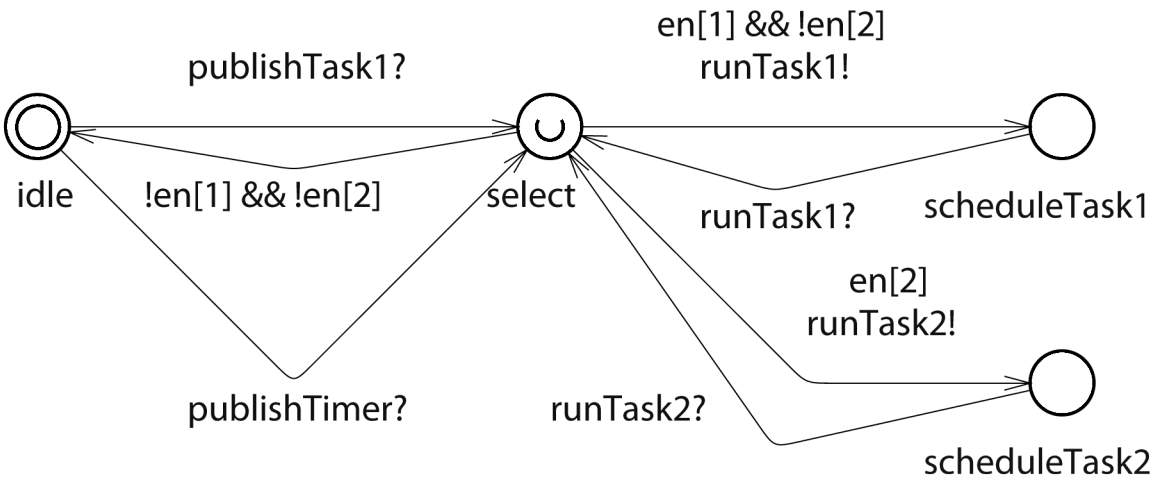

Figure 4.8: UPPAAL Timed Automaton Model for a Scheduler

Figure 4.8 shows the TA model of a simple fixed-priority scheduler, that manages the scheduling on a single non-preemptive processor. The scheduler starts in the idle location, and observes all the input events of tasks, since the scheduling decision can only follow an enabling event. If any task managed by the scheduler receives an input event, the scheduler moves to the select location. The select location is `urgent`, and therefore the scheduling decision may not advance the clocks, but will only follow the `committed` locations in other elements of the DRE SEMANTIC DOMAIN. This choice ensures that all events will be delivered to tasks before the scheduling decision is invoked.

Depending on how many tasks the scheduler manages, it has a corresponding schedule... location for each task. The `en[]` array is used to define the scheduling policy; by observing which tasks are enabled for execution, the scheduler selects one task for execution. Determinism in the scheduler is important as it increases the analysis scalability.

The scheduler shown in Figure 4.8 implements a simple fixed-priority scheduling policy; if `Task2` is enabled, it will be selected for execution, if `Task1` is enabled and `Task2` is not, then `Task1` will be scheduled for execution. If both `Task1` and `Task2`

are enabled, `Task2` will be scheduled for execution, as only the guards on the select → scheduleTask2 transition evaluate to true in that case. Therefore, the scheduling policy is deterministic.

Complex preemptive policies can also be synthesized using TA. Since TA is an extension of *Finite State Machines (FSMs)* with real-valued clocks, it can practically express any scheduling policy that can be expressed as a FSM, but can also incorporate time constraints in the scheduling policy. In this dissertation, we use fixed-priority scheduling for the case studies.

4.3.7 Modeling Constraints

This section introduces a set of constraints for well-formed models based on the DRE Semantic Domain.

- `Task` to `Task` *connections must be one-to-many.* Events are broadcasted from the source to every dependent. If multiple events are sent to the same task, however, events will be dropped, which is a failure we want to avoid.

- `Task` to `Channel` *connections must be one to many*, which provides a modeling construct – the event channel – to express one-to-many broadcasting and many-to-one event consumptions.

- `Channels` *can only have one dependent.* Since event channel has only one buffer it cannot keep track of the buffer of individual tasks.

- `Channels` *can only have one source.* We have previously allowed a single task to broadcast events to multiple channels. If a task A is connected to an event channel C that channel also receives events from a task B emitting an event from task A will be received by task B, as well, since connections are unidirectional. We therefore disable this modeling construct.

4.4 Specifying the DRE Semantic Domain as a Discrete Event System

In this section we define a formal model for event-driven DRE systems using the concept of DES. We utilize this model to define the formal performance evaluation problem using a continuous time model. We target AEDRE systems, therefore we propose a scheduling model, that inherently captures varying communication delays as special "tasks" with execution intervals. We define the DRE Semantic Domain on a preemptive platform, but restrict the performance estimation problem described in Chapter 6 to non-preemptive systems, due to undecidability issues as described in Chapter 7. Using the DRE Semantic Domain to specify the operational semantics of the Alderis DSML we define a formal MoC, that we refer to as DRE MoC.

4.4.1 Events

We represent the DRE Semantic Domain as an extension to DE systems in order to express execution intervals in continuous time. A DE system is a 5-tuple $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is a finite set of states, Σ is a finite alphabet of symbols that we refer to as *event labels*, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, q_0 is the initial state, and Q_m is the set of *marker states* (exiting states). A transition or *event* in G is a triple (q, σ, q') where $\delta(q, \sigma) = q'$, $q, q' \in Q$ are the *exit* and *entrance* states, respectively, and $\sigma \in \Sigma$ is the event label. The *event set* of G is the set of all such triples.

In DES, transitions depend only on the current state and the event label. In the DRE Semantic Domain, we define event labels as time stamped values from the domain of non-negative real numbers $\mathbb{R}_{\geq 0}$.

Timestamps model the (global) simulation time when the event has occurred. We define the function $\mathsf{time}(e) : E \rightarrow \mathbb{R}_{\geq 0}$ to return the timestamp value of event $e \in E$

where E is the set of all (infinite) events generated by the system.

In the DRE Semantic Domain, tasks receive a potentially infinite sequence of events (timestamped values as event labels) in chronological order. The task then outputs a timestamped event for each input event. We denote the sequence of input events of a task t_k as $I_k = \{i_{k0}, i_{k1}, \dots\}$, the sequence of output events as $O_k = \{o_{k0}, o_{k1}, \dots\}$, $t \in T, i_{k0}, i_{k1}, \cdots \in E, o_{k0}, o_{k1}, \cdots \in E$. The order of events in the output sequence of each task is the same as the order of events in the input sequence of the task, o_{k0} is the response for i_{k0}, o_{k1} is the response for i_{k1}, etc. The timestamps of input events must be smaller than or equal to their corresponding output events. We formalize this constraint as follows: $(\forall t_k \in T)(\forall i_k \in I_k)(\forall o_k \in O_k)$ $(\mathsf{time}\ (i_k) \leq \mathsf{time}\ (o_k))$. Note that the discrete event simulation is completely deterministic if timestamps are unique constants.

Each task can process only one event at a time, for each input event $i_{kx} \in I_k$, the corresponding output event $o_{kx} \in O_k$ has to be generated before the task can receive its next input event $i_{ky} \in I_k$. Channels in the set C provide FIFO buffers to store events that cannot be immediately processed by their target tasks.

The set TR denotes a special class of tasks called *timers*. A timer $tr \in TR$ is a task that generates output events such that for the timestamp of any two consecutive events $(o_{tr}, o'_{tr} \in E)$ $\mathsf{time}(o'_{tr}) - \mathsf{time}(o_{tr}) = \mathsf{period}_{tr}$. We define the period of the timer as $\mathsf{period}_{tr} = \mathsf{bcet}_{tr} = \mathsf{wcet}_{tr}$.

To distinguish the special classes of tasks from non-special tasks we refer to the set $T \setminus C \setminus TR$ as the set of *computation tasks*. Computation tasks model actual tasks being executed on the threads/machines.

Computation tasks, timers, and event channels in the DRE Semantic Domain can be composed using events; the output event of a task may serve as an input event to another task (tasks). In these cases the same event label triggers multiple transitions. We make the restrictions that timers cannot have input events and the

output events of event channel can only be inputs to computation tasks. Moreover, the event flow has to satisfy the dependencies in set D; if the output of a task $t_1 \in T$ is the input of task $t_2 \in T$ then the dependency (t_1, t_2) has to be present in set D. Similarly, if a dependency (t_a, t_b) is present in set D then the output event of t_a has to be the input event of t_b.

4.4.2 Task States, Schedulers

We define three states for each computation task; `init`, `wait`, and `run`. Whenever a task receives an event from another task (including event channels and timers) the transition from the `init` state to the `wait` state is triggered. We refer to tasks in the `wait` state as *enabled tasks*. Enabled tasks are ready to execute.

We model scheduling policies by utilizing priorities. We model schedulers as discrete event systems that compose with tasks using events. Our model for schedulers keeps track of enabled tasks by putting them in an execution queue. Whenever the execution queue is non-empty the scheduler chooses a task (or possibly several tasks) for execution by generating an event triggering the transition from the `wait` state to the `run` state in the selected task(s). We assume that tasks can distinguish between events coming from tasks and schedulers in the DRE Semantic Domain. Events generated by schedulers are also labeled as timestamps.

Figure 4.9 shows a possible DES representation of the DRE example shown in Figure 4.1. We create a finite state machine model for each task, channel, timer, and scheduler, that compose using events. We denote input events as $e?$, output events as $e!$. If the output event of a transition is the input event of another transition then the two transitions are synchronized; their events must have the same timestamps when the transitions are taken. If two (or more) transitions are synchronized either all of them has to be taken, or none of them. We model scheduling policies by introducing priorities between transitions. For example, a simple fixed priority scheduling policy

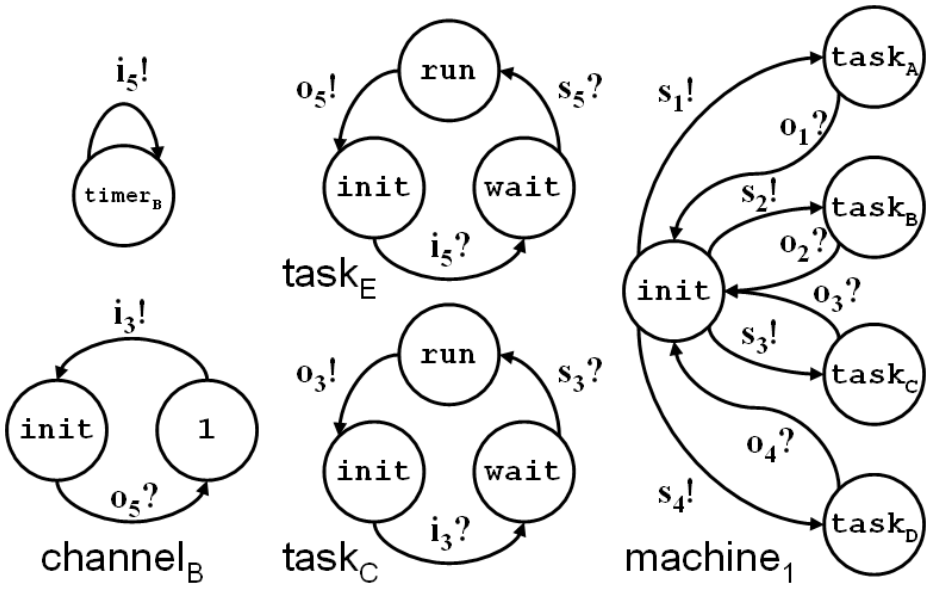

Figure 4.9: Composing Discrete Event Models using Events - Partial Representation of the DRE Example Shown in Figure 4.1

between tasks t_a, t_b, t_c, and t_d may be implemented by introducing priorities between the transitions marked as $s_1!, s_2!, s_3!, s_4!$. The priorities may enforce a fixed execution order between enabled tasks (a task is enabled if it is in the wait state).

Schedulers in the DRE Semantic Domain may utilize both a preemptive or non-preemptive scheduling policy. However, we restrict the formal performance estimation method in Section 6 to non-preemptive systems only.

Each computation task $t_k \in (T \setminus C \setminus TR)$ generates an output event with a timestamp between [bcet_k + $\mathsf{time}(s_k)$, wcet_k + $\mathsf{time}(s_k)$] where s_k is the event generated by the scheduler that triggered the transition from state `wait` to state `run` in task t_k.

In the DRE Semantic Domain every task has to be mapped to exactly one execution thread (defined as set TH in Section 4.1.1), and every execution thread must be mapped to exactly one machine (defined as set M in Section 4.1.1). Threads in the DES model are modeled as a non-preemptive scheduler, while machines are modeled as a preemptive scheduler between threads. This hierarchical scheduling model allows to express complex DRE systems.

Chapter 5

Real-time Model Checking of Software-Intensive Distributed Real-time Embedded Systems

This chapter describes how we applied the *Analysis Language for Distributed, Embedded, and Real-time Systems (*Alderis*) Domain-specific Modeling Language (*DSML*)* defined in Section 4.1 and the DRE Semantic Domain specified as *Timed Automata (*TA*)* in Section 4.3 for the real-time analysis of software-intensive mission-critical *Common Object Request Broker Architecture (*CORBA*)* avionics *Distributed Real-time Embedded (*DRE*)* systems. This section focuses on non-preemptive systems only, but the proposed method can be composed with the analysis of preemptive systems. We adapt the TA-based real-time verification method to preemptive systems in Chapter 7. Using the DRE Semantic Domain to specify the operational semantics of the Alderis DSML we defined a formal *Model of Computation (*MoC*)*, that we refer to as DRE MoC.

A promising infrastructure technology for software-intensive DRE systems is *component middleware*, which defines platform capabilities, and tools for specifying, implementing, deploying, and configuring *components* [78], and publish/subscribe services [41] that exchange messages between components. Components, in turn, are units of implementation, reuse, and composition that expose named interfaces called *ports*, which are connection points that components use to collaborate with each other. Component middleware helps simplify the development and validation of DRE systems by providing reusable services, and optimizations that support their functional, and *Quality of Service (*QoS*)* needs more effectively than conventional *ad hoc* software implementations.

Despite recent advances in component middleware, however, there remain signifi-

cant challenges that make it hard to develop large-scale DRE systems, including the lack of tools for effectively configuring, integrating, and verifying DRE systems built using components. To address these challenges, it is useful to analyze system behavior early in the life cycle, thereby enabling developers to select suitable design alternatives before committing to specific platforms or implementations. In particular, making these decisions in the design phase helps minimize mistakes that would otherwise be revealed in the testing and integration phases, when they are much more expensive to fix. Design-time analysis requires a means of expressing component behavior with respect to their QoS properties, defining the semantics for component interactions, and composing components to form subsystems.

In this section we illustrate and validate the concepts of model-based verification in the context of the Boeing Bold Stroke platform. We build on the ALDERIS DSML to model and analyze a complex DRE system, and the open-source *Distributed Real-time Embedded Analysis Method (*DREAM*)* framework for the model checking [31] of real-time properties. In particular, we consider the problem of deciding the schedulability of a given set of Bold Stroke tasks with event- *and* time-driven interactions. We represent the task model and scheduling policy via the DRE SEMANTIC DOMAIN. The TA formulation of the problem translates the schedulability problem into a reachability problem in which the set of tasks are schedulable if none of the corresponding TA can reach a state that was predefined to express missed deadlines. If this analysis completes successfully it implies that all tasks complete before their respective deadlines.

The contributions of this chapter are focused on the following areas:

- We formalize the problem of deciding the schedulability of DRE systems utilizing fixed priority scheduling using TA in Section 5.1.
- We describe the Boeing Bold Stroke execution framework in Section 5.2, and describe an approach for the modeling of multi-threaded software intensive DRE

systems by utilizing abstractions based on the threading model in Section 5.3.

- Section 5.4 introduces a case study loosely based on the Boeing Bold Stroke execution platform.
- Section 5.5 describes a model checking method based on TA for the real-time verification of non-preemptive DRE systems.

5.1 Problem Formulation

We utilize the DRE MoC specified as a TA system in Section 4.3 to define the real-time verification problem for DRE systems. We use the notation for the sequence of input events of a task t_k as $I_k = \{i_{k0}, i_{k1}, \dots\}$, the sequence of output events as $O_k = \{o_{k0}, o_{k1}, \dots\}$, $t \in T, i_{k0}, i_{k1}, \cdots \in E, o_{k0}, o_{k1}, \cdots \in E$. We specify deadlines for each computation task t_k using the mapping $d_k : T \rightarrow \mathbb{N}$. Deadlines for each task and each input event are counted from the timestamp of the input (enabling) event ($\mathsf{time}(\mathrm{i}_k)$).

Definition 5.1 (Schedulability): *A computation task $t_k \in T$ is schedulable if it always finishes its execution before its respective deadline. The* error *location is reachable in the TA model for task* iff *the computation task does not finish its execution before its respective deadline. The fact that the* error *location is unreachable implies the schedulability of the computation task t_k. The DRE MoC is schedulable if all tasks are schedulable, i.e. if none of the* error *locations are reachable in the TA model.*

Definition 5.2 (Run): *A* run *or* execution trace *of the DRE MoC is the chronological sequence of events occurred in the model. A run is* valid *if it is schedulable, that is if for all input (i_k) and their corresponding output (o_k) events in the execution trace* $\mathsf{time}(o_k) < \mathsf{time}(i_k) + d_k$*, otherwise it is* invalid.

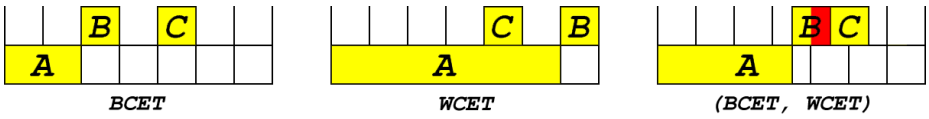

Figure 5.1: Motivating Example for a Non-WCET Deadline Miss

In event-driven systems it is not enough to consider the worst case times of tasks in general. It is a fact that the product of local worst case execution times does not necessarily result in worst case end-to-end computation times. We now demonstrate this problem in non-preemptive DRE systems to motivate our approach for the real-time verification of DRE systems.

Consider the example shown in Figure 7.3. Task A is running on `machine_1`, and tasks B and C are running on `machine_2`. Task A starts at time 0, and may finish its execution time anytime within the $[2, 6]$ interval. Task B starts its execution whenever task A finishes its execution, and executes for 1 time unit. Task C starts its execution at time 4, and executes for 1 time unit. We assume that task B has higher priority than task C, and that the deadlines for task B and task C are 1.2 time units. The system is schedulable when task A executes for its *Best Case Execution Time (BCET)* time as shown in the left of Figure 7.3, and it is schedulable when task A executes for its *Worst Case Execution Time (WCET)* time as shown in the middle of Figure 7.3. However, if task A finishes its execution at time 5.5, task C will miss its deadline as it has to wait for task B.

Therefore, the analysis has to capture execution intervals in continuous time, otherwise it may lead to false positives, and thus may result in unschedulable designs that cannot be detected at design time. We solve this problem for non-preemptive DRE in this chapter by capturing software-intensive DRE systems as TA models for real-time model checking. In Section 5.2 we describe the software architecture of the Boeing Bold Stroke execution platform, and describe our approach for modeling Bold Stroke using ALDERIS in Section 5.3.

5.2 Boeing Bold Stroke Execution Platform

The Boeing Bold Stroke architecture is an event-driven component-based DRE system platform built atop *(1) The ACE ORB (TAO)* [90], which implements key Real-time CORBA [79] features (such as thread-pools, lanes, and client-propagated and server-declared threading policies), and *(2)* TAO's Real-time Event Service [41], which implements the publish/subscribe pattern [19], and schedules and dispatches events via a federation of real-time event channels (any event channel mentioned in the rest of the chapter refers to the real-time event channel). Bold Stroke uses a Boeing-specific component model called *PRISM* [87], which implements a variant of the *CORBA Component Model (CCM)* [78] atop TAO. Following the CCM specification, *PRISM* defines the following types of *ports*, which are named interfaces, and connection points components use to collaborate with each other:

- **Facets**, which define named interfaces that process method invocations from other components.
- **Receptacles**, which provide named connection points to facets provided by other components.
- **Event sources and event sinks**, which indicate a willingness to exchange event messages with one or more components via event channels.

PRISM operation invocations provides blocking synchronous *call/return* semantics, where one component's receptacle is used to invoke an operation on another component's facet. Conversely, *PRISM*'s event propagation mechanism provides non-blocking asynchronous *publish/subscribe* semantics supported by real-time event channels connected via event sources/sinks. When a publisher pushes an event to an event channel all of its subscribed components are notified.

Although the CCM specification allows the dynamic creation and connection of components, *PRISM* follows common patterns in safety/mission-critical systems, and

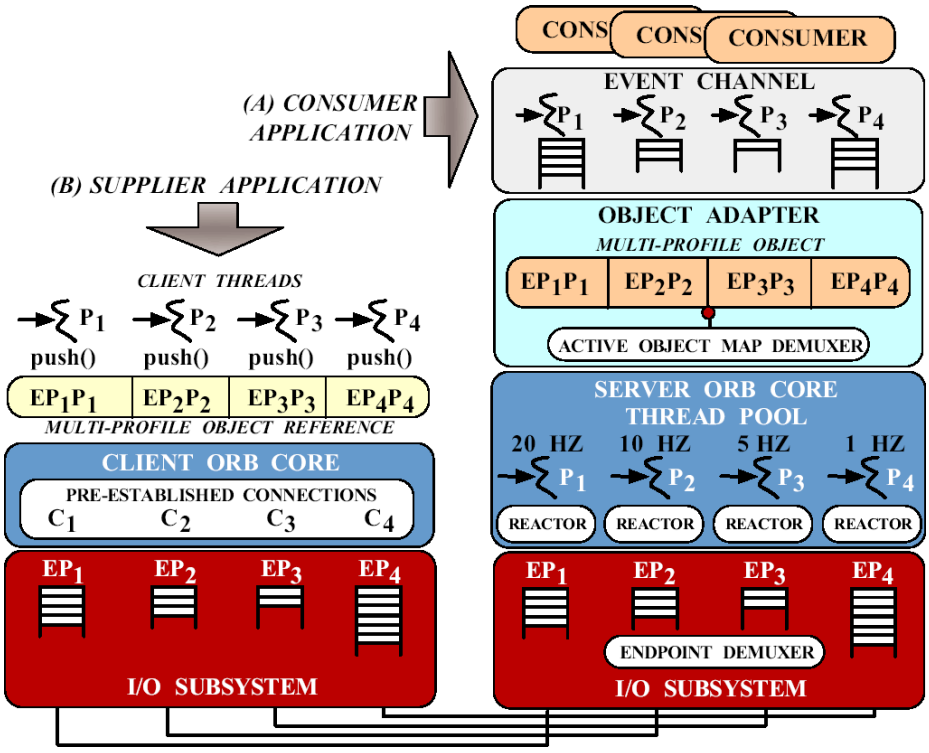

Figure 5.2: The Boeing Bold Stroke Execution Platform

enforces a static component allocation and configuration policy by creating and connecting components only during system initialization. Dynamical components in *PRISM* can reconfigure themselves by changing their behavior based on system mode settings, such as takeoff mode, landing mode, and threat-evasion mode.

Figure 5.2 shows the runtime architecture of the Bold Stroke execution platform, which consists of three primary layers: *(1)* the *Object Request Broker (ORB) layer*, which performs (de)marshalling, connection management, data transfer, event/request demultiplexing, error handling, concurrency, and synchronization, *(2)* the *real-time event channel* layer, which schedules and dispatches events in accordance with their deadlines and other real-time properties, and *(3)* the *application component layer*, which contain actions that are the smallest units of end-to-end processing that Bold Stroke application developers can manipulate.

Bold Stroke actions are largely event-driven, rather than strictly time-triggered. In particular, periodic real-time processing of frames is driven by asynchronous software timers that may drift apart from each other, so component interactions are unrestricted and asynchronous. This approach is intentional [29] and designed to increase flexibility and performance, though it has the side effect of impeding analyzability and strict predictability.

As a result of Bold Stroke's event-driven architecture, dependencies between actions can significantly influence the schedulability of avionics mission computing systems built atop it. Bold Stroke applications use priority-based scheduling, where actions that have the same priorities are scheduled non-preemptively in a *priority band* (also referred to as *rate group*) based on their sub-priorities. In this setting, preemptive scheduling is used between priority bands, whereas non-preemptive scheduling is used within a particular band.

A priority band is implemented by three types of threads: *(1)* the *dispatcher (worker) threads*, which reside in real-time event channels and execute all actions initiated by event propagations, *(2)* the *interval timeout thread*, which pushes timeout events at predefined intervals, and *(3)* *ORB threads*, which continually process request inputs from the ORB core executing actions initiated by operation invocations. This concurrency architecture implements the *Half-Sync/Half-Async* pattern [91], where a fixed pool of threads is generated in the server at the initialization phase to process incoming requests. This pattern ensures that Bold Stroke applications incur low latency and jitter for end-to-end actions [28].

An action has an assigned priority and sub-priority *(importance)* value for every real-time event channel to which it is subscribed. If two actions have the same sub-priority they will be ordered or scheduled non-deterministically according to the configuration. Every action has a *Worst Case Execution Time (WCET)* and a *Best Case Execution Time (BCET)* in the given scenario in which it is used. WCETs and

BCETs are computed by measuring the times corresponding to executing the tasks millions of times in a loop, do not include the time spent waiting to be scheduled, and are assumed to be independent of the scheduling policy. Actions can be initiated by two ways: operation invocations and event propagations. The Bold Stroke scheduling strategy is also configurable – by default its actions are scheduled in accordance with *Rate Monotonic Analysis* [64].

Facet-initiated actions invoked by a remote operation call inherit the QoS execution semantics from the invoking component and do not interact with TAO's runtime scheduler, which resides inside the real-time event channels. We therefore do not distinguish these actions from the invoking action in the scheduling perspective. The smallest unit of scheduling is an event-initiated action together with all the remote operation calls it can invoke. Since facet-initiated actions can also call other actions using remote operation calls, the complete call chain is an acyclic graph, with the event-initiated action as the root element. We call this smallest unit of scheduling an *invocation unit*.

An executing action may initiate actions on other priority bands, which are known as *cross-rate actions*. All processing inside a priority band must finish within the fixed execution period of the timer assigned to the band. This periodicity divides processing into *frames*, which are defined by the rate/period of the timer. For example, a 20 Hz timer will have a 50 ms frame and the overall execution time of the tasks in the timer's rate group must be smaller than 50 ms to fit within the frame. A priority band failing to complete outputs prior to the start of the next frame incurs a so-called *frame overrun* condition, where the band did not meet its completion deadline, *i.e.*, the frame completion time.

5.3 Abstractions Based on the Threading Model

The publish/subscribe architecture used in the the Boeing Bold Stroke execution framework defines two types of mechanisms for data exchange and dependencies.

- Remote method invocations follow conventional two-way function call semantics when a component issues a call from its receptacle to the target component's facet. These two-way facet/receptacle method calls will block if the called process is already executing, which can degrade performance significantly.

- Event propagations provide a more efficient asynchronous data flow semantics from event sources to event sinks supported by event channels. The event channel is built on the *Asynchronous Method Invocation (AMI)* [9] feature of the CORBA specification. The event channel is the implementation of an agent that manages the event passing between tasks. The caller thread A issues the method call on the event channel and resumes its execution, rather than waiting for the called thread B to process the event. When the called thread B finishes the execution it notifies the event channel, which issues the remote method on thread B as thread A's agent.

We assume a time interval for the delivery of the events that is independent for each processor, and is specified by *best case delay* and *worst-case delay*. The event passing between different processors is managed by the *remote* real-time event channel. They are modeled by the `Channel` modeling construct in ALDERIS and the DRE SEMANTIC DOMAIN. Race conditions may occur in remote event passing and are an important issue to be solved by analysis. Event passing on the same processor (same address space) is managed by *local* real-time event channels. They are modeled by the `Buffer` modeling construct in the DRE SEMANTIC DOMAIN. In this scenario the ORB delivers the event by invoking local functions - without marshaling/demarshaling

the requests. This technique is called a *collocation optimization* and is implemented in most CORBA ORBs, including TAO.

Collocated event passing does not follow the strict event passing semantics since a single thread is used to manage both the event sources and event channels. We therefore assume that the event channel notifies every task before the scheduler is invoked, thus there are no race conditions within a thread. This mechanism helps to enforce fixed priority scheduling for tasks that receive events at the same time and are deployed on the same thread.

Method invocations inherit the QoS execution semantics from the invoking task and do not interact with the runtime scheduler. If the caller and invoked tasks are deployed in the same address space of a processor, the invoked task executes within the same thread – the dispatcher (worker) thread – as the caller task. If the two tasks are deployed on separate machines the ORB uses a thread to transparently forward the method call to a task on the remote machine. This mechanism separates the scheduling of the dispatcher (worker) thread – which schedules tasks invoked by event propagations – and the ORB thread – which simply executes remotely invoked tasks.

The threading model described above provides a way to aggressively abstract out remote method calls from the model. For local method calls we simply add the WCET and BCET of the called and caller tasks. For remote method calls, we utilize non-blocking message passing semantics building on the real-time event channel and publish/subscribe communication paradigm using AMI.

Remote blocking function calls cannot be trivially expressed in ALDERIS, and should be avoided by design. If that is not possible, one option is to add the worst-case delay of the channel to the WCET of the call chain, and the best-case delay of the channel to the BCET of the call chain, and assume that the call chain executes as a single task. This approach assumes that the ORB thread is always ready to

serve requests. However, this assumption is overly optimistic in some cases, and therefore only applicable to soft real-time systems. Designers have the option of adding more ORB threads or let the ORB's *Portable Object Adapter (POA)* manage dynamic number of threads in the thread pool, but this approach does not guarantee hard real-time deadlines.

5.4 Non-preemptive Boeing Bold Stroke Application

To illustrate the model-based verification capabilities of DREAM, we examine a case study of a DRE system from the domain of avionics mission computing. Figure 5.4 shows the component-based architecture of this system, which is built upon the Boeing Bold Stroke real-time middleware described in Section 5.2. This application is deployed on a non-preemptive multi-processor platform.

As shown in Figure 5.3, this application is driven by five `Timer` components deployed on five CPUs. The `GPS` and `AIRFRAME` components are deployed on `CPU_1`. When the `1Hz Timer` component pushes an event the `GPS` component will be notified, and scheduled for execution by the OS (operating system) scheduler. The `GPS` component then pushes an event to the `AIRFRAME` component. The OS scheduler schedules the `AIRFRAME` component for execution, which calls back to the `GPS` component's facet using its receptacle to get the actual data required for execution. The `AIRFRAME` component pushes an event to each of the `NAV_STEERING`, `ROUTES`, `TACTICAL_STEERING`, and `NAV_DISPLAY` components. Since these components are deployed on different processors they are presented in lighter colors in the `1Hz Timer`'s band.

Computations on different processors are driven by their respective timers. Components do not necessarily execute with the timer's rate, however, as seen in the `NAV_DISPLAY` component's case. It is executed more often to serve remote requests

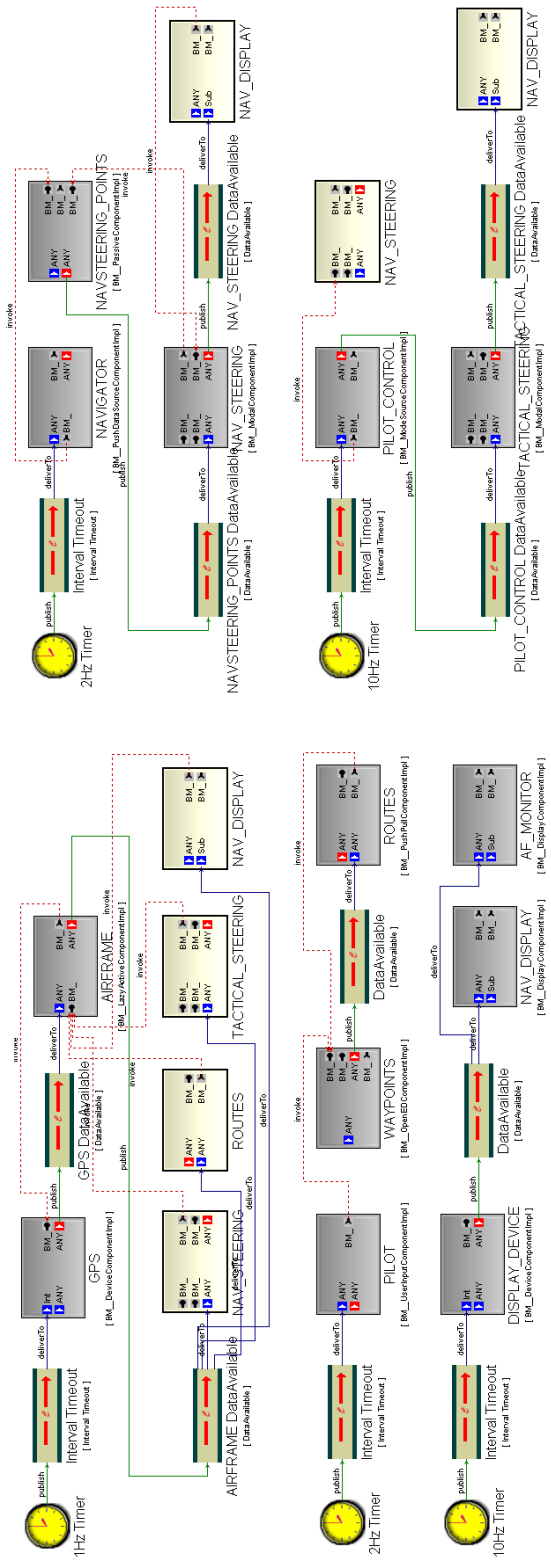

Figure 5.3: The Bold Stroke Application Model

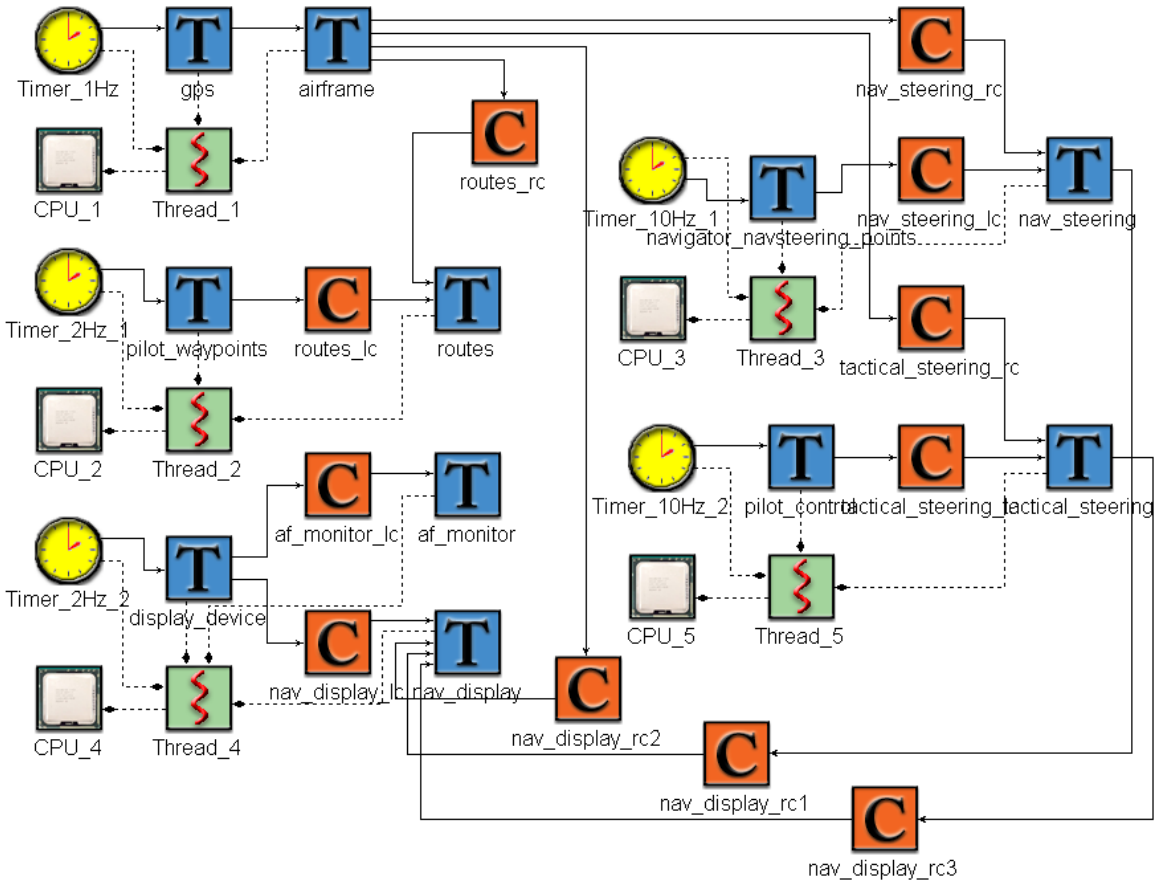

Figure 5.4: The ALDERIS Model of the Real-time CORBA Avionics Application

than to serve local requests on `CPU_3`.

Figure 5.4 shows the ALDERIS model of the real-time CORBA avionics application. The ALDERIS model does not capture blocking function calls in software. As described in Section 5.3, ALDERIS models *(1)* remote function calls as message passing using AMI, and *(2)* local method invocations are simply added to the caller task, since the two functions execute as a single task on a single thread. The `Channel` construct in ALDERIS models the CORBA event channel construct, which is practically a *First In First Out (FIFO)* that allows non-blocking message passing between tasks. We observe the following key challenges in the ALDERIS model shown in Figure 5.4:

- *Event flow, buffering.* Event propagations require buffering of the events (*i.e.*, for the `AIRFRAME` component) and concurrency management between event

channels that are publishing to the same component (*i.e.*, between the event channels that publish events to `AIRFRAME`).

- *Delays.* Communication between processors incur delays in the message propagation. Since the delays are not constant, race conditions may occur when a lower priority task receives an event earlier than a higher priority task, which can result in priority inversion.

- *Composition.* The problems above can be summarized as composition challenges, *i.e.*, *the schedulability of individual threads does not guarantee the overall schedulability of the system.*

5.5 Real-time Verification by Timed Automata Model Checking

We have used the open-source DREAM tool to generate the *Timed Automata (TA)* representation of the ALDERIS model shown in Figure 5.4. DREAM supports both the UPPAAL and Verimag IF model checkers, as discussed in Chapter 10. In this chapter we describe results using UPPAAL, but the results can be generalized to any TA model checker. The resulting TA representation can be checked for deadlock-freedom, bounded buffer sizes, and whether all deadlines are met.

In addition to model checking, UPPAAL provides built-in support for manual and automatic simulation. To improve efficiency, the model checking algorithms in UPPAAL are based on clock constraints equivalence rather than state equivalence. Systems in UPPAAL are modeled as a slightly modified variant of TA [12] as discussed in Section 4.3, and the specification is expressed in a restricted version of the *Timed Computational Tree Logic (TCTL)* [5], which is a temporal logic that can formalize

Component	CPU	Sub-priority	WCET	BCET	Deadline
GPS	CPU_1	VERY_HIGH	21	14	22
AIRFRAME	CPU_1	HIGH	53	33	54
PILOT_WAYPOINTS	CPU_2	VERY_HIGH	37	12	38
ROUTES	CPU_2	VERY_LOW	18	13	19
NAVIGATOR_NAVS...	CPU_3	VERY_HIGH	32	22	65
NAV_STEERING	CPU_3	HIGH	49	27	50
DISPLAY_DEVICE	CPU_4	VERY_HIGH	26	18	41
AF_MONITOR	CPU_4	HIGH	33	14	34
NAV_DISPLAY	CPU_4	MEDIUM	14	9	74
PILOT_CONTROL	CPU_5	VERY_HIGH	23	19	66
TACTICAL_STEERING	CPU_5	HIGH	58	47	59

Channel	Buffer Size	WC Delay	BC Delay
nav_steering_lc	2	0	0
routes_lc	2	0	0
tactical_steering_lc	2	0	0
nav_display_lc	2	0	0
af_monitor_lc	2	0	0
nav_steering_rc	2	2	1
routes_rc	2	2	1
tactical_steering_rc	2	3	1
nav_display_rc1	2	3	1
nav_display_rc2	2	3	1
nav_display_rc3	2	2	1

Table 5.1: Parameters for the Bold Stroke Application Shown in Figure 5.4

statements about system models. The UPPAAL semantic domain combines TA with dataflow semantics that can be used to express interactions between the automata.

