ABSTRACT

Techniques and Tools for Model-Based Design and Analysis of Embedded Real-Time Software

by

Zonghua Gu

Chair: Kang G. Shin

As Embedded Real-Time (ERT) systems become more complex and safety-critical, there is a trend to raise software development level of abstraction from programming languages to models. We have developed a suite of techniques and tools to improve industry acceptance of model-driven development of ERT software.

As a result of collaboration among multiple institutions, an end-to-end tool-chain has been developed for the design and analysis of ERT software, with Avionics Mission Computing (AMC) as the main target application. As part of the tool-chain, we have developed a tool called AIRES for model-level static analysis. Compared to traditional static analysis techniques that work at the level of programming languages, AIRES works at a higher level of abstraction, and provides valuable dependency and timing information to the engineer at an early stage of the design cycle.

AIRES mainly focuses on the static structural aspects while largely ignoring the dynamic behavior of component interactions. We use model-checking to formalize the natural language description of the dynamic behavior of the AMC software, and verify safety and liveness properties. We also present several techniques to improve scalability of model-checking by exploiting application-level domain semantics.

To bridge the gap between logical models and implementation on the physical execution platform, many UML tools come with automatic code generators that translate models into code in a programming language. However, current code generation technology generates functional code without considering non-functional and real-time issues. We have adapted the schedulability analysis algorithm by Harbour, Klein and Lehoczky to fit the native runtime model of UML-RT, a UML profile widely used in the telecom domain. This algorithm can be used during state-space exploration to synthesize an implementation architecture for a logical UML-RT model that satisfies timing constraints.

In summary, the techniques and tools developed in this thesis address multiple aspects of model-driven development of ERT software, in order to shift the focus of the software development process from programming language-level to the model-level, and reduce the overall system development cost.

Techniques and Tools for Model-Based Design and Analysis of Embedded Real-Time Software

by

Zonghua Gu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2004

To my dear parents

ACKNOWLEDGMENTS

First, I would like to express my gratitude and indebtedness to my mentor and advisor, Professor Kang G. Shin. I would never have come this far on this long and hard Ph.D. journey without his guidance and support, for which I am deeply grateful. I learned a lot more than real-time computing from Professor Shin during the past six years; in fact, I learned lessons of life that would benefit me for the rest of my life.

I would like to thank Professor Kevin Compton, Professor John Hayes and Professor Anna Stefanopoulou for serving on my thesis committee, and providing valuable comments on my thesis draft.

I owe not only this thesis, but also my entire life, to my mother Wang Huaiyu and father Gu Xuebin, who have dedicated themselves completely to their children, working hard all their life up to this day. I would also like to thank my sister Gu Zhonghui for her unconditional love and support.

I would like to thank my good friends Chen Zhigang, Wang Shige, Sun Wei, Dr. Wang Haining and Dr. Qiao Daji for the happy times we spent together in Ann Arbor; Dr. Dan Kiskis and Mohamed El-Gendy for attending the practice talk for my final defense, and providing valuable comments on my presentation material; B.J. Monaghan and Kirsten Knecht for their excellent administrative support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER I

Introduction

1.1 Embedded Real-Time Systems

What is an *embedded system*? Here is a definition from an online encyclopedia[103]:

> A specialized computer system that is part of a larger system or machine. Typically, an embedded system is housed on a single microprocessor board with the programs stored in ROM. Virtually all appliances that have a digital interface – watches, microwaves, VCRs, cars – utilize embedded systems. Some embedded systems include an operating system, but many are so specialized that the entire logic can be implemented as a single program.

Generally speaking, an embedded system is a device that you buy *not* for the reason that it has a computer chip in it. Instead, you buy it for other purposes, but it just happens to contain a micro-processor that controls its operation. Examples are cars, microwave ovens, medical devices, etc. Moore's law, dictating that processor speed doubles every 1.5 years, has held true for the past twenty years or so, and has resulted in faster processors at lower prices. As a result, micro-processors are put in more systems that were not traditionally considered computers, and are taking over more aspects of our daily lives. *Embedded software* is the software running on the computer that performs the core functions of the device. For example, the engine control software running in the car controls the air-fuel ratio into the combustion engine by accurately timing the duration that the fuel intake valve is open, which is in turn controlled by the user by pressing on the gas pedal. Embedded systems typically have the following characteristics:

- They are computational systems that are not first-and-foremost a computer.
- They are integral with physical processes, interacting with them through sensors and actuators.
- They are reactive, i.e., they have to react to the physical environment at its speed in real-time.
- They are heterogeneous, often with a mixture of software and hardware architectures.
- They are networked, so network delays must be considered in system analysis.

Embedded software development is a highly multi-disciplinary process. Depending on the application domain, it may involve control engineers, mechanical engineers, aerospace engineers, system engineers, as well as software engineers. Due to the ubiquity and safety-critical nature of embedded systems, it is crucial to have assurance on the correctness of software, both *functional correctness* (the software computes the right result), and *timing correctness* (the software computes the result at the right time). A *real-time system* is a system whose correctness depends not only on logical correctness, but also on the timeliness of its actions. It must produce a correct result within a deadline. Results produced too late (or sometimes too early) are considered useless or even harmful, even if it is logically correct. There are two types of real-time systems. In *hard real-time* systems, no deadline can be missed; otherwise catastrophe may occur. Hard real-time systems are often *safety-critical*, which means that failure may result in serious injury to human life or property. Examples are avionics control systems, automotive drive-by-wire systems, industrial automation systems, etc. In *soft real-time* systems, deadlines can be occasionally missed without serious consequences. Examples are multimedia transmission and reception systems, where deadline misses result in degraded user-perceived quality, but not more serious consequences. In *non-real-time* systems, only logical correctness matters while timing correctness does not. Examples are database systems, or a word processing application running on a desktop computer. Almost all systems have at least some kind of soft real-time constraints, or sometimes called *Quality of Service* (QoS) or *performance* constraints. A database that takes five minutes to process each query is obviously not useful. A typical QoS constraint may be defined as: 95% of all database queries must be processed within 1 second, and the maximum response time is less than 10 seconds. Embedded systems are often also real-time systems. Some are hard real-time, as with the engine control example, where it is imperative that the controller performs its function before a deadline in order for the car to be responsive to the driver's commands for safety reasons. Some are soft real-time, for example Personal Digital Assistants (PDA) and cell phones, where missed deadlines may result in user-perceived QoS degradation. We are mainly concerned with hard real-time systems in this thesis, e.g., Avionics Mission Computing (AMC) [80] and automotive control systems such as engine control and vehicle-to-vehicle adaptive cruise control [98].

A real-time system does not have to have a computer chip in it. It may be a pure mechanical system with timeliness constraints. It is a relatively recent phenomenon that computer chips are getting embedded into everyday products surrounding us. For example, a mechanical automotive brake system is a real-time system. When the user presses on the brakes, he expects immediate braking action. But no micro-controllers are involved.

A recent trend in automotive control is *x-by-wire* systems, where x stands for *brake*, *drive*, etc., where micro-controllers are used to control safety-critical functions without mechanical backup. The control algorithms can be implemented in hardware or software. We are mainly concerned with embedded systems implemented in software, often involving a real-time operating system and/or real-time networks.

1.2 Model-Driven Development of Embedded Software

It is more difficult to develop embedded software than desktop software, due mainly to resource constraints of the embedded micro-processor, which often lags a few generations behind and is much slower than state-of-the-art desktop processors for reasons of cost, energy consumption and reliability. Older, slower chips cost less, consume less energy (because they run at a slower clock speed), and are more reliable than newer, faster chips (because they have been tested more extensively in the field). Twenty years ago, embedded software engineers were preoccupied with optimization and fine-tuning of every line of code to achieve the highest running speed while fitting the software into the available memory space of the resource-constrained embedded processor. Typically, but not always, there is an inverse relationship between execution speed and code size, and it is up to the engineer to make design tradeoff decisions between them. Embedded software development was viewed as a niche area of software engineering, where many good software engineering principles and practices, such as object-orientation, modularity and layers of abstraction, were sacrificed in pursuit of the best performance possible. Embedded software engineers were viewed as hackers who were experts in writing arcane assembly language programs that are next to impossible to understand and maintain by anyone other than the author of the code.