Figure 5.5 and Figure 5.5 show the UPPAAL TA representation of avionics DRE case study shown in Figure 5.4, and demonstrates how the DRE SEMANTIC DOMAIN can represent DRE systems for compositional real-time analysis. The application consists of 11 computation tasks and 11 event channels, of which 5 are local, and used only for buffering. The application is deployed on 5 processors. The `Timer` components are a simple rate generators which publishes events at a predefined rate. We model them using `Timers` in the DRE SEMANTIC DOMAIN.

To satisfy real-time constraints and avoid unnecessary thread spawn delays, the

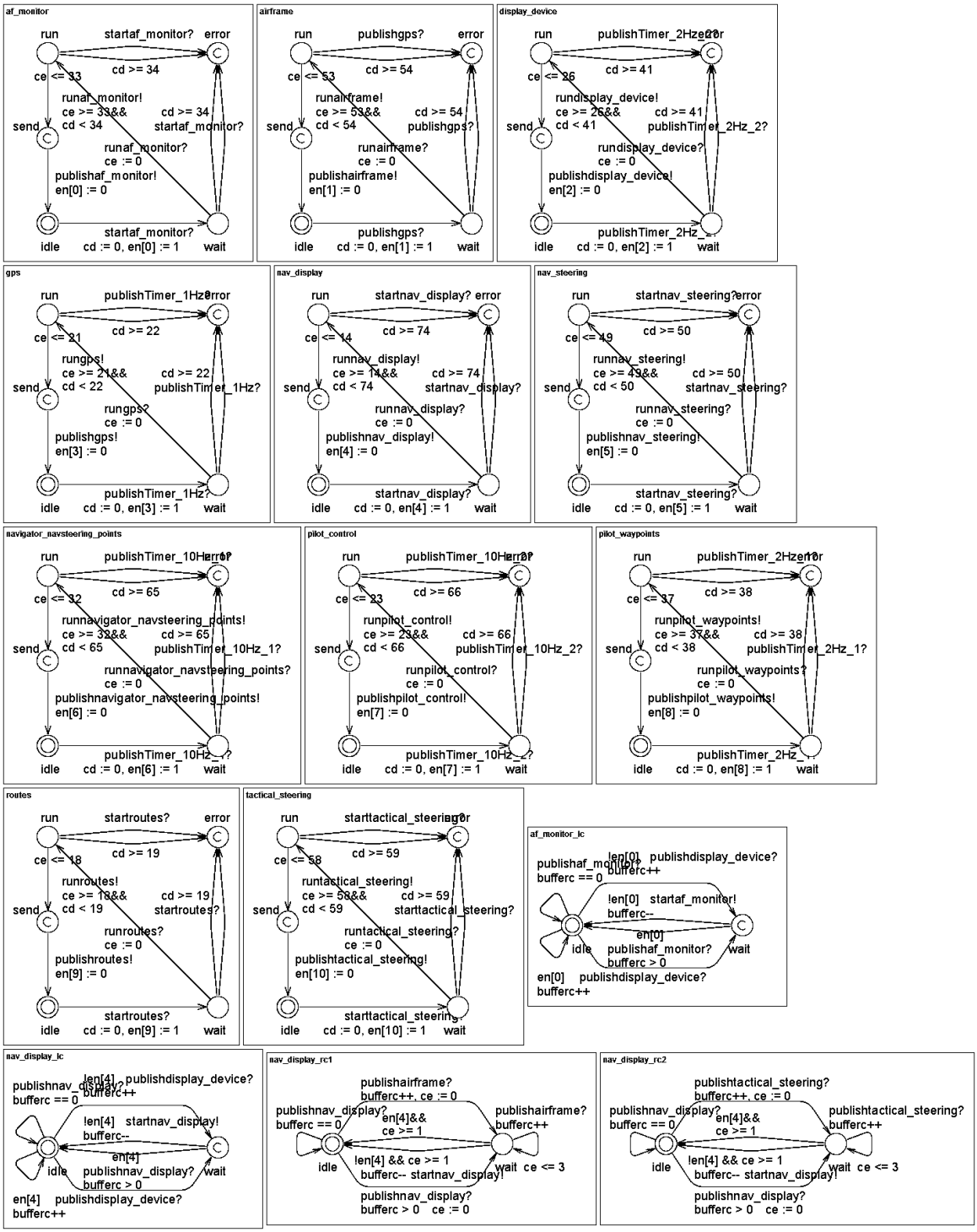

Figure 5.5: Uppaal Timed Automata Models for the Avionics Application Shown in Figure 5.4 (Part 1/2)

Figure 5.6: Uppaal Timed Automata Models for the Avionics Application Shown in Figure 5.4 (Part 2/2)

PRISM component middleware requires dedicated threads for each real-time event channel. In the DRE SEMANTIC DOMAIN, however, we can abstract out some of these threads to reduce the number of event channels and thus the state space. We have to model event channels explicitly *(1)* when we have to buffer events or *(2)* on remote event channels which have measurable delays. All the event channels satisfy one of the above conditions, except the timer's event channels that have been abstracted out in the model.

The scheduling policies are represented by `Schedulers` in the DRE SEMANTIC DOMAIN. We define 5 schedulers since the Bold Stroke application is deployed on a 5-processor architecture. The schedulers get more complex according to the scheduling policies. The automatic generation of the models provides a safe way to ensure the correct guard conditions and assignments.

The TA model shown in 5.5 corresponding to the Bold Stroke system shown in 5.4 has been shown to be schedulable by UPPAAL. We have checked the system for deadlocks and missed deadlines by using the following UPPAAL macro:

```
A[]  not  deadlock
```

This UPPAAL macro checks that eventually every task does not deadlock, *i.e.*, reach a state from which no transition is possible and time cannot progress. As discussed in Section 4.3, we modeled the `error` locations of tasks as `committed`, and therefore the absence of a deadlock implies the schedulability of the model.

If the above reachability macro evaluates to `true`, therefore, we have proved that there are no deadlocks in the system and every action always finishes its execution before the deadline. We also prove that every published event is consumed properly in the system and the event channels operate with limited buffer size. We have checked whether the system operates with finite buffer sizes with the TCTL formula:

```
A[]  (Channel.bufferc < Channel.lambdac)
```

UPPAAL produces a counter-example for invalid properties, which helps identifying the source of undesired behavior. Finally, we checked that eventually every task will execute using the formula:

```
E<>  Task.executing
```

The performance of the verification depends largely on the number of *non-deterministic branches* in the event flow, not the number of components. The ALDERIS model shown in Figure 5.4 and using the parameters shown in Table 5.1 can be analyzed in 164 seconds with ∼770 000 KB memory consumption on an Intel Core i7 920 processor running at 4GHz, using 6GB three-channel memory. Not only is the model sensitive to dependencies between tasks, but also the actual execution parameters.

In earlier work [67] we assumed constant execution times for tasks; when the BCET times equal the WCET times. In this case significant performance speedup can be observed, as the model can be verified in less than a second with 8 264K memory consumption. However, as we described in Section 5.1, it is insufficient to focus on worst-case time analysis in *Asynchronous Event-driven Distributed Real-time Embedded (AEDRE)* systems; execution intervals have to be captured in continuous time for the real-time verification.

To improve the analysis performance, designers must aim to ensure deterministic scheduling and behavior for critical system tasks. Our experiments confirm that the complexity grows exponentially with respect to the state space size. Finding the right abstraction is therefore crucial for tractable verification problems.

5.6 Concluding Remarks

This chapter presented a novel method for deciding the schedulability of non-preemptive DRE systems. We presented an approach for the modeling of software-

intensive DRE systems using the ALDERIS DSML, and utilized the open-source DREAM framework to generate a *Timed Automata (TA)* representation of the ALDERIS models. The proposed method can capture and verify properties of non-preemptive, event-driven component-based DRE systems that use the publish/subscribe communication pattern. The verification is automatic, exhaustive, and capable of producing counter-examples that help pinpoint sources of undesired behavior.

We have applied the proposed method to the Boeing Bold Stroke avionics mission computing platform, which is representative of state-of-the-practice DRE systems based on QoS-enabled component middleware. The goal of this chapter was to verify QoS properties that express the *behavior* of this DRE system, such as end-to-end deadlines, graceful degradation, and dependability. Key contributions of this chapter are as follows:

- We formalized the problem of deciding the schedulability of DRE systems utilizing fixed priority scheduling using TA.
- We described the Boeing Bold Stroke execution framework, and described an approach for the modeling of multi-threaded software intensive DRE systems by utilizing abstractions based on the threading model.
- We introduced a case study loosely based on the Boeing Bold Stroke execution platform, and described a model checking method based on TA for the real-time verification of non-preemptive DRE systems.

We extend the results of this chapter to preemptive DRE systems in Chapter 7. Extending the DREAM framework is a key part of our future work, which focuses on expressing the formal, heterogeneous composition of semantic domains to support better and more robust DRE systems development. The open-source DREAM implementation is available for download at `http://dre.sourceforge.net`.

Chapter 6

Performance Estimation of Distributed Real-time Embedded Systems by Discrete Event Simulations

This chapter describes how we applied the *Analysis Language for Distributed, Embedded, and Real-time Systems (*Alderis*) Domain-specific Modeling Language (*DSML*)* defined in Section 4.1 and the DRE Semantic Domain specified as a *Discrete Event (*DE*)* system in Section 4.4 for the formal performance analysis of *Distributed Real-time Embedded (*DRE*)* systems by *Discrete Event Simulations (*DES*)*. This method is applicable to non-preemptive systems only, but can be composed with the real-time verification of preemptive systems, such as the method introduced in Chapter 7. Using the DRE Semantic Domain to specify the operational semantics of the Alderis DSML we defined a formal *Model of Computation (*MoC*)*, that we refer to as DRE MoC.

Performance evaluation is a key challenge in the analysis of DRE systems. Major design parameters that influence performance include real-time properties, such as task execution times and communication delays, the degree of parallelism in computations, and the throughput of the communication architecture.

This chapter proposes a *Discrete Event Simulation (*DES*)*-based performance evaluation method for DRE systems, that employ *fixed-priority scheduling*. We introduce a formal model for DRE systems based on discrete event scheduling [21] using the concept of *logical execution time* [44], and the *Event Order Tree (*EOT*)* shown in Section 6.2. Nodes in the EOT represent events, and edges represent causality between the events. As events may arise non-deterministically, the tree may branch when different event orderings are possible. The proposed model explicitly captures the flow of data and communication effects (such as non-deterministic delays etc.) in

event-driven systems for *dynamic performance evaluation*.

In the proposed approach we do not store *timed states*, like *Timed Automata (TA)* model checking methods, just events and constraints on the (global) timestamps of real-valued events. Note that this approach represents real-time properties in *continuous time*. Storing timed states is the most significant contributor to memory consumption in model checking tools. The proposed method has minimal memory requirements, providing a way for runtime on-the-fly analysis in adaptable DRE systems.

In this stage of development we do not address the termination problem, as we do not try to identify previously visited timed states, but use a constant horizon as a time limit for the analysis. There are model checking methods that do not have this limitation in theory, but in practice all model checking methods suffer from the termination problem due to the state space explosion problem. Our preliminary results show that the proposed DES-based evaluation method can achieve *better coverage in large-scale DRE systems* than alternative methods as shown in Section 6.3. The abstract symbolic model allows *better simulation performance* compared to the actual simulations with comparable accuracy, providing an efficient method for design space exploration. This chapter has the following key contributions:

- We formalize the problem of estimating the end-to-end performance of DRE systems utilizing fixed priority scheduling using DES in Section 6.1.

- We describe a novel approach for the formal performance estimation of non-preemptive *Asynchronous Event-driven Distributed Real-time Embedded (AE-DRE)* systems based on DES in Section 6.2. This method is applicable to large-scale designs, and can provide partial results in case the models are too large for exhaustive analysis.

- Section 6.3 presents a large-scale avionics DRE case study and performance

comparisons to alternative methods, such as simulations and TA model checking.

6.1 Problem Formulation

We utilize the DRE MoC specified as a DE system in Section 4.4 to define the performance evaluation problem and schedulability problem for non-preemptive DRE systems. We use the notation for the sequence of input events of a task t_k as $I_k = \{i_{k0}, i_{k1}, \dots\}$, the sequence of output events as $O_k = \{o_{k0}, o_{k1}, \dots\}$, $t \in T, i_{k0}, i_{k1}, \cdots \in E, o_{k0}, o_{k1}, \cdots \in E$. We specify deadlines for each computation task t_k using the mapping $d_k : T \rightarrow \mathbb{N}$. Deadlines for each task and each input event are counted from the timestamp of the input (enabling) event ($\mathsf{time}(\mathrm{i}_k)$).

Definition 6.1 (Schedulability): *A computation task $t_k \in T$ is schedulable if it always finishes its execution before its respective deadline. The DRE MoC is schedulable if all tasks are schedulable. We formalize this condition using the DRE MoC as follows:* $(\forall t_k \in (T \setminus C \setminus TR))(\forall i_k \in I_k)(\forall o_k \in O_k)\ \mathsf{time}(o_k) < \mathsf{time}(i_k) + d_k$.

Definition 6.2 (Run): *A* run *or* execution trace *of the DRE MoC is the chronological sequence of events occured in the model. A run is* valid *if it is schedulable, that is if for all input (i_k) and their corresponding output (o_k) events in the execution trace* $\mathsf{time}(o_k) < \mathsf{time}(i_k) + d_k$, *otherwise it is* invalid.

Definition 6.3 (End-to-end computation time): *We define the end-to-end computation time between an input event i_{jn} of task t_j and an output event o_{km} of task t_k as the maximum possible difference between the events' timestamps along all the possible runs of the model* $\mathsf{end\text{-}to\text{-}end}(o_{km}, i_{jn}) = \mathsf{max}[\mathsf{time}(o_{km}) - \mathsf{time}(i_{jn})]$, *if* $\exists\{(t_j, t_a), (t_a, t_b), ..., (t_b, t_j)\} \in D$. *If task t_k does not depend on task t_j in the DRE MoC* ($\nexists\{(t_j, t_a), (t_a, t_b), ..., (t_b, t_j)\} \in D$), *we define* $\mathsf{end\text{-}to\text{-}end}(o_{km}, i_{jn}) = \infty$.

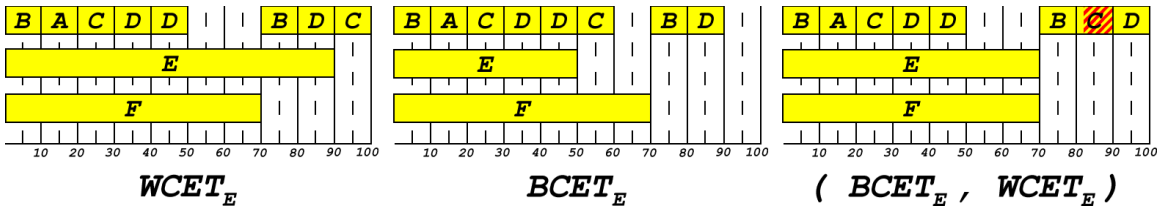

Figure 6.1: Execution Traces of the DRE Model Shown in Figure 4.1

Task	t_A	t_B	t_C	t_D	t_E	t_F
bcet	10	10	10	10	50	70
wcet	10	10	10	10	90	70
deadline	22	25	12	32	100	100

Table 6.1: Timing Information for the DRE Model Shown in Figure 4.1

It is a fact that the product of local worst case execution times does not necessarily result in worst case end-to-end computation times. We now demonstrate this problem in non-preemptive DRE systems to motivate our approach for formal performance evaluation using the simple DRE example shown in Figure 4.1. We define the period of each timer to be 100 time units, and the delay of each channel to be 0.

Figure 6.1 illustrates the first period of DRE model execution traces shown in Figure 4.1 using the parameters in Table 6.1. In this example, for most tasks the Best Case Execution Time bcet_k time equals the Worst Case Execution Time wcet_k time to reduce complexity, for easier illustration. We utilize fixed-priority scheduling in `machine_1` between tasks t_A, t_B, t_C, and t_D. Tasks t_E and t_F are executed concurrently and have their own schedulers. Note that *Earliest Deadline First (EDF)* scheduling would result in a deadline miss by task t_B, as it scheduled the sequence t_A, t_C, t_B. This illustrates that EDF is not optimal in the non-preemptive DRE MoC.

The execution trace in the left of Figure 6.1 demonstrates that the system is schedulable if all tasks execute using their *Worst Case Execution Time (WCET)*. The trace in the middle of Figure 6.1 shows that the system is schedulable when *Best Case Execution Time (BCET)* are considered during the execution trace. However, the trace in the right of Figure 6.1 shows that task t_C might miss its deadline if task t_E executes

for 71 time units. This example shows that the performance evaluation of event-driven non-preemptive DRE systems has to consider *execution intervals* rather than worst case execution times, and justifies the need for formal performance analysis.

6.2 Performance Estimation of DRE Systems by Discrete Event Simulations

This section describes the proposed DES-based performance evaluation method for event-driven DRE systems expressed using the DRE MoC. We introduce the *Event Order Tree (EOT)* and show how it can be utilized for performance estimation.

6.2.1 Event Order Tree

Definition 6.4 (Equivalent execution traces): *In the DRE MoC two execution traces are equivalent, if the two execution traces contain the same events, and the chronological order of events in both execution traces is the same.*

Note that for equivalence only the order of events have to be the same, not the timestamps of events (untimed equivalence). We propose a directed tree representation for the valid traces of a DRE model, called *Event Order Tree (EOT)*. Each node in the EOT represents an event and the (global) time constraint on the current event's timestamp. The path from the root of the EOT to a node represents possibly infinite number of equivalent execution traces of a DRE model. There is a directed edge from node A to node B if event B may be raised after event A, and there are no other events between them.

Figure 6.2 shows the EOT for the DRE model shown in Figure 4.1 (thicker borders explained in Section 6.3). Computations in the model are triggered by the timers, that generate events i_1, i_2, i_5, i_6, therefore we label the root as $\mathbf{i}_1\mathbf{i}_3\mathbf{i}_5\mathbf{i}_6$. All these events are

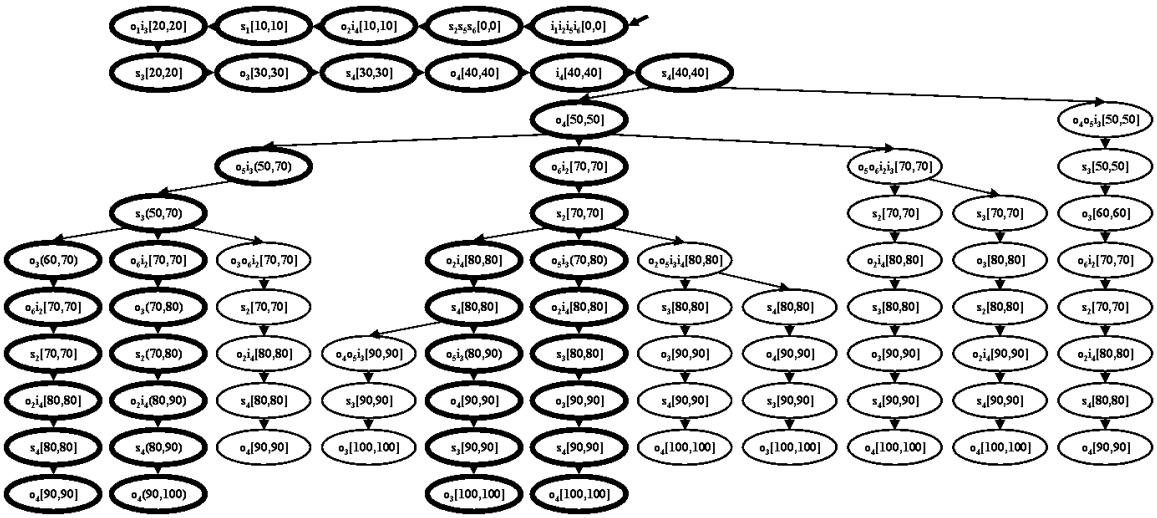

Figure 6.2: The Event Order Tree of the DRE Example in Figure 4.1 using the Parameters in Table 6.1

generated at (global) time 0, therefore we represent the constraint on their timestamps as **[0, 0]** in the root. The schedulers trigger the execution of tasks t_B, t_E and t_F by generating the s_2, s_5, s_6 events. We label the immediate child of the root in the EOT as $s_2 s_5 s_6$. Scheduling tasks for execution after they become enabled is instantaneous in our discrete event scheduler, therefore the time constraint on the timestamps of events $s_2 s_5 s_6$ remains $[0, 0]$. Although the time constraints in the root and the node marked as $s_2 s_5 s_6$ are identical, there is a causal ordering between them in the DES model; a task can only be scheduled for execution after it has received an input event. The causal orderings between events with the same timestamp correspond to zero-time transitions in other MoCs, such as TA.

Each path in the EOT from the root to the leaves imposes constraints on the execution intervals of tasks by defining constraints on the timestamps of events. For example, the path from the root to the leftmost leaf in the same tree requires task t_E to finish its execution after task t_D has finished its execution, and before task t_F finishes it execution in the $(50, 70)$ interval as shown by the constraint $\mathbf{o_5 i_3 (50,\ 70)}$ in the EOT. Therefore, the leftmost path in the EOT shown in Figure 6.2 restricts the

execution time of task t_E to $(50, 70)$.

Definition 6.5 (Branching intervals): *We refer to intervals implied by equivalent execution traces as* branching intervals. *Branching intervals are always subsets of the execution intervals* $[\mathsf{bcet}_k, \mathsf{wcet}_k]$ *of tasks.*

For example, in the case of task t_E its three branching intervals are: $[50, 50]$, $(50, 70)$, $[70, 70] \subset [50, 70]$, as shown by the constraints $\mathbf{o}_4\mathbf{o}_5\mathbf{i}_3$**[50, 50]**, $\mathbf{o}_5\mathbf{i}_3$**(50, 70)**, $\mathbf{o}_5\mathbf{o}_6\mathbf{i}_2\mathbf{i}_3$**[70, 70]** in the EOT.

6.2.2 Branches in the Event Order Tree

The execution of the DRE model on `machine_1` is deterministic, tasks execute in the order t_B, t_A, t_C, t_D, t_D, while tasks t_E and t_F execute on `machine_2` and `machine_3` in parallel. We reach the first non-deterministic choice in the ordering of events after task t_D starts executing for the second time within the period; if task t_E executes for its bcet then tasks t_D and t_E finish their execution simultaneously (constraint $\mathbf{o}_4\mathbf{o}_5\mathbf{i}_3$**[50, 50]**); otherwise task t_D finishes its execution first (constraint $\mathbf{o}_4$**[50, 50]**), and then task t_E finishes its execution. To capture this non-determinism the EOT has to branch at node $\mathbf{s}_4$**[40, 40]**. In the DRE MoC we also consider race conditions between tasks assigned to the same machine only if they receive events from tasks assigned to other machines.

Definition 6.6 (Race condition): *If for two tasks* $t_k, t_j \in T$, $\mathsf{machine}(t_k) = \mathsf{machine}(t_j)(\exists i_k \in I_k, i_j \in I_j)(\mathsf{time}(i_k) = \mathsf{time}(i_j))$, *and (*$\mathsf{machine}(t_s) \neq \mathsf{machine}(t_k) \vee \mathsf{machine}(t_r) \neq \mathsf{machine}(t_j), t_s \in T, t_r \in T, \{t_s, t_k\} \in D, \{t_r, t_j\} \in D$ *then there is a* race condition *between task* t_k *and* t_j.

For example, if tasks t_E and t_F finish at the same time there is a race condition between tasks t_B and t_C. We identify race conditions the following way: whenever

a set of tasks receives events with the same timestamp we check whether the tasks that generated that event are assigned to the same machine as the set of tasks. If not, race conditions may be present. If race conditions are present between a set of tasks we have to consider each task to receive its respective event first, therefore in these cases the EOT has to branch for each task.

Consider the node $\mathbf{o}_5\mathbf{o}_6\mathbf{i}_2\mathbf{i}_3$**[70, 70]** in the EOT shown in Figure 6.2. This node represents the execution trace where tasks t_B, t_A, t_C, t_D, t_D execute in this order in `machine_1`, and tasks t_E and t_F finish their execution at the same time. The EOT branches and we consider both the case when task t_B receives its start even first ($\mathbf{s}_2$), or when task t_C receives its start event first ($\mathbf{s}_3$) due to race conditions.

Definition 6.7 (Branching point): *We refer to nodes in the tree, where the EOT branches due to non-deterministic execution times, or race conditions as* branching points.

6.2.3 Real-time Verification by Discrete Event Simulations

In this subsection we propose a method for the real-time verification of a large class of DRE models using the EOT. The EOT is a symbolic representation of all distinguishable execution traces of the DRE model from a timing perspective as we show in this section. We build on the results of this section to propose a method for the on-the-fly construction of the EOT in Subsection 6.2.4, providing a way for formal performance evaluation with a systematic measurement of state space coverage.

Timers in the DRE MoC introduce periodicity in the models. Since the DRE MoC allows the use of multiple timers with different periods we need to find the least common multiplier of timer periods, which we refer to as *time limit*. We make the restriction that all tasks have to be in the `init` state when events timestamped with

the time limit are generated.

$$\forall(t_k \in T \setminus C \setminus TR)\ \mathsf{state}(t_k, time\_limit) = \texttt{init} \tag{6.1}$$

This restriction is sufficient, but not necessary for a schedulable DRE model, as in pipeline architectures the processing of older events may overlap with the processing of newer events at different stages in the pipeline, therefore we may not reach a condition where all tasks return to the `idle` state at once. In pipelined systems we can either verify the system to a limited horizon – which does not guarantee that the system will work properly after the time limit – or use other model checking techniques on the DRE MoC, such as TA, as described in [67, 65, 66]. In the rest of this section we show that if Equation 6.1 is satisfied, we can verify DRE systems by exhaustively enumerating all the execution traces corresponding to all paths in the EOT from the root to the leaves.

Theorem 6.1 (Repeatable property): *The EOT of a given DRE model repeats itself from all its leaves. We refer to this property of EOTs as* repeatable.

Proof (outline): We build on Equation 6.1 that tasks have to be in their initial states (`init`) when the time limit is reached, that is the least common multiplier of timer periods. The timers generate the same events that have appeared in the root, with the timestamps of the time limit. Since there is no relative difference between the timestamps of events generated by the timers the DRE model can exhibit the same execution traces as before. □

We only build the EOT until we reach the time limit on the timestamps of events. For example, the EOT shown in Figure 6.2 repeats itself from all leaves. It is important to note that even though a DRE system may utilize several timers with different periods there is only one EOT, rather than a forest. New events generated by faster timers are considered as branching points or are simply appended to the leaves if no

tasks are running when the timer generates a new event.

Theorem 6.2 (Finite number of nodes): *There is a finite number of nodes in the* EOT.

Proof (outline): In the DRE MoC a finite number of events are generated with timestamps within any interval, because *(1)* we only consider the boundaries of intervals, *(2)* timers generate events at discrete time steps, and (3) each task generates finite number of output events for each input event. Therefore, each branch has a finite number of children in the EOT. There is a finite number of branches – at most one for each executing tasks, and one for each enabled task that may be in race conditions. Since there is a finite number of branches and each branch has a finite number of children, the EOT has a finite number of nodes. □

Theorem 6.3 (Worst case execution trace): *We define the* worst case execution trace *of equivalent execution traces as the execution trace where tasks produce output events with maximum value timestamps from their branching intervals (as introduced in Definition 6.5). If the worst case execution trace of equivalent execution traces is valid, then all equivalent execution traces are valid.*

Proof (outline): The order of events in equivalent execution traces is fixed. Therefore, none of the tasks is forced to wait for longer when the execution times of some tasks are decreased, than when execution times are left unchanged. None of the tasks generate events with timestamps larger than in the worst case execution trace, otherwise the ordering of events would change. If the worst case execution trace is valid, then all equivalent traces are valid, since tasks within those execution traces generate events with timestamps less than or equal to the worst case execution trace, therefore they do not violate their deadlines. □

We have shown that the EOT has a finite number of leaves, therefore DRE models have finite number of equivalent execution traces. We have also shown that the real-

time properties of equivalent execution traces can be verified using a single discrete event simulation. The set of paths in the EOT from the root to the leaves gives all the possible equivalent execution traces of a DRE model. The exhaustive discrete event simulation of all the possible equivalent execution traces in the EOT of a DRE model consists of a finite number of discrete event simulations, therefore it is a valid method for the real-time verification of DRE models that satisfy Equation 6.1.

6.2.4 On-the-fly Detection of Branching Points in the Event Order Tree

This subsection describes how we can detect branching points at runtime, providing a way for the on-the-fly construction of the EOT. By enumerating all execution traces corresponding to the paths from the root of the EOT to the leaves, we can estimate the system's performance with 100% coverage. However, the exhaustive analysis of large-scale DRE systems is most often infeasible in practice due to the state space explosion problem. Therefore, in most practical scenarios we cannot build the whole EOT in advance due to storage constraints, and we can only enumerate some paths of the EOT due to time constraints. In these cases we obtain results using a partial state space search, and therefore we cannot guarantee their correctness. We can, however, achieve better coverage and confidence than with the existing methods, as shown in Section 6.3.

There are two major ways for building and analyzing the EOT. The first option is to build the EOT in a *Breadth First Search (BFS)* fashion. The BFS-based approach stores the EOT in the memory, and iteratively build the tree from the leaf-candidates. This approach requires that we store timing information (and times states) for all leaf-candidates of the EOT in order to quickly restore the timed state of the system corresponding to the actual leaf-candidates, and check for deadlines and end-to-end computation times. The BFS-based approach has significant memory overhead, and resembles an exhaustive model checking method.

Algorithm 6.1 Obtaining and Enumerating the Event Order Tree by Discrete Event Simulations

1: create the (empty) superset of race conditions R
2: set the execution time for all tasks $t_k \in T$ to their wcet_k time, and the next execution time for all tasks to their bcet_k time, respectively ($\forall t_k \in T$ $\mathsf{exec\_time}_k$ = wcet_k, $\mathsf{next\_time}_k$ = bcet_k)
3: *// enumerate all branching intervals*
4: **for all** permutations of $\mathsf{exec\_time}_k$ assignments, obtained using the $\mathsf{next\_time}_k$ variables **do**
5: clear the superset R
6: **call** discrete_event_simulation () described in Algorithm 6.2
7: *// enumerate all race conditions with the current* $\mathsf{exec\_time}_k$ *assignments*
8: **for all** permutations of events in superset R **do**
9: **call** discrete_event_simulation () described in Algorithm 6.2
10: **end for**
11: **end for**

In this section we propose a *Depth First Search (DFS)*-based approach to obtain the EOT. The DFS-based approach has minimal memory overhead, as it does not store the EOT in the memory. We detect branching points in the EOT during simulation traces, and then use this information to direct the discrete event scheduler to iteratively explore unique paths in the EOT.

Note that although there are several model checkers that implement a BFS-based or/and a DFS-based search algorithm to enumerate symbolic state spaces, they are optimized to check simple properties using some logic – such as *Linear Time Logic (LTL)* [75] or *Computational Tree Logic (CTL)* [25]. For the evaluation of embedded systems designers often want to find the maximum/minimum value of certain design parameters, or simply check whether the system performance gets better or worse when they change a design parameter. These conditions often require multiple model checker runs in order to translate these properties into a set of yes/no questions, which becomes impractical, cumbersome, and time consuming.

The key problem that we need to address is to detect branching points at run-time, and then exploit this information to construct new directed simulation traces in the future that enumerate traces representing unique branching intervals. We now

describe a simple and practical approach to address this problem.

In our implementation all events are globally observable, and *each task detects its own branching intervals*. As we discussed in Definition 6.5, branching intervals are always subsets of the execution intervals [bcet_k, wcet_k] of tasks, since the order of events in an execution trace can only change if an event is raised earlier/later than another event. Moreover, all branching intervals represent different orderings of events, therefore we only need to consider events within the [bcet_k, wcet_k] intervals of tasks to detect all branching points. Also, as described in Section 6.2.2, we consider race conditions in the models as well. During a single execution trace, whenever we encounter a race condition between events using Definition 6.6, we add the events in a set, and then add the set to the superset containing all sets of race conditions. For each event in a set we need to consider the possibility that it is executed first due to a race condition, and we need to consider all permutations between the sets in the superset. *We detect all events on-the-fly during simulations, and check all their possible permutations at the boundaries of branching intervals*, therefore we enumerate all the paths in the EOT. Algorithm 6.1 describes our algorithm for generating all permutations of events by iterative simulations. Note that we do not set all tasks' execution times to their next_exec time simultaneously, rather we generate all permutations.

6.3 Practical Application to Software-Intensive DRE Systems

In this section we evaluate the proposed DES-based performance estimation method as implemented in the open-source *Distributed Real-time Embedded Analysis Method (*DREAM*)* tool. Figure 6.3 shows the first case study used for the performance estimation. The DRE model is loosely based on a real-time *Common Object Request Broker Architecture (CORBA)* avionics application implemented in the Boeing Bold Stroke execution framework described in Section 5.2.

Algorithm 6.2 function discrete_event_simulation ()

1: run directed discrete event simulation, during which each task stores its start time as start_k
2: during the simulation all tasks t_k observe events e_i that are raised in the [start_k + bcet_k, start_k + $\mathsf{exec\_time}_k$] interval
3: **if** start_k + $\mathsf{next\_time}_k$ < $\mathsf{time}(e_i)$ **then**
4: *// we have encountered a branching point in the (*bcet_k, wcet_k*) interval*
5: store the value of $\mathsf{time}(e_i)$ - start_k in the $\mathsf{next\_time}_k$ variable
6: **else**
7: do nothing, event will be considered in subsequent simulations
8: **end if**
9: *// find all race conditions with the current* $\mathsf{exec\_time}_k$ *assignments*
10: **for all** race conditions detected between events $e_i, e_j, \ldots e_k$ during the simulation **do**
11: search for the set containing events $e_i, e_j, \ldots e_k$ in superset R (from Algorithm 6.1)
12: **if** the set is found **then**
13: do nothing
14: **else**
15: add the set $S = \{e_i, e_j, \ldots e_k\}$ to the superset R
16: **end if**
17: **end for**

The model consists of 98 tasks (including channels and timers) and 57 dependencies between tasks. Execution parameters for timers and channels are given in Table 6.2, and for computation tasks in Table 6.3.

Our approach for the modeling of the Bold Stroke framework is described in [67]. The case study shown in Figure 6.3 is described in [69], and is distributed with the open-source DREAM tool.

As described in Section 2.3, existing methods for real-time analysis and performance estimation have limited use in non-preemptive event-driven asynchronous systems. Static schedulability methods are not directly applicable to this scheduling model as demonstrated in Section 6.1, and are often overly conservative, limiting the accuracy of performance estimation results. Simulations capture dynamic effects in DRE systems, providing better accuracy, but the coverage of simulations is hard to measure. Model checking, on the other hand, provides a way for exhaustive

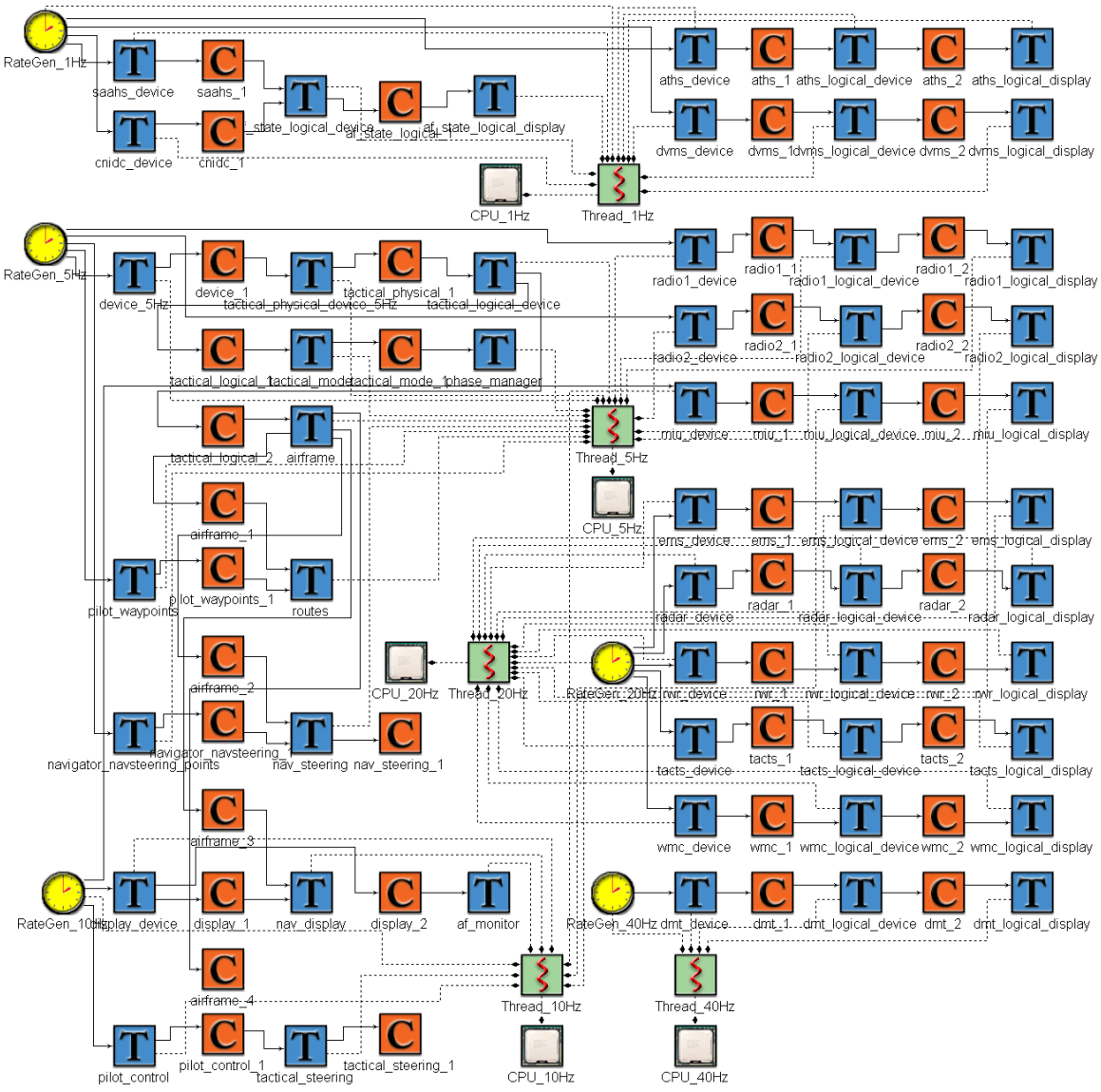

Figure 6.3: Mission-critical Avionics DRE System Case Study

analysis, but the abstractions required to prevent the state space explosion problem often result in decreased accuracy. The proposed DES-based performance estimation method combines model checking with simulations, therefore we compare the DES-based performance estimation results to random simulations, as implemented in the DREAM tool, and to TA model checking methods implemented in the UPPAAL model checker [26], and the Verimag IF toolset [15].

We focused on the analysis of two properties in our experiments; *(1)* we measured

Timer	Period
RateGen_1Hz	10000
RateGen_5Hz	2000
RateGen_10Hz	1000
RateGen_20Hz	500
RateGen_40Hz	250

Channel	WCDelay	BCDelay
aths_1	3	3
aths_2	5	5
dvms_1	6	6
dvms_2	5	5
radio1_1	8	8
radio1_2	6	6
radio2_1	5	5
radio2_2	7	7
miu_1	4	4
miu_2	8	8
ems_1	9	9
ems_2	3	3
radar_1	3	3
radar_2	5	5
rwr_1	6	6
rwr_2	4	4
tacts_1	6	6
tacts_2	3	3
wmc_1	5	5
wmc_2	7	7
dmt_1	8	8
dmt_2	11	11
saahs_1	9	9
cnidc_1	6	6
af_state_logical_1	5	5
airframe_1	20	8
airframe_2	30	10
airframe_3	20	20
airframe_4	30	30
device_1	15	6
display_1	20	8
display_2	10	10
nav_steering_1	25	13
navigator_navsteering_1	10	10
pilot_control_1	10	10
pilot_waypoints_1	30	22
tactical_logical_1	20	14
tactical_logical_2	25	10
tactical_mode_1	10	10
tactical_physical_1	30	10
tactical_steering_1	15	10

Table 6.2: Timer and Channel Parameters for the Real-time CORBA Case Study Shown in Figure 6.3

the *end-to-end computation time* of the application, as defined in Definition 6.3, from the first event generated by the timers, till the time when all tasks have finished their execution, and *(2)* we performed *schedulability analysis*, by formally analyzing whether any of the tasks may violate their deadlines.

Task	WCET	BCET	Subpriority	Deadline
aths_device	12	12	0	50
aths_logical_device	12	12	1	50
aths_logical_display	20	10	2	50
dvms_device	20	10	3	100
dvms_logical_device	20	10	4	50
dvms_logical_display	20	10	5	50
radio1_device	15	15	6	50
radio1_logical_device	15	7	7	50
radio1_logical_display	15	15	8	50
radio2_device	15	15	9	100
radio2_logical_device	15	8	10	50
radio2_logical_display	15	8	11	50
miu_device	10	8	12	50
miu_logical_device	10	10	13	50
miu_logical_display	10	8	14	50
ems_device	5	5	15	50
ems_logical_device	5	5	16	50
ems_logical_display	5	5	17	50
radar_device	3	3	18	50
radar_logical_device	3	3	19	50
radar_logical_display	3	3	20	50
rwr_device	3	3	21	50
rwr_logical_device	3	3	22	50
rwr_logical_display	3	3	23	50
tacts_device	3	3	24	50
tacts_logical_device	3	3	25	50
tacts_logical_display	3	3	26	50
wmc_device	3	3	27	50
wmc_logical_device	3	3	28	50
wmc_logical_display	3	3	29	50
dmt_device	10	10	30	50
dmt_logical_device	10	10	31	50
dmt_logical_display	10	10	32	50
saahs_device	10	10	33	250
cnidc_device	10	10	34	250
af_state_logical_device	10	10	35	300
af_state_logical_display	10	10	36	250
af_monitor	20	5	49	350
airframe	15	15	42	400
device_5Hz	15	15	37	500
display_device	20	8	47	500
nav_display	10	10	48	500
nav_steering	30	30	46	600
navigator_navsteering_points	30	8	45	1000
phase_manager	30	12	41	350
pilot_control	20	10	50	500
pilot_waypoints	10	10	43	800
routes	10	10	44	250
tactical_logical_device	30	8	39	250
tactical_mode	30	8	40	350
tactical_physical_device_5Hz	15	12	38	250
tactical_steering	15	15	51	350

Table 6.3: Task Parameters for the Real-time CORBA Case Study Shown in Figure 6.3

6.3.1 Comparison with Random Simulations

The main advantage of the DES-based method is that it gradually increases coverage over time. Random simulation-based methods do not have this property. Random simulations assign execution times following a uniform distribution from the [bcet_k, wcet_k] intervals of tasks. Let's denote the two endpoints of a branching interval as l_{k_i}, h_{k_i}, where l_{k_i} refers to the lower bound on the branching interval, and h_{k_i} refers to the higher bound on the branching interval $\mathsf{bcet}_k \leq l_{k_i} \leq h_{k_i} \leq \mathsf{wcet}_k$. Then we can formalize the probability that the random execution time is within the branching interval as follows:

$$P = \frac{h_{k_i} - l_{k_i}}{\mathsf{wcet}_k - \mathsf{bcet}_k} \tag{6.2}$$

Note that $\lim_{(h_{k_i} - l_{k_i}) \to 0} P = 0$, therefore the probability that an exact number is chosen randomly from a continuous-time interval is close to 0, even if we execute infinite number of simulations. Also, the smaller the branching interval, the less chance that we actually consider it during simulation. Since there is a higher chance that the execution time is picked from larger branching intervals, repetitive simulations will pick execution times from branching intervals that have already been chosen for simulation. To illustrate this problem, consider the EOT as shown in Figure 6.2. The nodes with the thinner borders correspond to execution traces, that represent race conditions, and cases when two (or more) events are released with the same timestamp. We see that these cases represent the majority of possible unique orderings of events in this simple example. In larger systems we can expect even worse results, as the number of branching intervals and race conditions may grow exponentially with respect to the number of tasks in the system.

As we have seen from Equation 6.2, the chance to find these execution traces using random simulations is close to 0. However, in the actual system the execution times

rarely follow a uniform distribution; it is quite probable that some execution times are more frequent than others, and that the real system encounters execution traces that were not considered during the simulation-based evaluation process. Since these traces are not simulated, designers will also fail to recognize how the system performance/schedulability might change due to dynamic effects such as race conditions or congestions. Therefore, we conclude that random simulations may be useful for the first steps of performance evaluation, but can achieve only partial coverage of the possible execution traces over time.

In contrast, the method presented in this chapter gradually increases coverage over time. Moreover, we consider each branching interval only once, and we check the worst case times directly, rather than a random number from the branching interval. Therefore, the proposed DES-based method can discover significantly more corner cases than random simulation-based performance estimation techniques.

To check whether our observations are relevant in large-scale systems, we ran experiments to compare random simulations and the DES-based method on the model shown in Figure 6.3. We ran experiments on an Intel Core i7 920 processor running at 4GHz using 6GB of three-channel RAM. On this test configuration, the DREAM 0.7 BETA release can simulate one execution trace of the DRE case study shown in Figure 6.3 in ∼20 ms. The fast performance is the result of the symbolic DES-based representation. We ran both random simulations and the DES-based method on the model shown in Figure 6.3 for a week. We used the open-source DREAM tool for the random simulations as well, therefore all improvements in the DES-based analysis are the result of the better state space coverage. We were able to simulate ∼30 million (3×10^7) non-equivalent execution traces (the execution order of tasks is different) of the case study using the DES-based method in one week. The implementation did not take advantage of multi-threaded execution available in modern multi-core architectures such as the Intel Core i7 processor, and scalability could be significantly

improved by a more efficient multi-core implementation. This coverage can only be achieved using model checking techniques within this short time. Our experiments show that the DES-based analysis can obtain higher bounds on the worst case end-to-end performance than random simulations.

The difference comes from the fact that the DES-based method has better state space coverage, and therefore it is more accurate for performance estimation than random simulations. Even though the DES-based method cannot always obtain the highest bounds on the end-to-end performance, the combination of model checking and directed simulations along the execution tree provided the best coverage that we could achieve within a week on this case study. This shows that the proposed DES-based verification method is practically applicable for the performance evaluation of large-scale systems.

6.3.2 Comparison with Timed Automata Model Checking Methods

We have used DREAM to generate TA representation from DRE models as described in [67]. UPPAAL and the IF toolset are two leading model checkers for real-time verification with several years of development history. Although both UPPAAL and IF build on the TA MoC they are inherently different. UPPAAL uses a traditional TA model [6] extended with integer-valued variables, IF, on the other hand, uses transition priorities to express time constraints.

We have not conducted extensive comparisons between DREAM and TA model checkers yet to reach meaningful conclusions on how their verification performance compares in general. In the case studies that we've analyzed, TA model checkers usually perform better than the proposed DES-based method, on smaller models, that have a high degree of non-determinism. Earlier we have successfully used UPPAAL for the real-time verification of DRE systems consisting of ∼30 tasks/event channels as described in Chapter 5. TA model checkers employ symbolic state representations

that allow for efficient heuristics and compact state space representation. Therefore, both UPPAAL and IF implement memory-bounded model checking.

On large-scale models, such as the case study shown in Figure 6.3, however, both TA model checkers run out of memory, and are unable to give partial results to designers. Although DREAM does not run out of memory on these examples, the verification time increases exponentially. The impact of this problem could be potentially reduced by implementing the model checker on a distributed platform. In our experience, TA model checkers are useful for the performance evaluation of DRE systems that can be modeled with less than ∼60–100 clocks (occasionally better on mostly deterministic models), therefore a compositional approach is required to cope with scalability issues. Moreover, since performance estimation has to be formalized as a yes/no question, designers have to "guess" what a close bound on the end-to-end computation time could be, and check whether the performance is smaller or not. Our experiences have shown that TA model checkers are well-suited for the real-time verification of small/medium size systems, but their practical application for the performance estimation of large-scale DRE systems is limited, and cannot be compared to the proposed DES-based method – or even random simulations – on large-scale systems, due to the state space explosion problem as a result of the exhaustive analysis.

6.4 Practical Application to an H.264 Decoder MPSoC Design

6.4.1 H.264/AVC Overview

The *H.264/Advanced Video Coding (AVC)* is a relatively new video compression standard that has been developed through the joint work of the International Organization for Standardization's *MPEG* group and the International Telecommunication Union's video coding experts group [52]. One of the main features of this standard is the significantly improved video quality, better compression efficiency and more

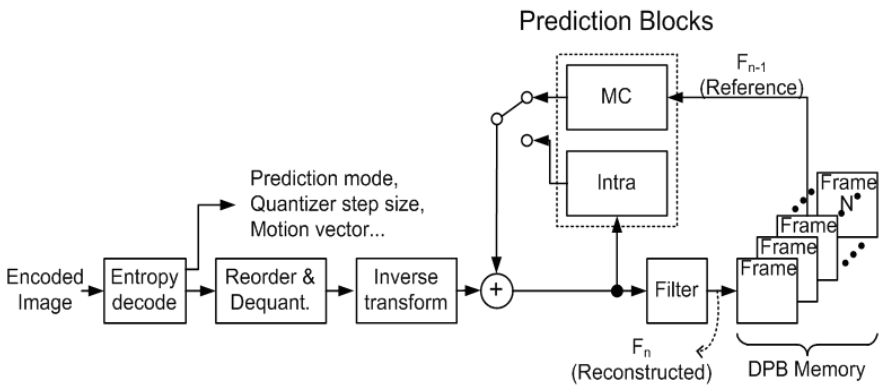

Figure 6.4: H.264 Decoder Algorithm

error robustness for various applications as compared to previous coding standards such as *H.263*, *MPEG-2*, and *MPEG-3*. The *H.264* standard is unique in its broad applicability across a range of bit rates and video resolutions (from low-bitrate mobile video applications to high-definition TV) and is gaining momentum in its adoption by industry.

To encode color images, *H.264* uses the YCbCr color space like its predecessors, separating the image into luminance (or "luma", brightness) and chrominance (or "chroma", color) planes. It is, however, fixed at 4:2:0 sub-sampling by default, *i.e.*, the chroma channels each have half of the vertical and horizontal resolution of the luma channel. A video frame is divided into slices, which can be one of three main types: *(1) Intra (I)*, that describe a full still image and contain a reference only to themselves, *(2) Predicted (P)*, that use one or more recently decoded slice(s) as a reference (*i.e.*, prediction) for picture construction, and *(3) Bi-directional predicted (B)*, that work like P slices with the exception that former and future I or P slices (in playback order) may be used as reference pictures. Each slice is further divided into *Macroblocks (MBs)* which are blocks of 16×16 pixels. The MBs are decoded from left to right, and then top to bottom. During the decoding process, the MBs are often further divided into smaller blocks, with the smallest unit being a 4×4 block.

Figure 6.4 shows a block diagram of the *H.264* decoder, which we use for decoding 176 × 144 square pixel *Quarter Common Intermediate Format (QCIF)* video intended for portable multimedia devices. The first stage of the decoding process is when numeric values are recovered from the binary codes of the compressed video using the Entropy Decoder. Entropy coding reduces statistical redundancies in a video stream by using either *Context-Adaptive Variable Length Coding (CAVLC)* or *Context-Adaptive Binary Arithmetic Coding (CABAC)*. Next, the *Reorder*, *Dequantization* and *Inverse transform* stages are used to recover residual data. This data is the difference of the inter- or intra-prediction made by the encoder and the actual value, for each MB. In the next stage, depending on the header information, either the motion compensation (*MC*, or inter-prediction) or *Intra* (intra-prediction) block is called. Inter-prediction exploits temporal redundancies by taking advantage of the fact that the content of a new frame in the video often has high correlation to the data in the previous frames. For each MB, the encoder looks for a piece (with the same size) of the previous frame that is similar to it, and then encodes that information by specifying the relative location of the block. The decoder then uses this information to reconstruct the block of the frame. Intra-prediction is used for frames that are not encoded using inter-prediction, such as the first frame of a new scene (which has low correlation with the previous frame) or frames added to limit error propagation due to inter-prediction. Intra-prediction exploits spatial redundancies by using part of a frame to predict the other parts. Finally, a *Deblocking filter* is used to improve perceptual quality of the reconstructed video. Since all the processing described so far process frames one block at a time, the filter helps to blur the edges of the blocks where imperfections are most visible.

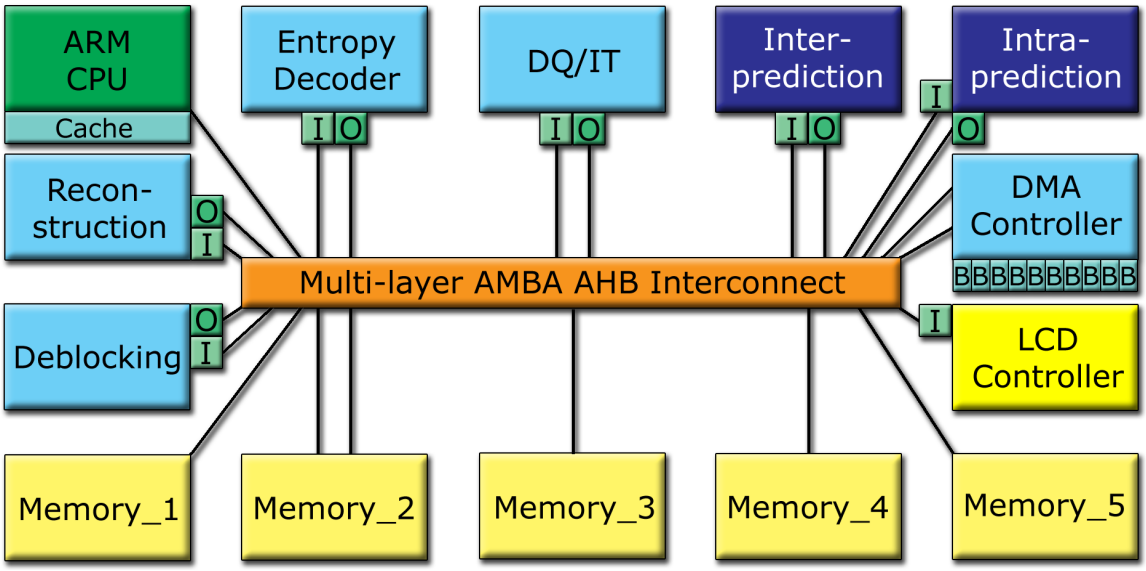

Figure 6.5: H.264 Decoder MPSoC Architecture

6.4.2 H.264 Decoder MPSoC Design

This section describes the multimedia *H.264* decoder *Multi-processor System-on-Chip (MPSoC)* design shown in Figure 6.5. The application utilizes a combination of SW blocks and custom *Application-specific Integrated Circuit (ASIC)* HW components for computation. Combining SW and HW computations on a common MPSoC architecture instead of utilizing a fully custom HW implementation has the advantage of better flexibility for future design changes and better reuse of available design IPs. Performance critical functionalities are implemented in custom ASIC HW, whereas more control intensive algorithms are often a better fit for SW implementations.

The *H.264* decoder MPSoC design shown in Figure 6.5 was built based on the concept of HW/SW co-design to take advantage of the strengths of both HW and SW implementations. The MPSoC is based on a fully connected *ARM Advanced Microcontroller Bus Architecture Advanced High-speed Bus (AMBA AHB)* bus matrix interconnect [8] (also referred to as crossbar switch or multi-layer interconnect). Advantages of using a bus matrix instead of a shared bus include *(1) increased through-*

put, (2) reduced congestions, (3) simpler arbitration, and (4) simplified performance analysis due to the simpler arbitration. Since all masters are connected to all slaves using the AMBA AHB bus matrix interconnect, masters do not need to wait for other masters when requesting access to the bus, but can transmit at will. A simple point arbitration policy is used to manage collisions when multiple masters attempt to access a single slave at the same time, by setting the `HREADY` signal on the bus, signaling masters that the slave is not ready yet for transmission. We utilize a simple fixed-priority scheduling algorithm as the point arbitration policy to manage which (waiting) master is granted access to a slave that becomes ready to serve the new request. Moreover, the design utilizes a *Direct Memory Access (DMA)* Controller for managing the transactions, alleviating the load on the *Central Processing Unit (CPU)*. We use 5 small memory blocks in this design to enable a pipelined decoder implementation that improves overall throughput, and ensure as few conflicts as possible between requests from the various stages. The I, O, and B blocks represent buffers.

We now describe the behavior of the MPSoC design. Details such as processing frequencies and buffer sizes are described in Section 6.4.3. The encoded multimedia stream that needs to be decoded is stored in `Memory_1` by an external DMA engine. The `CPU` signals the `DMA Controller` to copy the encoded frame from `Memory_1` to the input buffer of the `Entropy Decoder`. The `Entropy Decoder` decodes global parameters such as motion vectors used by the prediction engines, and writes results to its output buffer. The `DMA Controller` then fetches the results from the output buffer of the `Entropy Decoder`, and writes it to `Memory_2`. The `CPU` then signals the `DMA Controller` to fetch entropy information from `Memory_2` and copy it to the input buffer of the *Discrete Quantization/Inverse Transform (DQ/IT)* block. The CPU fetches the same data from `Memory_2`, and initiates an inter- or intra-prediction pre-processing depending on the encoded frame. The `CPU` and `DQ/IT` work in parallel, as they do not depend on each other. The `CPU` then writes the results of

its computation to Memory_3, and triggers DMA Controller to transfer macroblocks from Memory_3 to the Inter-prediction or Intra-prediction ASIC components (depending on whether the encoded frame is an P frame or a I frame) to decode macroblocks. The Inter-prediction or Intra-prediction ASIC blocks may also execute in parallel with the DQ/IT block. When the DQ/IT block finishes its computation, the DMA Controller fetches data from its output buffer, and writes results in Memory_4. Similarly, the DMA controller transfers the computation results from the output buffers of the Inter-prediction and Intra-prediction blocks to Memory_-4. The DMA Controller then fetches this data and writes it to the input buffer of the Reconstruction ASIC block, that is used to reconstruct macroblocks from the computation results of the DQ/IT and Inter-prediction/ Intra-prediction blocks. The DMA then transfers reconstructed macroblocks to Memory_5. The CPU then fetches reconstructed macroblocks from Memory_5, runs control-intensive pre-processing as a first step of the deblocking process, and instructs the DMA Controller to transfer the results to the input buffer of the Deblocking ASIC block, that smoothens out the visible borders of macroblocks. Finally, the decoded stream is sent to the *Liquid Crystal Display (LCD)* Controller by the DMA Controller, where it is displayed on the LCD screen. The entire decoding process takes place in a pipelined manner to improve decoder throughput.

6.4.3 Performance Parameters for the H.264 Decoder MPSoC Design

In order to facilitate accurate performance analysis, we utilized traditional simulation methods to obtain performance parameters for HW/SW components. As the first step, we have profiled the JM *H.264* reference C code (http://iphome.hhi.de/suehring/tml) to obtain cycle estimates on SW functions, and identify the computationally most intensive blocks. The Inter-prediction, Intra-prediction, and Deblocking ASIC blocks do not implement their respective processes fully in

HW, but rather use the CPU for pre-processing. The Entropy Decoder, DQ/IT and Reconstruction blocks are HW blocks that leave no pre-processing to the CPU.

We have fully implemented the Inter-prediction and Intra-prediction blocks together with the AMBA AHB bus in Verilog in order to get accurate estimates on performance parameters, and to allow functional verification using test vectors. The Entropy Decoder, DQ/IT, Reconstruction, and Deblocking ASIC blocks were partially implemented to allow for RTL cycle estimates.

In *H.264*, a frame is processed in units of MBs. The size of a macroblock can be computed as follows: 16×16 pixels (Luma) + 8×8 (Cb) +8×8 (Cr) = 384 Bytes. The *H.264* decoder MPSoC shown in Figure 6.5 works on QCIF resolution frames, therefore the size of a single frame is $99 \times 384 = 38016$ Bytes. We utilize a 32bit wide AMBA AHB bus matrix interconnect for communication, therefore to transfer a frame through the bus takes 9504 cycles in the ideal case. The worst case depends on congestions on the bus, and is therefore denoted as ? in both Table 6.4 and Table 6.5. The total number of bus cycles for processing a frame is 133056 cycles in the ideal case.

The size of input/output buffers in the DQ/IT, Inter-prediction, Intra-prediction, and Reconstruction ASIC blocks is 2KBytes (for each), that can store 5

Table 6.4: Cycle Estimates for Processing 1 Frame by HW/SW Blocks in Figure 6.5

Block	Worst Case	Best Case
Entropy Decoder	633600	76032
DQ/IT	65835	65835
Inter-prediction	102960	102960
Intra-prediction	59796	31624
Reconstruction	28611	28611
Deblocking	57024	57024
Inter-prediction (CPU)	4448129	1596938
Intra-prediction (CPU)	3794455	147572
Deblocking (CPU)	4887118	3255122
Bus cycles	?	133056

Table 6.5: Execution Time Estimates for Processing 1 Frame by HW/SW Blocks (in μs)

Block	WCET	BCET
Entropy Decoder	6336	760
DQ/IT	659	659
Inter-prediction	1030	1030
Intra-prediction	598	316
Reconstruction	287	287
Deblocking	571	571
Inter-prediction (CPU)	11121	3992
Intra-prediction (CPU)	9487	368
Deblocking (CPU)	12218	8137
Bus cycles	?	133

macroblocks at one time ($384 \times 5 = 1920$B). The input/output buffer sizes of the `Entropy Decoder`, `Deblocking` blocks and the input buffer of `LCD Controller` block are 40KBytes, therefore they can store a whole frame (frame size is 38016B). The DMA engine has multiple buffers to manage transactions between masters/slaves on the bus, and the CPU has 32KB L1 data and instruction caches.

Table 6.4 shows the best and worst case cycle estimates for the *H.264* decoder MPSoC. The execution time of the `Entropy Decoder` ASIC block depends on the input data, therefore we see variation in the execution times. The `Intra-prediction` block either utilizes 4×4 blocks or 16×16 blocks, hence the variation in execution time. And finally, SW blocks executed on the CPU are control-dependent, and exhibit large variation in execution time. The worst case time for bus cycles depends on dynamic factors that have to be captured by the performance verification method.

Our *performance requirement* for the *H.264* decoder is to be able to decode *H.264* streams real-time in *30 frames/second*. Parameters that we had to determine included the frequencies of ASIC blocks, the frequency of the AMBA AHB bus matrix interconnect (preferably the same as the ASIC blocks for simple synchronization), and the frequency of the ARM CPU. The unoptimized *H.264* reference code comes at a high price, as we need a 400MHz ARM processor for the decoding, even though most of the

heavy lifting is performed in custom ASIC blocks. The frequencies of both the AMBA AHB bus matrix interconnect and the ASIC blocks are set to 100MHz. An industrial implementation should optimize the *H.264* SW before deploying it in a commercial application, however this is not the focus of this chapter.

Using these parameters we have calculated the estimated worst and best case execution times of all blocks. The sum of the worst case execution times for P frames (inter-prediction) is 32355 microseconds (referred to as μs from now on), and for I frames (intra-prediction) it is 30289μs, that suggests that 30 frames can be decoded in 970650μs. This gives us a tight but feasible bound to achieve 30 fps *H.264* QCIF decoding. However, to provide guarantees on the end-to-end performance of the MPSoC design, we need to consider dynamic effects, such as congestions on the bus, and varying execution times.

6.4.4 Formal Modeling of the H.264 Decoder MPSoC Design

Figure 6.6 shows the ALDERIS model for the *H.264* decoder. The SW tasks and ASIC blocks are denoted as `T`, whereas `C` represent communication channel *First In First Outs (FIFOs)* (set C in Section 4.1.1), which in turn model bus transactions. As seen from the design, there are 13 channels representing the bus transactions, but the link between the `Mem4` and `Reconstruction` blocks carries two frames (that get linked in the `Reconstruction` block).

All tasks have their own thread of computation, except for the CPU, that has a single thread to schedule the `CPU (Inter-prediction)`, `CPU (Intra-prediction)`, and `CPU (Deblocking)` tasks. Threads are denoted as icons with the two threads, and are connected to tasks by dashed lines.

Computations are driven by the `Timer`, that sends out events periodically, representing the various frames. The MPSoC design shown in Figure 6.6 depicts the case when a P frame is processed; when I frames are processed, the `Intra-prediction`

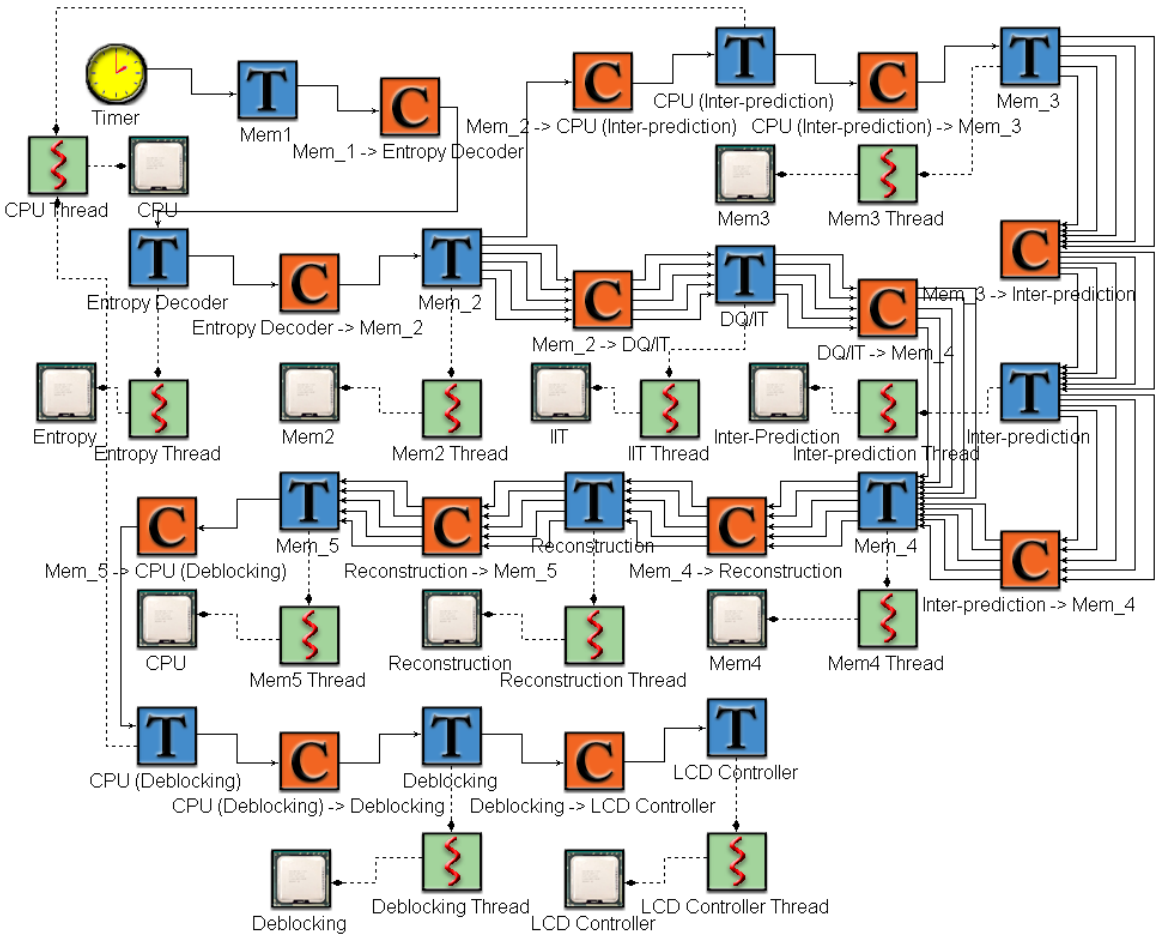

Figure 6.6: Formal Modeling of the H.264 Decoder MPSoC Design

and `CPU (Intra-prediction)` blocks are called instead of the `Inter-prediction` and `CPU (Inter-prediction)` blocks.

Arrows in the block represent dependencies in the set D. Between some block in the design we have depicted several arrows (*i.e.* between the `Mem2` task and the `DQ/IT` task through the FIFO). The larger number of arrows between blocks represent multiple transactions. In these cases, we are passing 5 macroblocks in each transaction (that fit in the 2K buffers described in Section 6.4.3). Therefore, 20 transactions are required to transfer all 99 macroblocks between blocks connected by several arrows.

Execution parameters for tasks are specified in Table 6.5. We use 95μs as parameter for bus transaction delays per frame. This parameter approximates the 9504 cy-

cle/frame bus transaction delay using 100MHz frequency, as described in Section 6.4.3. We assume 1 μs access time to read from/write to memories. We specify the period of the `Timer` to be 33333 μs, to represent 30 frames/second. We also use 33333 μs as the deadline for the end-to-end computation of the *H.264* decoder. We have used the model shown in Figure 6.6 with the parameters given in Table 6.5 for the performance verification of the *H.264* decoder MPSoC design, as described in Section 6.4.5.

6.4.5 Performance Verification of the H.264 Decoder MPSoC Design by DES

We have implemented the performance analysis method in the open-source DREAM tool. The task graph shown in Figure 6.6 is translated to a discrete event system. DREAM simulates the execution of this system by setting each task to use its worst case execution time. Each task t_k records events that happened within their $[\mathsf{bcet}_k, \mathsf{wcet}_k]$ execution interval. Using this information, directed simulations are run with all permutations of events. As each execution trace considers only the logical execution times but not the actual computation, each simulation is in the order of milliseconds, and depends largely on the number of tasks. The performance of the overall analysis is mainly influenced by the degree of non-determinism in the analyzed system.

Using the model described in Section 6.4.4, the open-source DREAM tool computed the worst case end-to-end execution time of a P frame (inter-prediction) to be 32709μs, and of an *I* frame (intra-prediction) to be 30643μs. We see that the manual estimates on the worst case end-to-end computation times in Section 6.4.3 were quite close, but could not be guaranteed. By combining cycle estimate information obtained by SW profiling and *Register-Transfer Level (RTL)* simulations with model checking methods we were able to obtain tight and reliable bounds on the end-to-end performance of the multimedia streaming *H.264* decoder MPSoC application.

6.5 Concluding Remarks

This chapter presented an approach to model DRE systems as *Discrete Event (DE)* systems using a continuous-time model, and proposed a method for formal performance evaluation and real-time verification. The proposed method explicitly captures the data flow, and models communication and execution intervals using a non-preemptive scheduling model. The DRE MoC provides a formal executable model allowing to bridge the gap between simulations and formal verification. Our benchmarks based on a large-scale avionics case study show that the DES-based performance evaluation method can achieve better coverage than alternative methods, and provides a way for the systematic measurement of coverage. We also applied the proposed DES-based performance estimation method to an H.264/AVC MPSoC design. The proposed approach allows to efficiently explore large design spaces early in the design flow, provides formal guarantees on real-time constraints, and can produce counter-examples when real-time properties are violated. The DES-based performance estimation and verification method has been implemented in the open-source DREAM tool available at `http://dre.sourceforge.net`.

Chapter 7

Conservative Approximation Method for the Real-time Verification of Preemptive Systems

Asynchronous event-driven communication is widely used in modern *Distributed Real-time Embedded (DRE)* systems. Reducing synchronizations in event-driven systems can simplify implementation, prevent blocking waits, reduce energy consumption, and provide better throughput and flexibility. Providing formal guarantees on real-time properties in asynchronous event-driven systems, however, remains a key challenge.

Stopwatch Automata (SA) [77] were proposed as a *Model of Computation (MoC)* that can express preemptable tasks in asynchronous event-driven systems. It was shown that reachability analysis on the composition of SA as task graphs (integration graphs) is undecidable [55, 59] if the following conditions are met: *(1)* tasks use event-based asynchronous triggering (*i.e.* a target task starts whenever its source finishes) on a distributed platform, *(2)* execution times are specified as continuous-time intervals, *(3)* preemptions may occur anytime within the continuous-time execution interval. In this dissertation, we refer to systems that satisfy these three conditions as *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)*. PEARSE are a subset of DRE systems, that do not require global synchronization.

This chapter presents a conservative approximation method for the verification of PEARSE models, using *Timed Automata (TA)* [6] model checking methods. The reachability problem is decidable on TA, therefore we provide an implementable method for the automatic real-time verification of PEARSE models.

The proposed method approximates each stopwatch automaton (S) using an ap-

proximate timed automaton (T). We show that the stopwatch automaton accepts all the time traces that the timed automaton accepts by showing that the language that T accepts is a subset of the language that S accepts ($L(T) \subseteq L(S)$). This problem is known as the language inclusion problem [47]. Since $L(T) \subseteq L(S)$ holds, there are no timed traces that the timed automaton accepts, but the stopwatch automaton does not accept, therefore the approximation is conservative. Accordingly, the proposed analysis provides a sufficient condition to determine the schedulability of preemptive event-driven asynchronous real-time systems with execution intervals (PEARSE).

The remainder of this chapter is organized as follows. Section 7.1 describes the problem statement; Section 7.2 presents the proposed method for the verification of preemptive scheduling, and proves the conservative nature of the approximation; Section 7.3 demonstrates the approach on a real-time *Common Object Request Broker Architecture (CORBA)* application and Section 7.4 presents concluding remarks.

7.1 Problem Formulation

7.1.1 Stopwatch as a Model for a Preemptable Real-time Task

A stopwatch is a clock that can be reseted, stopped, and resumed, providing a simple model for a preemptable real-time task. The execution time of a task can be represented as a stopwatch as shown in Figure 7.1. Time is represented as clock c_t, and the stopwatch clock is c_{sw}. The valuation of these clocks is v_t and v_{sw} as defined in Section 3.4.

The stopwatch makes a transition to the stop location from its initial (idle) location when it receives an enable_i event that signals that the task is ready for execution. The ? sign after an event denotes an input (receive) event, and the ! sign after an event denotes an output (send) event as used in [12]. Whenever the task is scheduled for execution, the stopwatch makes a transition to the run location, and whenever the

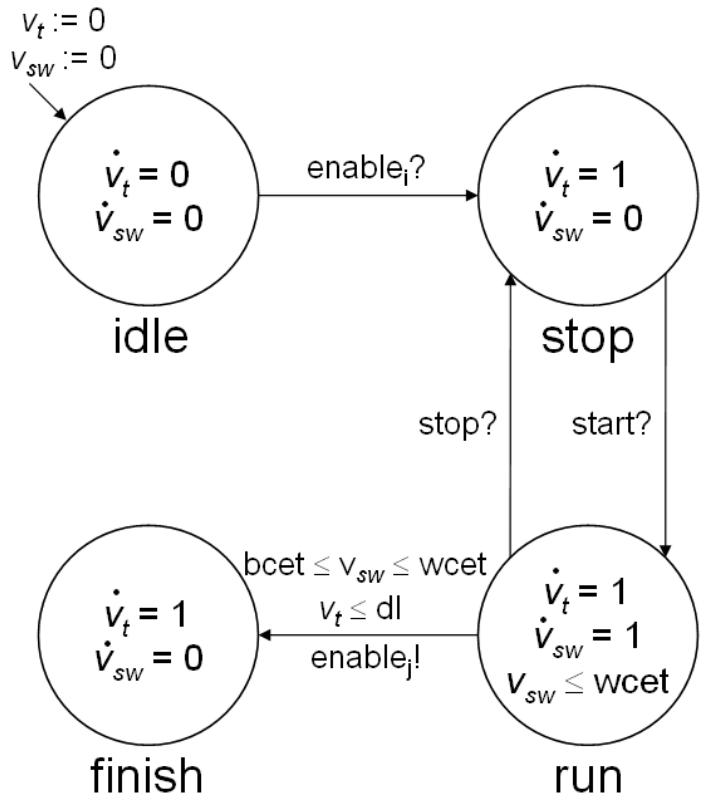

Figure 7.1: Task Stopwatch Automaton (TSA) – Model of a Preemptable Real-time Task

task is preempted, the stopwatch moves to the stop location. v_{sw} represents the valuation of the stopwatch clock. We refer to the stopwatch shown in Figure 7.1 as *Task Stopwatch Automaton (TSA)* in this dissertation. When the task finishes its execution, the TSA moves to the finish location. We say that the TSA *executes* if and only if it is in the run location, and it is *preempted* if and only if it is in the stop location, and we refer to the time spent in the run location as *execution time*. Figure 7.1 can be extended to model periodic tasks by adding a transition to the idle location from the run location instead of the finish location. In this chapter we use the simple one-time executing task shown in Figure 7.1 for simplicity.

A task may have a *Best Case Execution Time (BCET)*, that corresponds to the shortest, and a *Worst Case Execution Time (WCET)*, that corresponds to the longest time in which the task may finish its execution. Time for execution is counted from

the time of the $\mathsf{enable_i}$ event (v_{t_0}), and does not include time spent in the stop location. The following constraints are implied by the definitions of the stopwatch model, best case and worst case times:

$$0 \leq v_{sw} \leq v_t,\ 0 \leq v_{sw} \leq \mathsf{wcet},\ 0 \leq \mathsf{bcet} \leq \mathsf{wcet} \tag{7.1}$$

Deadlines, denoted dl for a given task, are constraints on the maximum time from the time of the $\mathsf{enable_i}$ event to the time when the automaton makes a transition to the finish location.

Definition 7.1 *A real-time task is schedulable if it always finishes its execution before its respective deadline. A task is then schedulable if and only if* $v_t \leq \mathsf{dl}$ *when* $v_{sw} = \mathsf{wcet}$.

The alphabet of the TSA is $\Sigma = \{\mathsf{start}, \mathsf{stop}, \mathsf{enable_i}, \mathsf{enable_j}\}$. The start and stop events are controlled by a (set of) scheduler(s), the task receives the $\mathsf{enable_i}$ event from its source task, and sends out the $\mathsf{enable_j}$ event when it finishes its execution. See Section 7.1.2 for the formal definition of composition rules.