However, there have been major changes in the way embedded systems are developed in recent years. As processors are becoming faster and cheaper every year as dictated by Moore's law, it becomes more affordable to use micro-processors with relatively high-performance within embedded systems. For example, the CPU contained within a modern cellular phone has more computation power than the first ENIAC computer built in 1946. Costs of computer memory has also been dropping fast. As a result, the major concern of embedded systems engineers has shifted from optimizing for performance to coping with complexity. Figure 1.1 shows the Volvo S-80, as an example to illustrate the degree of complexity reached in modern automobiles.

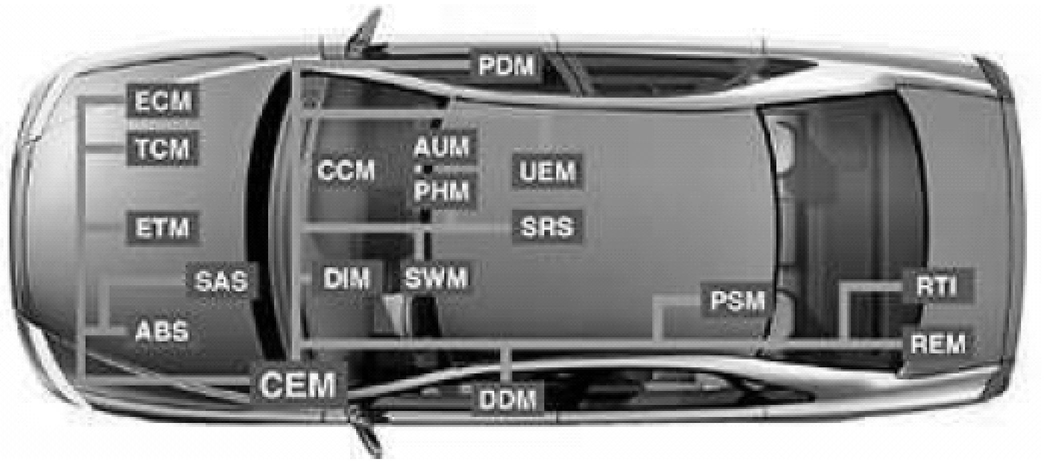


Figure 1.1: The Volvo S-80, with two CAN bus segments connected through gateways. There are 18 control units on the buses, and the CAN bus loads are around 60%. This figure is taken from [96].

As Embedded Real-Time (ERT) systems become more complex and mission- or safety-critical, the traditional development process of manual coding followed by extensive and lengthy testing is becoming inadequate. The overarching concern for an embedded system developer is no longer to optimize software at very low levels in order to squeeze every ounce of performance out of it,[1] but to ensure high-level system correctness, modularity and maintainability at the expense of some performance loss. Engineers are willing

[1]This may still be true for certain domains such as digital signal processing for mass-produced consumer products, where performance optimizations can result in large savings in hardware costs.

to adopt good software engineering principles in exchange for some performance loss as long as it is within an acceptable range. Embedded software engineering has moved from a niche area to a vibrant research area, with several international conferences and journals devoted to it.

In order to increase developer productivity, the abstraction level for software development has been raised from assembly language to modern programming languages such as C/C++ and Java. There is a recent trend to raise the level of abstraction further to the *model-level*, and rely on automatic or semi-automatic code generators to produce code in a traditional programming language. Examples of this approach include Unified Modeling Language (UML) [91], Model-Driven Architecture (MDA) [90], and Model-Integrated Computing (MIC) [88]. In particular, the MIC approach advocates using *domain-specific* models throughout the engineering process that allow both system analysis (to determine the overall characteristics of the system) and synthesis (to generate configuration or functional code for the system). The term *Model-Driven Development* (MDD) is commonly used as a general term for relying on models in the software development process. The benefits of adopting MDD are:

- Models are at a higher level of abstraction than programming languages, and therefore engineer productivity is typically higher when developing models as compared to developing code,
- Formal models can be analyzed and validated to prove their correctness,
- Real-time implementations can be synthesized from models.

In order to illustrate the MDD approach to embedded software development, let's first visit the definition of models. The Marriam-Webster dictionary provides several definitions, the most relevant of which is:

A usually miniature representation of something; also, a pattern of something to be made.

We can find a lot of models in the toy store, such as model trains, fire-trucks, houses, action figures, dolls, etc. They fit the dictionary definition perfectly: miniature representation of the real thing. Models do not have to be three-dimensional; they can be purely conceptual, and visualized on a computer, or drawn on a napkin. In fact the most common form of models in software engineering are box-and-arrow diagrams describing software architectures such as client-server, three-tier, peer-to-peer, etc. The purpose of developing a model is to use it as a blueprint to build the real thing. An architect draws on paper the structure of a building, i.e., a model, and hands it off to the building contractor, who then uses bricks and concrete to build it. A civil engineer develops a mathematical model of a bridge in the form of differential equations, and analyzes it to make sure that it satisfies certain requirements such as stability, durability, etc., before handing it off to the bridge contractor to build it.

Models for software have a distinct advantage over models for other domains like civil engineering. If models are formal enough, we can directly transform from models into the final product (executable code), since both are fundamentally bit pattern representations in the computer. This changes the job of a software engineer from writing code to building models.

Figure 1.2 shows the typical waterfall system development life-cycle. We discuss it in the context of automotive embedded control systems. First, the control engineers identify system requirements, e.g., the cruise control system must maintain the car speed within a certain range of deviation from the set-point chosen by the driver, and design a control law that fulfills the requirements using tools such as Matlab/Simulink [65]. They then hand over the control law to the software engineers, who then choose a target embedded

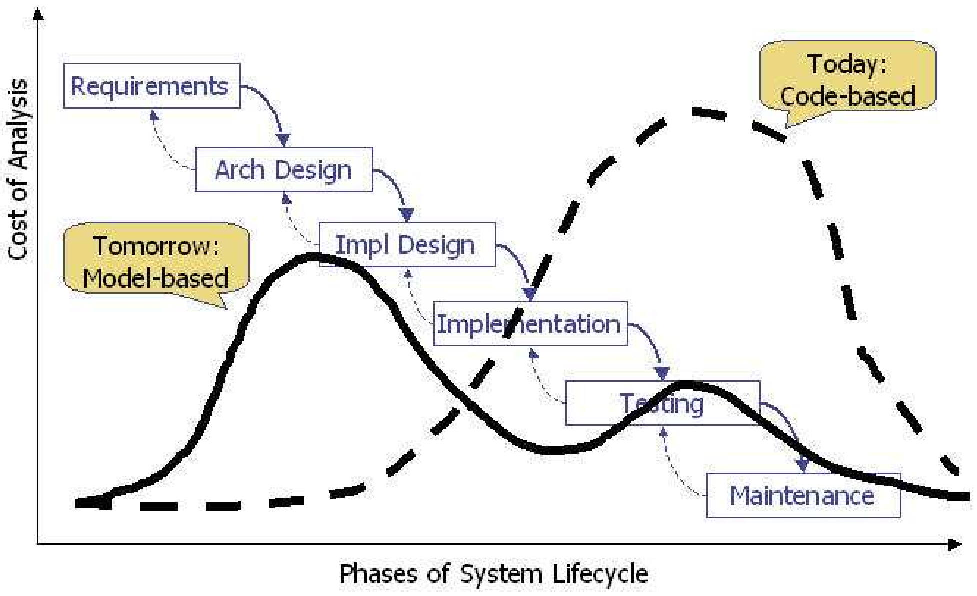


Figure 1.2: System development life-cycle and analysis costs.

platform, and write code to implement the control algorithms. After a series of stages, from system design, to module design, to implementation, to module test, to system integration, the software engineers hand over a running system to the test and calibration engineers, who then perform extensive testing and calibration of the entire system to see if the vehicle performs as expected. The final stage is maintenance after the system delivery. Current industry practice relies heavily on the testing stage at the code-level to verify timing constraints. Timing problems are identified by detecting deadline misses on the target micro-controller. Unfortunately, problems detected at such late stages typically result in system redesigns and are very costly both in terms of time and money. What we would like to do, is to move the analysis to an earlier design stage, and perform model-based analysis. This will reduce the efforts required at the testing stage, and reduce the overcall cost of software development

Model-Driven Development (MDD) has met with great success in the enterprise software domain, but has been accepted less in the embedded systems domain. Even though many MDD tools have been created or customized for real-time embedded software, the current state-of-the-practice in industry still largely relies on low-level programming using C or even assembly. Even object-oriented programming languages such as C++ and Java have not been widely adopted except in larger embedded systems where per-unit hardware cost is of less concern than software development cost. The DARPA Model-Based Integration of Embedded Software (MoBIES) program, started in 2000, has been exploring model-based approaches for embedded software composition and analysis, especially emphasizing non-functional issues such as timing, synchronization, dependability and resource constraints. The goal of the MoBIES program is to develop mathematical models and interface standards to integrate physical descriptions of application domains, high-level specifications of program functionality and process models for software tools. With the MoBIES technology, the embedded software engineer can construct a domain-specific embedded software design framework by assembling different tools. These tools can be developed using different modeling languages specialized for different design concerns and different modeling aspects of the target embedded software. Therefore, various cross-cutting issues in embedded software design and analysis, including real-time scheduling, size/weight/power limitations, fault-tolerance, safety and security, can be addressed by tools in the integrated tool-chain.