A *timed word* is of the form $(\sigma_0, \tau_0)(\sigma_1, \tau_1)\ldots(\sigma_n, \tau_n)$, where $\sigma_0, \sigma_1, \ldots, \sigma_n \in \Sigma$ denote events, and $\tau_1, \tau_2, \ldots, \tau_n \in \mathbb{R}_{\geq 0}$ denote the timestamps of events. The set of timed words is the *timed language* on which the TSA operates. We can express the syntax of the (untimed) language that the TSA accepts using the following regular expression:

$$S_{L(S)} = \mathsf{enable_i}\ \mathsf{start}\ (\mathsf{stop}\ \mathsf{start})^*\ \mathsf{enable_j} \tag{7.2}$$

The timestamps of all events have to be less than dl in a timed word in order for the TSA to accept the word. We denote the timestamps of events in the $[0 \ldots \mathsf{dl}]$ interval as $\tau_1, \tau_2, \ldots, \tau_e$, where τ_1 denotes the $\mathsf{enable_i}$ event, and t_e denotes the last start event. Note that e is always an even number according to Equation 7.2. The TSA accepts the timed language $L(S)$ as described in Equation 7.2, and satisfies the following

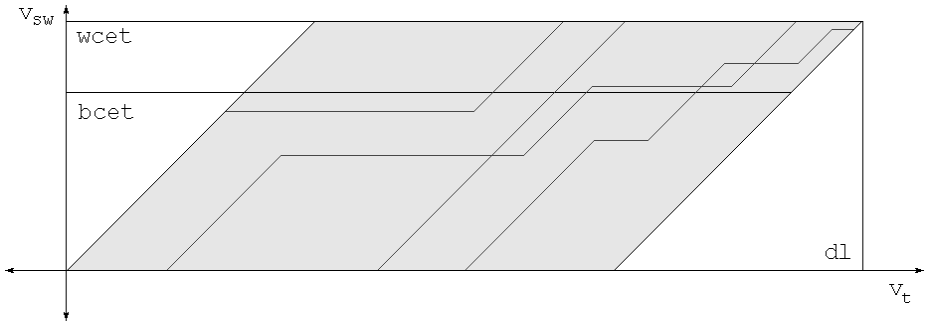

Figure 7.2: Clock Constraints on Stopwatches for Schedulability

constraint:

$$\sum_{i=1}^{\frac{e}{2}} \tau_{2i} - \tau_{2i-1} \leq \mathsf{dl} - \mathsf{wcet} \tag{7.3}$$

Equation 7.3 states that a task is schedulable if it spends at most dl - wcet time in the stop location in the TSA within the $[0, \mathsf{dl}]$ interval for clock c_t, since in this case it can execute for wcet time before its deadline. Figure 7.2 shows the constraints implied on the TSA clock for a schedulable task. v_{sw} is in the [0,wcet] domain, v_t is in the $[0, \mathsf{dl}]$ domain, and the slope of v_{sw} is at most 1 ($\dot{v}_{sw} \in \{0, 1\}$). The valid clock assignments define a parallelogram, and all other assignments result in possible deadline violations. Note that since this model is an initialized stopwatch automaton, reachability is decidable [45], therefore we can verify the schedulability of a single preemptable task, given that we know when start/stop events occur. The darker lines show a few stopwatch clock valuation traces that model the execution of schedulable real-time tasks.

7.1.2 Composable Stopwatch Automata as a Model for PEARSE

In Section 7.1.1 we described how TSA can model a preemptable real-time task. In this section we consider how the composition of stopwatches can represent PEARSE. In this chapter we represent PEARSE as task graphs $G_S = (X_S, E_S)$, where the set of vertices represent real-time tasks modeled as stopwatches, and edges represent

dependencies between the tasks $E_S \subseteq X_S \times X_S$. A vertex with no incoming edge(s) is a *source*, and a vertex with no outgoing edge(s) is a *terminal*. This task graph model is a subclass of *integration graphs* defined in [55]. In our model, each hybrid automaton in the integration graph is a TSA.

There are two ways in which TSA models compose in G_S; serial and parallel composition. When *parallel composition* is used, the composed automata operate independently from each other. In graph G_S, two TSA compose using parallel composition, if none of them is reachable from the other on a directed path. We denote parallel composition between two vertices $x_i, x_j \in X_S$ as $x_i \oplus x_j$. The $\oplus$ operator is associative, distributive and commutative.

Assume that there is an edge in graph G_S between two vertices $(x_i, x_j) \in E_S$, so task x_j depends on task x_i. Then the $\mathsf{enable_j}$ transition in TSA_i, and the $\mathsf{enable_i}$ transition in TSA_j can only be taken simultaneously. We refer to this case as *serial composition*, and define it as follows. Denote the language of x_i as $L(S)_i$, and the language of x_j as $L(S)_j$. Denote the alphabet of x_i as $\Sigma_i = \{\mathsf{start_i}, \mathsf{stop_i}, \mathsf{enable_{i_i}}, \mathsf{enable_{j_i}}\}$, and the alphabet of x_j as $\Sigma_j = \{\mathsf{start_j}, \mathsf{stop_j}, \mathsf{enable_{i_j}}, \mathsf{enable_{j_j}}\}$. By definition, the serial composition of x_i and x_j means that the timestamp of $\mathsf{enable_{j_i}}$, and the timestamp of $\mathsf{enable_{i_j}}$ is the same. We denote serial composition between two vertices $x_i, x_j \in V_S$ as $x_i \otimes x_j$. The $\otimes$ operator is associative, distributive, but not commutative, since if x_i depends on x_j is not the same case as when x_j depends on x_i.

For the sake of simplicity we do not consider buffers or communication delays between tasks. Further, without losing generality, we disallow multiple sources to tasks $(\forall(x_a, x_b, x_j \in V_S)((x_a, x_j) \in E_S \wedge (x_b, x_j) \in E_S) \rightarrow x_a = x_b)$, each task may depend on at most one task directly. We do allow multiple dependents for tasks, as these can be modeled as synchronizations between transitions, and do not require buffers.

We point out that this model can be easily extended to include *First In First*

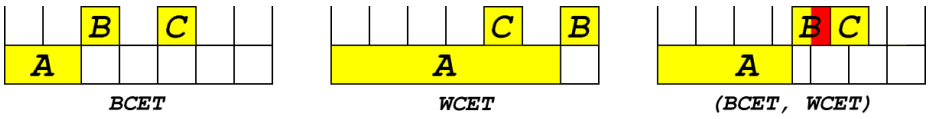

Figure 7.3: Motivating Example for a Non-WCET Deadline Miss

Out (FIFO) channels modeled as TA, that can express many-to-many connections, and communication delays as well. Moreover, the model can be extended to include periodic tasks modeled as TA that broadcast enable_i events with some rate. The start and stop events may be controlled by a (set of) scheduler(s), providing a MoC that can express PEARSE in a formal setting. Please see [67] for an approach to model real-time CORBA applications using TA as a MoC for real-time analysis.

7.1.3 Problem Description

Integration graphs may be used to compose stopwatches using events, to express PEARSE as a network of stopwatch models using preemptive scheduling. Although the reachability analysis of a single initialized stopwatch automaton is decidable [45], reachability analysis on integration graphs is undecidable in general, more specifically if the conditions defining PEARSE are met: *(1)* tasks use event-based asynchronous triggering (*i.e.* a target task starts whenever its source finishes) on a distributed platform, *(2)* execution times are specified as continuous-time intervals, *(3)* preemptions may occur anytime within the continuous-time execution interval [55, 59].

In this chapter we propose a conservative approximation method for the verification of preemptive scheduling in PEARSE designs. We approximate SA using TA by discretizing clocks, to "store" time passed before a preemption. The practical benefit of this method is that we provide a decidable technique for the real-time verification of PEARSE.

In event-driven systems it is not enough to consider the worst case times of tasks in general. Consider the simple example shown in Figure 7.3. Task A is running on

machine_1, and tasks B and C are running on machine_2. Task A starts at time 0, and may finish its execution time anytime within the $[2, 6]$ interval. Task B starts its execution whenever task A finishes its execution, and executes for 1 time unit. Task C starts its execution at time 4, and executes for 1 time unit. We assume that task B has higher priority than task C, and that the deadlines for task B and task C are 1.2 time units. The system is schedulable when task A executes for its BCET time as shown in the left of Figure 7.3, and it is schedulable when task A executes for its WCET time as shown in the middle of Figure 7.3. However, if task A finishes its execution at time 5.5, task C will miss its deadline as it has to wait for task B. Therefore, the analysis has to capture execution intervals in continuous time, otherwise it may lead to false positives; unschedulable designs that cannot be detected at design time.

We achieve this goal by mapping preemptive scheduling to non-preemptive scheduling. TA can express non-preemptive scheduling with execution intervals [32, 67], and reachability analysis is decidable on TA [6].

7.2 Conservative Approximation of Integration Graphs

In this section we describe the conservative approximation method for the reachability analysis of integration graphs. We implement this approximation in two steps. In the first step, we map each TSA in graph G_S (defined in Section 7.1.2) to a timed automaton. We refer to this timed automaton as *Task Timed Automaton (TTA)* (described in detail in Section 7.2.1). In the second step we consider how approximation errors can be considered in the analysis of task graphs. We denote the language that the TTA accepts as $L(T)$. Then we show that $L(T) \subseteq L(S)$, that implies the conservative nature of the approximation.

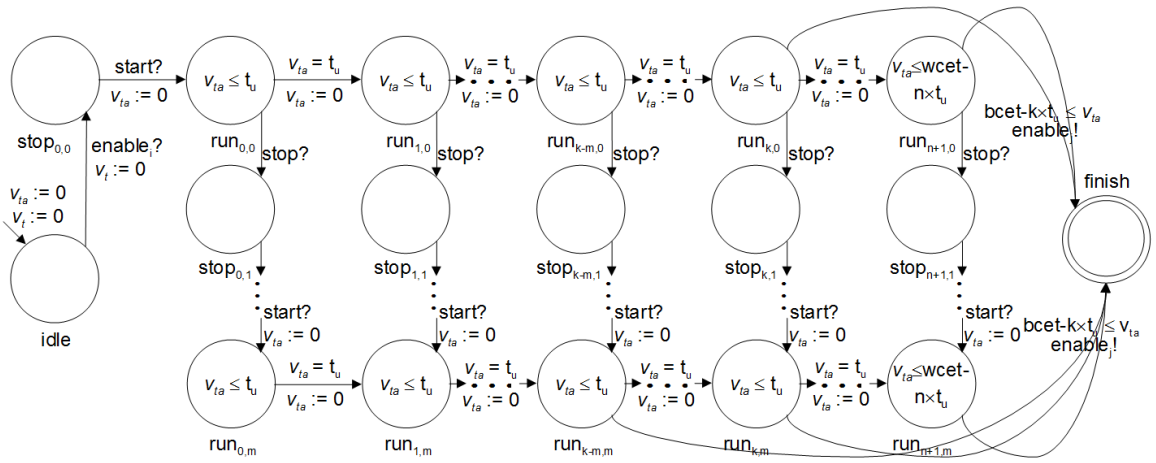

Figure 7.4: Task Timed Automaton (TTA) – Approximating a Preemptable Real-time Task

7.2.1 Mapping the TSA to TTA

In this section we show how the TSA can be mapped to the TTA. We represent preemptable tasks as TSA as shown in Figure 7.1. We introduce a generic timed automaton template for preemptable tasks as shown in Figure 7.4. The locations denoted as $\mathsf{run}_{x,y}$ in Figure7.4 represent the run location of the TSA, the $\mathsf{stop}_{x,y}$ locations represent locations where the task is preempted (location stop in Figure 7.1). The x index represents discrete checkpoints denoting time passed, and y represents the number of preemptions encountered during the run of the TTA. We say that the TTA *executes* if and only if it is in a $\mathsf{run}_{x,y}$ location, and it is *preempted* if and only if it is in a $\mathsf{stop}_{x,y}$ location, and we refer to the time spent in $\mathsf{run}_{x,y}$ locations as *execution time*.

We denote the time unit used for the discretization as t_u. We partition the WCET time of a task to n equal-size intervals, and a smaller interval representing the fraction of the division of WCET by t_u. If t_u is a divisor of WCET, then the locations $\mathsf{run}_{n,y}$ representing the fractions are not required, and the value of n and m needs to be decreased by 1 in all guards in Figure 7.4. We define the constants used in the template as follows:

$$n = \lfloor \frac{\mathsf{wcet}}{t_u} \rfloor, 1 \leq m \leq \frac{\mathsf{dl}}{t_u}, k = \lfloor \frac{\mathsf{bcet}}{t_u} \rfloor \tag{7.4}$$

Key restriction: *Since the TTA is an approximation of the TSA, it can only express bounded numbers of preemptions (per task).* Note that this bound *does not restrict periodic execution.* If the execution of the task graph is repeatedly triggered by a periodic (or aperiodic) event, the task can get preempted limited times during a single execution of the task graph. However, there is no bound on the number of executions, and thus the overall number of preemptions.

The constant m represents the maximum number of preemptions that the TTA can capture. This is a weak restriction, as in practical systems tasks are not preempted infinitely often. In fact, frequent preemptions can significantly degrade the system performance and therefore should be avoided by design. Moreover, since task graphs (and dependencies between tasks) are fixed, one can calculate how many times a task may get preempted during a single execution of the task graph. *The value of m is not affected by how many times the task graph is executed, or whether the design is periodic.* The value of m is proportional to $\frac{\mathsf{dl}}{t_u}$, and the maximum value of m is $\frac{\mathsf{dl}}{t_u}$. We prove this statement at the end of Section 7.2.2, where we describe the timed language that the TTA accepts.

Each $\mathsf{run_{x,y}}$ location represents t_u time spent executing, due to the invariants $v_{ta} \leq t_u$ (see Definition 3.1). Since the difference between the WCET and BCET time of a task may be larger than the time unit t_u used for the discretization, we may need to introduce transitions from several $\mathsf{run_{x,y}}$, $x \in \{0 \dots n-1\}, y \in \{0 \dots m\}$ locations, to provide a way for the automaton to jump to the finish location (shown as curvy arrows in Figure 7.4). We introduce a transition from each $\mathsf{run_{x,y}}$, $k \leq x + y$ location to the finish location, where the constant k from Equation 7.4 is used to calculate the indeces of $\mathsf{run_{x,y}}$ locations from which the finish location is reachable, as the example shows in Figure 7.4.

The model shown in Figure 7.4 does not use any variables other than the valuation of a single clock (v_{ta}), and the valuation of the global time clock (v_t). The constants m, n, k used in the guards can be computed before the verification process, and therefore do not require extensions to TA [6]. However, there exist some extensions to TA that allow the use of integer variables, and the value of k, n, m can be encoded in integer variables, and therefore pre-computing these constants is not required when using modern model checkers such as UPPAAL [26] or the Verimag IF toolset [15]. The TTA model introduced in Section 4.3.3 is a more compact representation than the one shown in Figure 7.4 due to the use of integer variables that encode the x indeces of $run_{x,y}$ locations in the `et` variable, and the y indeces of $run_{x,y}$ locations in the `pre` variable. Thus, we encode the $\mathsf{run_{x,y}}$ locations in Figure 7.4 as two integer variables. The `pr` variable defines the precision of the approximation, and is denoted as time unit t_u in Figure 7.4.

7.2.2 Analysis of the Timed Automaton Approximation

In Section 7.2.1 we presented the construction rules for the approximation of the TSA using the TTA. In this section we prove the conservative nature of the approximation using the language inclusion problem.

The alphabet of the TTA is the alphabet of the TSA, $\Sigma = \{\mathsf{start}, \mathsf{stop}, \mathsf{enable_i}, \mathsf{enable_j}\}$. However, the TTA can only capture a bounded number of preemptions (per task). The syntax of the (untimed) TTA language can be described using regular expressions as follows:

$$S_{L(T)} = \mathsf{enable_i\ start\ (stop\ start)}^{0\ldots m}\ \mathsf{enable_j} \tag{7.5}$$

We see that $S_{L(T)} \subseteq S_{L(S)}$. The $\mathsf{run_{x,y}}$ locations of the TTA represent "checkpoints" in time; they store the discrete clock value and the number of preemptions occured

during the execution of the task. If a task is in the $\mathsf{run_{a,b}}$ location, then it has executed for at least $a \cdot t_u$ time, and has encountered b preemptions.

Definition 7.2 *We define the valuation $v_x = v_{ta} + x$, where v_{ta} is the valuation of c_{ta} as in Figure 7.4, and x is the index of locations $\mathsf{run_{x,y}}$ or $\mathsf{stop_{x,y}}$, where the automaton resides.*

Valuation v_x has a discrete-time component, that stores the checkpoints that have already been encountered, and a continuous-time component, that measures the time between checkpoints. Since timed automaton clocks cannot be resumed like a stopwatch, the valuation v_x is only an approximation of v_{sw}, that models the actual execution time accurately. Note that v_x is not used in the TTA model directly, we define it for the sole purpose of measuring the imprecision of using v_x to approximate clock value v_{sw}. To establish the relationship between the TSA and the TTA, we compare them on timed words that follow the syntax of the (untimed) regular expression $S_{L(T)}$. We assume that both the TSA and the TTA receive events in the same order, and with the same timestamps.

Proposition 7.1 *For any timed word r that follows the syntax of the (untimed) regular expression $S_{L(T)}$, during the co-simulation of word r on both the TSA and the TTA $v_{sw} - m \cdot t_u \leq v_x \leq v_{sw}$ holds from time 0 to the timestamp of the last event, where v_{sw} is the valuation of TSA clock c_{sw} shown in Figure 7.1, v_x is the valuation defined in Definition 7.2, and m is the number of* stop *events in the timed word (the number of preemptions).*

Proof: Since both the TSA and the TTA receive events with the same timestamps, their transitions will happen simultaneously. Whenever the TSA receives a start event, it makes a transition to the run location, and whenever it receives a stop event it moves to the stop location. Similarly, whenever the TTA receives a start event, it makes a

transition to one of the $\mathsf{run}_{\mathsf{x,y}}$ locations, and whenever it receives a stop event it moves to one of the $\mathsf{stop}_{\mathsf{x,y}}$ locations.

The stopwatch clock valuation v_{sw} increases with a slope of 1 between start and stop events in the run location, and is constant between stop and start events. The valuation v_x also increases with a slope of 1 between start and stop events in the $\mathsf{run}_{\mathsf{x,y}}$ locations, due to the continuous-value timed automaton clock valuation v_{ta} shown in Figure 7.4. Whenever a preemption happens, v_{ta} is reset (shown in Figure 7.4). Therefore, with each preemption, valuation v_x loses the value stored in the continuous-time component (v_{ta} becomes 0), whereas clock v_{sw} keeps its value, therefore, $v_x \leq v_{sw}$.

When the TTA receives a stop event, the clock valuation v_{ta} is in the $[0, t_u]$ interval, due to the invariants $v_{ta} \leq t_u$. Since the clock is reset, v_x decreases by at most t_u time, while v_{sw} hold its value when a preemption occurs. If there are m preemptions, the clock value v_x may decrease by at most $m \cdot t_u$ time compared to clock value v_{sw}, therefore $v_{sw} - m \cdot t_u \leq v_x$. □

Proposition 7.1 is the key to quantify the imprecision of the timed automaton approximation. It shows that valuation v_x increases slower than – or in the case when no preemptions occur with the same slope as – clock value v_{sw} due to the fact that the TTA does not keep track of the time spent in the $\mathsf{run}_{\mathsf{x,y}}$ location where it has received the stop event. These results imply that for the same timed word, it takes at least as much time for the valuation v_x to reach a given value, as it does for the stopwatch clock valuation v_{sw}.

Recall that WCET and BCET denote the longest and shortest possible execution times of a task, respectively, not including the time spent in the stop location. If we denote the time that the TSA has spent in the run location as τ_{run}, then $v_{sw} = \tau_{run}$, based on the fact that v_{sw} is simply a function of v_t.

In the TTA model we also use the WCET, BCET parameters to obtain guards on transitions. Proposition 7.1 shows that valuation v_x follows clock value v_{sw} with some

imprecision. The time that the TTA spends in $\mathsf{run}_{\mathsf{x,y}}$ locations equals τ_{run}, because anytime the TTA receives a start event it makes a transition to a $\mathsf{run}_{\mathsf{x,y}}$ location, and anytime it receives a stop event it moves to a $\mathsf{stop}_{\mathsf{x,y}}$ location, just like the TSA does. Therefore, according to Proposition 7.1, $v_{sw} - m \cdot t_u \leq v_x \leq v_{sw} = \tau_{run}$.

Definition 7.3 *We refer to the valuation of global time clock* (v_t) *when* $v_x =$wcet *as actual worst case execution time, and denote it as* t_{wcet}. *The valuation of global time clock* (v_t) *when* $v_x =$bcet *is referred to as actual worst case execution time, and denoted as* t_{bcet}

Proposition 7.2 *For any timed word* r *that follows the syntax of the (untimed) regular expression* $S_{L(T)}$, *if* $v_x =$wcet *holds anytime during the run of word* r *on the TTA, then* wcet$\leq t_{wcet}$.

Proof: The invariants $v_{ta} \leq t_u$ in the TTA imply that the TTA spends at most t_u time in each $\mathsf{run}_{\mathsf{x,y}}$ location. Time spent in $\mathsf{stop}_{\mathsf{x,y}}$ locations is not part of the execution time and therefore does not contribute to t_{wcet} (only to the deadline). Moreover, we reset the continuous-time component (clock valuation v_{ta} in Figure 7.4) whenever we leave a $\mathsf{stop}_{\mathsf{x,y}}$ location, therefore the value of clock valuation v_x when it leaves the stop location is less than or equal to its value when it entered the location. Neither t_{wcet}, nor v_x increases by passing through $\mathsf{stop}_{\mathsf{x,y}}$ locations, and therefore we can abstract these location out here. If the TTA receives no stop events, it spends t_u time in n locations, then wcet$-n \cdot t_u$ time in the last location, therefore if no preemptions occur, t_{wcet} is wcet in the TTA.

Create a directed graph $G_M = (V_M, E_M)$ such, that for each $\mathsf{run}_{\mathsf{x,y}}$ location add a vertex $v_{x,y}$ in graph G_M. For all vertices $v_{x,y}, x \in 0 \ldots n-1, y \in 0 \ldots m-1$ add a directed edge from $v_{x,y}$ to $v_{x+1,y}$, and a directed edge from $v_{x,y}$ to $v_{x,y+1}$. Add a terminal vertex z to graph G_M, and add edges from each $v_{n,y}$ to z. We only add edges from $v_{n,y}$ locations because we are interested in the worst case execution time

of the task, and therefore we require the automaton to go through (at least) n $\mathsf{run_{x,y}}$ locations. The edges in G_M specify the order in which $\mathsf{run_{x,y}}$ locations can follow each other in the TTA, the source represents the initial idle location, and the terminal represents the finish location. The graph G_M is an abstract model of the TTA shown in Figure 7.4. The shortest path in G_M from $v_{0,0}$ to t is the $s, v_{0,0}, v_{1,0}, \ldots v_{n,0}, t$ path, that corresponds to the case when the TTA receives no stop events. Anytime the TTA receives a stop event, it needs to go through additional $\mathsf{run_{x,y}}$ locations, and therefore t_{wcet} increases. □

Proposition 7.3 *For any timed word r that follows the syntax of the (untimed) regular expression $S_{L(T)}$, if $v_x =$bcet holds anytime during the run of word r on the TTA, then $t_{bcet} \leq$bcet.*

Proof: We build on the G_M graph introduced in Proposition 7.2. Since we are interested in the best case, we add an edge from each $v_{x,y}$ vertex to t, where $k \leq x+y$, not just from $v_{n,y}$ vertices. Since we introduced edges from each $\mathsf{run_{x,y}}$, $k \leq x + y$ vertex, the shortest path from s to t is $k+2$ ($k \cdot v_{x,y}$ vertices, plus start (s) and terminal (t)locations), regardless of the value of y, that represents the number of preemptions. Each vertex represents a $\mathsf{run_{x,y}}$ location, where we spend at most t_u time, due to the invariant $v_{ta} \leq t_u$, and therefore v_t is at most $k \cdot t_u$ when the TTA reaches a location that has a transition to the finish location. The guard on all transitions from $\mathsf{run_{x,y}}$ locations to the finish location is bcet$-k \cdot t_u$, and bcet$-k \cdot t_u + k \cdot t_u =$bcet, therefore t_{bcet} is bcet or less in the TTA. □

Now that we established the relation between the TSA and the TTA, we focus on the language that the TTA accepts. The timestamps of all events have to be less than dl in a timed word in order for the TSA to accept the word. We discard the $\mathsf{enable_j}$ event, since it might not correspond to the worst case. Similarly to the notations used with the TSA, we denote the timestamps of events in the $[0 \ldots \mathsf{dl}]$ interval as $t_1, t_2, \ldots, t_e$, where t_1 denotes the $\mathsf{enable_i}$ event, and t_e denotes the last start event.

The number of preemptions $m = \frac{e}{2}$ in Proposition 7.1, since every second event is a stop event (and e is even). Therefore, $\sum_{i=1}^{\frac{e}{2}} t_u = m \cdot t_u$, corresponding to the maximum difference between clocks c_{sw} and v_x on any timed word over Σ, as shown in Proposition 7.1. Accordingly, we conclude, that the TTA accepts the timed language that has a syntax as described in Equation 7.5, and satisfies the following constraint:

$$\sum_{i=1}^{\frac{e}{2}} \tau_{2i} - \tau_{2i-1} + t_u \leq \mathsf{dl} - \mathsf{wcet} \tag{7.6}$$

Equation 7.6 states that a task is schedulable if it spends at most $\mathsf{dl}-\mathsf{wcet}-m \cdot t_u$ time in $\mathsf{stop}_{\mathsf{x,y}}$ locations in the TTA within the $[0, \mathsf{dl}]$ interval, since in this case it can execute for WCET time before its deadline. We need to subtract $m \cdot t_u$ from the available time to compensate for the imprecision of the TA approximation described in Proposition 7.1.

We now show that the maximum value of m is $\frac{\mathsf{dl}}{t_u}$ in Equation 7.4. From Equation 7.6 we see that the maximum value for m defined in Equation 7.4 is $\frac{\mathsf{dl}}{t_u}$, since the number of preemptions $m = \frac{e}{2}$, therefore in this case $\sum_{i=1}^{\frac{\mathsf{dl}}{t_u}} t_u = \mathsf{dl} \geq \mathsf{dl}-\mathsf{wcet}$, therefore the constraint specified in Equation 7.6 will never be satisfied when the number of preemptions encountered is more than $\frac{\mathsf{dl}}{t_u}$.

7.2.3 Language Inclusion Problem for a Single TTA/TSA Pair

The language inclusion problem for SA can be described as follows; given two SA T and S, are all timed traces accepted by T also accepted by S? For the proof we proceed with the common method of complementation and emptiness checking of the intersection [47]: $L(T) \subseteq L(S)$ if and only if $L(T) \cap \overline{L(S)} = \emptyset$.

Theorem 7.1 *The TSA accepts the timed language over Σ that the TTA accepts: $L(T) \subseteq L(S)$.*

Proof: $S_{L(T)} \subseteq S_{L(S)}$, therefore it is sufficient to show that $L(T) \subseteq L(S)$ holds over timed words that can be expressed using the (untimed) syntax $S_{L(T)}$. Let t_{stop} denote the expression $\sum_{i=1}^{\frac{e}{2}} \tau_{2i} - \tau_{2i-1}$, the time that the TSA spends in the stop location. The timestamps of events are the same in the TTA and TSA traces, as we compare the timed languages $L(T)$ and $L(S)$ word by word. Then, the time constraints on the timed language $L(T) \cap \overline{L(S)}$ can be expressed as $t_{stop} + \sum_{i=1}^{\frac{e}{2}} t_u \leq \mathsf{dl{-}wcet} \cap t_{stop} > \mathsf{dl{-}wcet}$. Since $t_u \in \mathbb{R}_{\geq 0}$, therefore $0 \leq \sum_{i=1}^{\frac{e}{2}} t_u$. Since t_{stop} cannot be both smaller than or equal to dl - wcet and less than dl - wcet, therefore the intersection of $L(S)$ and $L(T)$ is the empty set, that implies that the TSA accepts the timed language over Σ that the TTA accepts, and $L(T) \subseteq L(S)$ holds. □

7.2.4 The Effects of Composing TTA on the Approximation

We now show that since t_{wcet} of the TTA is larger than t_{wcet} of its corresponding TSA, and t_{bcet} of the TTA is smaller than t_{bcet} of its corresponding TSA, therefore the composition of TTA models is a conservative approximation of the composition of TSA.

Theorem 7.1 shows that the TSA accepts the language that the TTA accepts ($L(T) \subseteq L(S)$). As described in Section 7.1, the composition of TSA as a task graph turns reachability analysis undecidable [55], which motivated our work to approximate TSA task graphs using TTA task graphs. In this Section we show that the composition of TTA as a task graph (denoted as G_S in Section 7.1.2) does not invalidate the results of Theorem 7.1. For the conservative approximation of TSA task graphs, we replace each TSA in the task graph with a TTA. We denote this graph as $G_T = (X_T, E_T)$, where each vertex in set X_T is a TTA. As each timed automaton is also a stopwatch automaton, TTA compose using events the same way as TSA do. Graphs G_S and G_T are a representation of applying the $\oplus$ parallel composition operator, and the $\otimes$ serial composition operator to TSA and TTA models, respectively.

Therefore, we need to consider how these operators may influence the timestamps of events.

When parallel composition is used between two automata $(x_k \oplus x_l), x_k \in X_T, x_l \in X_T$, then the two automata do not depend on each other and can be analyzed independently. Therefore, the parallel composition of TSA and TTA models does not influence the timestamps of events.

We now consider the case when serial composition is used between two automata $(x_k \otimes x_l), x_k \in X_T, x_l \in X_T$. Denote the language of x_k as $L(T)_k$, and the language of x_l as $L(T)_l$. Denote the alphabet of x_k as $\Sigma_k = \{\mathsf{start_k}, \mathsf{stop_k}, \mathsf{enable_{k_k}}, \mathsf{enable_{l_k}}\}$, and the alphabet of x_l as $\Sigma_l = \{\mathsf{start_l}, \mathsf{stop_l}, \mathsf{enable_{k_l}}, \mathsf{enable_{l_l}}\}$. Since the timestamp of $\mathsf{enable_{l_k}}$, and the timestamp of $\mathsf{enable_{k_l}}$ is the same by definition, therefore the timestamps of events in Σ_l may be influenced by the timestamp of $\mathsf{enable_{l_k}}$.

The $\mathsf{enable_{l_k}}$ event signals the end of the execution of TTA_k, and is raised when v_x is within the $[\mathtt{bcet_k}, \mathtt{wcet_k}]$ interval. Therefore, the timestamp of $\mathsf{enable_{l_k}}$ is influenced by the imprecision between v_x and v_{sw} described in Proposition 7.1. Proposition 7.2 shows, that if $v_{x_k} = \mathtt{wcet_k}$ holds, then $\mathtt{wcet_k} \leq t_{wcet_k}$. Also, Proposition 7.3 shows, that if $v_{x_k} = \mathtt{bcet_k}$ holds, then $t_{bcet_k} \leq \mathtt{bcet_k}$. Therefore, if $v_{x_k} = \mathtt{wcet_k}$ holds for a TTA, then the timestamp of $\mathsf{enable_{l_k}}$ is in the $[t_{bcet_k}, t_{wcet_k}]$ real-valued interval, and $[\mathtt{bcet_k}, \mathtt{wcet_k}] \subseteq [t_{bcet_k}, t_{wcet_k}]$. This implies that TTA_k can generate all the timestamps for event $\mathsf{enable_{l_k}}$, that its corresponding TSA_k model can generate, if $v_{x_k} = \mathtt{wcet_k}$ can be satisfied. If it cannot, then the TTA will report the task as unschedulable (that may or may not be true). Therefore, the proposed approximation method provides a sufficient, but not necessary condition, in general, to determine the schedulability of TSA models composed using the $\oplus$ and $\otimes$ operators.

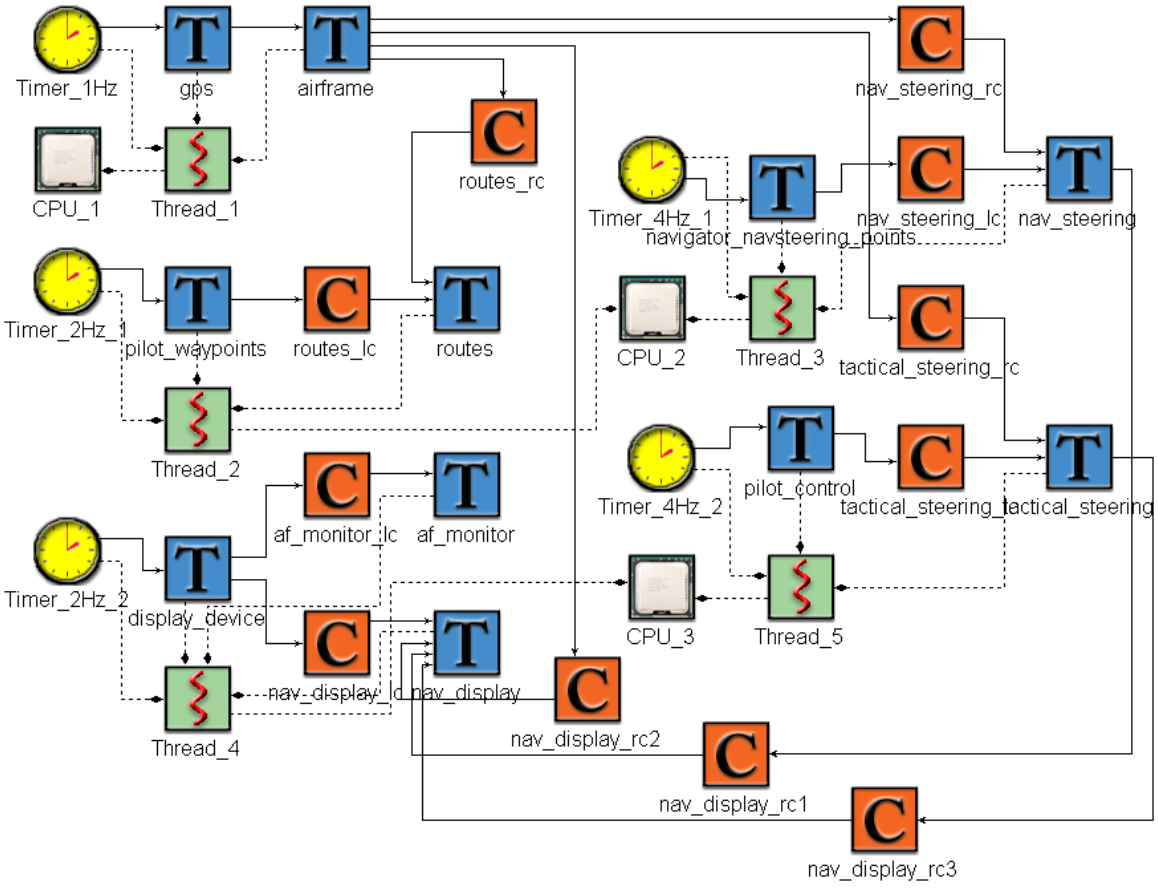

Figure 7.5: Real-time CORBA Avionics Application

7.3 Practical Application

We applied the proposed conservative approximation method to analyze a PEARSE design shown in Figure 7.5, loosely based on the real-time CORBA avionics application described in Section 5.4. In this section, we map the application to a preemptive execution platform for the real-time verification, with slightly different execution parameters.

Tasks represent software components, and are denoted as `T`, FIFO event channels are denoted `C`. Timers send out events periodically, driving the computation in the design. Arrows represent dependencies between tasks. Tasks are mapped to threads as defined by the dashed lines. Within each thread, fixed-priority non-

Timer	Period
Timer_1Hz	1000
Timer_2Hz_1	500
Timer_2Hz_2	500
Timer_4Hz_1	250
Timer_4Hz_2	250

Task	WCET	BCET	Deadline
gps	21	18	100
airframe	53	50	100
pilot_wayp...	37	35	300
routes	18	15	250
display_device	26	26	250
af_monitor	33	32	150
nav_display	14	12	150
nav_steering	69	65	150
navigator...	42	42	100
pilot_control	43	37	80
tactical_st...	58	52	100

Table 7.1: Parameters for the Real-time CORBA Case Study Shown in Figure 7.5

preemptive scheduling is used, and fixed-priority preemptive scheduling is used between threads. Both CPU_2 and CPU_3 use preemptive scheduling. FIFOs are scheduled non-concurrently (*i.e.* they are always ready to execute). Communication between software tasks is fully asynchronous and event-driven. Overall, there are 11 tasks in the design and 11 FIFO buffers, that execute on 5 threads on 3 CPUs. Execution parameters for tasks are shown in Table 7.1.

Each task is represented as a TTA using the UPPAAL syntax defined in Section 4.3.3 and FIFO channels are modeled as the `Channel` and `Buffer` constructs introduced in Section 4.3.4 and Section 4.3.5. Figure 7.6 and Figure 7.7 show the TA representation of the real-time CORBA avionics application shown in Figure 7.5. As discussed in Section 4.3.2, error locations are defined as `committed` for each TTA to ensure that the model deadlocks whenever a deadline is exceeded, or event is lost. Thus, when no deadlocks occur then all deadlines are met.

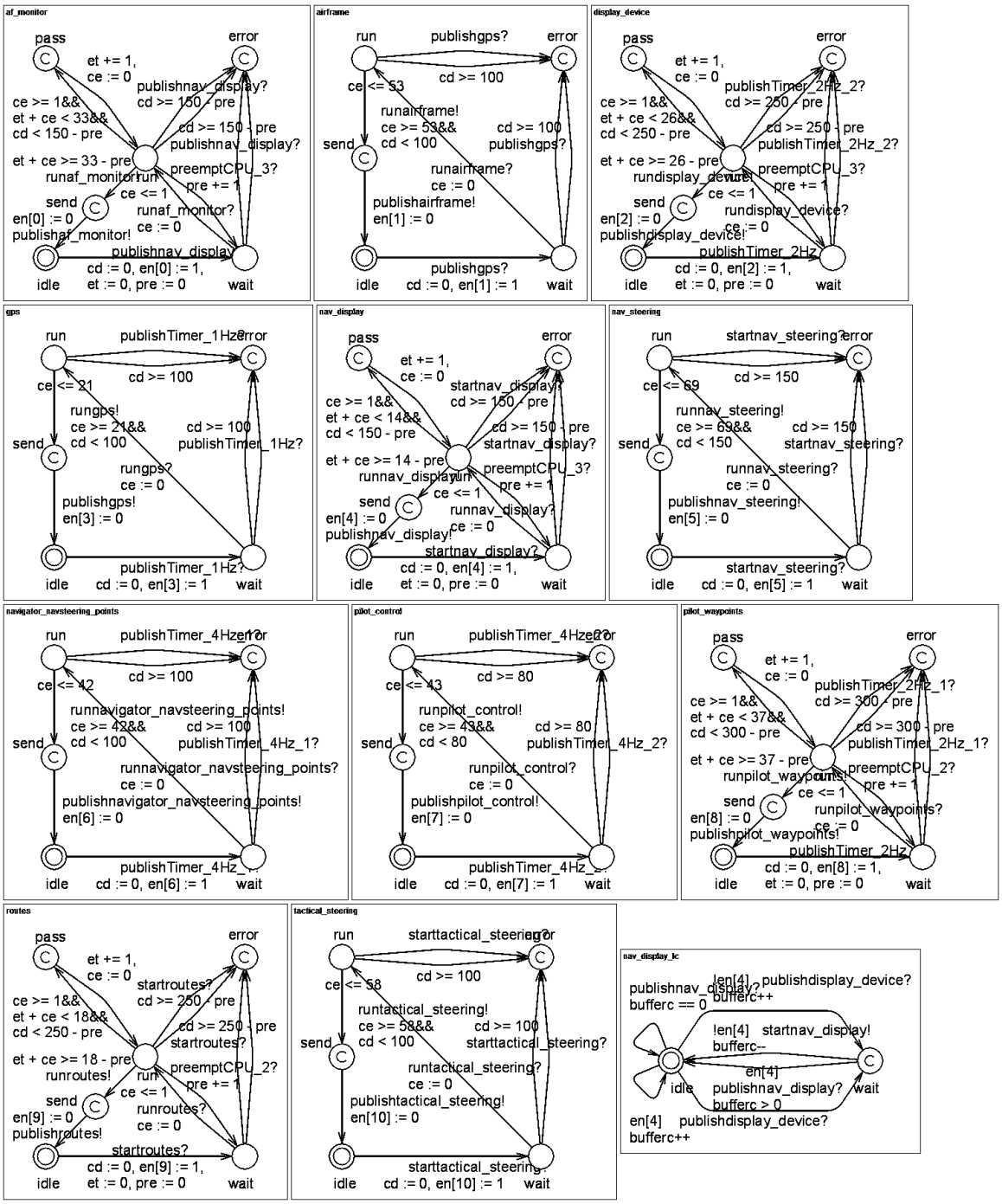

Figure 7.6: Uppaal Timed Automata Models for the Avionics Application Shown in Figure 7.5 (Part 1/2)

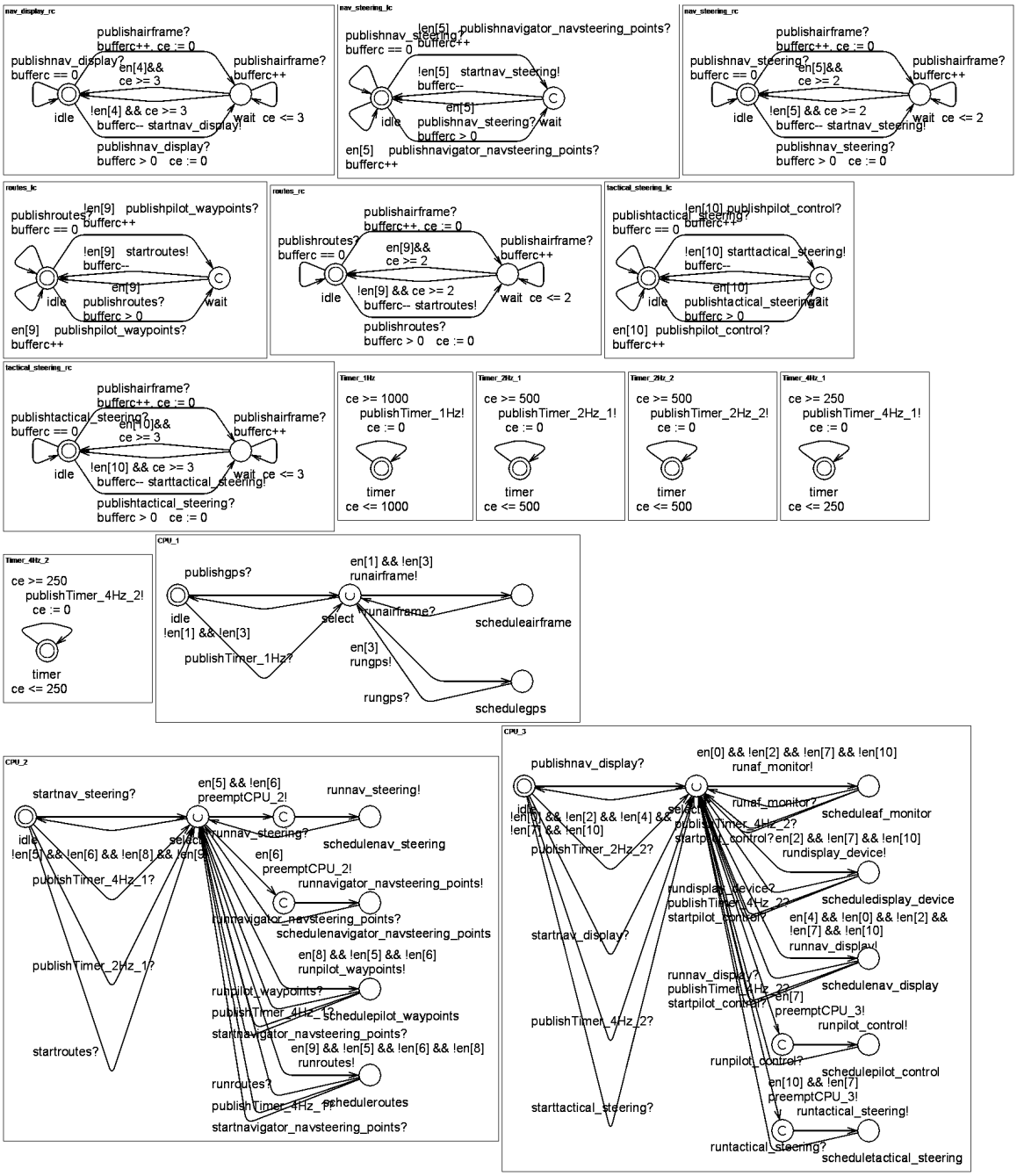

Figure 7.7: Uppaal Timed Automata Models for the Avionics Application Shown in Figure 7.5 (Part 2/2)

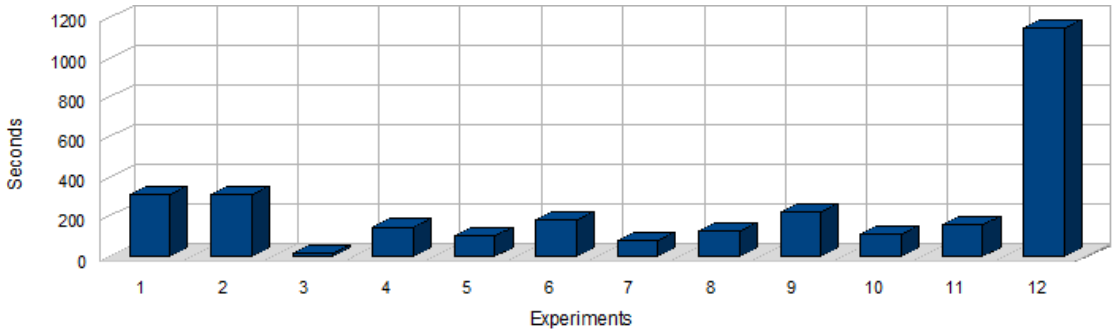

Figure 7.8: Model Checking Time

We have checked the schedulability of the TA model using the the UPPAAL model checker [26] by issuing the `A[] not deadlock` macro (Experiment 1). We then ran experiments where in each step we halved both the BCET and WCET of a single task (first `gps`, then `airframe` etc.) We used the highest possible precision for preemptive tasks, the value of the clock is saved at every integer value during the execution of a task. Experiments were executed on an Intel Core i7 i920 processor running at 4GHz, using 6GB three-channel RAM. Model checking time for the 12 experiments is shown in Figure 7.8, and the memory used is shown in Figure 7.9.

Both the verification time and memory consumption vary as a function of non-determinism, which is influenced by many factors, including the actual execution parameters, the size of execution intervals, the number of concurrently executing tasks, as well as the number of tasks. That said, complexity cannot simply be judged as a factor of size.

In Experiment 3, where the WCET and BCET parameters for the `airframe` task are halved (to 27 and 25, respectively), the change results in a deadline miss in the `nav_display` task. This means that we did not find a sufficient condition for schedulability, and the design may or may not be schedulable. All other experiments proved the design schedulable with the given parameters. Decreasing the execution parameters for the `tactical_steering` task greatly increased verification time and memory consumption. The cause of the complexity increase is unknown to us, but

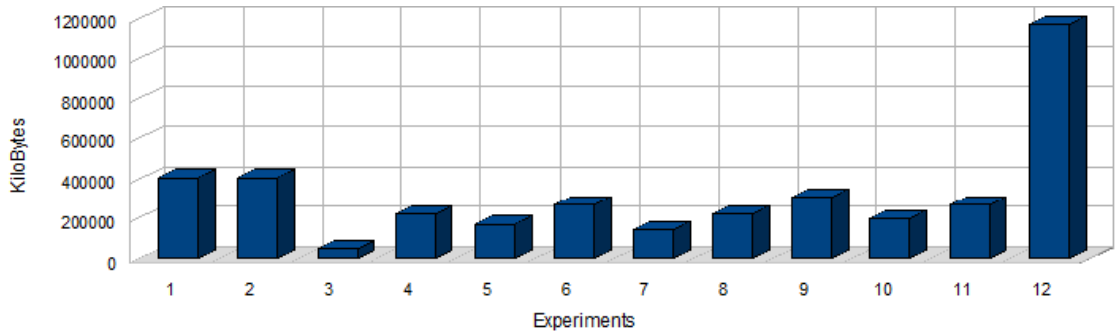

Figure 7.9: Model Checking Memory Consumption

we suspect that the changes have increased the non-determinism in the model (*i.e.* by introducing race conditions, or non-deterministic execution order).

Several improvements may increase scalability in real-life problems. First of all, the proposed method allows for *hierarchical model checking*, since preemptive components can be encapsulated into non-preemptive "wrappers", acting as a black box. Since intervals for communication are captured, there is no need to model all components at once. We plan to investigate this direction in the future. Second, UPPAAL does not take advantage of multi-core processors or distributed clusters. Model-checking algorithms that are CPU-bound rather than memory-bound – such as the algorithm described in [70] – have the potential to leverage multi-core hardware and may provide better performance in the future. There is room for optimization in current model checkers. Given the resources, the real-time verification of large-scale designs is within reach.

7.4 Concluding remarks

This chapter presented a conservative approximation method for the verification of *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)*. The proposed method is based on TA model checking methods, and inherently captures asynchrony and dependencies between tasks and provides a way for

the formal analysis of practical embedded systems. We have shown that the approximation provides a sufficient, but not required condition to determine the schedulability of distributed asynchronous event-driven systems using preemptive scheduling. The practical application of the method was shown on a real-time CORBA avionics application. This chapter presents a conservative approximation method for the verification of *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)*. The proposed method is based on TA model checking methods, and inherently captures asynchrony and dependencies between tasks and provides a way for the formal analysis of practical embedded systems. We have shown that the approximation provides a sufficient, but not required condition to determine the schedulability of distributed asynchronous event-driven systems using preemptive scheduling. The practical application of the method was shown on a real-time CORBA avionics application. The conservative approximation method for PEARSE has been implemented in the open-source *Distributed Real-time Embedded Analysis Method (*DREAM*)* tool available at `http://dre.sourceforge.net`.

Chapter 8

Combining Transaction-level Simulations and Model Checking for MPSoC Verification and Performance Evaluation

Modern *Multi-processor Systems-on-Chip (MPSoCs)* are deeply embedded electronic systems operating in resource-constrained environments, that consist of several heterogeneous components such as programmable processors, custom logic blocks, memories, and peripherals, all of which are connected together via an interconnection network. The complexity and functionality of these systems often rivals that of high-performance processors from a decade ago, at a fraction of price and energy costs. MPSoC designs must satisfy increasingly complex performance constraints for emerging applications, that is becoming more and more challenging for system designers because of the large number of components on a chip that have multifaceted dependencies and interactions with each other.

While MPSoC designs themselves can be viewed as *Distributed Real-time Embedded (DRE)* systems, the communication subsystem in MPSoC designs has a major impact on both design and analysis. Unlike software-intensive DRE systems that communicate over packet-switched networks, MPSoC designs often utilize complex bus matrix architectures, where access to the bus is managed by an *arbiter* (or several arbiters). In particular, deadlock-freedom and livelock-freedom is not guaranteed by bus protocols, but is a key requirement for designers.

Bus interconnect standards such as the *ARM Advanced Microcontroller Bus Architecture Advanced High-speed Bus (AMBA AHB)* [8] and *CoreConnect* [48] are commonly used to integrate heterogeneous components into MPSoC designs. Bus protocols provide reliable communication in MPSoC systems by specifying standard methods for

interaction between components connected to the bus. Key issues that bus protocols must address include synchronization, dealing with concurrent requests, transmission errors, preventing deadlocks, and *Quality of Service (QoS)* support for MPSoC designs.

Despite the fact that bus protocols have a critical role in providing a reliable platform in MPSoC systems, their specifications are typically written as a combination of natural languages and timing diagrams. Although this approach is effective in explaining basic use cases to developers, it cannot cover every possible use case, and introduces *ambiguity* in the specification. This ambiguity is especially troublesome when heterogeneous *Intellectual Property (IP)* blocks have to be integrated on the bus, as different vendors might implement the ambiguous parts of the specification differently. Thus the *interoperability* of such components can be at risk.

Although most vendors provide test vectors to validate and certify whether components work with a bus protocol, there is no well-defined methodology to check whether the system as a whole satisfies high-level design constraints. Simulation-based approaches have been found useful in designing large-scale embedded systems, however, they can only show the presence of errors, not their absence. Moreover, simulations are time-consuming limiting designers to a few test cases. The contributions of this chapter focus on the following areas:

- We define a formal model for the AMBA AHB protocol based on the *Finite State Machine (FSM) Model of Computation (MoC)*. We develop a cycle-accurate model for MPSoC designs based on AMBA AHB. This work is described in Section 8.1.

- We describe a set of digital camera MPSoC design alternatives based on the *JPEG2000* [53] still image compression standard in Section 8.2.

- We describe our approach for the functional verification of the digital camera design alternatives in Section 8.3. We describe an ambiguity in the AMBA AHB

protocol specification, and our approach to resolve the ambiguity.

- We utilize the formal models developed in Section 8.1 to propose a method for performance estimation by combining transaction-level simulations and model checking in Section 8.4.

8.1 Formal Modeling of the AMBA AHB protocol

Over the past decade and a half, several bus-based on-chip communication architecture standards have been proposed to handle the communication needs of emerging MPSoC designs. Of these, the ARM Microcontroller Bus Architecture (AMBA) version 2.0 [8] is one of the most widely used on-chip communication standards to interconnect components in MPSoC designs. The goal of this standard is to provide a flexible, high-performance bus architecture specification that is technology-independent, takes up minimal silicon area and encourages IP reuse across designs. The AMBA 2.0 bus architecture standard defines three buses: *ARM Advanced Microcontroller Bus Architecture Advanced High-speed Bus (AMBA AHB)*, *ARM Advanced Microcontroller Bus Architecture Advanced Peripheral Bus (AMBA APB)* and *ARM Advanced Microcontroller Bus Architecture Advanced System Bus (AMBA ASB)*. The AMBA AHB bus is used for high bandwidth and low latency communication, primarily between *Central Processing Unit (CPU)* cores, high performance peripherals, *Direct Memory Access (DMA)* controllers, on-chip memories and interfaces such as bridges to the slower AMBA APB bus. AMBA APB is used to connect slower peripherals such as timers, *Universal Asynchronous Receiver/Transmitters (UARTs)* etc. and uses a bridge to interface with AMBA AHB. It is a simple bus that does not support the advanced features of the AMBA AHB bus. The AMBA ASB bus is an earlier version of the high-performance bus that has been superseded by AMBA AHB in current designs. Since we use the AMBA AHB bus standard in this chapter, we present a brief overview

of is features next.

The *ARM Advanced Microcontroller Bus Architecture Advanced High-speed Bus (AMBA AHB)* is a high-speed, high-bandwidth bus that supports multiple masters. AMBA AHB supports pipelined data transfers for high speed memory and peripheral access without wasting precious bus cycles. Burst transfers allow optimal usage of memory interfaces by giving advance information of the nature of the transfers. AMBA AHB also allows split transactions which maximize the use of the system bus bandwidth by enabling high latency slaves to release the system bus during the dead time while the slave is completing its transaction. AMBA AHB architectures can have various topologies such as single shared bus, multi-layer (or hierarchical) shared bus and bus matrix. A designer can select any of these topologies based on MPSoC communication requirements and customize it to optimize overall bandwidth and improve performance. An AMBA AHB bus consists of an address bus (typically 32 bits wide) and separate (or shared) read and write data buses that have a minimum recommended width of 32 bits, but can have any values ranging through 8, 16, 32, 64, 128, 256, 512 or 1024 bits, depending on application bandwidth requirements, component interface pin constraints and the bit width of words accessed from memory modules (*i.e.* embedded DRAM).

When a master needs to send or receive data in AMBA AHB, it requests an arbiter for access to the bus by raising the `HBUSREQx` signal. The arbiter, in turn, responds to the master via the `HGRANTx` signal. Depending on which master gains access to the bus, the arbiter drives the `HMASTERx` signals to indicate which master has access to the bus (this information is used by certain slaves). When a slave is ready to be accessed by a master, it drives the `HREADYx` signal high. Only when a master has received a bus grant from the arbiter via `HGRANTx` and detects a high `HREADY` signal from the destination slave, will it initiate the transaction. The transaction consists of the master driving the `HTRANSx` signal, which describes the type of transaction

(sequential or non-sequential), the HADDRx signals which are used to specify the slave addresses, and HWDATAx if there is write data to be sent to the slave. Any data to be read from the slave appears on the HRDATAx signal lines. The master also drives control information about the data transaction on other signal lines; HSIZEx (size of the data item being sent), HBURSTx (number of data items being transferred in a burst transaction), HWRITE (whether the transfer is a read or a write) and HPROTx (contains protection information for slaves which might require it).

8.1.1 Modeling AMBA AHB by Finite State Machines

In this section we formalize our model for the AMBA AHB protocol. The notion of time used in the protocol specification is discrete (bus cycle), therefore the protocol can be represented as a *Discrete Event (DE)* system. There are several models of computation that can express DE systems, well-known examples include FSMs, Petri-nets and data-flow networks. *Timed Automata (TA)* and *Hybrid Automata (HA)* are extensions to FSMs in order to express the continuous evaluation of system variables, and are therefore too heavyweight to represent cycle-based bus protocols. The DE model is simpler than TA or HA, and offers a more scalable approach for verification by utilizing *Binary Decision Diagrams (BDDs)* [18]. We propose the use of the FSM MoC for the representation of the AMBA AHB bus protocol as a DE system. We chose FSMs mainly because they are supported by several model checkers [1] [56] [46].

We have created a cycle-accurate model of the AMBA AHB bus in order to model bus transactions accurately. To ensure that a single split transfer or RETRY response does not deadlock the bus as described in [88], we assume that the arbiter only grants access to a new master when the HRESP signal is OK and the HTRANS signal is IDLE. This introduces an extra cycle arbitration delay in the model. We model arbitration delays, pipelining, and busy slaves (HREADY is 0) in the bus as well. We have also modeled the two-cycle response times for RETRY and SPLIT responses according to

the AMBA AHB specification. We implemented a round-robin arbiter, mainly to avoid starvations that might arise when a fixed-priority arbiter is used. We consider `RETRY` responses from the slave (`HRESP = RETRY`), as well as split transfers.

We do not model `HLOCK` signals, that are set by a master than needs uninterrupted access to the bus during a transaction. When `HLOCK` is set by the master, the arbiter simply holds its state as it is forced to grant access to the locking master as long as the signal remain active. It is easy to see that a master that asserts `HLOCK` and does not deassert it essentially causes a deadlock. Unless the master is faulty, this condition should not occur in practice. The `HLOCK` signal set by a master overrides the arbiter, and therefore it is the responsibility of the master to ensure that it eventually deasserts this signal. As long as the master deasserts `HLOCK`, no deadlocks occur, as the arbiter continues where it left off before the `HLOCK` signal was set.

8.1.2 Modeling AMBA AHB Masters

We modeled a generic AMBA AHB master as a FSM with six states (`idle, busreq, haddr, read, write, error`) as shown in Figure 8.1. This FSM provides a "black box" model for AMBA AHB masters as seen from the bus. The master requests access to the bus, then reads, and finally writes to the bus. The `error` state is used to check for inconsistent replies from the slave/arbiter, and turns protocol checking into a reachability problem; in correct protocols, the `error` state should not be reachable. By specifying how much time the FSM spends in each state we can capture performance analysis in a formal setting.

Algorithm 8.1 shows the NuSMV syntax for the FSM shown in Figure 8.1. Transitions are specified within the `case ...esac;` block. Transitions are ordered deterministically; the next value of `state` will be specified by the first guard that evaluates to `true`. The figure is only partial; the `HADDR`, `HTRANS`, `HRDATA`, `HWDATA`, and `BUSREQ` signals depend on the `state` of the master. The `MASTER_STATE` variable is used for the

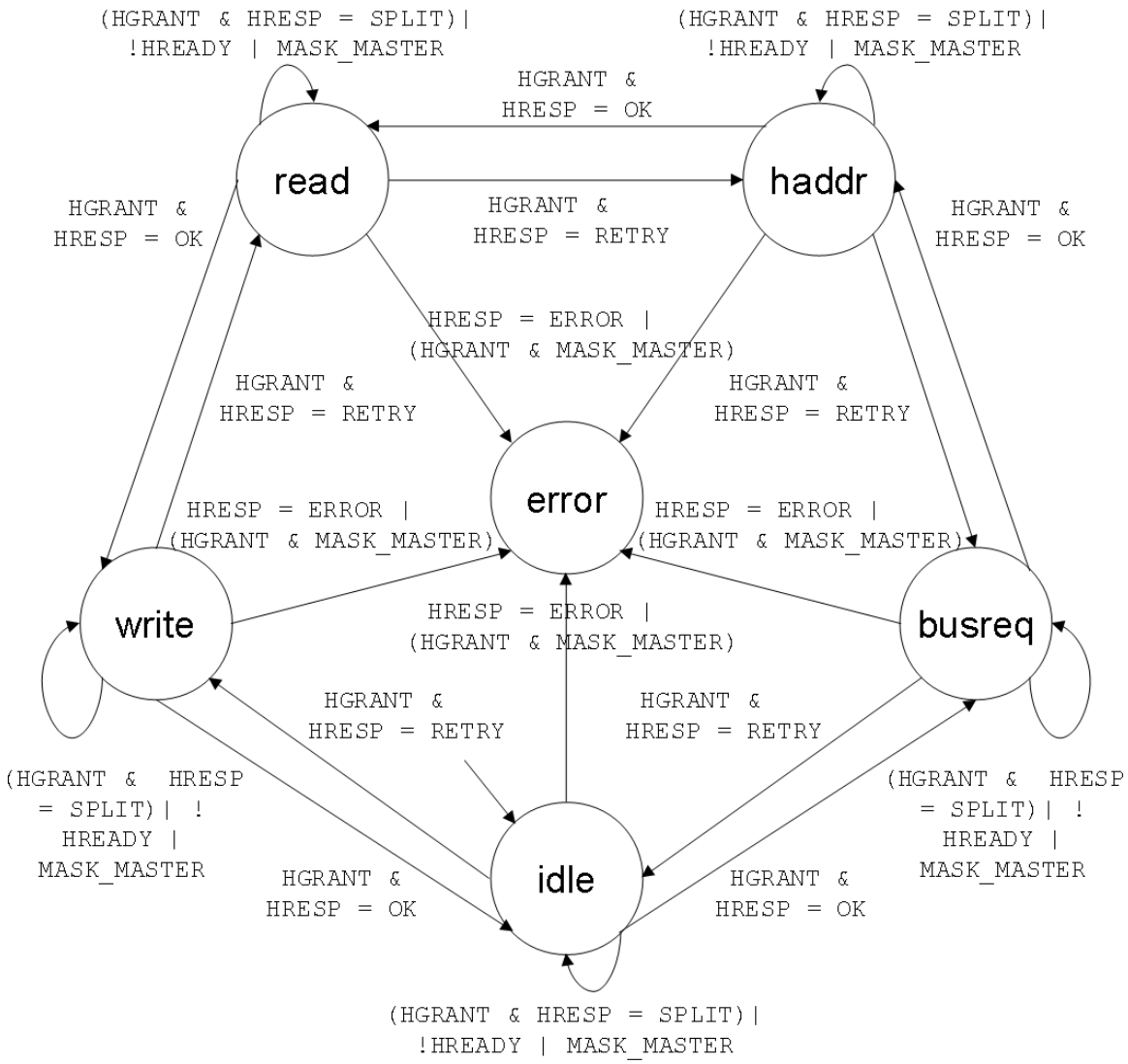

Figure 8.1: Finite State Machine Model of an AMBA AHB Master

performance evaluation described in Section 8.4 to provide a fast and simple method to track the master's current state from the arbiter.

The *Best Case Execution Time (BCET)* and *Worst Case Execution Time (WCET)* parameters are given as inputs to the AMBA AHB master. The `READSIZE` and `WRITESIZE` parameters specify the size of the data read from and written to the bus. The BCET, WCET, `READSIZE`, and `WRITESIZE` parameters are provided by the simulations.

Algorithm 8.1 Partial NuSMV Finite State Machine Model for an AMBA AHB Master

```
MODULE master_read_write (BUSREQ, HGRANT, MASTER_STATE, MASK_MASTER, BCET,
WCET, READSIZE, WRITESIZE, START, FINISH)
  VAR
   state :  idle, busreq, haddr, read, write, busy, error;
   prev_state :  idle, busreq, haddr, read, write, busy, error;
   io :  read, write;
   ET : 0..MAXET;
   SIZE : 1..MAXSIZE;
   HADDR : boolean;
   HTRANS : IDLE, NONSEQ, SEQ, BUSY;
   HWDATA : boolean;
  ASSIGN
   init (state) := idle;
   init (io) := read;
   init (SIZE) := 1;
   init (prev_state) := idle;
   next (prev_state) := idle;
   next (state) :=
    case
     HRESP = ERROR : error;
     MASK_MASTER & HGRANT : error;
     HRESP = SPLIT & HGRANT : state;
     !HREADY : state;
     MASK_MASTER : state;
     HRESP = RETRY & HGRANT : prev_state;
     state = idle & START & READSIZE = 0 :  busy;
     state = idle & START : busreq;
     state = idle :  idle;
     state = busreq & HGRANT : haddr;
     state = busreq & !HGRANT : state;
     state = haddr & HGRANT : read;
     state = read & HGRANT : write;
     state = write & HGRANT & SIZE = READSIZE & io = read :  busy;
     state = write & HGRANT & SIZE = WRITESIZE & io = write :  idle;
     state = write & HGRANT & SIZE < READSIZE & io = read :  haddr;
     state = write & HGRANT & SIZE < WRITESIZE & io = write :  haddr;
     state = busy & ET < BCET : busy;
     state = busy & ET = WCET : busreq;
     state = busy & BCET <= ET : busy, busreq;
     1:  error;
    esac;
...
```

8.1.3 Modeling AMBA AHB Slaves

We modeled a generic AMBA AHB slave using four states (`idle`, `write`, `read`, `error`) as shown in Figure 8.2. The transitions of the slave have to be synchronized with the master – *i.e.* the slave has to be in the `read` state when the master is in the `write` state – otherwise the slave (and the master) will enter the `error` state. Thus by verifying that the `error` state is unreachable from both the master and the slave we can prove that the master and slave communicate with each other as expected.

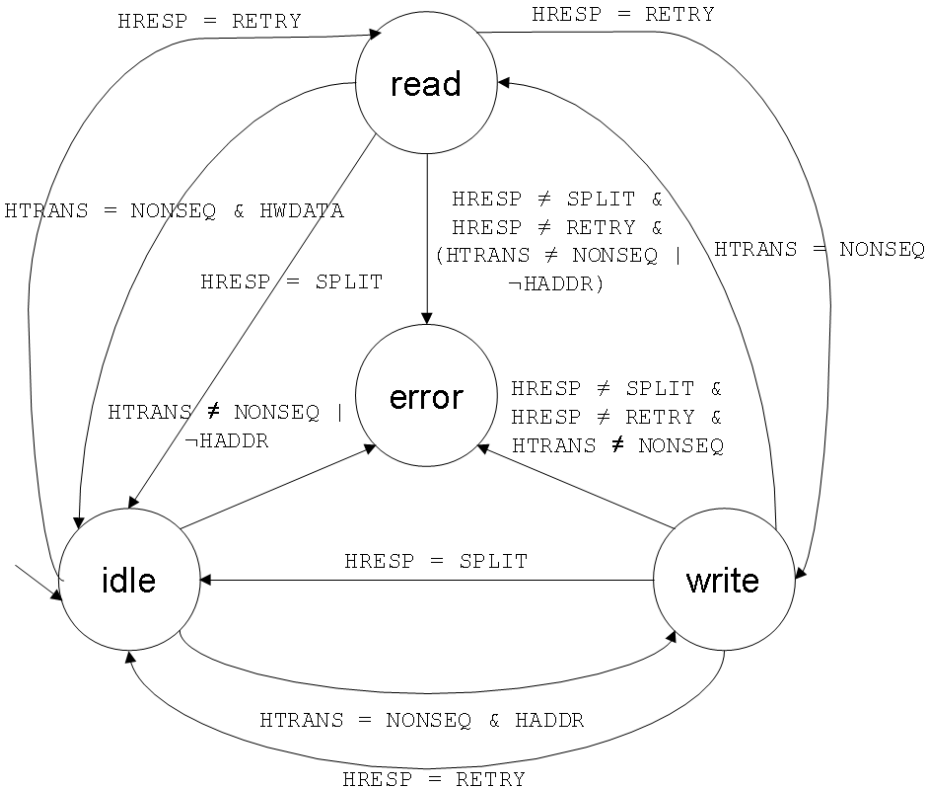

Figure 8.2: Finite State Machine Model of an AMBA AHB Slave

Algorithm 8.2 shows the NuSMV syntax for the FSM shown in Figure 8.2. Transitions are specified within the case block. Algorithm 8.2 is only partial; the `HREADY` and `HRESP` signals are assigned values non-deterministically for the functional verification. The slave records split transactions by storing the master's address in the `MASK_MASTER1`, `MASK_MASTER2`, and `MASK_MASTER3` flags. These flags are managed by the slave (the arbiter also maintains its own flags for which master is masked) and are cleared when the slave issues an `HSPLITx` signal. The `extended` variable is used to extend the duration of `RETRY` and `SPLIT` responses for two clock cycles according to the AMBA AHB specification.

Algorithm 8.2 Partial NuSMV Finite State Machine Model for an AMBA AHB Slave

```
MODULE slave (HADDR, HTRANS, HWDATA, HRDATA, HREADY, HRESP, HMASTER,
HSPLIT, MASK_MASTER1, MASK_MASTER2, MASK_MASTER3, SLAVE_STATE)
  VAR
   state :  {idle, write, read, error};
   prev_state :  {idle, write, read, error};
   extended :  boolean;
  ASSIGN
   init (state) := idle;
   init (prev_state) := state;
   init (extended) := 0;
   next (prev_state) := state;
   next (state) :=
    case
     SLAVE_STATE != x :  SLAVE_STATE;
     HRESP = SPLIT : idle;
     !HREADY : state;
     HTRANS = BUSY : state;
     HRESP = RETRY : prev_state;
     state = idle & HTRANS = NONSEQ & HADDR : write;
     state = idle :  state;
     state = write & HTRANS = NONSEQ : read;
     state = read & HTRANS = NONSEQ & HWDATA : idle;
     1 :  error;
    esac;...
```

8.1.4 Modeling an AMBA AHB Round-robin Arbiter

We modeled a round-robin arbiter to evaluate the case studies described in Section 8.3 and Section 8.4. The arbiter is specific to the AMBA AHB protocol, and captures most bus signals used by the master and the slave. The design is too complex to show in a figure, therefore we use the NuSMV syntax to describe the arbiter's functionality. We describe the partial implementation of arbiter for two masters and one slave for simplicity, for more details and case studies please visit `http://alderis.uci.edu/amba2`. We also define a dummy `default` master for the bus that expresses the case when none of the two masters have access to the bus. The arbiter keeps track of both masters current state, and their previous state. An alternative approach would be to check the masters' state directly, however that would require a dedicated wire from the masters to the arbiter, which is unnecessary hardware overhead. Therefore, the arbiter keeps track of masters' state using the following simple rules. Whenever the slave sets the RETRY signal, repeat the previous transaction:

```
next (master1_state) :=
case
    -- Roll back for retrys
    HRESP = RETRY & !lasterror :  master1_prev_state;
```

Follow the transitions in the FSM shown in Figure 8.1:

```
    master1_state = idle & HREADY & HRESP = OK & HGRANT1 :  grant;
    master1_state = grant & HTRANS != IDLE & HREADY & HRESP = OK & HGRANT1
    :  transmit;
    master1_state = transmit & HTRANS = IDLE & HREADY & HRESP = OK &
    HGRANT1 :  idle;
```

Mask masters (prevent them from acquiring access to the bus) whenever the slave set the `SPLIT` response:

```
    HREADY & HRESP = SPLIT & HGRANT1 & !lasterror :  mask;
    master1_state = mask & HSPLIT = master1 :  idle;
```

The `lasterror` variable is introduced to hold the master's state if the previous response was either `RETRY` or `SPLIT`, according to the AMBA AHB specification. The default behavior is to hold the master's state.

```
    lasterror :  master1_state;
    master1_state;
esac;
```

The arbitration policy determines which master will be given preference in the next bus cycle when asking access to the bus by setting its `BUSREQ` signal. We express the arbitration policy with the help of the `preferred` variable. Given that we implemented a round-robin arbiter, the value of the `preferred` variable alternates between `master1` and `master2`. The first set of rules expresses that if a master requests access to the bus, it is eventually granted access to the bus. When both masters request access, one of them will be granted access non-deterministically:

```
next (preferred) :=
    -- Master starts the transmission
    !HGRANT1 & !HGRANT2 & master1_state != mask & master2_state != mask &
    HREADY & HRESP = OK & BUSREQ1 & BUSREQ2 :  master1, master2;
    !HGRANT1 & !HGRANT2 & master1_state != mask & HREADY & HRESP = OK &
    BUSREQ1 & !BUSREQ2 :  master1;
    !HGRANT1 & !HGRANT2 & master2_state != mask & HREADY & HRESP = OK &
    !BUSREQ1 & BUSREQ2 :  master2;
```

When one master (*i.e.* `master1`) is split and the other one (`master2`) is transmitting to the slave, the slave may decide to split (mask) the current transmitting master (`master2`), and unsplit the masked master (`master1`). If the masked master (`master1`) requires access to the bus, grant access immediately, otherwise grant access to the `default` (dummy) master.

```
    -- Cross-splitting
    HGRANT1 & master1_state = transmit & master2_state = mask & HRESP =
    SPLIT & !BUSREQ2 & HSPLIT = master2 :  default;
    HGRANT2 & master2_state = transmit & master1_state = mask & HRESP =
    SPLIT & !BUSREQ1 & HSPLIT = master1 :  default;
    HGRANT1 & master1_state = transmit & master2_state = mask & HRESP =
    SPLIT & BUSREQ2 & HSPLIT = master2 :  master2;
    HGRANT2 & master2_state = transmit & master1_state = mask & HRESP =
    SPLIT & BUSREQ1 & HSPLIT = master1 :  master1;
```

When one master is masked and the slave sends a `SPLIT` response, mask the other master as well. If none of the masters are masked and the slave sends a `SPLIT` response, mask the active master, and grant access to the other master if it requests access to the bus. If not, move on to the default master.

```
-- Split masters
HGRANT1 & master2_state = mask & HRESP = SPLIT : default;
HGRANT2 & master1_state = mask & HRESP = SPLIT : default;
HGRANT1 & master2_state != mask & HRESP = SPLIT & BUSREQ2 :  master2;
HGRANT2 & master1_state != mask & HRESP = SPLIT & BUSREQ1 :  master1;
HGRANT1 & HRESP = SPLIT & !BUSREQ1 & !BUSREQ2 :  default;
HGRANT2 & HRESP = SPLIT & !BUSREQ1 & !BUSREQ2 :  default;
```

If a master finishes the transaction, grant access to the other master if it requires access to the bus. If not, move on to the default master.

```
-- Master finishes the transaction - round-robin
HGRANT1 & master1_state = transmit & HTRANS = IDLE & HREADY & HRESP =
OK & !BUSREQ1 & !BUSREQ2 :  default;
HGRANT2 & master2_state = transmit & HTRANS = IDLE & HREADY & HRESP =
OK & !BUSREQ1 & !BUSREQ2 :  default;
HGRANT1 & master1_state = transmit & master2_state != mask & HTRANS =
IDLE & HREADY & HRESP = OK & BUSREQ2 :  master2;
HGRANT2 & master2_state = transmit & master1_state != mask & HTRANS =
IDLE & HREADY & HRESP = OK & BUSREQ1 :  master1;
```

If a master cancels its BUSREQ signal, either grant access to the other master (if it has its BUSREQ signal set), or move to the default master.

```
-- Master gives up its BUSREQ
HGRANT1 & master1_state = idle & master2_state != mask & HTRANS = IDLE
& HREADY & HRESP = OK & !BUSREQ1 & BUSREQ2 :  master2;
HGRANT2 & master2_state = idle & master1_state != mask & HTRANS = IDLE
& HREADY & HRESP = OK & !BUSREQ2 & BUSREQ1 :  master1;
HGRANT1 & master1_state = idle & HTRANS = IDLE & HREADY & HRESP = OK &
!BUSREQ1 & !BUSREQ2 :  default;
HGRANT2 & master2_state = idle & HTRANS = IDLE & HREADY & HRESP = OK &
!BUSREQ1 & !BUSREQ2 :  default;
```

When a slave requests a masked master to be unsplit, then unsplit that master.

```
    -- Unmasking masters
    !HGRANT1 & !HGRANT2 & master1_state = mask & HSPLIT = master1 :
    master1;
    !HGRANT1 & !HGRANT2 & master2_state = mask & HSPLIT = master2 :
    master2;
    HGRANT1 & master2_state = mask & HRESP = SPLIT & BUSREQ2 & HSPLIT =
    master2 :  master2;
    HGRANT2 & master1_state = mask & HRESP = SPLIT & BUSREQ1 & HSPLIT =
    master1 :  master1;
    HGRANT1 & master1_state = transmit & master2_state = mask & HTRANS =
    IDLE & HREADY & HRESP = OK & BUSREQ2 & HSPLIT = master2 :  master2;
    HGRANT2 & master2_state = transmit & master1_state = mask & HTRANS =
    IDLE & HREADY & HRESP = OK & BUSREQ1 & HSPLIT = master1 :  master1;
    HGRANT1 & master1_state = idle & master2_state = mask & HTRANS = IDLE &
    HREADY & HRESP = OK & BUSREQ2 & !BUSREQ1 & HSPLIT = master2 :  master2;
    HGRANT2 & master2_state = idle & master1_state = mask & HTRANS = IDLE &
    HREADY & HRESP = OK & BUSREQ1 & !BUSREQ2 & HSPLIT = master1 :  master1;
```

By default, leave the current preferred master.

```
    1 : preferred;
esac;
```

As seen from the formalism above, the arbiter design is rather complex even in the case of two masters. If designers want to support the advanced features of the AMBA AHB protocol, they need to verify the functionality and performance of the design. In Section 8.3 we describe how we verified the correctness of the MPSoC designs shown in Figure 8.4 using the formal models described in this section.

8.2 Digital Camera MPSoC Design Alternatives

In this section we describe three alternative MPSoC designs for a digital camera using the AMBA AHB bus. The digital camera used for the case study implements the new *JPEG2000* [53] still image compression standard developed by the JPEG committee. The advantages of *JPEG2000* over its predecessor *JPEG* include lossy

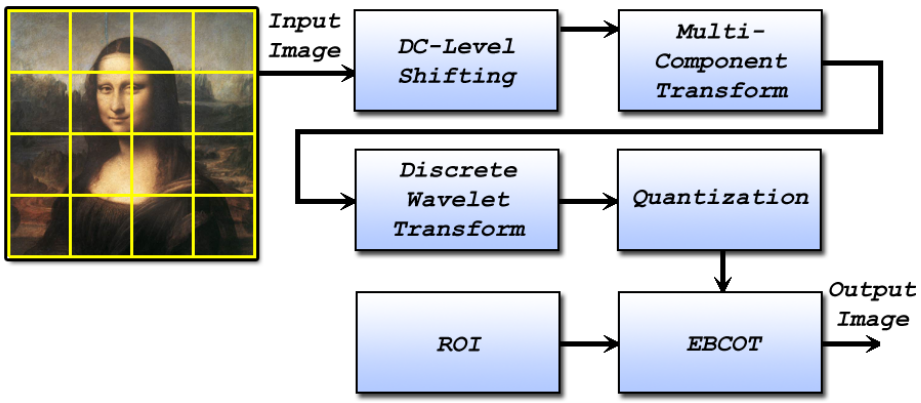

Figure 8.3: JPEG2000 Encoder Block Diagram

to lossless compression, *Region Of Interest (ROI)*, multiple resolution representation, error resiliency, etc. The *JPEG2000* encoder is divided into three main parts: image transformation, quantization and entropy coding. Unlike JPEG, which relies on the more commonly used *Discrete Cosine Transform (DCT)*, *JPEG2000* uses the *Discrete Wavelet Transform (DWT)* as it facilitates the notion of progressive image transmission. *JPEG2000*'s choice of entropy coding is based on the *Embedded Block Coding with Optimal Truncation (EBCOT)* [95].