As part of the MoBIES program, in addition to *Technology Developers* (TD) from various universities, several industrial partners have been involved to provide *Open Experimental Platforms* (OEP), application examples used for demonstration and validation of technology developed by the technology developers. Throughout this thesis, we have attempted to use such OEPs whenever possible, in particular, the *Avionics Mission Computing* (AMC) [80] software provided by our industrial partner Boeing. However, other application examples are also used, such as the Train-Gate-Controller [41] example and the Elevator Control [28] example, in order to better illustrate certain concepts.

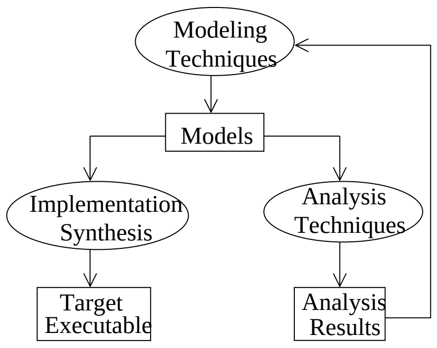


Figure 1.3: The Model-Driven Development process.

There are three main steps to the Model-Driven Development (MDD) process, as shown in Figure 1.3: first, we need modeling techniques to build models, which serve as the central artifact of the development process. Once the models are built, we need model analysis techniques to prove or disprove correctness of the models. If analysis results reveal any problems, e.g., the system does not satisfy certain functional or real-time correctness criteria, it is necessary to use feedback from analysis results to redesign the models. After we are certain that the models are correct, we need implementation techniques to generate runtime executable that can be downloaded and run on the target platform.

Current industry practice is still quite far from the visions of MDD. Models such as the Unified Modeling Language (UML) [91] often serve in a documentation role that the engineer can refer to while performing manual coding. Therefore, the link between model and code is weak and easily broken in the process of system maintenance and evolution, when code is modified or enhanced without the corresponding change at the model-level, or vice versa. Engineers almost never do any model analysis at an early design stage. Instead, they heavily rely on the testing stage for verification, which often incurs a high cost, as shown in Figure 1.2. We need a more automated and integrated development process than the current industry practice. In this thesis, we present a suite of techniques and tools that aim to ease industry adoption of MDD by addressing the issues of model analysis and implementation synthesis.

It is important to have tools that are both usable by average engineers and scalable to larger, realistically sized systems. Some formal languages, such as Z and VDM, were designed to makes software specifications mathematically-precise and instill rigor into the software development process. But they have largely fallen into oblivion except in some highly specialized areas, e.g., systems with very high safety and confidence requirements, because they are based on mathematical logic, and carry a steep learning curve before they can be used effectively. Therefore, we have placed a lot of emphasis on building user-friendly and scalable tools that encapsulate technical details and are usable by average engineers.

1.3 Main Contributions

This thesis makes contributions in three related topics of model-driven development, namely, *model-level static analysis*, *applications of model-checking*, and *implementation synthesis from UML-RT*. In the context of Figure 1.3, the first two topics fall into the general category of *analysis techniques*, and the last area falls into the category of *implementation synthesis* techniques.

1.3.1 Model-Level Static Analysis

Traditional static analysis techniques work at the level of programming languages, and study control and data dependency relationships associated with functions and variables. Control dependency refers to flow of control through a sequential program, and data dependency refers to the locations of definitions and uses of the program variables. When the level of abstraction is raised from programming languages to models, it is necessary to perform *model-level* static analysis; that is, to study dependency relationships associated with components and ports, which often involve real-time, concurrency and distribution. In Chapter III, we describe our tool called AIRES for model-level static analysis, developed as part of the MoBIES tool-chain[2], which is an end-to-end tool-chain for model-based design, analysis and code generation for Avionics Mission Computing (AMC) software. The main functionalities of AIRES include model-level dependency visualization and anomaly detection, forward/backward slicing of dependency graphs, execution rate assignment, timing analysis and component allocation. All the algorithms implemented in AIRES are of polynomial complexity and scalable to large, realistically-sized systems. AIRES provides valuable dependency and timing information at the model and architectural-level to the engineer at an early design stage, which used to be buried in the source code underneath layers of abstraction and not readily accessible to the engineer. For example, the engineer had to use a debugger to trace through hundreds of thousands of lines of code to track down a cyclic dependency bug. AIRES has been evaluated by our industrial partner with the Goal-Quality-Metric [77] methodology, and received positive feedback.

1.3.2 Applications of Model-Checking

Formal verification techniques are often employed to verify safety-critical applications for correctness assurance. *Model-checking* [3] is one of the formal verification techniques that has gained tremendous popularity in recent years, especially for hardware and protocol verification. In simple terms, model-checking works by exhaustively searching through the system state space to look for “bad” states that cause violation of any system correctness specifications. There are two types of model-checkers. Untimed model-checkers focus on verification of *functional correctness*, with examples such as SMV [66], Spin [43] and La-

[2]The MoBIES tool-chain was developed collaboratively by the University of Michigan, Vanderbilt University and Southwest Research Institute, as well as engineers from Boeing.

belled Transition Systems Analyzer (LTSA) [61]. Real-time and hybrid model-checkers focus on verification of real-time and hybrid systems for their *timing correctness,* with examples such as UPPAAL [8] and HyTech [42]. We discuss applications of both types of model-checkers, using an untimed model-checker to verify functional properties of embedded *software*, and a timed model-checker to verify properties of embedded *system*, which typically consists of embedded software interacting with embedding physical environment.

AIRES mainly focuses on the static structural aspects while largely ignoring the dynamics of component interaction. In Chapter IV, we use the untimed model-checker LTSA [61] to construct a formal model of the AMC software and check for safety and liveness properties. The AMC software is component-based with many different types of components, each with its unique functionality and interfaces, acting as basic building blocks of a complete system. The documentation from our industrial partner provides descriptions of the dynamic behavior of various component types in natural language, which is not formal and subject to misinterpretation and misunderstanding. Formalization with Labelled Transition Systems Analyzer (LTSA) gives a formal and unambiguous description of system behavior, and enables application of model-checking to prove or disprove certain properties related to system behavioral dynamics, which are not possible with static analysis alone. One of the major impediments to industry adoption of model-checking is lack of scalability, since it involves exhaustive exploration of the system state space, which grows exponentially to an unmanageable size for realistically-sized systems. This is commonly called “state-space explosion”. We discuss several techniques to alleviate the state-space explosion problem and improve scalability by exploiting application-level domain semantics.

An ERT system typically consists of embedded software and embedding physical environment with tight coupling and interaction in between. Current work on real-time verification typically focuses on either one or the other, but not both. To extend the scope of modeling and analysis from *software* to *system* level, we propose an integrated approach in Chapter V to modeling and analysis of ERT systems where the embedded software and the embedding physical environment are modeled and analyzed within the same modeling formalism. By adopting this approach, the engineer can have an integrated view of the entire system when making design decisions, and perform optimization operations such as maximizing total system utility given resource constraints, or minimizing total system cost given application requirements.