8.2.1 JPEG2000 Encoder Description

Figure 8.3 shows the block diagram for the *JPEG2000* encoder. Designers have the option of implementing a distributed compression method, where the image is broken up into tiles, and the compression is carried out for each tile separately. Although this feature is not required by the *JPEG2000* specification we have decided to implement it to improve the concurrent processing in the system and thus the overall performance of the MPSoC. Tile size varies, from smaller sizes – 64×64 pixels for memory restrained designs – to 512×512 for better compression quality. These parameters vary and designers need to consider the requirements for their specific designs.

After the image is tiled, each tile is passed though the *DC Level Shifting* step

which converts the tile pixels from unsigned integers to two's complements. In the next step the tile is passed through the *Multi-Component Transform (MCT)*, which is in charge of transforming the input tile from RGB color format to either YUV by using the *Reversible Color Transform (RCT)*, or to YCbCr by using the *Irreversible Color Transform (ICT)*. RCT can be used in both lossless and lossy compression whereas ICT can only be used for lossy compression. After the tile has been transformed, it is processed by the DWT, which further decomposes the tile into different levels of decomposition. For every pass DWT makes on a tile, depending on the number of decomposition levels needed, DWT generates four sub-bands, denoted as LL, HL, LH, and HH, where LL represents the downsampled tile (half the width/height of the previous tile), and the other three sub-bands represent a residual version of the tile which are used for the image reconstruction process. Once DWT has processed the tile it is passed through the quantization step only when lossy compression is used. The user has the option to declare ROIs, that are encoded independently from the rest of the image based on user specifications. This allows to use lossless compression for some (interesting) parts of the image, while using lossy compression for the rest of the image. Finally, the image (or tile) is processed with EBCOT, which produces the final bitstream for the image.

EBCOT can be further subdivided into two parts, commonly known as *Tier-1* and *Tier-2*. Tier-1 is the most computation intensive part of *JPEG2000*. Because of the complexity of both DWT and Tier-1, most designers choose to implement these functionalities in hardware. Tier-2, however, is very control intensive, therefore it is often implemented in software on the main CPU.

8.2.2 Description of MPSoC Design Alternatives

We consider three design alternatives for the implementation of the digital camera as shown in Figure 8.4. In Design 3 all communication between the heterogeneous IP

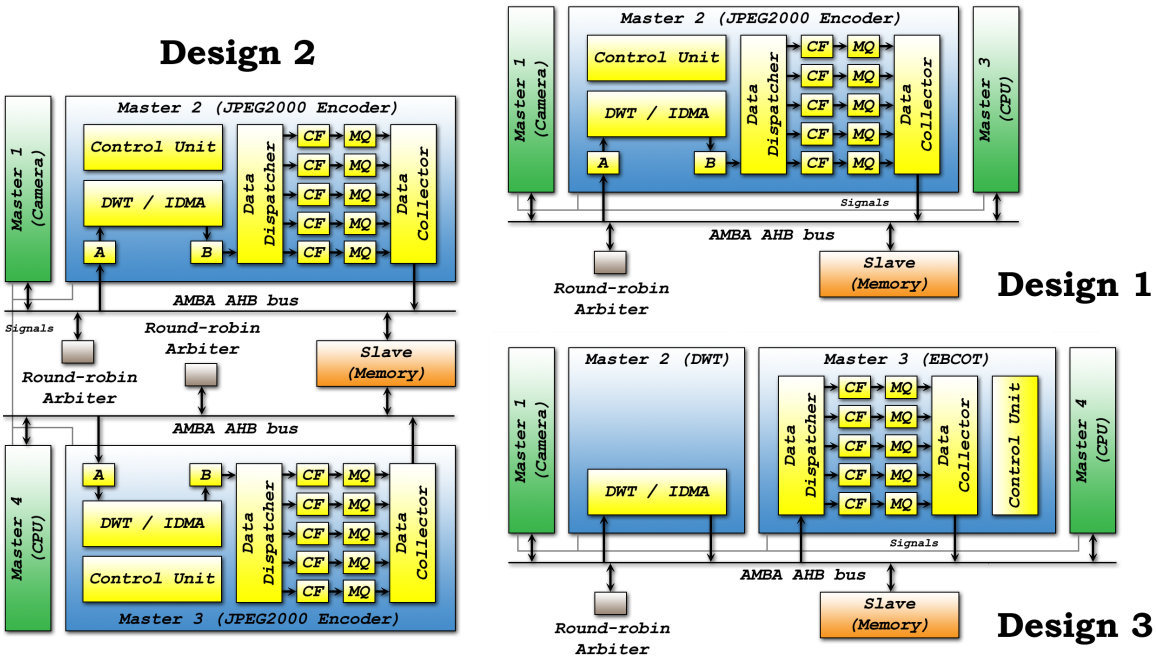

Figure 8.4: Design Alternatives of the Digital Camera Case Study using the JPEG2000 Encoder

blocks uses the AMBA AHB bus, in Design 1 the DWT and EBCOT functional blocks are combined into a single chip, and Design 2 uses two AMBA AHB buses and two *JPEG2000* encoders to reduce congestion on the buses and increase the throughput of the digital camera MPSoC. In all of the architectures shown in Figure 8.4, the DWT module has an internal DMA engine that fetches the tiles from main memory to either DWT's local memory or bank `A` of the tile memory, depending on whether DWT is currently processing a tile or not. The DWT module is capable of lossless and lossy compression and implements DC Level Shifting and MCT. In Design 1 and 2 shown in Figure 8.4 the DWT unit writes the transformed image to bank `B` in tile memory. The DWT unit can only write new information if `Data Dispatcher` has finished fetching all codeblocks for the previous tile from bank `B`. Otherwise, the DWT module may be blocked by the slower `Data Dispatcher`. In Design 3 the AMBA AHB bus is used to directly write to the (slave) memory therefore in this design the `Data Dispatcher` never blocks the DWT unit.

The `Data Dispatcher` module reads the codeblocks in the tile memory and performs the quantization step on them. Its main job is to feed bitplanes onto each *Bit Plane Coder (BPC)* so that at any given time, there could be up to N different bitplanes being processed by the BPC modules. A BPC is actually the module in charge of performing Tier-1 on incoming DWT coefficients, and it is subdivided into two parts, the *Context Formatter (CF)* and the *Arithmetic Coder (MQ-Coder)*. These blocks are denoted as `CF` and `MQ` in Figure 8.4. Finally, the processed data is collected by the `Data Collector`, from which it is written to the main memory through the AMBA AHB bus.

8.3 Functional Verification of AMBA-based MPSoC Designs

This section describes how we utilized the NuSMV models introduced in Section 8.1 for the functional verification of design alternatives discussed in Section 8.2. We have considered two problems in our formal analysis. A *deadlock* can be observed in the FSM model as a state where no transitions are enabled. A *livelock* can be observed as a state from which only a subset of states is reachable. A livelock can express situations like starvation, where a master is not granted access to the requested slave. We used the NuSMV tool to verify deadlock-freedom and liveness properties in a single bus system with two, three, and four masters and one slave, using round-robin arbitration.

For the functional verification we have considered the case when all the masters are allowed to concurrently request access to the bus and carry out read/write transactions in an arbitrary (non-deterministic) manner, and the slave can arbitrarily split/unsplit masters and issue `RETRY` responses. This covers all the valid uses of the bus and therefore can be applied to prove the correctness of the designs. The proposed functional verification does not take the internal computation of components

into consideration, rather it treats them as "black boxes" that use the bus according to the specification. The results described in this section are therefore applicable to any MPSoC that uses any of the architectures show in Figure 8.4 with a round-robin scheduler.

Design 1 shown in Figure 8.4 employs three masters on the same bus therefore we verify the functional correctness of this design using three masters and a slave, and we verify Design 3 using four masters and a slave. Design 2 shown in Figure 8.4 has two buses, both of which can access the main memory. In our analysis we assume that the memory can be accessed from both buses with no risk of deadlocks. This requirement is provided by the use of a memory unit with two interfaces for data access, and is guaranteed by hardware. Therefore, we can verify Design 1 by independently verifying the two AMBA AHB buses with two masters.

We have used the open-source NuSMV model checker to verify *Computational Tree Logic (CTL)* [25] properties on the FSMs. During this process we have discovered several trivial deadlock cases that are covered by the AMBA AHB specification. For example, we were able to show that a `SPLIT` response followed immediately by a `RETRY` response deadlocks the system, as the master receiving the `RETRY` response has not started transmitting on the bus yet. The AMBA AHB specification, however, requires the slave to issue an `OK` response following the `SPLIT` response. Similarly, we found that the combination of a `RETRY` response and a low `HREADY` signal may deadlock the bus because the master is required to keep its state when the `HREADY` signal is low, but is also required to repeat the last transmission since the response is `RETRY`. These ambiguities, however, do not have a high practical value as their use does not seem to be logical in real-life MPSoCs.

To keep consistency all the formulas described in this section apply to the 4-master Design 2 and Design 3, which we adapted to Design 1 by simply removing any assumptions/constraints on (the non-existent) `Master4`. We have assumed that the following

formulas evaluate to true infinitely often (using the `JUSTICE` NuSMV keyword) in all MPSoC designs for the analysis: `HREADY`, `HRESP = OK`, `HSPLIT = master1`, `HSPLIT = master2`, `HSPLIT = master3`, `HSPLIT = master4`. This was necessary to avoid trivial erroneous cases, such as when the slave is never ready to receive data, or when it continuously sends `RETRY` responses. Using these assumptions we were able to prove several properties in all design alternatives shown in Figure 8.4. First we showed that the `error` state is unreachable in all the masters and the slave by using the CTL formulas (x refers to the index used for all masters):

```
AG (masterx.state != error),
AG (slave.state != error).
```

The AMBA AHB protocol permits a simple way for livelock by allowing the slave to arbitrarily split masters. If the slave splits a master and does not unsplit it, we end up in a livelock condition as the split master never gets a chance to serve requests. Moreover, if the slave splits all the masters and does not unsplit them the system deadlocks. We showed these conditions by checking the following CTL formula:

```
EF (MASK_MASTER1 & MASK_MASTER2 & MASK_MASTER3 & MASK_MASTER4).
```

We had to specify rules within the slave to enforce that whenever two masters are split one of them will eventually be unsplit. Then we tried to verify whether the system can always recover from a deadlock caused by the slave by splitting all the masters on the bus by using the following CTL formula:

```
AG ((MASK_MASTER1 & MASK_MASTER2 & MASK_MASTER3 & MASK_MASTER4) ->
AF (!MASK_MASTER1 | !MASK_MASTER2 | !MASK_MASTER3 | !MASK_MASTER4)).
```

However, to our surprise we found that this property is not necessarily true in all designs. The proposed model checking method uncovered the undocumented ambiguity in the AMBA AHB specification described below.

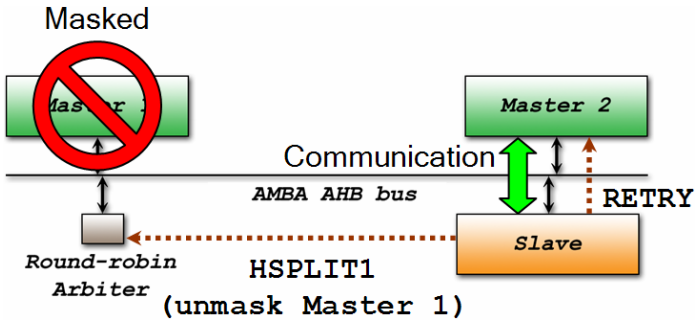

Figure 8.5: Ambiguity in the AMBA AHB Specification

8.3.1 Ambiguity in the AMBA AHB Specification

Despite the fact that the functional verification of the AMBA AHB protocol has been addressed before by various researchers [88] [93] [7], we were able to uncover an ambiguity in the protocol that has not yet been documented by other authors. Consider an MPSoC system based on the AMBA AHB protocol, using two masters (`master_1`, `master_2`) and a slave. The arbiter has to keep track of the masters' state in order to manage the split transfers. This could be implemented by providing dedicated wires between the masters and the arbiter, however this is impractical in most cases as it requires extra computation and hardware. An alternative method is to monitor the bus traffic to obtain the master and slave states. The arbiter may use the `HTRANS` signal to check whether the master is idle or transmitting (`NONSEQ, SEQ`), the `HBURST` signal to predict the remaining cycles from the transfer, and the `HRESP` signal to monitor whether the active master and slave has to step back to repeat a transaction.

The AMBA AHB protocol allows three types of responses by the slave: `OK` signals that the transaction in the previous clock cycle has been successfully completed, `RETRY` signals that the slave wants the master to repeat the transaction from the previous clock cycle, and `SPLIT` is a signal to the arbiter to mask the master. A slave issues the `SPLIT` response when it predicts that it will be unable to receive data – a rather ambiguous definition in the AMBA AHB specification. Later, a slave can signal the

arbiter using the HSPLITx signal that it is now ready to process data and requests that the arbiter unmasks a previously masked master.

Let's assume that the slave has previously split master_1 (master_1 is masked by the arbiter), and is in a transaction with master_2 as shown in Figure 8.5. The slave can unmask master_1 by issuing an HSPLIT1 signal using the masked master's address to the arbiter. Consider that the slave tries to unmask master_1 by setting HSPLIT1 when it issues a RETRY response. The AMBA AHB specification is ambiguous on what the arbiter should do in this case. The specification says that a master has to repeat the last transaction when it receives the RETRY response. If the arbiter monitors the bus signals to keep track of the masters' states it will try to go back to its previous state to keep synchronized with the master and the slave. However, if the arbiter implements this behavior it will not unmask master_1 as the client requests. Since there is no acknowledgement for HSPLITx signals the client thinks that master_1 is already unmasked, and won't request that the arbiter unmasks it again. This may result in deadlock as master_1 never gets access to the bus again. The AMBA AHB specification states, that "A slave which issues RETRY responses must only be accessed by one master at a time." Thus we could not reach an agreement whether the "access" refers to access through the bus or access by being split by the slave - which would cover this deadlock, but would also imply that a slave cannot issue a RETRY response if it may split a master.

8.3.2 Resolving the Ambiguity

Once we recognized the possibility for deadlock we tried to resolve the ambiguity and show the correctness of the design. We have disallowed the simultaneous use of the HRESP = RETRY and the HSPLITx signal – that have caused a deadlock as described above – using a simple constraint, and tried to verify whether the system can always recover from a deadlock caused by the slave by splitting all the masters on the bus

by using the CTL formulas below:

```
AG ((MASK_MASTERx & MASK_MASTERy) ->
AF (!MASK_MASTERx | !MASK_MASTERy)),

AG ((MASK_MASTER1 & MASK_MASTER2 & MASK_MASTER3) -> AF (!MASK_MASTER1 | !MASK_-
MASTER2 | !MASK_MASTER3)),

AG ((MASK_MASTER1 & MASK_MASTER2 & MASK_MASTER3 & MASK_MASTER4) ->
AF (!MASK_MASTER1 | !MASK_MASTER2 | !MASK_MASTER3 | !MASK_MASTER4)).
```

Using the constraint that disallows the simultaneous use of the `HRESP = RETRY` and the `HSPLITx` signal we were able to show that the MPSoC design works correctly. We were able to show that all bus requests by the masters eventually get served in the constrained MPSoC designs by the arbiter by checking the following CTL formulas:

```
SPEC AG (masterx.state = busreq -> AF HGRANTx), SPEC AG (masterx.state = busreq
-> AF masterx.state = write).
```

These formulas ensure that the system does not deadlock or livelock and show the correctness of our designs.

8.4 Performance Evaluation of AMBA-based MPSoC Designs

This section describes the proposed method for performance evaluation that combines the transaction-level simulation approach with model checking.

8.4.1 Simulation-based Evaluation

The abstraction we chose for simulation-based evaluation is *Cycle Count Accurate At Transaction Boundaries (CCATB)* [84], and the functional behavior of each module is "cycle-count-accurate". Each one of the blocks in Figure 8.4 is implemented as `SC_-`

Table 8.1: JPEG2000 Encoding SystemC Simulation Results for Design 1 Shown in Figure 8.4 using 64×64 pixel Tiles (for a single tile). Scale: cycles

Image	DWT	Tier-1 BC	Tier-1 WC	Tier-2	Input	Output	End-to-end
baboon	194188	517005	741519	9122240	12288	11099	10335043
boat	194188	165141	737046	8750875	12288	10046	10044857
goddesses	194188	513846	772461	8663630	12288	11456	9996487
goldhill	194188	242055	747954	8672436	12288	10376	9978464
lena	194188	461601	769239	8689815	12288	11979	10024198

Table 8.2: JPEG2000 Encoding SystemC Simulation Results for Design 2 Shown in Figure 8.4 using 64×64 pixel Tiles (for a single tile). Scale: cycles

Image	DWT	Tier-1 BC	Tier-1 WC	Tier-2	Input	Output	End-to-end
baboon	194188	612189	741951	9122240	12288	10546	10385552
boat	194188	362484	741915	8750875	12288	9482	9999909
goddesses	194188	373950	743544	8663630	12288	11811	9936290
goldhill	194188	483885	742743	8672436	12288	10481	9936927
lena	194188	206181	741753	8689815	12288	9689	9950482

Table 8.3: JPEG2000 Encoding SystemC Simulation Results for Design 3 Shown in Figure 8.4 using 64×64 pixel Tiles (for a single tile). Scale: cycles

Image	DWT	Tier-1 BC	Tier-1 WC	Tier-2	Input	Output	End-to-end
baboon	194188	513675	731124	9122240	12288	8622	10351259
boat	194188	390141	721467	8750875	12288	7842	9963563
goddesses	194188	505197	766737	8663630	12288	8157	9926747
goldhill	194188	411192	736254	8672436	12288	8592	9907612
lena	194188	416304	756117	8689815	12288	8031	9943368

`MODULE` which is a special class in *SystemC* used to declare modules. Communication between modules is implemented through `SC_PORTS` using `SC_SIGNALS`.

Within each `SC_MODULE` there may be several concurrently executing threads, declared as `SC_THREAD` in *SystemC*. For instance, DWT has a tiling engine thread that emulates DMA and fetches the tiles from main memory, a compute thread that emulates the DWT lifting kernel and wakes up when the controller signals that there is a tile ready to be processed, a read thread that fetches tiles from tile memory, and a write thread that writes DWT coefficients to tile memory. The `Data Dispatcher` has two threads, one that reads DWT coefficients from tile memory and the main data dispatcher thread that distributes the bitplanes among all of the bit plane coders in

Table 8.4: JPEG2000 Encoding SystemC Simulation Results for Design 1 Shown in Figure 8.4 using 128×128 pixel Tiles (for a single tile). Scale: cycles

Image	DWT	Tier-1 BC	Tier-1 WC	Tier-2	Input	Output	End-to-end
baboon	751393	2315254	3151948	9010373	49152	36537	14290609
boat	751393	1764568	3086892	8758372	49152	41719	13990027
goddesses	751393	1843190	3219664	9451990	49152	42391	14823509
goldhill	751393	2325098	3173076	8768459	49152	41645	14090307
lena	751393	2364360	3241400	8793070	49152	37578	14172351

Table 8.5: JPEG2000 Encoding SystemC Simulation Results for Design 2 Shown in Figure 8.4 using 128×128 pixel Tiles (for a single tile). Scale: cycles

Image	DWT	Tier-1 BC	Tier-1 WC	Tier-2	Input	Output	End-to-end
baboon	751393	2530709	2872989	9010373	49152	37755	13921490
boat	751393	1693281	2851434	8758372	49152	33136	13578227
goddesses	751393	1897617	2998701	9451990	49152	43176	14502501
goldhill	751393	1935384	2901609	8768459	49152	32449	13682391
lena	751393	1759751	2904408	8793070	49152	32471	13690690

Table 8.6: JPEG2000 Encoding SystemC Simulation Results for Design 3 Shown in Figure 8.4 using 128×128 pixel Tiles (for a single tile). Scale: cycles

Image	DWT	Tier-1 BC	Tier-1 WC	Tier-2	Input	Output	End-to-end
baboon	751393	1778247	2863026	9010373	49152	32298	13819444
boat	751393	1723257	2790918	8758372	49152	36460	13479349
goddesses	751393	1778346	2939823	9451990	49152	29430	14315158
goldhill	751393	1758681	2884545	8768459	49152	31797	13603176
lena	751393	1763469	2947986	8793070	49152	29820	13674139

round robin fashion.

The CF and the MQ-Coder both have three separate threads, a read (from input *First In First Out (FIFO)*) thread, a write (to output FIFO) thread and a compute thread. The `Data Collector` module also has two threads, one for reading from the bit plane coder output FIFOs in round-robin fashion, and one for writing the encoded data back to main memory. The exhaustive verification of the *SystemC* model is practically infeasible due to the large number of threads and the degree of non-determinism present in the simulation models.

The *SystemC* model is configured using a configuration script that sets up its parameters based on the input image that the model will process. The parameters

include tile width, tile height, image width, image height, DWT decomposition levels, etc. The script configures and runs the model for a given amount of test images. From each simulation run we obtain the execution intervals for processing tiles, and the size of compressed tiles sent over the AMBA AHB bus.

Tables 8.1–8.6 show the parameters that we have obtained by the *SystemC* simulations. We have run simulations on five different pictures using 64 × 64 and 128 × 128 pixel images as input for the compression. The Tier-1 columns describe the measured execution time of the Tier-1 *JPEG2000* compression in execution cycles, Tier-2 columns correspond to the software implementation of Tier-2 on the main CPU. The Input column shows the size of the tile as input to the DWT and Tier-1, Output specifies the worst case size of the tile after the compression in Tier-1.

We see that both Design 2 and 3 improve upon the performance of Design 1 slightly on average, however for the baboon image Design 1 has the lowest worst case end-to-end computation time. The main source of performance gain in Design 2 is the reduced congestion on the AMBA AHB bus. In Design 3 performance gain is obtained because of the non-blocking communication between the DWT and the EBCOT unit. Design 2 does not seem to benefit from using 2 *JPEG2000* encoders as the performance bottleneck is the CPU. However, to measure the expected performance gain in case a faster CPU is used we ran simulations to obtain the average throughput of the `JPEG2000 Encoder(s)` in all designs as shown in Tables 8.7–8.8. These tests adhere to our expectations that the two `JPEG2000 Encoders` should have nearly twice as much throughput as a single one.

8.4.2 Model Checking-based Performance Evaluation

This section describes how we utilized model checking to evaluate the worst case behavior of the digital camera design alternatives shown in Figure 8.4 based on the simulation results described in Subsection 8.4.1 above. The simulations give us very

Table 8.7: Average Throughput of the JPEG2000 Encoders SystemC Simulation Results using 64×64 pixel Tiles. Scale: tile/sec

Image	Design 1	Design 2	Design 3	Design 2 vs 1	Design 3 vs 1
baboon	186.71	365.65	187.70	1.9583	1.0052
boat	198.68	371.32	198.84	1.8690	1.0008
goddesses	190.59	364.43	187.55	1.9121	0.9841
goldhill	188.70	365.23	188.36	1.9355	0.9982
lena	190.72	368.43	187.80	1.9318	0.9847

Table 8.8: Average Throughput of the JPEG2000 Encoders SystemC Simulation Results using 128×128 pixel Tiles. Scale: tile/sec

Image	Design 1	Design 2	Design 3	Design 2 vs 1	Design 3 vs 1
baboon	48.47	93.11	49.86	1.9211	1.0287
boat	50.89	94.18	52.39	1.8506	1.0295
goddesses	49.03	94.03	49.99	1.9179	1.0196
goldhill	49.03	93.16	50.02	1.9003	1.0203
lena	48.84	93.80	49.94	1.9204	1.0224

Table 8.9: Parameters used for Performance Evaluation by Model Checking. Scale: 10^4 cycles

Case study	M_1	M_2 BC	M_2 WC	M_3 BC	M_3 WC	M_4 BC	M_4 WC
Design 1 64×64	1	35	97	866	913	N/A	N/A
Design 1 128×128	1	251	400	875	946	N/A	N/A
Design 2 64×64	1	40	94	40	94	866	913
Design 2 128×128	1	245	376	245	376	875	946
Design 3 64×64	1	19	19	39	77	866	913
Design 3 128×128	1	75	75	172	295	875	946

Table 8.10: Worst Case Bounds on the End-to-end Computation Times of the Designs Shown in Figure 8.4 obtained using Model Checking. Scale: cycles

Tile size	Design 1 WCET	Design 2 WCET	Design 3 WCET
64×64 pixel tiles	10670000	10580000	10540000
128×128 pixel tiles	17000000	16800000	16600000

accurate results for the end-to-end processing of image tiles, however they can only cover a few execution traces of the system. The formal model checking approach provides the means to evaluate larger design spaces to obtain the *worst case end-to-end execution time* of the overall MPSoC designs. The formal models used for the evaluation are cycle-accurate on the bus transaction-level.

We have shown in Section 8.3 how we proved the overall functionality of the sys-

tem. The FSM models used for the performance evaluation are more lightweight than for the functional verification described in Section 8.3; we do not consider split transactions, `RETRY` responses, or blocking slaves (`HREADY` is assumed to be set), as these functionalities are not used in the digital camera design alternatives shown in Figure 8.4. Although these assumptions are not required for the performance analysis they increase the model checking scalability.

We have used simple Boolean variables to model the interrupts and signals in the digital camera, thus enforcing the dependencies between components. Although finite state machine is inherently an untimed model of computation it can capture time on a discrete time scale as transitions can be ordered. In our analysis we have declared a global time variable that is increased at every cycle.

Since most model checkers (including NuSMV) are optimized for property checking by giving yes/no answers, we had to conform to several restrictions. First, the performance analysis has to be be expressed as a reachability problem: when `Master_3` has written its data to the memory is the execution time always smaller than some value x? Formally using CTL formula: `AG (finish -> TIME <` x`)`, where `finish` is the signal generated by the CPU (`master_1`) when it is in the `write` state and it has written all the required data to the memory. Second, the state space used by NuSMV is influenced by the range of variables used in the model. To overcome this problem we have increased the timescale of the simulation (from cycles to 1000 cycles), thereby creating an abstraction of the system that is cycle-approximate with the highest precision available without hitting the state explosion problem. Table 8.10 summarizes the results of formal model checking on the worst case execution bounds of the digital camera design alternatives.

The proposed method is computationally intensive, and its performance degrades exponentially with respect to the state space size of the analyzed system. This might present scalability issues when trying to apply the method for large-scale MPSoCs. In

this case, the combination of several techniques may be used. First, we might increase the scalability by increasing the timescale of the analysis, at the loss of some precision. Second, the method can be hierarchically composed. End-to-end execution times for MPSoCs can also be represented as intervals thus providing a way to encapsulate larger MPSoC designs as a single component. Third, we can limit how many execution traces we want to capture in the models *i.e.* by using constant execution times.

8.4.3 Evaluating the Performance Estimation Results

Figure 8.6 summarizes the experiments performed for the performance estimation of the digital camera MPSoC design alternatives shown in Figure 8.4. Each group of 3 bars shows the cycle estimates we obtained for a design alternative with a given tile size.

The first bar (`D_1 64×64 Block`) illustrates the block performance estimates for Design 1 with 64×64 pixel tiles given in Table 8.1. The bar shows the best case cycle estimate for the Tier-1 block, the difference between Tier-1 WCET and Tier-1 BCET (so that the height of the two blocks corresponds to Tier-1 WCET), the Tier-2 BCET estimate, and the difference between Tier-2 WCET and Tier-2 BCET. Naturally, Tier-1 and Tier-2 of the EBCOT algorithm are the most computationally intensive blocks of the camera design, and therefore provide a lower bound on the expected performance of the digital camera design.

The second bar (`D_1 64×64 Sim`) illustrates the end-to-end performance of Design 1 obtained by simulations as shown in Table 8.1. These results include all blocks for the performance estimation, as well as the overhead of the communication, such as reading/writing tiles from/to the memory through the AMBA AHB bus.

To obtain the true worst case performance estimates, we build on model checking to explore the finite state machine models augmented with execution parameters for each block as described in Section 8.4.2. The third bar (`D_1 64×64 MC`) shows the

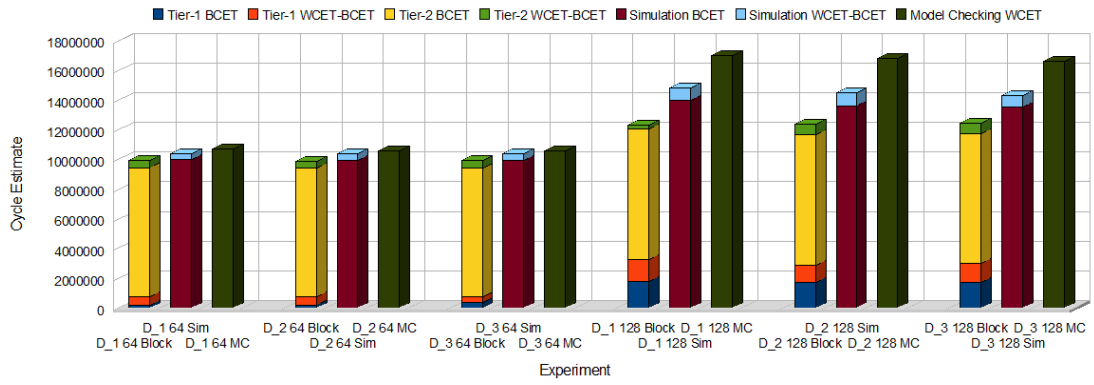

Figure 8.6: Performance Estimation of MPSoC Design Alternatives Shown in Figure 8.4

worst case performance estimate that we were able to prove assuming that the best and worst case execution time parameters used for masters/slaves were correct. This approach does not provide a "hard" bound on the worst case performance estimate, since we build on simulation results to estimate the performance of the individual blocks. In realistic design problems, however, designers are less concerned about the performance of individual blocks than choosing the right design alternative that provides the best performance based on their assumptions.

8.4.4 The Impact of Transaction-level Simulations and Model Checking on the Accuracy of the Performance Estimates

In Figure 8.7, we quantified the difference between the worst case estimates obtained by analytic results, simulations, and model checking shown in Figure 8.6. The communication overhead and bus congestions are responsible for the difference in the worst case performance estimates obtained by analytic calculations and simulations. The first bar of each group of 3 bars shows the communication overhead estimated by simulations as opposed to not considering the communication subsystem. For 64×64 tiles the difference is less than 5%, showing that the AMBA AHB bus rarely encounters any congestion, and the overhead is nearly negligible. Once we consider 128×128 tiles,

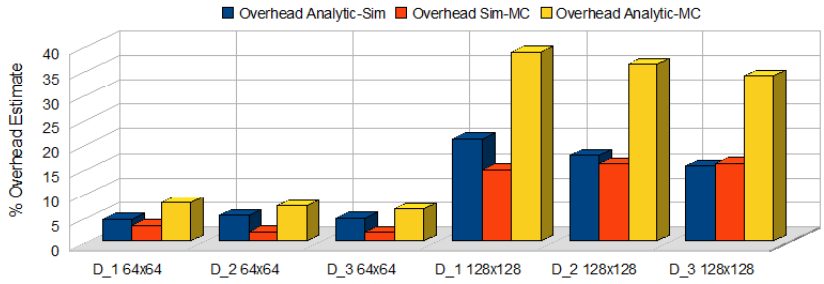

Figure 8.7: Communication Overhead Estimates by Simulations and Model Checking

we see that simulations estimate that the communication through the AMBA AHB bus is responsible for ∼15–20% overhead in the worst case. Since larger tiles are used, the number of memory accesses while processing a tile increase significantly, simply because we deal with larger data sets. The first bars in each group in Figure 8.7 show that simulations are essential to accurately predict the impact of the communication overhead.

Let us now consider the practical impact of applying model checking to obtain end-to-end performance estimates. The primary difference between simulation results and model checking results are due to the fact that model checking considers the end-to-end worst case execution time by performing an exhaustive state space search. During simulations, we can only cover a few execution traces of the MPSoC designs, and therefore we cannot estimate the impact of non-deterministic delays and congestions. Moreover, we cannot quantify the coverage of the performance estimates either.

The second bar of each group of 3 bars shows the communication overhead estimated by model checking as opposed to simulations. This basically shows that our simulations are not "pessimistic" enough, and cannot find the actual worst case end-to-end performance of the design. This is mainly due to the fact that not all functional blocks experience their worst case behavior at the same time, and the simulations do not encounter the maximum number of possible congestions on the AMBA AHB bus. As with simulations, the impact of considering model checking for

performance estimation grows as the complexity of the design grows, and accounts to nearly 15%. This practically means that the actual performance of the design may be 15% worse than the worst case performance estimate obtained by simulations.

Finally, the third bar of each group of 3 bars shows the communication overhead estimated by model checking as opposed to the analytical method, where we did not consider the AMBA AHB bus. In this case, the difference may be more than 35%, showing that analytical methods simply cannot estimate the worst case performance of the digital camera MPSoC designs with acceptable accuracy.

By considering both simulations and model checking, we obtained performance estimates for the worst case end-to-end performance of the digital camera design early in the design flow, and improved the accuracy of the performance estimate compared to simulations by around 3% when 64×64 pixel tiles are used, and nearly 15% when 128×128 tiles are used.

The performance analysis shows that Design 3 offers the best worst case end-to-end processing, while Design 2 is second (as the bottleneck is the CPU not the encoder block), and Design 1 is the slowest alternative. Our results show that the formal performance analysis is able to provide tight worst case execution numbers for the end-to-end processing of the digital camera MPSoC design alternatives.

The proposed formal performance evaluation method is unique compared to simulation-based evaluations as it covers orders of magnitude larger design spaces. The application of the method allows designers to avoid the common mistake of underestimating the worst case performance of MPSoCs as a result of inadequate coverage by simulations.

Chapter 9

Cross-abstraction Real-time Analysis of Bus Matrix MPSoC Designs

This chapter builds on methods originally developed for the real-time verification and performance estimation of software-intensive *Distributed Real-time Embedded (DRE)* systems, including the methods presented in Section 5, Section 6 and Section 7. Bus protocols and arbitration policies have a major impact on key design parameters in *Multi-processor System-on-Chip (MPSoC)* designs such as throughput and delays, and present new challenges for functional verification. A key contribution of this chapter is to show how methods for the analysis of DRE systems can be adapted to MPSoC designs utilizing fully connected bus matrix interconnects, and how point arbitration policies can be expressed by the non-preemptive scheduling of task graphs.

Model-based design is an emerging paradigm that aims to manage this complexity by systematically capturing key properties of MPSoCs, such as their structure, parameters of individual components, and their interactions. This chapter introduces the model-based *Cross-abstraction Real-time Analysis (*Carta*)* framework for the cross-abstraction analysis of MPSoC designs, that combines the concepts of component-based design, domain-specific modeling, simulations, and model checking to provide a unified framework for the functional and performance analysis of MPSoCs with bus matrix interconnection networks. The design flow aims to address three major challenges in the formal analysis of MPSoC designs: *(1) functional verification* - to ensure that the system will not be trapped in a deadlock or livelock state, *(2) performance estimation* - in order to obtain tight bounds on the worst case performance of the MPSoC design, and *(3) verification of real-time properties* - to prove whether individual deadlines for tasks and performance estimates hold for the MPSoC design.

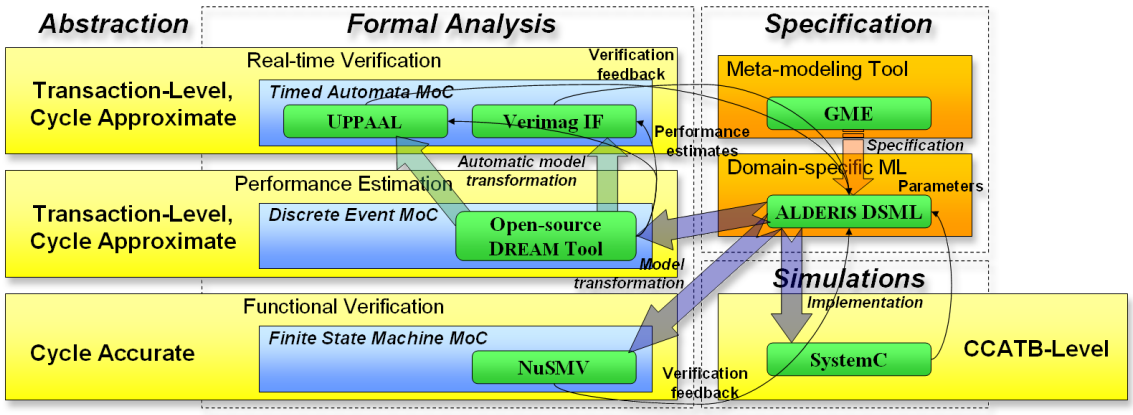

Figure 9.1: The CARTA Model-based Analysis Framework

Finding the right abstraction is a key challenge for the model-based analysis of MPSoC designs. Figure 9.1 shows our proposed cross-abstraction real-time analysis framework that provides a way to utilize the right level of abstraction for each analysis method. The three challenges addressed by our proposed analysis framework – functional verification, performance estimation, real-time verification – require different approaches, models of computation, abstractions, and tools for formal analysis. We pick the model of computation and abstraction level for each analysis method that provides the most efficient analysis.

For *functional verification*, it is important to accurately capture the signals in the bus matrix interconnection network. If the analysis model is too abstract, certain problems can remain undetected in the design phase. In our framework, we make use of a cycle-accurate *Finite State Machine (FSM) Model of Computation (MoC)* to capture the bus protocol, and arbitration algorithms. Using this model, we can efficiently check for all combinations of communication signals that satisfy the protocol to verify that no deadlocks and starvations occur in the MPSoC design.

The effectiveness of *performance estimation* and *real-time verification*, on the other hand, is primarily limited by scalability issues. Cycle-accurate FSM models quickly lead to the state space explosion problem, when used for performance estimation,

or real-time verification. Therefore, we need to raise the abstraction to transaction-level formal models. Transaction-level abstractions are well-established in the domain of simulation-based design exploration [81]. Transaction-level formal models in our context are event-driven, and communicate via asynchronous message passing. Timing information in the models is captured as time intervals associated with events. We apply the transaction-level modeling concept to increase the scalability of formal methods in the CARTA framework.

The CARTA framework builds on various modeling and analysis tools created by the research community and the authors. We utilize the *Generic Modeling Environment (GME)* [61] as a modeling tool for designing *Analysis Language for Distributed, Embedded, and Real-time Systems (*ALDERIS*)* [68] models (`http://alderis.ics.uci.edu`), as described in Section 9.2. *Domain-specific Modeling Languages (DSMLs)* in our approach are defined by the concept of model-integrated computing [11], that promotes the use of *meta-modeling* to create custom modeling languages which are a good fit for a specific problem domain. ALDERIS captures key properties of a MPSoC design, such as computation units, inter-component dependencies, the mapping of tasks to HW or SW, execution times and delays, and key constraints that the design has to satisfy. MPSoC designs are specified using the ALDERIS DSML, and drive the CARTA model-based analysis framework.

ALDERIS models of MPSoC designs are executable *Discrete Event (DE)* models with formal semantics. These models can be transformed into FSM models with cycle-accurate timing accuracy. We utilize the open-source *Distributed Real-time Embedded Analysis Method (*DREAM*)* tool for the performance estimation of ALDERIS models, and the NuSMV [1] model checker for the functional analysis of the FSM models. The DREAM tool also generates a direct *Timed Automata (TA)* representation of ALDERIS models, that can be analyzed by the UPPAAL [26] and Verimag IF [15] TA model checkers.

Simulations form an integral part of the proposed design flow, and provide accurate task execution times and delays to the ALDERIS models. For the purposes of simulation, we capture MPSoC designs in the *SystemC* [101] modeling language at the *Cycle Count Accurate At Transaction Boundaries (CCATB)* modeling abstraction [84]. *SystemC* is a *C++* library that provides a rich set of primitives for modeling communication and synchronization. The CCATB modeling abstraction is a form of transaction-based bus cycle accurate model that enables fast and accurate performance estimation for MPSoC designs. CCATB captures transactions in the design using function calls which allow a significant speedup in simulation. For instance, in the *ARM Advanced Microcontroller Bus Architecture Advanced High-speed Bus (AMBA AHB)* [8] on-chip communication architecture, hundreds of signals can transition during a data read or write issued from a processor to a memory. In the CCATB model, a `read()` or `write()` function call captures the functionality of the hundreds of signals, while still maintaining cycle accuracy required for meaningful exploration. This leads to a reduction in modeling time, and improves simulation speed by several orders of magnitude over signal-accurate *C++* or *Register-Transfer Level (RTL)* models. CCATB also performs additional optimizations, such as effectively clustering static MPSoC delays and incrementing simulation time in chunks, to further improve simulation speed. The contributions of this chapter are focused on the following areas:

- We utilize the ALDERIS DSML introduced in [68] for the formal modeling of MPSoCs with bus matrix interconnection networks. ALDERIS was proposed as a DSML for the modeling of software-intensive DRE systems. In this chapter we extend the use of ALDERIS to complex bus matrix architectures commonly used in modern MPSoCs. The novelty in this chapter is to show how complex bus designs can be abstracted out as transaction-level models, and how we translate resource allocation to the ALDERIS task graph model. The ALDERIS DSML is used as a high-level specification of the MPSoC, and directly drives the

functional verification, performance estimation, and performance verification methods. This is described in more detail in Section 9.2.

- We describe an approach for the functional verification of MPSoCs using AMBA AHB bus matrix interconnection networks. We extend earlier work on the verification of simple bus designs [73] to handle complex bus matrix structures. We use finite state machine models of the AMBA AHB protocol with cycle-accurate timing information to formally verify deadlock-freedom. We describe this method in Section 9.3.

- We utilize a *Discrete Event Simulation (DES)*-based formal performance estimation method described in [70] to estimate the real-time performance of a networking MPSoC design. By switching to more abstract representation of the design, we achieve significant speedups and scalability increase with negligible accuracy loss. Section 9.4 describes results for this work.

- We build on our earlier work on the real-time verification of DRE systems [67, 65, 66] to propose a method to verify estimates for real-time performance using TA model checkers. By incorporating TA model checkers in the design flow, we can prove that the model satisfies the performance estimates achieved by the DES-based method. Section 9.5 presents this TA-based real-time verification method.

- Finally, the major contribution of this chapter is that it tightly integrates all of the above steps in the CARTA model-driven design analysis framework. This approach provides a way to use time-accurate models for functional verification, and more abstract representations for scalable performance estimation. By adopting the right abstractions to different steps of the analysis, we can significantly increase scalability when needed, while also retaining accuracy for

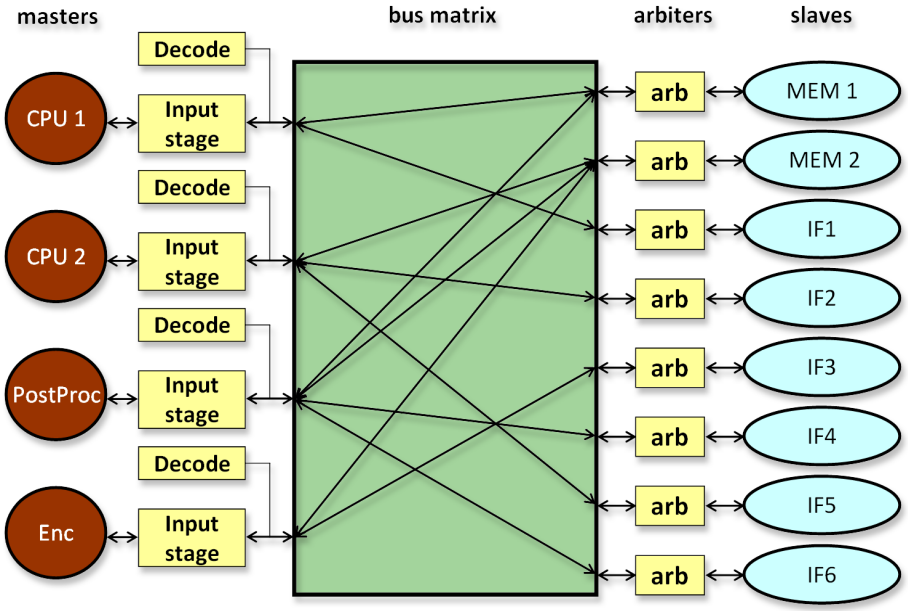

Figure 9.2: Networking Router MPSoC HW Design

steps where it is needed. We compare the scalability of the proposed methods in Section 9.6.

9.1 Networking Router MPSoC Design

To demonstrate the effectiveness of our design flow, we use CARTA to explore a networking router case study. This design is a high-level abstraction of a router design by Conexant Systems, that was first described in [83]. This system is used for data packet forwarding, processing and encoding. The MPSoC design implementation for this case study consists of multiple processors, memories and network interfaces, and a bus matrix interconnection network. The different application tasks are mapped onto the MPSoC components shown in Figure 9.2. The figure shows a simplified version of the hardware platform, without peripherals (*i.e.* timer, interrupt controller, *Universal Asynchronous Receiver/Transmitter (UART)*, etc.) and without the *Direct Memory Access (DMA)* engine, for clarity. The major components in the MPSoC

are the two embedded processors (CPU_1 and CPU_2), a post processing *Application-specific Integrated Circuit (ASIC)* (PostProc), an encrypt engine ASIC (Enc), on-chip memories (MEM 1 and MEM 2), network interfaces to communicate with external components(IF1 to IF6), and an AMBA AHB bus matrix interconnection network to facilitate inter-component communication on the MPSoC.

Figure 9.3 shows a high level overview of the software design modeled as a task graph, as well as the mapping of the different tasks onto the MPSoC components. Computation tasks (T) mapped to the same processing unit communicate directly with each other. The blocks marked as (B) represent accesses to the bus matrix, either to read/write one of the memory units (M), or to communicate with the external environment through network interfaces (IF). Processor CPU_1 is used to execute tasks associated with the intrusion detection functionality, while processor CPU_2 executes tasks associated with the simple protocol translation and packet forwarding functionalities. The encrypt engine ASIC block is optimized for data encoding, and executes tasks that perform different encodings on packet data. A post-processing ASIC block is used to speed up the back-end of the protocol translation, as well as the encoding functionalities.

We now describe the networking router application case study in more detail. The system receives data packets from the network interface (IF) components. The receiver (PktRx) tasks are responsible for data packet pre-processing and preliminary decoding. Subsequently, the system performs multiple functions - intrusion detection, simple protocol translation, forwarding, and packet encoding. The intrusion detection functionality consists of a Chk task that is used to check if the packets have not been subjected to some suspicious activity, as would be the case if there is an intrusion. The HdrCal task is used to perform data packet processing to detect intrusions, based on user-defined intrusion signatures. If an intrusion is detected, then reports of suspicious activities are generated. Finally the packets are stored in physical memory Mem_1 by

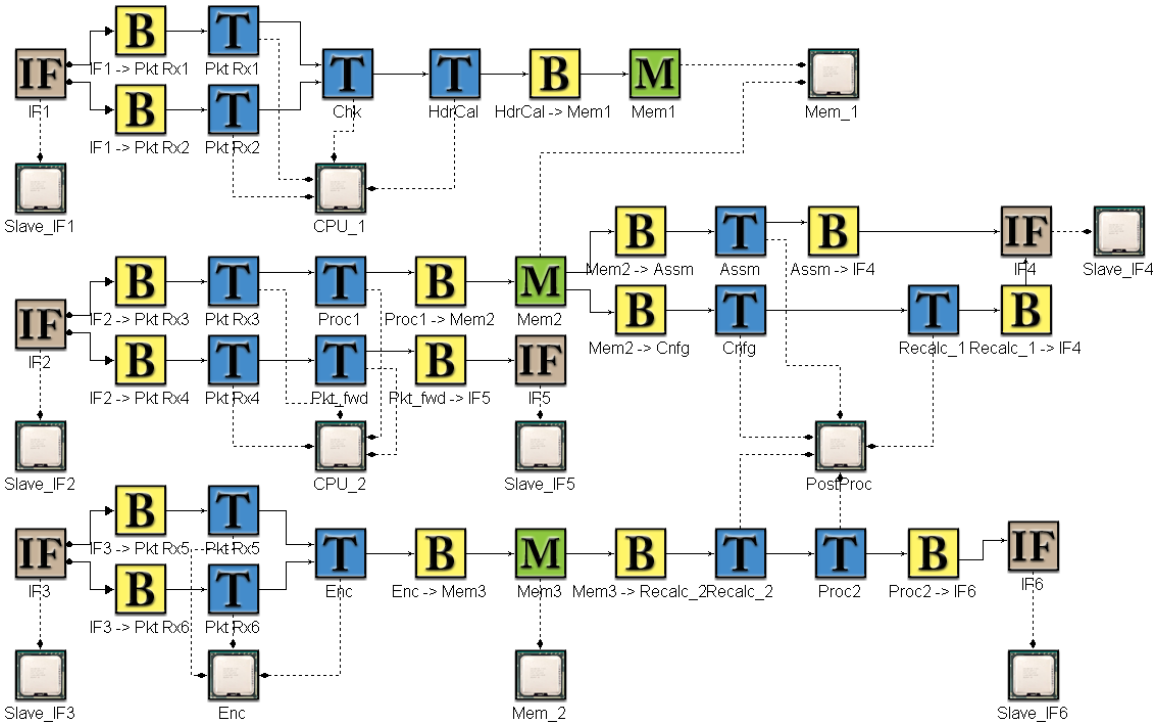

Figure 9.3: Networking Router MPSoC SW Design

the `Mem1` task from where another subsystem either blocks the flow, or passes the packets on to the next router. In the simple protocol translation functionality, the `Proc1` task is used to strip the source protocol and store the data packets into physical memory `Mem_1` by task `Mem2`. The memory also consists of a set of translational templates. These templates are used by the `Cnfg` task to strip the source protocol header, and then append the new protocol information to the data. The `Recalc_1` task finally reorders the data payload and new header, and sends out the packets to an outgoing network interface. The `Assm` task is used instead of the `Cnfg` and `Recalc_1` tasks if the protocol translation is fairly lightweight (for instance if the source and destination protocols are similar). The forwarding functionality consists of a task `Pkt_fwd` which receives a packet and updates the header data (*i.e.*, updating time stamps and stripping source routing fields), and then forwards the data to the output interface. `Enc` implements the packet data encoding functionality, and stores

the results into physical memory `Mem_2` by task `Mem3`. Subsequently, task `Recalc_2` is responsible for reordering the data payload and creating a header. Task `Proc2` is used to finally perform post-processing the packet data headers before forwarding them to the output network interface.

9.2 Modeling Bus Matrix-based MPSoC Designs

9.2.1 Modeling the Router MPSoC using Alderis

The networking router design shown in Figure 9.2 uses an AMBA AHB bus matrix interconnection network. This architecture simplifies the arbitration policy – simple point arbitration is used at the slaves (either memories or network interfaces) to determine which master gets access to the slave. If the slave is not already busy serving a master, any master can get access to it immediately, and transmit data through its dedicated bus to the slave. If the slave is already busy serving a request, then the masters trying to access the slave are forced to wait. When the slave is free again, the master with the highest priority gets access to the slave. This simple point arbitration policy provides a great fit for simple task scheduling problems – the memory is modeled as a task, and scheduling this task for execution represents the access of the memory by other tasks. This abstraction provides a simple, but accurate model to capture the event flow between tasks, as well as memory and interface accesses through the bus matrix.

In the following we describe how we used the Alderis DSML to model the case study described in Section 9.1. Tasks and interfaces are modeled as Alderis tasks. We introduce *First In First Out (FIFO)* buffers between tasks in order to *(1)* capture communication delays on the bus, and *(2)* buffer events, in case the task is already executing, and not able to receive the event yet. Dependencies in the Alderis task graph follow the SW design dependencies. However, there are some differences, as the

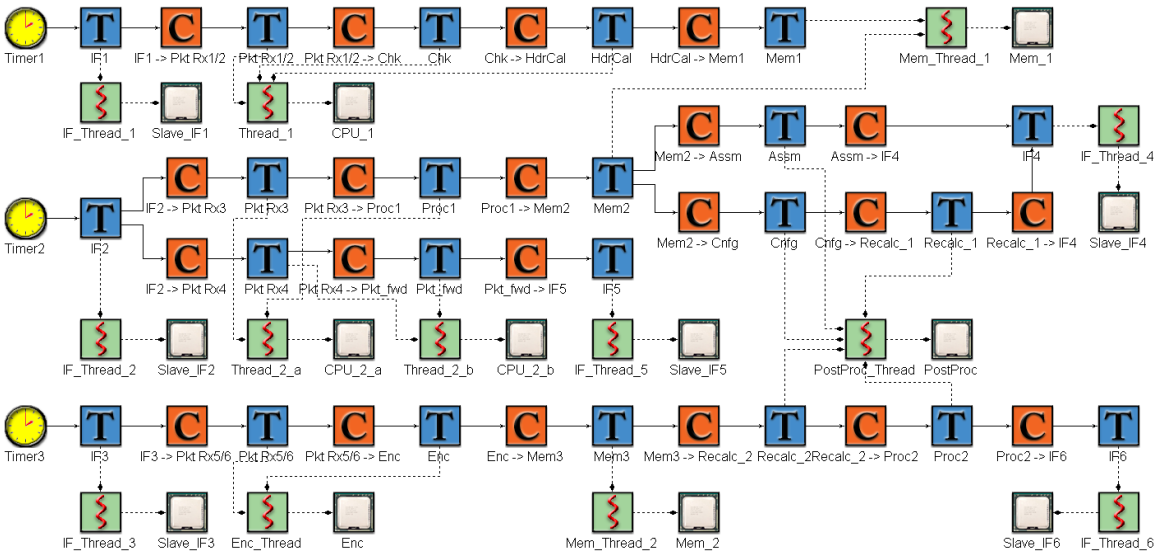

Figure 9.4: ALDERIS Model of the Router MPSoC in the GME Tool

ALDERIS models capture concurrency, real-time properties, and scheduling as well. We add timers to the model that represent the sampling rates over the input interfaces. Timers periodically push events, that represent data received from the environment. The task graph is event-driven and asynchronous, therefore further event propagation is not synchronized, timers are solely used as a triggering mechanism.

Figure 9.3 shows the dependencies in the SW design. It does not, however, capture the semantics of the event flow. There are four branches in the model (from `IF1`, `IF2`, `IF3`, and `Mem2`). The branches from `IF1`, `IF2`, and `IF3` represent a choice where only one path is taken. The branch from `Mem2` to `Assm` and `Cnfg` represent broadcast – both dependents receive the event, and process it when they are scheduled for execution.

The `CPU_2` HW component shown in Figure 9.3 is modeled as two independent (logical) HW units in the ALDERIS model shown in Figure 9.4 (`CPU_2_a` and `CPU_2_b`). This change is a direct result of the semantics of the DRE MoC [70]. In the DRE MoC, tasks and timers broadcast events to all dependents, and cannot selectively send events to a subset of dependents. The main reason that justifies this behavior is that we want to capture all execution paths in the model for performance estimation,

rather than choosing between non-deterministic branches.

IF2 is the interface for the protocol translation functionality. Depending on whether the data is forwarded or stored in the memory for further processing, one of two distinct event paths is taken from IF2. Although both paths execute on the same processing unit CPU_2, these two paths cannot be active at the same time; either the PktRx3, Proc1 path is taken (and then Mem2 and so on), or the PktRx4, Pkt_fwd, IF5 path. Even though these paths are mapped to the same processing unit, they are not concurrent. Therefore, we execute both paths in parallel for performance estimation. By introducing a new logical processing unit, CPU_2_b, we can simultaneously explore both paths at the same time.

We encounter the same situation at the branch from IF1; either PktRx1, or PktRx2 is chosen to process the packages arriving through the interface. Therefore, we have the option to introduce a new logical processing unit, to capture the fact that PktRx1 and PktRx2 do not execute at the same time. We choose a different option in this case, however, since the rest of the processing is the same in both cases; the only difference is between the receiving tasks PktRx1 and PktRx2. We simply substitute a new logical task (PktRx1/2), with a *best case execution time* (bcet) of min[bcet(PktRx1), bcet(PktRx2)], and a *worst case execution time* (wcet) of max[wcet(PktRx1), wcet(PktRx2)]. Regardless of which execution path is taken (PktRx1 or PktRx2), task PktRx1/2 captures the execution intervals of both tasks, and therefore can be used for worst case performance estimation. This method increases scalability with minimal loss of accuracy, and is therefore preferable in the early stages of the design flow. We apply the same idea for the branch from IF3, and substitute tasks PktRx5 and PktRx6 with the PktRx5/6 logical task. Finally, the branch from Mem2 follows broadcast semantics to dependents, and therefore requires no changes in the ALDERIS representation. The resulting ALDERIS model shown in Figure 9.4 captures dependencies between tasks, the mapping of tasks to the target

platform, as well as timing information of the networking router MPSoC design, and allows formal analysis as described in the following sections.

9.3 Functional Verification of AMBA AHB Bus Matrix MPSoC Designs

In this section we extend the modeling of AMBA AHB communication buses introduced in Section 8.3 to AMBA AHB bus matrix interconnects. In a fully connected bus matrix, there is a dedicated AMBA AHB bus between each master and slave. Point arbitration is used to manage access to slaves (such as memories). The bus matrix design increases throughput and concurrency, as multiple transactions can take place simultaneously between different masters/slaves.

In bus matrix designs, the functional verification is simplified. Each bus in the bus matrix connects a single master to a single slave. Therefore, congestions on the bus are greatly reduced, and can only appear when the slave is busy serving another master. There are altogether 11 buses in the networking router case study, as shown in Figure 9.2. Point arbitration is managed by a simple fixed-priority arbiter. Priorities for masters are defined as follows (in decreasing order): `CPU 1`, `CPU 2`, `PostProc`, `Enc`.

Since there is only one master and slave on each bus, the arbiter has no reason to interrupt a transfer by a master. Therefore, `HLOCK` signals have no practical use in a fully connected bus matrix. Nevertheless, it depends on the master whether it uses this feature or not. A master that does not deassert `HLOCK` could cause a livelock by disallowing the slave to serve other masters. However, it is the responsibility of the master to manage the `HLOCK` signal, and a master that does not deassert `HLOCK` would be considered faulty. When the master deasserts `HLOCK`, the arbiter is free to continue where it left off, so no livelocks can occur.

Algorithm 9.1 NuSMV Specification of an AMBA AHB Arbiter Managing a Single Master and Slave

```
MODULE bus_matrix_arbiter (HTRANS, HREADY, HRESP, BUSREQ, HGRANT, HMASTER,
HSPLIT)
  VAR
   master_state :  idle, mask, grant, transmit;
   master_prev_state :  idle, mask, grant, transmit;
   lasterror :  boolean;
   preferred :  master, default;
  ASSIGN
   init (master_state) := idle;
   init (master_prev_state) := master_state;
   init (lasterror) := 0;
   init (preferred) := default;
   next (master_prev_state) := master_state;
   next (preferred) :=
    case
     -- Master starts the transmission
     !HGRANT & master_state != mask & HREADY & HRESP = OK & BUSREQ :
master;
     -- Split master
     HGRANT & HRESP = SPLIT : default;
     -- Master finishes the transaction
     HGRANT & master_state = transmit & HTRANS = IDLE & HREADY & HRESP =
OK & !BUSREQ :
      default;
     -- Master gives up its BUSREQ
     HGRANT & master_state = idle & HTRANS = IDLE & HREADY & HRESP = OK &
!BUSREQ : default;
     -- Unmasking masters
     !HGRANT & master_state = mask & HSPLIT = master :  master;
     1 :  preferred;
    esac;
   next (master_state) :=
    case
     -- Roll back for retrys
     HRESP = RETRY & !lasterror :  master_prev_state;
     master_state = idle & HREADY & HRESP = OK & HGRANT : grant;
     master_state = grant & HTRANS != IDLE & HREADY & HRESP = OK & HGRANT
: transmit;
     master_state = transmit & HTRANS = IDLE & HREADY & HRESP = OK &
HGRANT : idle;
     HREADY & HRESP = SPLIT & HGRANT & !lasterror :  mask;
     master_state = mask & HSPLIT = master :  idle;
     lasterror :  master_state;
     !HREADY : master_state;
     1 :  master_state;
    esac;
   next (lasterror) :=
    case
     HRESP = RETRY : 1;
     HRESP = SPLIT : 1;
     1 :  0;
    esac;
   HMASTER :=
    case
     preferred = master :  master;
     preferred = default :  default;
    esac;
   HGRANT :=
    case
     preferred = master :  1;
     1 :  0;
    esac;
```

We have used the NuSMV [1] model checker to verify *Computational Tree Logic (CTL)* [25] properties on the networking router MPSoC design. We specified AMBA AHB masters, slaves, and the arbiter in NuSMV code. Algorithm 9.1 shows the NuSMV specification for an AMBA AHB arbiter managing a single master and slave. This model is sufficient to verify the correctness of the arbiter, as in a fully connected bus matrix each bus connects a single master to a single slave. Point arbitration is managed at the slave side; whenever the slave is busy serving a transaction, it signals this fact by setting the `HREADY` signal to low (other implementations are also possible).

In Section 8.3.1 we have described an ambiguity in the AMBA AHB specification, that may arise when a slave splits a a master, and requests retransmission by setting the `HRESP = RETRY` response. In a fully connected bus matrix, each AMBA AHB bus connects a single master and slave, and the slave has little reason to split the master. Nevertheless, we use the simple fix of disallowing the slave to split a master and set the `HRESP` signal to `RETRY` in the same cycle to avoid possible deadlocks.

To show that no deadlocks can occur in the model, we have shown that the `error` state is unreachable in masters and slaves by checking the following CTL formulas (where x refers to the index of masters, y is the index of slaves):

```
AG (MASTERx.state != error), AG (SLAVEy.state != error).
```

We have specified rules to enforce that whenever a master is split, it will be eventually unsplit in order to avoid livelocks. We have verified this property using the following CTL formulas:

```
AG ((MASK_MASTERx) -> AF (!MASK_MASTERx)).
```

Finally, we checked whether starvations are possible by checking the following formulas:

```
SPEC AG (MASTERx.state = busreq -> AF HGRANTx),
SPEC AG (MASTERx.state = busreq -> AF MASTERx.state = write).
```

9.3.1 Experiments

We have run experiments using the NuSMV tool on an Intel Core i7 processor running at 4GHz with 6GB triple-channel DDR3 RAM. For the arbiter connecting a single master and slave, the verification time took less than a second, with 6700KB memory consumption. For 2 masters, the analysis took less than a second with 11700K memory consumption. For 3 masters the analysis took 28 seconds with 92080KB memory consumption. Since in this study we consider fully connected bus matrices only, scalability issues do not arise, as each master is connected to each slave by a dedicated AMBA AHB bus.

We have found that starvations are possible in general when high-priority masters continually request access to slaves, and therefore lower-priority masters do not get served. Therefore, we need to consider the actual dependencies in the model, and perform real-time analysis to ensure that this condition does not occur. We need to consider the actual communication between tasks in the MPSoC design, and consider whether the starvation may occur in the actual MPSoC design.

This problem, however, cannot be adequately addressed at the cycle accurate abstraction due to the long computation times, that would lead to state space explosion. There is no theoretical limitation on performing real-time verification at the cycle accurate abstraction; the limitation is present simply due to the state space explosion present at low-level abstractions.

In the next section we describe how we obtained worst case performance estimates on the networking router case study, and how we used the results in the final stage for real-time verification using TA in Section 9.5. We address the problem of starvation in Section 9.5.

9.4 Formal Performance Estimation by Discrete Event Simulations

Section 9.2 demonstrates how MPSoC designs based on fully connected AMBA AHB bus matrix interconnects can be modeled using ALDERIS. This approach allows to utilizes the DES-based formal performance estimation method for the performance analysis of MPSoC designs based on fully connected AMBA AHB bus matrix interconnects.

We used the open-source DREAM tool – that implements the DES-based simulation-guided performance estimation method – for the performance estimation of the networking router MPSoC design. Table 9.1 shows the parameters used for the model shown in Figure 9.4. We have obtained the deadlines from application requirements, and the execution time estimates by using fast and accurate system-level simulation at the CCATB abstraction [84].

Figure 9.5 illustrates the DES-based performance estimation method. The dependencies imply a partial ordering on the execution order of tasks, as well as the events during the discrete event simulation. We distinguish three major types of events in Figure 9.5; *input* events, denoted as "i", *output* events denoted as "o" and *start* events denoted as "s". Each event is followed by a number to indicate which task is responsible for the event. This is simply a syntactic short-hand to keep the nodes in the tree small. Numberings for tasks are given in the last column of Table 9.1.

As seen from Figure 9.5, the application starts with tasks `IF1`, `IF2`, and `IF3` receiving input events. The second node shows that the three tasks are scheduled (started) by the scheduler for execution. Since `IF2` has the smallest execution time between `IF1`, `IF2`, and `IF3`, therefore `IF2` will be the first to finish its execution. When `IF2` finishes its execution, it generates an output event, that is broadcasted to both `PktRx3` and `PktRx4`.

The first non-deterministic branch occurs after `Pkt_fwd` (numbered 10) is scheduled

Task	CPU	Priority	WCET	BCET	Deadline	Numbering
IF1	Slave_IF1	1	200	-	201	1
IF2	Slave_IF2	2	100	-	101	2
IF3	Slave_IF3	3	400	-	401	3
IF4	Slave_IF4	4	200	-	201	22
IF5	Slave_IF5	5	200	-	201	14
IF6	Slave_IF6	6	400	-	401	23
Mem1	Mem_1	7	100	-	201	16
Mem2	Mem1_1	8	100	-	201	13
Mem3	Mem_2	9	100	-	101	15
PktRx1/2	CPU_1	10	1442	770	1443	4
Chk	CPU_1	11	450	220	451	8
HdrCal	CPU_1	12	2400	1340	2401	12
PktRx3	CPU_2_a	13	1530	1400	1531	5
PktRx4	CPU_2_b	14	670	590	671	6
Pkt_fwd	CPU_2_b	15	600	200	601	10
Proc1	CPU_2_a	16	2800	2400	2801	9
Cnfg	PostProc	17	1600	1300	1601	18
Assm	PostProc	18	800	600	2401	17
Recalc_1	PostProc	19	3000	2000	3801	20
PktRx5/6	Enc	20	2100	1100	2101	7
Enc	Enc	21	4900	3800	4901	11
Recalc_2	PostProc	22	1750	1500	6381	19
Proc2	PostProc	23	5600	4800	5601	21

Table 9.1: Parameters for the Networking Router MPSoC Design Shown in Figure 9.4

for execution. The earliest time when Pkt_fwd may finish its execution is at time 890, as can be computed by adding the best case execution times of itself and its sources IF2, PktRx4 (100+590+200=890). We can also compute that Pkt_fwd will finish its execution by time 1370 (100+670+600=1370). For task PktRx1/2 we can similarly obtain that it will finish its execution between 970 and 1642. Therefore, the execution intervals of Pkt_fwd and PktRx1/2 overlap, and we need to consider two cases; when Pkt_fwd finishes first, and when PktRx1/2 finishes first. It is also possible that the two tasks finish simultaneously, in which case race conditions may be considered, resulting in the same two options. Total orderings become important when multiple tasks are mapped to the same thread or processing unit. For example, both Mem1 and

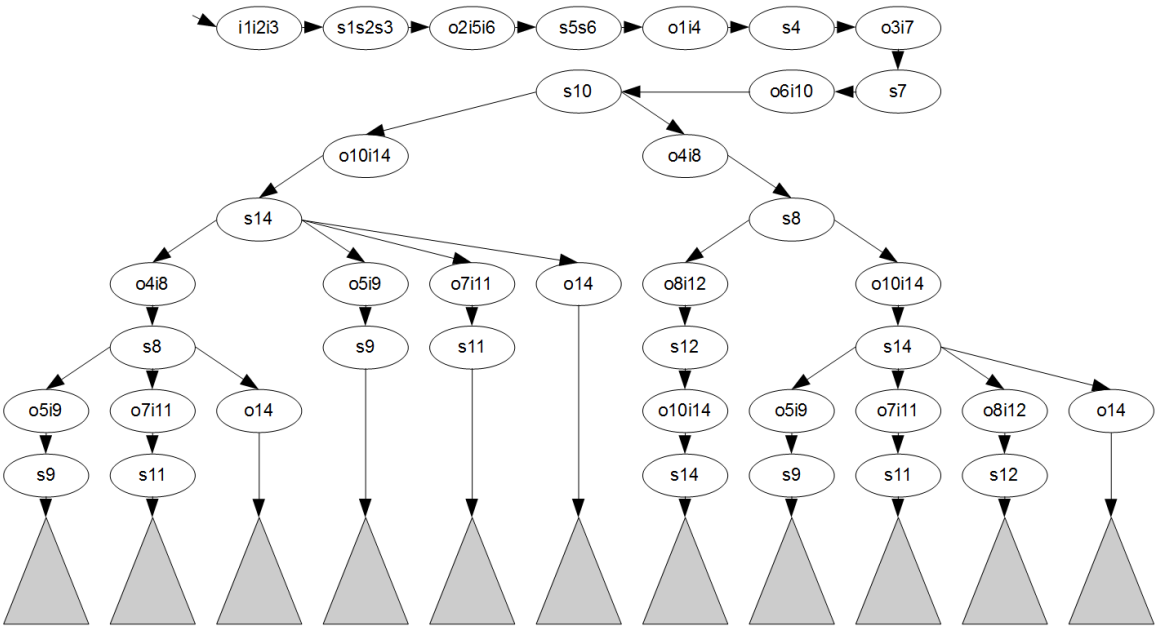

Figure 9.5: A Partial View of the Event Order Tree for the Example Shown in Figure 9.4 using the Parameters in Table 9.1

`Mem2` denote memory accesses in the same memory module. Likewise, tasks `Cnfg`, `Recalc_1`, `Recalc_2` and `Proc2` are all mapped to the same thread, and therefore it is important to consider the order in which they may be scheduled for execution. Figure 9.5 only illustrates the top of the tree, and the gray rectangles refer to subtrees of the corresponding nodes.

The size of the event order tree grows very fast even for moderate-size examples, and pre-computing the tree is not possible due to resource constraints. Rather, the proposed method builds the event order tree on-the-fly, that captures all the possible total orderings of events. Branches are discovered during simulation-time, and then subsequently enumerated. Algorithm 6.1 describes how the event order tree is constructed on-the-fly.

By enumerating the event order tree, one can obtain the worst case bound on the overall performance of the model. To enumerate the tree, the discrete event simulation step described in 6.2 are performed repeatedly, where the execution time of tasks is

continuously updated to capture all possible permutations of execution times. Since the analysis is based on repeated simulations, we refer to this approach as "simulation-guided model checking". The event order tree captures all permutations of events, and is therefore exhaustive; it does not produce false positives. On the other hand, the method is built on the assumption that discrete event simulations using limited horizon are sufficient, after which the system returns to an initial idle state, which may not be the case for all types of real-time systems. For a description of the formal performance estimation method and proofs on the validity of the performance estimation method please see Section 6.2.3.

The DES-based method is more accurate for performance estimation than static analysis methods, as it captures dynamic effects such as congestions on the bus, and race conditions. The advantage of the DES-based method compared to ad-hoc simulations is the increased state space coverage. Compared to pure model checking methods, the main advantage is that it does not run out of memory on large-scale examples, as it is based on fast iterative simulations, and is therefore *Central Processing Unit (CPU)*-bound. Moreover, most model checkers are tailored to answer yes/no questions, but the DES-based method can directly obtain the worst case bound on the end-to-end computations.

9.4.1 Experiments

The open-source DREAM tool computed the worst case end-to-end performance of the networking MPSoC design modeled as shown in Figure 9.4 in 590 seconds on an Intel Core i7 920 processor running at 4GHz using 6GB triple channel memory, with only 776KB memory consumption. The implementation of the DES-based method was not optimized to take advantage of the multi-core architecture, and therefore executed on a single thread. The DES-based performance estimation method is easy to parallelize as it consists of repetitive simulations, and we are considering a multi-

core implementation in the future.

The overall end-to-end performance estimate is 17780 cycles. We use this information as a bound on the end-to-end performance of the networking router MPSoC design. Each cycle in the model represents $5ns$. The lowest period of the timers that still does not violate the end-to-end bound is therefore $\frac{17780\times5}{10^3} = 88.9\mu s$. This means that the highest possible frequency for the sampling is ∼11.248KHz. Note that with each sampling the networking router MPSoC processes several packets, and therefore provides reasonable performance. The analysis is exhaustive, and therefore the bound is tight. In the next section we perform real-time verification on the model to prove that the performance estimates hold, and that no starvation occurs in the MPSoC design.

9.5 Real-time Verification using Timed Automata

Real-time verification is optional in most cases, as the performance estimation method is based on an exhaustive state space search, and is therefore a model checking method itself. There are three cases when the use of the extra verification step is justified. First, the DES-based performance estimation method does not capture timed states to improve scalability, but rather utilizes a *limited horizon* for the simulations. Although this approach is sufficient in most cases where the simulation periodically returns to the initial state, it cannot be applied to all models in general. For example, obtaining the required time horizon for simulating a pipeline architecture may be error-prone. In other words, while the DES-based method does not produce false positives, it relies on the assumption that a limited horizon is sufficient for the analysis. In cases where this condition cannot be proven, the use of the additional verification step is required.

Second, for some complex MPSoC designs the performance estimation method *might not terminate* due to the state space explosion problem. TA-based model checkers in

some cases (but not always) achieve better scalability. In this case, the use of the real-time verification method is justified, as it might prove the validity of the worst case performance estimate.

Third, the TA-based model checker can be utilized to *prove properties* on the design that could not be carried out at the cycle-accurate level due to the state space explosion problem. Our FSM-based analysis described in Section 9.3 has shown that starvations may be present in the design due to the fixed-priority point arbitration algorithm utilized in the slaves. By capturing the MPSoC design shown in Figure 9.4 as TA, we can prove that no livelocks and starvations are present in the model. Note that real-time properties such as execution times may influence starvations, and therefore have to be considered for the analysis.

In this section we utilize the TA model checking due to the third condition; it is hard to prove that no deadlocks and livelocks are present using the DES-based method. During DES, we can easily identify situations where a task did not execute at all, but it is hard to identify whether a blocking occurs simply due to waiting for resources, or an actual deadlock or livelock. The TA-based method removes any doubt, and the models can be automatically generated from DREAM.

Figure 9.6 shows the partial TA representation in the UPPAAL tool for the networking router MPSoC design shown in Figure 9.4. The locations denoted with U are urgent locations, and C denotes committed locations [12], both of which imply that time cannot pass in that location, and the outgoing transitions needs to be taken immediately upon entering the location. Tasks have two clocks, c_e and c_d. Tasks, channels, timers, and schedulers compose together as a network of TA, providing an abstract model for scheduling. The translation from ALDERIS models to the TA representation is described in detail in Chapter 5 and Chapter 7. The translation process itself is based on refinement, DREAM generates a TA model for each task, FIFO buffer, and timer in the ALDERIS models using a template. Scheduling policies are specified

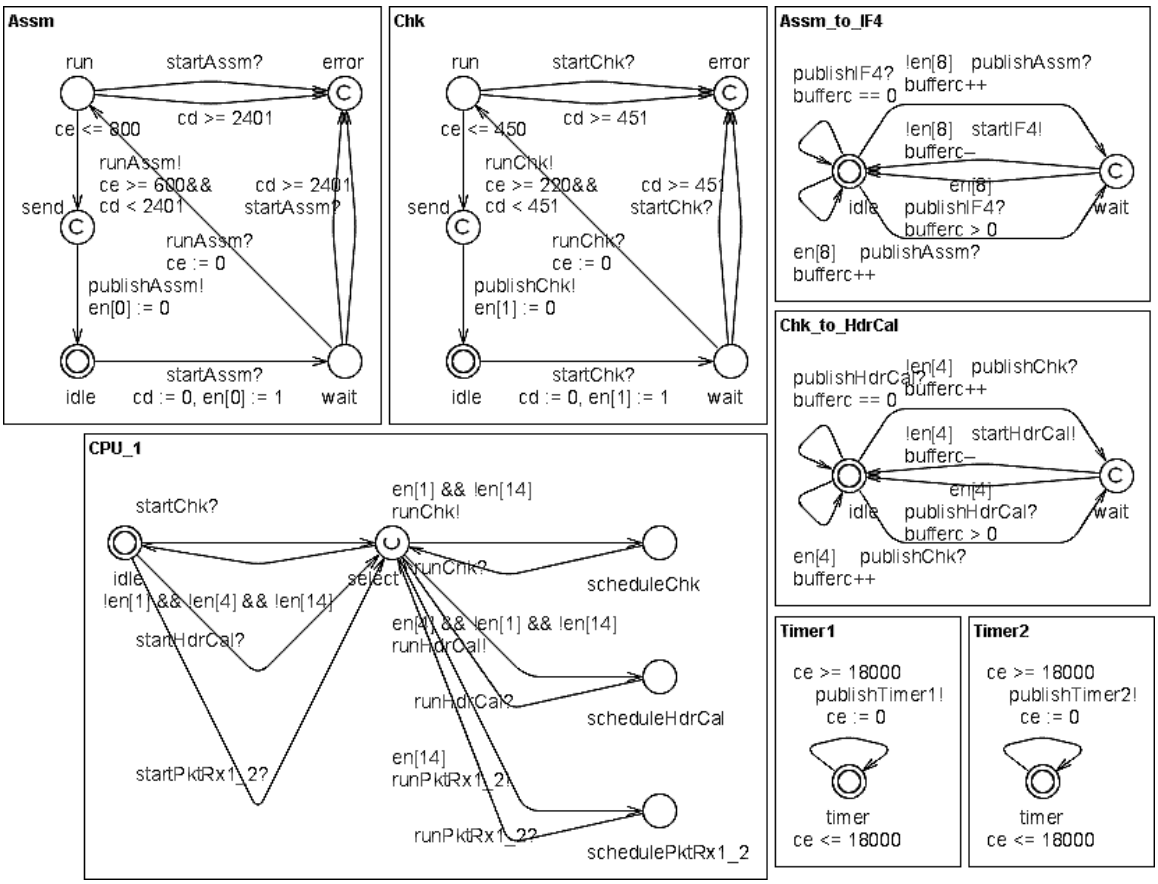

Figure 9.6: Partial Timed Automata Model of the Networking Router MPSoC Design Shown in Figure 9.4 in UPPAAL

as automata, where transitions may trigger the execution of tasks.

The translation is implemented in the open-source DREAM tool and is fully automated. DREAM generates TA representations for the UPPAAL and Verimag IF model checkers. In this section we arbitrarily focus on UPPAAL, as both tools are TA model checkers. We have verified that no task can violate its deadline by checking the following (by now perhaps familiar) UPPAAL macro:

```
A[]  not  deadlock
```

Since the TA are created in such a way to deadlock when a deadline is violated, this analysis proves the real-time schedulability of the system. Note that this result does not contradict the FSM-based analysis. The FSM-based analysis shows that deadlocks

may be present in general in MPSoC designs utilizing AMBA AHB with fixed-priority point arbitration. When we consider the actual communication in the MPSoC design by considering dependencies and when components request access to resources in the TA model, we can prove that no deadlock are present in the actual MPSoC design. In short, deadlocks may be present in general, but are not present in the router case study used in this chapter.

9.5.1 Experiments

We have run experiments using the UPPAAL model checker on an Intel Core i7 920 processor running at 4GHz using 6GB triple channel memory. The verification time took less than a second for the design illustrated in Figure 9.6, with 9140KB memory consumption. We have rounded up the period of the timer to 18000 cycles for simplicity, and we were able to prove the end-to-end execution time of 17780 cycles. The real-time verification shows that any frequency that is lower than the corresponding 11.1KHz is guaranteed not to violate the end-to-end deadline, or the individual deadlines of tasks.