For this purpose, the modeling languages of the untimed model-checkers such as LTSA are not adequate, since they are only capable of modeling the software part, not the physical environment, which is inherently continuous and real-time. Therefore, we need real-time or hybrid modeling formalisms and model-checkers to construct a system-level model. We use Time Petri-Nets (TPN) [67] as the modeling formalism to illustrate this methodology. We also describe an automated translation procedure from TPN models into semantically-equivalent Timed Automata (TA) [8] models, thus enabling the use of mature model checkers for TA such as UPPAAL [8] for analysis of TPN models. Even though TPN is used for illustration purposes, the concept of integrated modeling and analysis is general and applies to other real-time and hybrid modeling formalisms such as Hybrid Automata [42].

1.3.3 Implementation Synthesis from UML-RT

In Chapter VI, we consider the problem of generating a real-time multi-threaded implementation from UML-RT models. UML-RT [78] is a UML profile that describes the system architecture in terms of components, ports and connectors. Despite the word "real-time" in its name, UML-RT does not have explicit constructs for expressing timing and schedulability constraints. Commercial code generators for UML-RT generate functional code in programming language code, but largely ignore real-time issues, resulting in a gap between model and real-time implementation. The native runtime model of UML-RT assigns one or more active objects to each OS thread running at a fixed priority. This runtime model does not fit the task model of the classic Rate Monotonic Analysis (RMA) [53] technique. We have developed schedulability analysis algorithms for this runtime model based on the Harbour, Klein, Lehoczky (HKL) algorithm [38] with added blocking time caused by the Run-To-Completion (RTC) semantics of UML-RT. This algorithm can be used during state-space exploration to find a suitable implementation architecture for a logical UML-RT model.

In summary, the techniques and tools developed in this thesis address multiple aspects of the model-driven development process for ERT software in order to shift the focus of the software development process from programming language-level to the model-level, and reduce the overall system development cost.

1.4 Related Work

There has been a lot of work in the programming languages and compilers area on static analysis techniques, such as control and data dependency analysis and program slicing [44], used for program optimization, program understanding and reverse engineering. This reflects the fact that most of the software development activities have centered at the programming languages level. As the MDD approach gains more ground, we should develop similar static analysis techniques that work at the abstraction level of models and architecture. This is actually a simpler job than analyzing programming languages, which are highly expressive with complicated control-flow structures using *while* and *for* loops, or even *goto* statements. Dependency relationships at the model-level are typically much simpler and more intuitive, due to the graphical nature of models, and (generally) simpler control flow constructs. However, static analysis at model-level potentially has a much larger impact than static analysis at the programming language level, since it happens at an earlier design stage.

Previous work on software timing analysis has centered around analyzing a sequential software program to find out its Worst-Case Execution Time (WCET) using control and data flow dependency information extracted from static analysis at the programming language level. Once we acquire the WCET information for software components, there are many important system-level issues to be addressed, such as multi-threading, concurrency, component and task allocation, and execution platform selection, in order to achieve system-level performance objectives such as schedulability, load-balancing, fault-tolerance, etc. These issues are visible at the model and architecture level, and necessitates techniques and tools that work at a higher level of abstraction than WCET tools for programming languages.

A number of CASE tools for model-driven development of embedded software have been developed in recent years. Examples include Virginia Embedded Systems Toolkit (VEST) [85] from University of Virginia, TimeWeaver [17] from Carnegie Mellon University, Cadena [40] from Kansas State University, MetaH [9] from Honeywell Technologies, Ptolemy [12] and Giotto [52] from University of California, Berkeley, etc. Each of these tools has a different focus on the various aspects of embedded software development. For example, VEST's focus is on *prescriptive aspects*, which differ from Aspect-Oriented Programming by being prescriptive instead of descriptive, and are applied at the early design stage instead of the programming stage. MetaH is an Architectural Description Language and toolset for developing real-time, fault-tolerant, securely-partitioned, multi-processor software in the avionics domain. It is based on the programming language Ada, and uses a custom-generated runtime executive instead of Commercial Off-The-Shelf (COTS) middleware like CORBA in AMC. The focus of the MoBIES tool-chain is on providing an integrated toolset with open architecture and standard interfaces to facilitate easy plug-in of third-party tools, and the focus of AIRES is on model-level dependency and timing analysis to provide the engineer insight into dependency and real-time information at an early design stage.

The publish/subscribe model of computation, as implemented in Real-Time CORBA [76] and used in AMC, has been widely adopted in a variety of application domains, including both ERT and enterprise distributed systems. Garlan [25] describes a model-checking framework for publish/subscribe systems. The key feature of this framework is a reusable, parameterized state machine model that captures runtime event management and dispatch policy. Generation of models for specific systems is then handled by a translation tool that accepts as input a set of component descriptions together with a set of properties, and maps them into the framework where they can be checked using the model-checker SMV [66]. Our modeling approach works at a higher level of abstraction and ignores details related to the internals of the CORBA middleware such as queuing and dispatch policies. For example, we use a single synchronization primitive to represent the flow of an event from the publisher component to the subscriber component. Even though it is a coarse approximation of the actual behavior, it is adequate for our purpose of verifying application-level logical properties, given the assumption that the middleware behaves correctly. This significantly reduces system state space and allows us to check for much larger models than the case of modeling the system at a more detailed level.

Research in real-time and hybrid systems initially focused on modeling of physical systems, and was later extended to deal with software dynamics such as CPU and network scheduling, e.g., the TIMES tool [5] is designed to model and analyze real-time task sets that do not conform to the assumptions of traditional schedulability analysis techniques, e.g., a task set with data-dependent triggering patterns. However, it does not have the concept of integrated modeling and analysis. Some authors have recognized the significance of tight integration of embedded software with its physical environment, and the need for an integrated analysis framework. For example, Seto [79] proposed an integrated approach to controller design and task scheduling, where task frequencies are allowed to vary within a certain range as long as such a change does not affect critical control functions such as maintenance of system stability. An algorithm was developed to optimize the overall system control performance while maintaining schedulability by adjusting task frequencies. Our work is complementary to Seto's since we focus on the model-checking technique instead of analytical derivation.

Most commercial UML tools have the capability to generate code in programming languages such as C/C++ or Java. However, they focus on functional issues, and largely ignore non-functional and real-time issues such as multi-threading, concurrency and schedulability. A number of implementation alternatives for UML-RT have been proposed, for example, by Saksena [75] and Kim [64], for mapping a logical UML-RT model into a multi-threaded real-time executable. Their approaches associate priorities with end-to-end application scenarios instead of capsules, and treat each scenario as a task, so that classic RMA [53] techniques can be applied to analyze schedulability. However, they both have some shortcomings. Saksena's approach requires the engineer to stick to a programming discipline of dynamically adjusting capsule priorities to reflect the priority of the currently executing end-to-end transaction. This approach hurts the encapsulation of capsules by mixing system-level concerns (scenarios) with component-level concerns (capsules). It also involves runtime system-call overheads that may or may not be acceptable to certain resource-constrained embedded systems. Certain small RTOSes may not even provide APIs for dynamic priority change. Kim's approach creates shared data when multiple scenarios cut through the same capsule, and breaks a key advantage of UML-RT, which eliminates the need for error-prone concurrency control mechanisms (mutexes, semaphores and monitors) by using asynchronous message passing as the main communication mechanism among capsules instead of shared data.

Both Saksena's and Kim's approaches require modifications to the native runtime model of UML-RT as implemented in the RoseRT CASE tool [46]. Even if Saksena's or Kim's approaches were widely adopted, there would still be a lot of legacy applications that will never be changed. Therefore, instead of modifying the computational model of design tools to fit scheduling algorithms, we modified scheduling algorithms to fit the native runtime model of design tools. This should make our approach more easily acceptable to industry.

The thesis is structured as follows: in Chapter II, we provide some background knowledge and related work that are necessary to understand the rest of the thesis. Chapter III describes model-level static analysis techniques and the AIRES tool, using Avionics Mission Computing (AMC) as the example application domain. In Chapter IV, we discuss the application of model-checking techniques to verify functional properties of the AMC software, and several techniques for improving scalability of model-checking. In Chapter V, we describe an integrated approach for modeling and analysis of ERT systems with tight coupling between embedded software and embedding physical environment. Chapter VI considers the problem of synthesizing a real-time implementation from logical UML-RT models. Finally, in Chapter VII, we draw some conclusions and discuss possible future work.

CHAPTER II

Background Knowledge and Related Work

In this chapter, we describe background knowledge and related work in order to set the stage for the rest of the thesis. This chapter is structured as follows. Section 2.1 discusses the MIC approach to building domain-specific modeling environments and generators. Section 2.2 provides an introduction to formal verification techniques, focusing on the model-checking approach and two model-checkers used in later chapters: the untimed model-checker LTSA, and the real-time model-checker UPPAAL. Section 2.3 introduces UML and its extensions for embedded real-time systems. Finally, Section 2.4 introduces real-time scheduling theory.

2.1 Model-Integrated Computing

Model-Integrated Computing (MIC) [88] is developed at Vanderbilt University, and embodies the model-based design paradigm: models are used not only to design and represent, but also to synthesize, analyze, integrate, test, and operate embedded systems. Models capture not only what the dynamics and expected properties of the system are, but also what is assumed about the system's environment. Based on the MIC approach, the Generic Modeling Environment (GME) [57] is developed as a configurable toolset for creating domain-specific modeling and program synthesis environments through a *meta-*

model that specifies the modeling paradigm of the application domain. The *meta-model* captures all the syntactic, semantic and presentation information regarding the application domain, and defines the family of models that can be created using the resulting modeling environment. It contains descriptions of the entities, attributes, and relationships that are available in the modeling environment, and the constraints that define what modeling constructs are legal. As an example, Figure 2.1 shows the meta-model for hierarchical finite state machines (HFSM), and Figure 2.2 shows a HFSM model conforming to the meta-model.

As an example of the MIC approach, a chemical engineer trying to design a chemical reaction experiment should ideally have a modeling environment within which he can directly express concepts such as reactants, catalyzers, flow pipes, vessels, cooling liquids, etc., possibly with intuitive graphical icons for each entity. This involves developing a meta-model for the domain at hand, and associating intuitive bitmap images to the domain concepts. This should be more user-friendly than requiring the user to write programs in C++ and build the domain abstractions manually. We can draw another analogy with an example from computer science. Turing Machines are general-purpose and can be used to express and simulate any sequential computer algorithm, but no one uses it as a programming language, because it is very inefficient and verbose to encode an algorithm with Turing Machines. Instead, engineers use programming languages such as C, with its built-in constructs for expressing iterative loops and numerical computation, to write complex algorithms succinctly. Just as a programming language like C makes it easier for programmers to implement computer algorithms, a domain-specific modeling environment makes it easier for engineers to build models in their respective domains.

Model transformation is defined as transforming a model conforming to meta-model *A* to another model conforming to a different meta-model *B*. It is often termed *semantic*

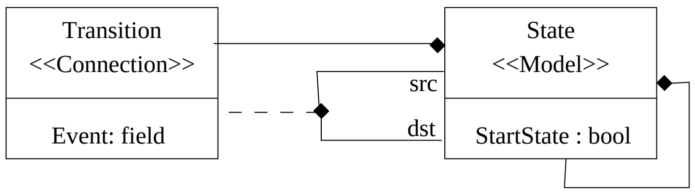


Figure 2.1: The meta-model for hierarchical finite state machines (HFSM). The source state can be connected to the destination state with a transition, and each state can contain other states or transitions to form a hierarchy.

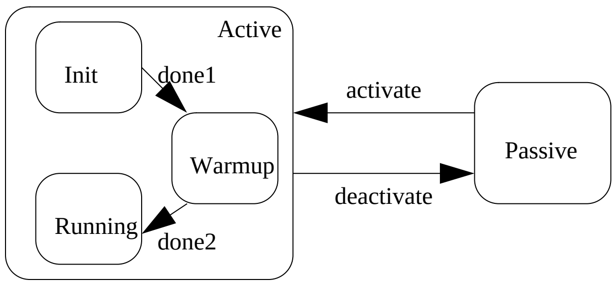


Figure 2.2: An example HFSM conforming to the meta-model in Figure 2.1. Each transition is labeled with the event that triggers it.

translation since it changes model semantics in the process of transformation, which is generally a more difficult problem than *syntactic translation* such as file format conversion from postscript to PDF, where document syntax is changed while semantics remains unchanged. One example transformation is flattening the hierarchy of a HFSM and generate an equivalent flat FSM, possibly with a much larger number of states than the original HFSM. Another example is the classic problem of generating a deterministic finite automaton from a non-deterministic finite automaton.

Figure 2.3 shows the relationship between meta-modeling and domain modeling in the MIC approach. The *meta-modeler* first uses a meta-modeling environment to construct a meta-model for the application domain under consideration, e.g., high-performance embedded signal processing. He then synthesizes a domain-specific modeling environment (DSME) based on the meta-model, and hands it off to the *domain modeler*, who then uses the DSME to develop a model instance of application domain under consideration, for ex-

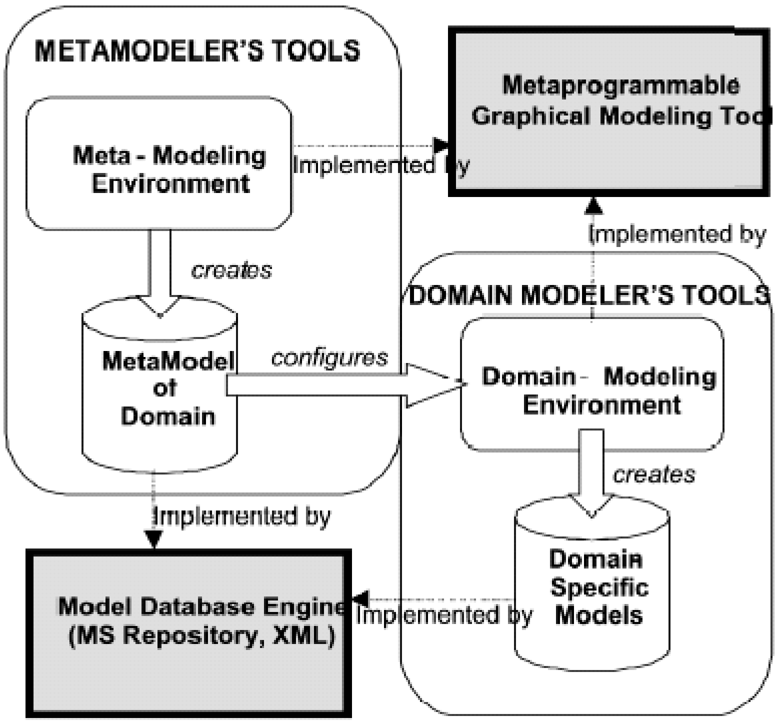


Figure 2.3: Relationship between meta-modeling and domain-modeling in the MIC approach. This figure is taken from [57].

ample, a particular high-performance signal processing system. Both the meta-models and models are stored in the form of XML files to facilitate easy interchange of models with third-party tools. The DSME encodes the syntactical rules specified in the meta-model, and ensures that only legal models conforming to the meta-model are generated. The domain modeler is able to directly use familiar domain abstractions to model the system instead of manually imposing the domain abstractions onto a general purpose modeling language such as UML.

The key factor that distinguishes the model-based approach from other techniques is that models are used as input to *generators*, also called *model interpretors*, that translate them into other artifacts used for analysis or at runtime in the application. The two major applications of generators are:

- To translate models into the input language of analysis tools and to translate the analysis results back into the modeling language, e.g., to generate models used for schedulability analysis from application models.

- To translate models into executable code that can then be compiled and run on a target execution platform.

There are three techniques for developing generators:

- *Direct implementation:* using a set of C++ APIs provided by GME, the designer can write code to manually traverse the data structures contained within the source model, and write out the corresponding portions in the target model, which can be in the form of a model useful for performing analysis, or in the form of a programming language to be compiled and run. This technique is simple and works well for situations where the transformations are easy to capture in a procedural form.
- *Pattern-Based Approach:* use the design pattern *Visitor* [19] to traverse the input model in the form of an abstract syntax tree, and take actions at specific points during the traversal. This approach allows concise and maintainable implementation of generators compared to the direct implementation approach.
- *Metagenerators:* use *Graph Grammars* and *Graph Rewriting* [48] to develop a mathematically precise mapping from input metamodel to output metamodel. This is the most sophisticated approach to developing translators. One drawback is that both the source meta-model and target meta-model must be captured within GME, which is often not the case. For example, the meta-model of a programming language like C++ or Java is very complicated and not easily captured cleanly within GME. In that case it is necessary to resort to the direct implementation approach.