9.6 Comparing the Results of the Analysis Methods

Figure 9.7 illustrates the analysis time and memory consumption used for the analysis of the networking router case study. During the functional verification phase, we considered a cycle-accurate model of the AMBA AHB bus, and found that deadlocks may be present due to the fixed-priority arbitration. The analysis time took less than one second for buses containing a single master and slave, however we observed increased analysis time and memory consumption as more and more masters were added to the bus. This was mainly the result of the more complex arbitration policy needed. Since in fully connected bus matrices each master is connected to each slave,

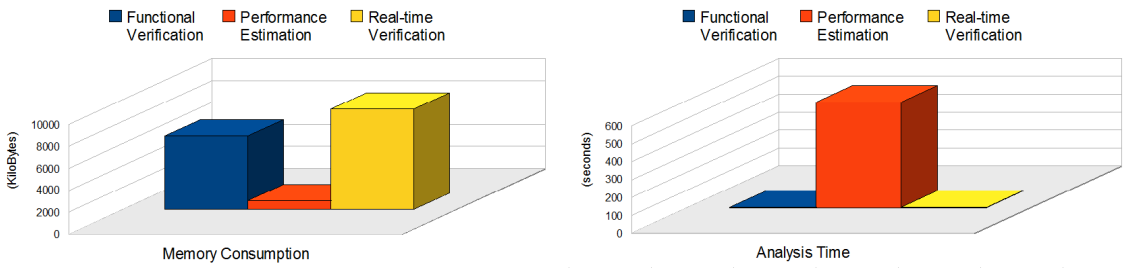

Figure 9.7: Analysis Time and Memory Consumption for the Networking Router Case Study

analysis scalability was not an issue.

For performance estimation, we relied on the DES-based simulation-guided model checking method. Analysis time was higher, while memory consumption was lower. Generally, we find that memory-bound model checkers have better performance than the CPU-bound method if the example is actually small enough to fit in the main memory. Once the model size increases beyond the main memory size, memory-bound model checkers are not useful in practice, as performance degrades significantly, and no partial results are given. For this example, we find that the model is small enough to fit in memory, and the various optimizations result in improved performance compared to the DES-based method, although both methods are relatively fast.

For real-time verification, the UPPAAL tool shows impressive performance by proving deadlock-freedom, as well as proving the real-time schedulability of the router design. Results are similar to the NuSMV results, even though the problem analyzed is different: UPPAAL considers the interactions between tasks – similarly to the DES-based method – on a transaction-level abstraction. Here we see the advantage of using multiple abstractions for the analysis; performing the same analysis using the NuSMV tool at the cycle-accurate level would result in serious performance penalty, and possibly even state space explosion. Our results show the practical applicability of the CARTA framework for the cross-abstraction analysis of MPSoC designs.

Chapter 10

Simulation-guided Model Checking: The Dream framework

In this chapter we describe the *Distributed Real-time Embedded Analysis Method (*Dream*)* (`http://dre.sourceforge.net`) for the model-based analysis of *Distributed Real-time Embedded (DRE)* systems. The Dream project focuses on the practical application of formal analysis methods to automate the verification, development, configuration, and integration of *Asynchronous Event-driven Distributed Real-time Embedded (AEDRE)* systems. The open-source Dream tool is a prototype implementation of the analysis methods described in Chapter 5, Chapter 6 and Chapter 7. This chapter describes the design of the open-source Dream tool.

Dream is a model-based analysis method and tool for the real-time verification and performance estimation of DRE systems. The Dream design flow utilizes the concept of *Domain-specific Modeling Languages (DSMLs)* to capture simulations and model checking in a formal framework. Dream models can be specified using the *Analysis Language for Distributed, Embedded, and Real-time Systems (*Alderis*)* DSML introduced in Section 4.1. Alderis models can be constructed using the *Generic Modeling Environment (GME)* tool [61], but Dream accepts Alderis models in *Extensible Markup Language (XML)* format, which is simply a textual representation of the Alderis abstract syntax presented in Section 4.1.1. Alderis can express both AEDRE and *Time-triggered Distributed Real-time Embedded (TTDRE)* systems using the DRE Semantic Domain, either as *Timed Automata (TA)* as defined in Section 4.3, or as a *Discrete Event (DE)* system as defined in Section 4.4.

The major goal behind the development of Dream is to help designers bridge the gap between their domain of knowledge, and formal model checking, and to promote

the practical application of formal methods to DRE systems' development. DREAM, therefore, combines methods from modeling, simulations, model checking, and tool integration to facilitate this model-driven analysis flow.

10.1 Functionality Provided by DREAM

This section discusses the methods implemented in the open-source DREAM tool.

10.1.1 Random Simulations

DREAM implements a DE simulator with formally defined execution semantics using the ALDERIS *Model of Computation (MoC)* defined as a DE system in Section 4.4. DE simulations are used for random testing, and also drive the simulation-guided model checking algorithm for real-time verification and performance estimation.

The *Discrete Event Simulation (DES)* of a large-scale DRE model consisting of ∼100 tasks takes around 20ms on an Intel Core i7 920 processor running at 4GHz with 6GB three-channel memory. Therefore, the design space exploration is orders of magnitude faster than traditional simulation-based approaches.

10.1.2 Real-time Verification of Non-preemptive DRE Systems by Timed Automata

DREAM implements the real-time model checking method described in Section 5 based on the TA MoC using the UPPAAL [26] and Verimag IF [15] tools. ALDERIS models are automatically translated to an equivalent TA representation based on the DRE SEMANTIC DOMAIN introduced in Section 4.3. The models for the tasks and scheduling algorithms are automatically generated from DREAM allowing rapid evaluation of system designs.

The distributed multi-threaded fixed-priority scheduler is modeled as TA, that specifies priorities between transitions triggering the execution of the TA. When multiple tasks are enabled, the guards/priorities on transitions control which task will be allowed to access resources. This provides a formal model that captures the event-driven triggering commonly used in AEDRE systems, and provides an abstract model of execution.

DREAM can generate TA models for both UPPAAL [26] and the Verimag IF toolset [15]. The TA models represent the DRE system models specified using ALDERIS in a formal framework. The generated UPPAAL and IF models express the same behavior, but are not directly comparable since the semantics for specifying clock constraints, broadcast event passing, buffering, and the property checking are implemented differently in UPPAAL and IF. UPPAAL and IF are both timed automata model checkers. While the semantics of TA models differ slightly in UPPAAL and IF, they both are capable model checker tools, and can express the DRE SEMANTIC DOMAIN for practical real-time analysis.

10.1.3 Performance Estimation and Real-time Verification by DES

DREAM implements the performance estimation method based on DES introduced in Section 6. DREAM implements a DE scheduler, and can perform symbolic simulations of ALDERIS models using the DRE SEMANTIC DOMAIN specified as a DE system, as defined in Section 4.4.

DREAM models explicitly capture the event-flow and non-deterministic communication effects, such as varying delays etc. for dynamic performance evaluation. DREAM does not store timed states for the analysis, like TA model checking methods, as this is a significant contributor to memory consumption in model checking tools. By utilizing Algorithm 6.1, DREAM builds the event order tree introduced in Section 6.2.1 on-the-fly, implementing a *Central Processing Unit (CPU)*-bound method

for performance estimation and real-time verification.

Note that this approach represents real-time properties in continuous time. We do not address the termination problem at this stage of development, as we do not try to identify previously visited timed states, but use a constant horizon as a time limit for the analysis. Although there exist model checking methods, such as TA model checking, that do not have this limitation in theory, in practice all model checking methods suffer from the termination problem due to the state space explosion problem. The proposed method has minimal memory requirements, can provide partial results if the model is too large for exhaustive analysis, provides a way to measure coverage, and can provide counter-examples when properties are violated. Our results show that this approach can reach better coverage for the performance estimation of large-scale DRE system designs than alternative methods.

The DES-based performance estimation method is applicable only to non-preemptive systems due to decidability issues discussed in detail in 7.1. This means that the exhaustive verification and performance estimation can be ensured only in the case of non-preemptive systems. However, the DE scheduler can certainly perform symbolic simulations on preemptive models. A conservative approximation method for the performance estimation and real-time verification of ALDERIS models may be feasible in theory, given that such an approximation was found for TA as shown in Chapter 7. However, this dissertation and DREAM does not address the formal performance estimation problem based on DES for preemptive systems.

10.1.4 Real-time Verification of Preemptive DRE Systems by Timed Automata

Stopwatch Automata (SA) [77] were proposed as a MoC that can express preemptable tasks in asynchronous event-driven systems. It was shown that reachability analysis on the composition of SA as task graphs (integration graphs) is undecidable [55, 59]

in general.

DREAM implements the real-time model checking method for *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)* described in Section 7. ALDERIS models are automatically translated to TA models using the *Task Timed Automaton (TTA)* introduced in Section 7.2. The resulting TA model is an approximation of the SA model, and can be utilized for real-time model checking using the UPPAAL and Verimag IF tools.

To the best of our knowledge, the model checking method implemented using DREAM + UPPAAL + the Verimag IF toolset is the first practical implementation for the real-time model checking of *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)*. Alternative methods based on *Stopwatch Automata (SA)* and *Hybrid Automata (HA)* model checkers are not guaranteed to terminate, and have worse scalability than the proposed method, as discussed in Section 2.2.3.

10.1.5 Task Mapping Problem on a Distributed Platform by Genetic Algorithms

DREAM implements a method to obtain the mapping of tasks to threads in the ALDERIS model such that real-time constraints are satisfied. The task mapping problem is an extension to the job shop scheduling problem, that is NP-complete [14]. In this section we propose a solution for the task mapping problem using DES directed by a genetic algorithm, shown in Algorithm 10.1. Algorithm 10.1 is a pseudo-code representation of the task mapping method implemented in DREAM.

The major advantage of Algorithm 10.1 is that it is scalable for large-scale systems as well, and performs well for finding task mappings that satisfy real-time constraints. Obtaining a feasible mapping for a model of this size is in the order of seconds.

The disadvantage of the approach is that it builds on random algorithms, therefore

Algorithm 10.1 Heuristic for the Task Mapping Problem

1: we suggest the initial values $s = \frac{3 \times n}{8}, p_1 = 30\%, p_2 = 10\%, p_3 = 65\%$
2: generate $n \in \mathbb{N}$ design alternatives (called solutions) by randomly mapping all tasks to threads using $\mathsf{thread}(t_k)$
3: run a single simulation for all solutions, where the execution time for tasks is picked randomly from the execution interval
4: **for** i = 0; i < NUMBER OF SIMULATIONS; ++i **do**
5: calculate a fitness value for all solutions by counting how many tasks have missed their deadlines during the simulation, and weigh it by how much they have missed it
6: sort the n solutions in decreasing order based on the fitness value computed in the previous step
7: **if** there is a solution that satisfies constraints **then**
8: stop and run model checking on that task mapping solution
9: **end if**
10: put the first s solutions in set A and the second s solutions in set B
11: // Single parent mutation in set A
12: **for all** solutions in set A **do**
13: **for all** task t_k in the solution **do**
14: randomly modify $\mathsf{thread}(t_k)$ with p_1 probability
15: **end for**
16: **end for**
17: // Two parent mutation in set B
18: randomly pick two solutions x and y from set B
19: **for all** x and y pairs $\in B$ **do**
20: **for all** task t_k **do**
21: **if** $\mathsf{thread}(t_k)$ in solution $x = \mathsf{thread}(t_k)$ in solution y **then**
22: randomly modify $\mathsf{thread}(t_k)$ in both solutions with p_2 probability
23: **else**
24: randomly modify $\mathsf{thread}(t_k)$ in both solutions with p_3 probability
25: **end if**
26: randomly regenerate solutions that are not in A or B
27: **end for**
28: **end for**
29: **end for**

the worst case behavior of the model is not considered during the analysis. Therefore, the heuristic described in Algorithm 10.1 provides only the first step in the proposed methodology for the task mapping problem. In the second step, we designers should perform a formal real-time analysis on the most promising design alternatives utilizing the methods described earlier in this section.

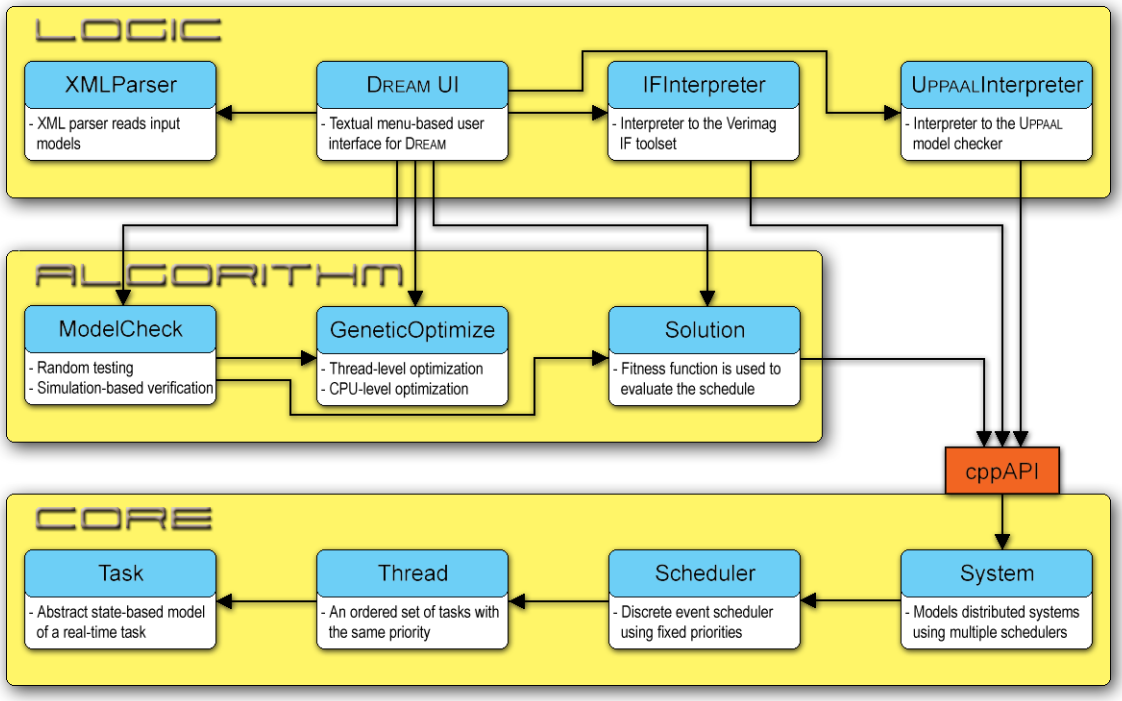

Figure 10.1: The Modular DREAM Design

10.2 Design and Implementation

The driving force behind the design of the DREAM tool is to create a tool which captures high-level system design in a formal setting. Tasks are also captured at a rather high level abstraction, using only 4 states: (idle, wait, run, and preempted). The main reason for the aggressive abstraction is to allow the real-time verification of a large number of tasks on heterogeneous platforms. Task computations are omitted for both the TA-based real-time verification and the DES-based performance estimation, but the DES-based method can be extended to capture simple computations, even during the exhaustive simulation-based verification.

The design of DREAM is split up into three major modules; `Core` implements the discrete event scheduler on a preemptive distributed platform, `Algorithm` implements various algorithms for verification and optimization that build on the `Core`, and `Logic` implements the DREAM UI and the interpreters which create TA models from the

internal representation in the `Core`.

The `Core` implements systems with varying complexity. The `Task` class implements a real-time task which models computations in the system. Tasks may have best and worst case execution times, and sub-priorities for non-preemptive scheduling. Tasks can be assigned to threads, which are a list of tasks with a priority for preemptive scheduling. A task's priority is the priority of the thread – tasks that are assigned to the same thread have the same priorities. A processing node is represented as a fixed priority scheduler. A scheduler manages a thread-pool of (possibly) several threads (with distinct priorities). Higher priority threads are favored against lower priority threads, whenever a high priority thread becomes enabled it preempts any lower priority threads that are running. DREAM can simulate hyper-threaded systems as well where several threads may run concurrently on the same execution node at the same time. The `System` class represents a distributed system consisting of several processing nodes and provides a *C++* API for the other two modules. The *C++* API uses the visitor pattern to get access to the `Core`.

The `Algorithm` module uses this *C++* API to obtain direct pointers to the tasks using the visitor pattern. Using the pointers the algorithm can optimize parameters by running several simulations, and specify whether best, worst, or random execution times are assumed for each task. The `Solution` class maintains the connection to the `Core` using the pointers and it can compute a fitness function for the current system model. If the system is schedulable 0 is returned, otherwise an error value is computed from the number of tasks that have missed their deadlines. The `GeneticOptimize` class uses genetic algorithms to optimize the scheduling by generating several possible priority assignments (using the `Solution` class). 2-parent and 1-parent mutations are used on the best solutions to obtain even better ones, while the worst candidates are replaced by random solutions. The `ModelCheck` class also builds on the `Solution` class. It implements simulation-based model checking for systems using

non-preemptive scheduling. The method builds on the `Core` which updates the states and clocks for the tasks for each event and checks whether a task's execution time is longer than its deadline.

The `Logic` module implements the text-based menu from which DREAM services can be accessed. The UPPAAL and IF interpreters generate TA models directly from the `Core`. An XML parser is used to read models into DREAM.

DREAM is implemented in *ANSI C++*. The source code compiles using gcc for Linux as well as Visual Studio 7.1 and 8.0 compilers, and should be easy to port to other platforms as well. STL data structures are used within DREAM for the sake of simplicity. The implementation is not particularly optimized but is relatively easy to read. All classes and methods are documented and available online. The website of the open-source DREAM tool is available at `http://dre.sourceforge.net`.

Chapter 11

Concluding Remarks and Future Work

Technological advances enabled an "information revolution" in the past decades, changing the way we search for information, solve problems, travel, communicate and interact with each other. *Embedded systems* are computation platforms that increasingly interact with the physical world, defining the concept of *Cyber-physical Systems (CPS)*. CPS are at the forefront of the information revolution, and have the potential to dwarf accomplishments in the field of computer science to date.

Distributed Real-time Embedded (DRE) systems provide the platform for the implementation of CPS, that increasingly run in open environments, in less predictable conditions than previous generations of real-time and embedded systems (such as micro-controllers) that are specialized for specific application domains (such as traffic control). DRE systems provide a highly adaptive and flexible infrastructure for reusable resource management services, thereby providing a platform for flexible and adaptive CPS.

The use of DRE systems is pervasive, ranging from small-scale *Multi-processor System-on-Chip (MPSoC)* designs operating in resource-constrained environments such as cell phone platforms, medical devices and sensor networks all the way to large-scale software-intensive systems of systems used in avionics, ship computing environments, and in *Supervisory Control and Data Acquisition (SCADA)* systems managing regional power grids.

Section 11.1 discusses the challenges addressed by this dissertation, Section 11.2 summarizes key contributions, and Section 11.3 presents future directions for research.

11.1 Challenges in the Design of Distributed Real-time Embedded Systems

Traditional mission-critical real-time systems extend the concept of the *time-triggered architecture* [58] to distributed and embedded systems. We refer to this class of systems as *Time-triggered Distributed Real-time Embedded (TTDRE)* systems. By separating the invocation of tasks from their activation, TTDRE systems achieve deterministic time behavior; by synchronizing the start of execution of tasks with the global clock designers achieve a high degree of *predictability*, and are able to express which tasks are allowed to execute at any point in time.

The vast majority of DRE systems, however, fall in the category of *Asynchronous Event-driven Distributed Real-time Embedded (AEDRE)* systems. AEDRE systems are based on a reactive, event-driven communication paradigm, where the execution of tasks is triggered asynchronously, depending on when they are invoked by external events, or other tasks. Event-driven systems provide a natural abstraction for DRE systems, as they closely resemble biological systems; whenever external events occur, the reaction follows as soon as possible.

As DRE systems are increasingly complex and diverse, and the question whether AEDRE or TTDRE systems should be used depends on the application domain, and key design constraints. In most heterogeneous DRE systems, AEDRE and TTDRE systems are used simultaneously; critical functionality may be provided by time-triggered components, while non-critical functionality may be provided by event-driven components.

The *composition* of time- and event-driven systems is a significant challenge that may have significant impact on the design of modern DRE systems, providing designers with the option to design systems that have the advantages of both AEDRE and TTDRE systems, thus leading to greater design freedom and flexibility. Developing analysis

methods for AEDRE systems, however, remains a key challenge.

11.2 Key Technical Contributions of this Dissertation

This dissertation proposed a model-based design methodology to address three major challenges in the formal analysis of DRE systems: *(1) functional verification* – to ensure that the system will not be trapped in a deadlock or livelock state, *(2) performance estimation* – in order to obtain tight bounds on the worst case performance of the DRE design, and *(3) verification of real-time properties* – to prove whether individual deadlines for tasks and performance estimates hold for the DRE design. The novelty of our approach lies in *(1) combining formal methods and symbolic simulations* for the system-level evaluation of DRE designs early in the design flow, and *(2) utilizing multiple abstractions* to trade off analysis accuracy and scalability. The key technical contributions of this dissertation are as follows:

- **Definition of a formal semantic domain for AEDRE systems:** we described the DRE Semantic Domain – a formal executable domain for the analysis of DRE systems. We reviewed common methods to specify semantics, then described our approach for the modeling of DRE systems by meta-modeling, and introduced the *Analysis Language for Distributed, Embedded, and Real-time Systems (*Alderis*) Domain-specific Modeling Language (DSML)*.

- **A model checking method for the real-time verification of non-preemptive AEDRE systems by** *Timed Automata (TA)***:** we specified TA models for the compositional analysis of DRE systems. We described the refinement-based transformation process that allows the analysis of Alderis models by TA model checking methods.

- **A performance estimation method for AEDRE systems using *Discrete Event Simulations (DES)*:** we described a novel DES-based performance

estimation method for DRE systems. The DES-based method is applicable to large-scale DRE systems as it is based on repetitive simulations of the model, and therefore does not suffer from memory consumption limits. Moreover, it can provide partial results in case the models are too large for exhaustive analysis.

- **A conservative approximation method for the verification of preemptive AEDRE systems by TA:** Preemptable tasks can be expressed using *Stopwatch Automata (SA)* [77]. The reachability problem on the composition of SA as a task graph is undecidable in general, since it can be mapped to the halting problem [55]. The schedulability of preemptive multi-processor systems is undecidable using TA in the generic case [59], as *Timed Automata (TA)* cannot directly model stopwatches. Therefore, model checking *Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE)* is a challenging problem that is undecidable in the generic case. This dissertation presented a novel conservative approximation method for the practical model checking of PEARSE. To the best of our knowledge, the proposed method is *the first decidable – and therefore practically applicable – method for the real-time verification of AEDRE systems designed as Preemptive Event-driven Asynchronous Real-time Systems with Execution Intervals (PEARSE).*

Utilizing the key technical contributions of this dissertation, we applied the analysis methods to the following problem domains:

- **Cross-abstraction verification and performance estimation of MPSoCs:** While MPSoC designs themselves can be viewed as DRE systems, the communication subsystem in MPSoC designs has a major impact on both design and analysis. Unlike software-intensive AEDRE systems that communicate over packet-switched networks, MPSoC designs often utilize complex bus matrix architectures, where access to the bus is managed by an *arbiter* (or several ar-

biters). Bus protocols and arbitration policies have a major impact on key design parameters such as throughput and delays, and present new challenges for functional verification. In particular, deadlock-freedom and livelock-freedom is not guaranteed by bus protocols, but is a key requirement for designers.

This dissertation introduced an approach for the combination of transaction-level simulations and model checking for formal MPSoC performance estimation and real-time analysis. We then extended the real-time analysis to bus matrix MPSoC designs. This dissertation described how methods for the analysis of AEDRE systems can be adapted to MPSoC designs utilizing fully connected bus matrix interconnects, and how point arbitration policies can be expressed by the non-preemptive scheduling of task graphs.

- **The open-source** *Distributed Real-time Embedded Analysis Method (*DREAM*)* **framework for the simulation-guided verification and performance estimation of AEDRE systems:** DREAM is an open-source tool and method for the model-based real-time verification and performance estimation of DRE systems. It implements the key technical contributions of this dissertation; *(1) Timed Automata (TA)*-based real-time verification of non-preemptive AEDRE systems using the UPPAAL [26] and Verimag IF [15] model checkers, *(2)* DES-based method for performance estimation, *(3)* conservative approximation method for the verification of PEARSE. DREAM models may be specified using ALDERIS, a modeling language based on the DRE SEMANTIC DOMAIN. The DREAM project focuses on the practical application of formal analysis methods to automate the verification, development, configuration, and integration of AEDRE systems. DREAM is available for download at `http://dre.sourceforge.net`.

11.3 Future Directions

The model-based design and early exploration of DRE systems is a challenging problem. This dissertation investigated the functional verification, real-time analysis and performance estimation of AEDRE systems in detail. However, there remain significant challenges to apply the results of this work to complex DRE system designs. The work presented in this dissertation can be extended in the following directions:

- **Hierarchical (compositional) approach to verification:** we developed methods for the analysis of preemptive and non-preemptive DRE systems. The proposed methods are based on the DRE Semantic Domain, and allow hierarchical verification by composing events. In most practical DRE systems, preemptive and non-preemptive components are used simultaneously, and components interact through a packet-switched network (in the case of large-scale software-intensive DRE systems), or through an on-chip interconnect (in the case of MPSoCs). Therefore, component interactions can be captured in a high-level model, where components are represented as "black boxes". The timing of component interactions can then be captured and verified through interfaces, that specify the time intervals when the component is sending/receiving events. This approach would enable the real-time analysis of medium- to large-scale DRE systems.

- **Energy (and power) verification:** The DRE Semantic Domain captures the timed behavior of AEDRE systems. While the focus of this dissertation is on real-time properties, the DRE Semantic Domain can also express energy consumption at the task-level, including frequency scaling and dynamic voltage scaling, as shown in our earlier work [68]. By annotating the analysis models with accurate power information, the model checker can compute energy consumption over timed traces, and prove energy consumption properties. In

theory, both leakage and dynamic power could be captured in the proposed framework. By directly building on our existing work on real-time analysis, a task-level energy estimation approach is well within reach.

- **Multi-core implementation of the DES-based real-time analysis method described in Chapter 6:** The DES-based performance estimation and real-time verification method is CPU-bound, unlike most model checking techniques. The recent advance of multi-core processors provides a platform that could significantly improve the performance of the proposed DES-based analysis method.

Beside these immediate extensions, the following improvements seem within reach with more extensive changes:

- **Integration with real-time calculus:** Modular Performance Analysis [33] is an approach to characterize DRE systems merely by describing incoming and outgoing event rates, message sizes, and execution times. Resources and the distributed execution platform is defined in similar terms, and *Real-Time Calculus* is then used to compute upper and lower bounds of the system performance. Integrating real-time calculus into the proposed model-based design framework would provide better analysis scalability at the price of accuracy, giving designers more options for tradeoffs.

- **Real-time kernel based on the open-source DREAM tool:** DREAM implements a fixed-priority real-time scheduler that is used for DES. Implementing a lightweight kernel that utilizes the same model of computation would allow designers to implement DRE software that can be automatically analyzed by DREAM.

- **Run-time analysis:** next-generation DRE systems need to be increasingly adaptive to address challenges in CPSs. While static analysis provides an ap-

proach for the design of mission-critical systems, run-time analysis allows more flexible DRE systems, and improved adaptability. Given the small footprint of DREAM, a method could be developed that considers heuristics based on DES for dynamic run-time reconfiguration.

- **Integration with Model Predictive Control/Supervisory Control:** Model Predictive Control [76] is a very effective method for the run-time control of DRE systems. Since the DRE SEMANTIC DOMAIN is based on a *Discrete Event (DE)*/TA formalism, combining the proposed method with supervisory control of DE systems [103] would improve run-time adaptation and control in mission-critical systems.

Bibliography

[1] A. Cimatti and E. Clarke and E. Giunchiglia and F. Giunchiglia and M. Pistore and M. Roveri and R. Sebastiani and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, 2002.

[2] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 113–126, 2002.

[3] A. Agrawal, G. Karsai, and A. Ledeczi. An End-to-End Domain-Driven Development Framework. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct 2003.

[4] Alberto Sangiovanni-Vincentelli. Defining Platform-based Design. *EEDesign of EETimes*, February 2002.

[5] R. Alur, C. Courcoubetis, and D. L. Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104(1):2–34, 1993.

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

[7] H. Amjad. Verification of AMBA Using a Combination of Model Checking and Theorem Proving. *Electronic Notes in Theoretical Computer Science, Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVoCS 2005)*, 145:45–61, 2006.

[8] ARM. AMBA Specification rev 2.0, IHI-0011A, 1999.

[9] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the Middleware 2000 Conference*, 2000.

[10] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45–52, 2003.

[11] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing Applications using Model-driven Design Environments. *IEEE Computer*, 39:33–40, 2006.

[12] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. *Lecture Notes on Concurrency and Petri Nets*, 3098:87–124, 2004.

[13] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 91:64–83, 2003.

[14] J. Blazewicz, J. Lenstra, and A. R. Kan. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, pages 11–24, 1983.

[15] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. *Formal Methods for the Design of Real-Time Systems, LNCS 3185*, pages 237–267, 2004.

[16] V. A. Braberman and M. Felder. Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification. *Lecture Notes in Computer Science, Proceedings of the joint 7th ESEC-FSE Conference*, 1687:494–510, 1999.

[17] S. Bradley, W. Henderson, and D. Kendall. Using Timed Automata for Response Time Analysis of Distributed Real-Time Systems . In *Proceedings of the 24th Workshop on Real-Time Programming (WRTP)*, pages 143–148, 1999.

[18] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

[20] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, 29:5–26, January 2005.

[21] C. G. Cassandras. *Discrete Event Systems, Modeling and Performance Analysis*. Irwin, 1993.

[22] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, pages 138–152, 2000.

[23] P. Chauhan, E. M. Clarke, Y. Lu, and D. Wang. Verifying IP-Core based System-On-Chip Designs. In *Proceedings of IEEE ASIC SOC Conference*, pages 27 – 31, 1999.

[24] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. CALM and Cadena: Metamodeling for Component-Based Product-Line Development. *IEEE Computer*, 39(2):42–50, 2006.

[25] E. Clarke and E. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. *Logic of Programs, Lecture Notes in Computer Science*, 131:52–71, 1981.

[26] A. David, G. Behrmann, K. G. Larsen, and W. Yi. A Tool Architecture for the Next Generation of Uppaal. Technical report, Uppsala University, 2003.

[27] D. de Niz, G. Bhatia, and R. Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 231–242, 2006.

[28] M. Deshpande, D. C. Schmidt, C. O'Ryan, and D. Brunsch. Design and Performance of Asynchronous Method Handling for CORBA. In *Proceedings of Distributed Objects and Applications (DOA)*, 2002.

[29] B. S. Doerr and D. C. Sharp. Freeing Product Line Architectures from Execution Dependencies. In *Proceedings of the 11th Annual Software Technology Conference*, 1999.

[30] V. D'silva, S. Ramesh, and A. Sowmya. Synchronous protocol automata: a framework for modelling and verification of SoC communication architectures. In *IEEE Proceedings of Computers and Digital Techniques*, volume 152, pages 20–27, January 2005.

[31] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[32] C. Ericsson, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *Proceedings of Real-Time Computing Systems and Applications (RTSCA)*, pages 182–189, 1999.

[33] Ernesto Wandeler and Lothar Thiele and Marcel Verhoef and Paul Lieverse. System architecture evaluation using modular performance analysis - a case study. *Software Tools for Technology Transfer (STTT)*, 8(6):649–667, Oct. 2006.

[34] P. H. Feiler, B. Lewis, and S. Vestal. The SAE AADL Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *Workshop on Model-driven Embedded Systems*, 2003.

[35] T. Gerdsmeier and R. Cardell-Oliver. Analysis of Scheduling Behaviour using Generic Timed Automata. *Electronic Notes in Theoretical Computer Science*, 42:143–157, 2001.

[36] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt. Integrated Adaptive QoS Management in Middleware: An Empirical Case Study. *Real-time Systems*, 24(2–3):101–130, 2005.

[37] A. Goel and W. R. Lee. Formal Verification of an IBM CoreConnect Processor Local Bus Arbiter Core. In *Proceedings of the 37th Design Automation Conference (DAC)*, pages 196–200, 2000.

[38] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *Science of Computer Programming: Special Issue on Model Driven Architecture*, 73:39–58, 2008.

[39] Z. Gu, S. Wang, S. Kodase, and K. G. Shin. An End-to-End Tool Chain for Multi-View Modeling and Analysis of Avionics Mission Computing Software. In *Proceedings of Real-Time Systems Symposium (RTSS)*, pages 78–81, 2003.

[40] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[41] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The Design and Performance of a Real-Time CORBA Event Service. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 184–200, 1997.

[42] S. Hartmann and R. Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operations Research*, pages 394–407, 2000.

[43] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):110–122, 1997.

[44] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[45] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998.

[46] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004.

[47] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[48] IBM. 32-bit Processor Local Bus Architecture Specifications ver 2.9, SA-14-2531-01, 2001.

[49] IEEE. VHDL (IEEE 1076 Standard), 2000.

[50] IEEE. Verilog (IEEE 1364 Standard), 2001.

[51] IEEE. SystemVerilog (IEEE 1800 Standard), 2005.

[52] ITU-T VCEG, ISO/IEC MPEG. ISO/IEC 14496-10 International Standard (ITU-T Rec. H.264), 2003.

[53] JPEG Committee. ISO/IEC JTC1/SC29/WG1 N1855, JPEG 2000 Part I: Final Draft International Standard (ISO/IEC FDIS15444-1). 8.2000.

[54] J.Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual, 1998.

[55] Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Decidable Integration Graphs. *Information and Computation*, 150(2):209–243, 1999.

[56] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[57] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza. *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[58] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, Oct. 2001.

[59] P. Krcal, M. Stigge, and W. Yi. Multi-Processor Schedulability Analysis of Preemptive Real-Time Tasks with Variable Execution Times. In *Proceedings of FORMATS*, pages 274–289, 2007.

[60] K. Lahiri, A. Raghunathan, and S. Dey. System-Level Performance Analysis for Designing On-Chip Communication Architectures. *IEEE Transactions on Computer Aided-Design of Integrated Circuits and Systems*, 20:768–783, 2001.

[61] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, and J. Sprinkle. Composing Domain-Specific Design Environments. *IEEE Computer*, pages 44–51, Nov 2001.

[62] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5), May 2006.

[63] E. A. Lee, C. Hylands, J. Janneck, J. D. II, J. Liu, X. Liu, S. Neuendorffer, S. S. M. Stewart, K. Vissers, and P. Whitaker. Overview of the Ptolemy Project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001.

[64] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of ACM*, 20(1):46–61, January 1973.

[65] G. Madl and S. Abdelwahed. Model-based Analysis of Distributed Real-time Embedded System Composition. In *Proceedings of EMSOFT*, 2005.

[66] G. Madl, S. Abdelwahed, and G. Karsai. Automatic Verification of Component-Based Real-Time CORBA Applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, pages 231–240, 2004.

[67] G. Madl, S. Abdelwahed, and D. C. Schmidt. Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking. *Real-Time Systems*, 33:77–100, Jul 2006.

[68] G. Madl and N. Dutt. Domain-specific Modeling of Power Aware Distributed Real-time Embedded Systems. In *Proceedings of the 6th Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2006.

[69] G. Madl and N. Dutt. Tutorial for the Open-source DREAM Tool. In *CECS Technical Report*, 2006.

[70] G. Madl, N. Dutt, and S. Abdelwahed. Performance Estimation of Distributed Real-time Embedded Systems by Discrete Event Simulations. In *Proceedings of EMSOFT*, 2007.

[71] G. Madl, N. Dutt, and S. Abdelwahed. A Conservative Approximation Method for the Verification of Preemptive Scheduling using Timed Automata. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 255–264, 2009.

[72] G. Madl, S. Pasricha, N. Dutt, and S. Abdelwahed. Cross-abstraction Functional Verification and Performance Analysis of Chip Multiprocessor Designs. *IEEE Transactions on Industrial Informatics, Special Section on Real-time and (Networked) Embedded Systems (submitted for publication)*, 2009.

[73] G. Madl, S. Pasricha, Q. Zhu, L. A. D. Bathen, and N. Dutt. Formal Performance Evaluation of AMBA-based System-on-Chip Designs. In *Proceedings of EMSOFT*, pages 311–320, 2006.

[74] G. Madl, S. Pasricha, Q. Zhu, L. A. D. Bathen, and N. Dutt. Combining Transaction-level Simulations and Model Checking for MPSoC Verification and Performance Evaluation. *ACM Transactions on Design Automation of Electronic Systems (submitted for publication)*, 2009.

[75] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[76] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, June 2000.

[77] J. McManis and P. Varaiya. Suspension Automata: A Decidable Class of Hybrid Automata. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV)*, pages 105–117, 1994.

[78] Object Management Group. CORBA Component Model, 2002.

[79] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/02-08-02 edition, Aug. 2002.

[80] OSCI. SystemC ver 2.1 (IEEE 1666 Standard), 2005.

[81] S. Pasricha. Transaction Level Modeling of SoC with SystemC 2.0. In *Synopsys User Group Conference (SNUG)*, May 2002.

[82] S. Pasricha and N. Dutt. *On-chip Communication Architectures: System on Chip Interconnect*. Morgan Kauffman, 2008.

[83] S. Pasricha, N. Dutt, and M. Ben-Romdhane. BMSYN: Bus Matrix Communication Architecture Synthesis for MPSoC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, (TCAD)*, 26(8):1454–1464, 2007.

[84] S. Pasricha, N. Dutt, and M. Ben-Romdhane. Fast Exploration of Bus-based Communication Architectures at the CCATB Abstraction. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–32, 2008.

[85] Rafik Henia and Arne Hamann and Marek Jersak and Razvan Racu and Kai Richter and Rolf Ernst. System Level Performance Analysis - the SymTA/S Approach. *IEE Proceedings on Computers and Digital Techniques*, 152:148–166, 2005.

[86] K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MpSoC Performance Verification. *IEEE Computer*, 36:60–67, April 2003.

[87] W. Roll. Towards Model-Based and CCM-Based Applications for Real-Time Systems. In *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 75–82, 2003.

[88] A. Roychoudhury, T. Mitra, and S. R. Karri. Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Design, Automation and Test in Europe (DATE)*, pages 828–833, 2003.

[89] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation*. World Scientific Publishing Co., Inc., 1997.

[90] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar. A High-Performance Endsystem Architecture for Real-Time CORBA. *IEEE Communications Magazine*, 14(2), 1997.

[91] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, 2000.

[92] J. Stankovic, R. Zhu, R. Poornalingham, C. Lu, Z. Yu, M. Humphrey, and B. Ellis. VEST: An Aspect-based Composition Tool for Real-time Systems. In *Proceedings of the IEEE Real-time Applications Symposium (RTAS)*, pages 58–69, 2003.

[93] K. W. Susanto and T. F. Melham. An AMBA-ARM7 Formal Verification Platform. In *International Conference of Formal Engineering Methods (ICFEM)*, pages 48–67, 2003.

[94] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, pages 110–112, Apr. 1997.

[95] D. Taubman. High performance scalable image compression with EBCOT. *IEEE Transactions on Image Processing*, 9:1158 – 1170, July 2000.

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

[97] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44:205–227, 2002.

[98] Venkita Subramonian and Christopher Gill and César Sánchez and Henny B. Sipma. Reusable Models for Timing and Liveness Analysis of Middleware for Distributed Real-Time Embedded Systems. In *Proceedings of EMSOFT*, pages 252–261, 2006.

[99] S. Vestal. MetaH User's Manual, Version 1.27. Technical report, Honeywell Technology Center, 1998.

[100] S. Vestal. Formal Verification of the MetaH Executive Using Linear Hybrid Automata. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium*, pages 134–144, 2000.

[101] J. R. W. Muller and W. Rosenstiel. *SystemC Methodologies and Applications*. Kluwer Academic Publishers, 2003.

[102] S. Wang and K. Shin. Task Construction for Model-Based Design of Embedded Control Software. *IEEE Transactions on Software Engineering*, 32(4):254–264, 2006.

[103] W. M. Wonham. *Supervisory Control of Discrete-Event Systems*. Monograph, 2008.

[104] T.-Y. Yen and W. Wolf. Performance Estimation for Real-Time Distributed Embedded Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, 1998.

[105] Y. Zhao, J. Liu, and E. A. Lee. A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 259–268, 2007.