2.2 Formal Verification

As the complexity of safety-critical systems grows, it becomes increasingly important to ensure the correctness of these systems and embedded software controlling them. Building models is the necessary first-step towards higher levels of of assurance, but it is also very important to have effective analysis techniques that can be applied to the models in order to derive useful properties. Some example analysis techniques are real-time schedulability analysis, dependency analysis with graph theoretical methods, etc. If the model is executable, then simulation can be used for verification. In fact simulation is the most widely-used technique in industry today. However, simulation only explores part of the system state space. It is useful for finding bugs and proving that a system's incorrectness, but cannot be used to prove that a system is 100% correct.

There are another family of analysis techniques that are characterized as *formal methods* [27], defined as using mathematical techniques to prove system correctness. There are mainly two types of formal methods:

- *Theorem-proving*: the system is specified in a logical system and deductive verification techniques are applied to prove properties. Examples are the PVS and Larch theorem provers. This approach is mostly manual even with state-of-the-art theorem provers that provide help in proof-checking.

- *Model-checking:* instead of manual deduction like in theorem-proving, the model-checking technique uses exhaustive state-space exploration to prove properties.

2.2.1 Model-Checking

We are mainly concerned with the model-checking technique in this thesis. As shown in Figure 2.4, an automated tool called a *model-checker* is used to check a model M against

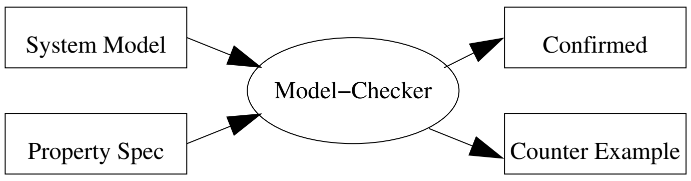


Figure 2.4: The model-checking process.

its expected property ϕ, i.e., to prove or disprove the formula $M \vdash \phi$. The outcome is either a correctness confirmation that property ϕ holds, or a counter example showing the execution trace leading to the property violation if ϕ is not true. For example, a model M of a missile launch system is checked against the expected property ϕ that the missile will not be launched (engines ignited) unless all the release latches are disengaged. If ϕ is true for M, the model checker will confirm its validity. If ϕ is not true, it will show what states or portions of M (a counter example) that will result in the premature launch.

Typically, the system model M is specified in some type of state machine notation, and the model-checker explores the full state-space of M to determine the validity of desired property ϕ. There are two approaches to model-checking:

- *Temporal logic based:* this involves a finite state mechine model of M with a temporal logic representation of ϕ. One representative example of this approach is SMV (Symbolic Model Verifier) [66] from Carnegie Mellon University.

- *Automata based:* this involves automata models for both M and ϕ. One representative example is the model-checker Spin [43].

Model-checking has gained tremendous popularity in recent years, especially in the hardware design and verification area. In simple terms, model-checking works by exhaustively searching through the system state space to look for “bad” states that cause violation of any system correctness specifications. There are two types of model-checkers. One type focuses on verification of *functional correctness*, with examples such as Symbolic

Model Verifier [66] (SMV) from Carnegie Mellon University, Spin [43] from Lucent Bell Labs, and those based on process algebra such as the Labelled Transition Systems Analyzer [61] from Imperial College London. The other type focuses on verification of real-time and hybrid systems for their *timing correctness*, with examples such as UPPAAL [8] and HyTech [42].

2.2.2 The Untimed Model-Checker LTSA

Modeling Language

Finite State Processes (FSP) [61] is a light-weight process algebra developed by Jeff and Magee Kramer, with formal semantics defined with Labelled Transition Systems (LTS). Labelled Transition System Analyzer (LTSA) is a model-checker that performs reachability analysis on FSP models in order to check for safety and liveness properties.

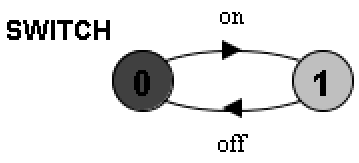


Figure 2.5: A simple FSP model for a light switch.

Primitive components are defined as finite-state processes using action prefix `->`, choice `|` and recursion. If `x` is an action and `P` a process, then `(x->P)` describes a process that initially engages in the action `x` and then behaves exactly as process `P`. Figure 2.5 shows a simple FSP model for a light switch, which toggles between the off and on states:

```
SWITCH = (on->off->SWITCH).
```

We can write an equivalent specification using recursion:

```
SWITCH = OFF, OFF = (on->ON), ON = (off->OFF).
```

If `x` and `y` are actions, then `(x->P|y->Q)` describes a process which initially engages in either of the actions `x` or `y`, and the subsequent behavior is described by `P` or `Q`, respectively. A model for a drinks machine is:

```
DRINKS = (red->coffee->DRINKS|blue->tea->DRINKS).
```

If the red button is pressed, it dispenses a cup of coffee; if the blue button is pressed, it dispenses a cup of tea.

The alphabet of a process is the set of actions it can engage in. The interface operator @ is used to specify the set of action labels which are visible at the interface of the component and thus may be shared with other components. The alphabetic extension operator + is used to specify extension of process alphabet to include a set of action labels.

Primitive processes can be composed with the parallel composition operator —— to form a *composite process*. Processes interact via synchronization on common message labels in the traditional style of process algebra. That is, if processes in a composition have a common shared action, all processes must execute the shared action at the same step. For example:

```
MAKER = (make->ready->MAKER).
USER = (ready->use->USER).
||MAKER_USER = (MAKER||USER).
```

The MAKER process and USER process share a common action `ready`, so they must execute that action synchronously while the other actions can be interleaved.

Property Specification

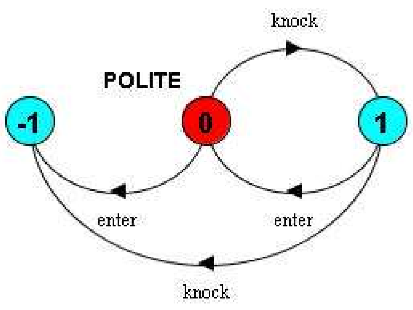


Figure 2.6: The property specification that event *knock* must happen before event *enter*.

A property is an attribute of a program that is true for every execution of that program. Properties of interest for concurrent programs fall into two categories: *safety* and *liveness*. A safety property asserts that nothing bad happens during execution, and a liveness property asserts that something good eventually happens. Example safety properties are mutual exclusion and absence of deadlock; example liveness property is that a process is eventually granted a resource if it makes a request for it.

LTSA does not support property specification in the form of temporal logics such as LTL in Spin and CTL in SMV. Rather, properties are specified using property automata.

A safety **property** P defines a deterministic process that asserts that any trace including actions in the alphabet of P, is accepted by P. Figure 2.6 shows the LTSA process generated from the safety specification:

```
property POLITE = (knock->enter->POLITE).
```

An event `enter` without a preceding `knock`, or two adjacent `knock` events will result in reaching the error state denoted -1. Another common safety property is *deadlock freedom*, which is built into the LTSA model checker and need not be explicitly specified. Instead, there is an explicit menu option for checking deadlocks.

A liveness property **progress** $P = a_1, a_2, \ldots, a_n$ defines a progress property P, which asserts that in an infinite execution of a target system, at least one of the actions $a_1, a_2, \ldots, a_n$ will be executed infinitely often. For example, for a coin toss process

```
COIN = (toss->heads->COIN|toss->tails->COIN)
```

We can specify progress properties

```
progress HEADS = {heads}.
progress TAILS = {tails}.
```

which states that if we toss the coin infinite number of times, we can expect to see both *heads* and *tails* infinitely often. This is verified to be correct given the **fair choice** scheduling policy assumption built into the LTSA tool, which states that *if a choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.*

2.2.3 Timed Automata and UPPAAL

The Real-Time Model-Checker UPPAAL [8] is based on Timed Automata [4]. A timed automaton is a standard finite-state automaton extended with a finite collection of real-valued clocks, which proceed at the same rate and measure the amount of time that has elapsed since they were last reset. The UPPAAL definition of TA has added a few extensions to the standard definition of [4], such as integer variables, CCS-style synchronous communication, urgent channels, etc.

A transition $l \xrightarrow{g,\alpha,\phi} l'$ means that the automaton in state l will perform the α transition instantaneously when the guard g is true and reach state l' while resetting the variables in ϕ to zero. In UPPAAL, both the guard g and the variables to be reset ϕ can contain clock as well as data variables. We use C to denote the finite set of clock variables ranges over by $x, y, z, \ldots$, and use V to denote the set of data variables ranged over by $i, j, k, \ldots$.

Guards Over Clock and Data Variables

We use $G(C, V)$ to stand for the set of *guards*, i.e., the set of formulas ranged over by g, generated by the following syntax $g ::= a|g \wedge g$, where a is a constraint in the form: $x \sim n$ or $i \sim n$ for $x \in C$, $i \in V$, $\sim \in \geq, \leq, =$, and n being a natural number. Each control location l may also contain an invariant condition in the form of $x \leq n$, meaning that the clock x must not increase beyond value n before the location l is exited.

Reset Operations

To manipulate clock and data variables, we use R to denote the set of possible reset operations in the form of $w := e$, where w is a clock or data variable, and e is an expression. A reset operation on a clock variable should be in the form $x := n$, where n is a natural number; a reset operation on a data variable should be in the form $i = c * i + c'$, where c and c' are integer constants.

Channels, Urgent Channels, and Synchronization

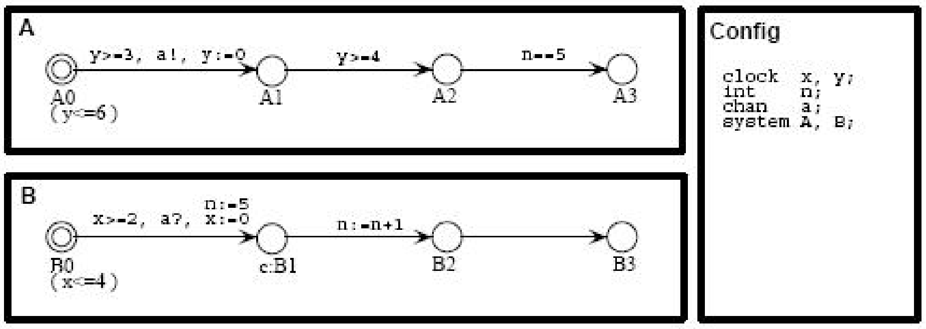
We assume that processes synchronize with each other via channels. Let A denote the set of channels, and U denote a subset of A that are urgent channels on which processes must synchronize whenever possible. We use $\mathcal{A} = \alpha?|\alpha \in \mathcal{A} \cup \alpha!|\alpha \in \mathcal{A}$ to denote the set of actions that processes can perform to synchronize with each other. Conceptually $\alpha?$ means that the process receives an input from channel α, and must synchronize with another process that performs an output to channel α, denoted by $\alpha!$. This is CCS-style pair-wise synchronization, as opposed to the CSP-style broadcast synchronization, where all processes with the same action label α must synchronize together with no distinction between input and output.

Automata with Clocks and Data Variables

An automaton $\mathcal{A}$ over actions A, clock variables C, and data variables V is a tuple (N, l_0, E), where N is a finite set of nodes called *control nodes*, l_0 is the initial node, and $E \subseteq N \times G(C, V) \times \mathcal{A} \times 2^R \times N$ corresponds to the set of edges.

Concurrency and Synchronization

To model networks of processes, we introduce a parallel composition operator. Assume that $A_1, \ldots, A_n$ are automata with clocks and data variables. We use $\bar{A}$ to denote parallel composition $(A_1| \ldots |A_n)$.



```
clock   x, y;
int     n;
chan    a;
system A, B;
```

Figure 2.7: Two example Timed Automata modeled with UPPAAL. This figure is taken from [99].

Figure 2.7 shows two automata A and B composed together in UPPAAL. There are two global clocks x and y, a global variable n, and a global channel a. Initially both clocks x and y are set to 0. In automaton A, the invariant $y \leq 6$ in control location A_0 means that A_0 must be left within 6 time units, while the guard $y \geq 3$ on the transition edge from A_0 to A_1 means that this transition can only be taken when $y \geq 3$. The combined effect of the invariant and guard means that A must make a transition from A_0 to A_1 between time 3 and time 6. The shared channel a forces automata A and B to synchronize on their respective transitions $A_0 \longrightarrow A_1$ and $B_0 \longrightarrow B_1$. Upon making the transitions, clocks x and y are both reset to 0. The guard can involve data variables such as $n == 5$; the assignment operation can also involve data variables such as $n := n + 1$.

Property Specification

The UPPAAL model-checker can be used to check for invariant and reachability properties in the form of *all reachable states must satisfy* β, or *some reachable states must satisfy* β, where

$$\beta = a|\beta_1 \wedge \beta_2|\neg\beta$$

a is an atomic formula being an atomic clock or data constraint, or a component location. Atomic clock or data constraints are either integer bounds on individual clock or data variables (e.g., $1 \leq x \leq 5$), or integer bounds on differences between clock or data variables (e.g., $1 \leq x - y \leq 5$).

We can also specify and check *bounded liveness* properties, not directly, but using testing automata. For example:

$$\psi = \phi \text{Until}_{<t} a$$

states that property a must become true before t time units, and ϕ must hold true before then.

2.2.4 Timed Petri-Nets

A TPN is characterized by a 7-tuple $N = (P, T, B, F, I, M_0, D)$, where

- P is a finite set of *places* p_i.
- T is a finite set of *transitions* t_i.
- B is the *backward incidence* function $B : T \times P \rightarrow N$, where N is the set of nonnegative integers.
- F is the *forward incidence* function $F : T \times P \rightarrow N$.
- I is the *inhibitor edge incidence* function $I : T \times P \rightarrow \{0, 1\}$. The input place to an inhibitor edge is called an *inhibitor input place*.
- M_0 is the *initial marking* function $M_0 : P \rightarrow N$.
- D is a mapping $D : T \rightarrow Q^* \times (Q^* \cup \infty)$, which associates a *delay interval* $\tau = [lb, ub]$ with each transition $t \in T$, where Q^* is the set of rational numbers.

A transition t is said to be *enabled* when each input place p_i has at least $B(t, p_i)$ tokens, and each of inhibitor input place p_j is empty. A transition T with delay interval

$\tau = [lb, ub]$ is fired as soon as it is enabled, unless disabled by the firing of a conflicting transition that removes tokens from some of T's input places. The firing takes at least lb time units, but no more than ub time units. During the firing, the tokens at the input places have been consumed, but the tokens at the output places have not been produced. If a transition with delay interval $\tau = [lb, ub]$ becomes enabled at time θ_0, it fires immediately and consumes its input tokens, then finishes and produces its output tokens at sometime during the interval $[\theta_0 + lb, \theta_0 + ub]$.

Note that our definition of TPN requires each transition to be *urgent*, that is, fired as soon as enabled, unless disabled by a conflicting transition at that instant, while in the original definition[74], no bound is imposed on when a transition may fire after it is enabled. Also, note the distinction between Timed Petri Net and *Time Petri Net* [67]. In a Time Petri Net, transition T has to be enabled continuously for $[lb, ub]$ time units before it can fire, and the firing is instantaneous: input tokens are consumed and output tokens are produced at the same time. During the time interval $[lb, ub]$, T may be disabled by the firing of a conflicting transition.

2.3 Unified Modeling Language

Object-Oriented (OO) design methodologies have become the mainstream in enterprise software development. Unified Modeling Language (UML) [91] is the *de facto* standard for OO design and development. It is a result of merging several OO design methodologies proposed in the 1980s such as Booch, OMT and OOSE.

2.3.1 Diagram Types in the UML Standard

UML consists of the following main types of diagrams: Use Case Diagram, Class Diagram, State Transition Diagram, Collaboration Diagram, Sequence Diagram, Activity Diagram and Deployment Diagram.

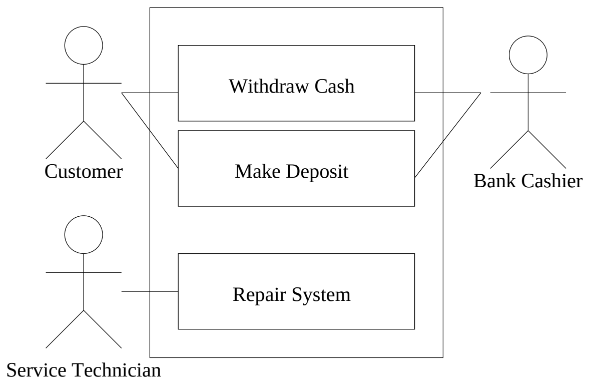


Figure 2.8: An example UML Use Case Diagram.

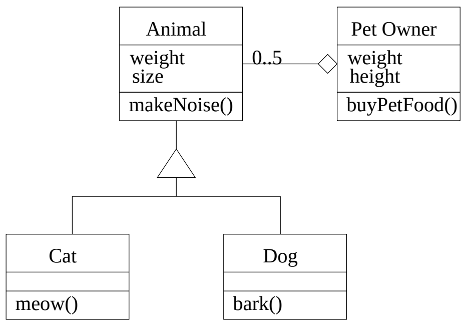


Figure 2.9: An example UML Class Diagram.

Use Case Diagram A Use Case Diagram describes interaction scenarios between the external users of the system, also called actors, and the system itself. For example, Figure 2.8 shows an example Use Case Diagram for a banking system, where actors *customer* and *bank cashier* participate in use cases *withdraw cash* and *make deposit*, and actor *service technician* participates in use case *repair system*. Use Case Diagrams are a good way to express basic functionalities of the system at early design stages due to its informal nature.

Class Diagram The Class Diagram is central to the OO methodology, used to express the concepts of *encapsulation*, *polymorphism* and *inheritance*. Encapsulation means that a class hides its private data members and methods from the external view. This facilitates reuse of the class in different contexts. Polymorphism means that sometimes the type of

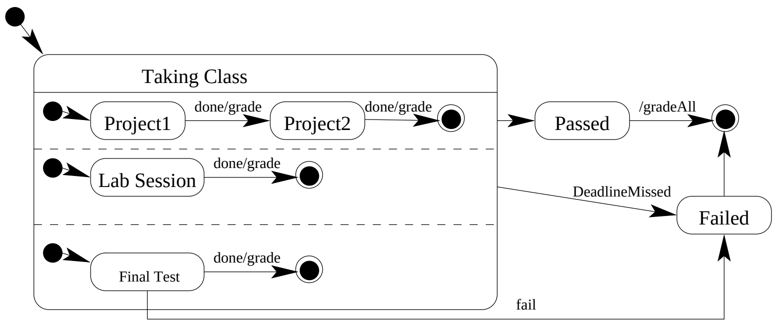


Figure 2.10: An example UML Statechart Diagram.

an object cannot be determined statically, but can only be determined dynamically at run-time. Inheritance means that classes may form specialization/generalization hierarchies. Figure 2.9 shows an example Class Diagram, which specifies *inheritance* relationships between the *animal* class and the *cat/dog* classes, and an *association* relationship between a pet-owner and 0-5 animals.

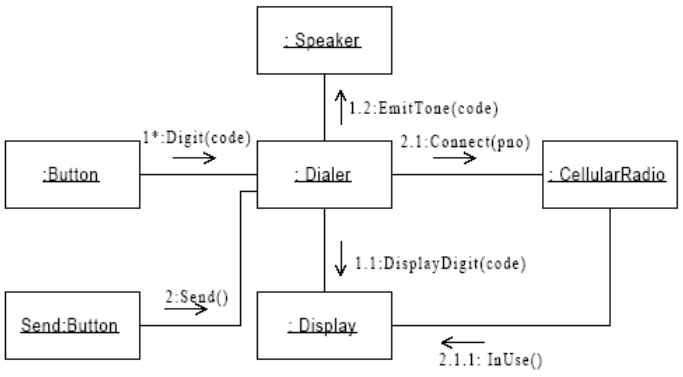


Figure 2.11: An example UML Collaboration Diagram.

State Transition Diagrams A State Transition Diagram is associated with a class, and describes its dynamic behavior. Most UML tools implement a variant of state transition diagram called *Statechart* [39] by David Harel. It is similar to the concept of Finite State Automata (FSA) in theoretical computer science, but with added concepts of hierarchy and other attributes on transitions such as action, guard, etc. Figure 2.10 shows an example statechart describing the steps needed to take a college course. The initial state is denoted

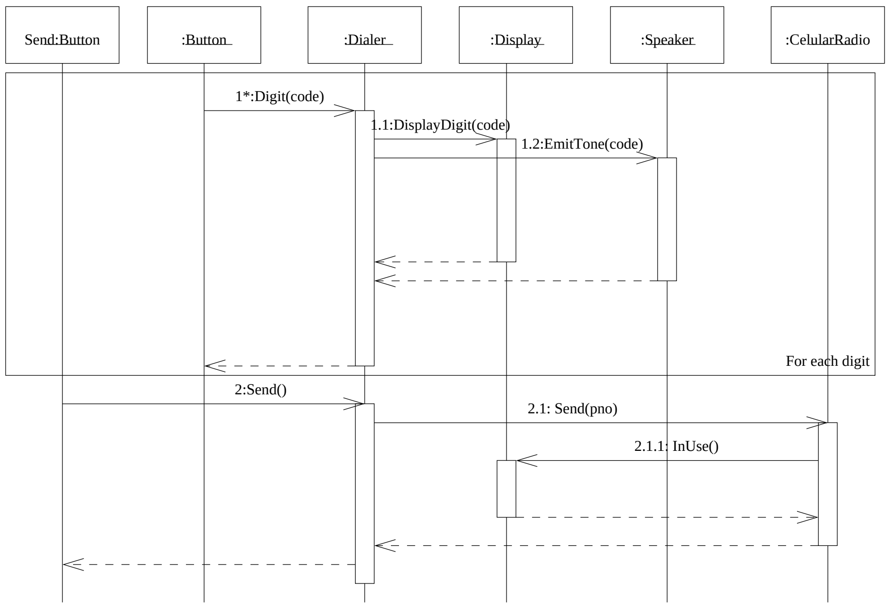


Figure 2.12: An example UML Sequence Diagram.

by a filled circle, and final state is denoted by a filled circle contained within an empty circle. The state *Taking Class* contains 3 concurrently-executing parallel state machines separated by dotted lines called *swim lanes*. In order to take a class, the student must complete two projects sequentially and receive grades for both. He also needs to complete a lab session, as well as taking a final test and receive grades for both. If he fails the final test, or he misses any deadlines for his projects, labs or tests, the *fail* state is reached; otherwise the *passed* state is reached. The statechart always ends up on the final state.

Collaboration Diagram A Collaboration Diagram describes the collaboration among a group of objects. It is similar to the Use Case Diagram in the sense that it describes an interaction scenario, but at a much more fine-grained level. Figure 2.11 shows an example Collaboration Diagram for a cell phone making a call. Note that the rectangles denote objects, not classes, as indicated by the underlined class names. The syntax of an object name is "*Object Name* : *Class Name*", for example "Send : Button" means that the object named *Send* is an instance of the class named *Button*. The other objects are anonymous

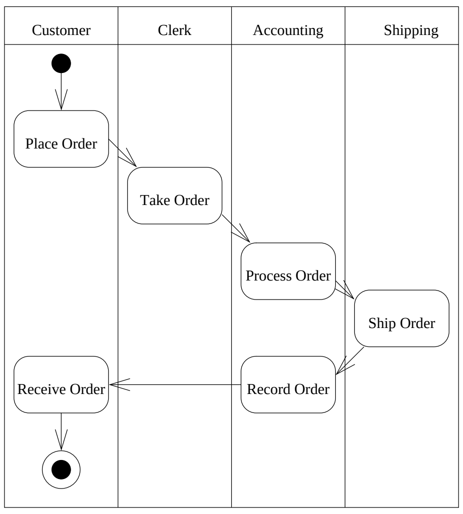


Figure 2.13: An example UML Activity Diagram.

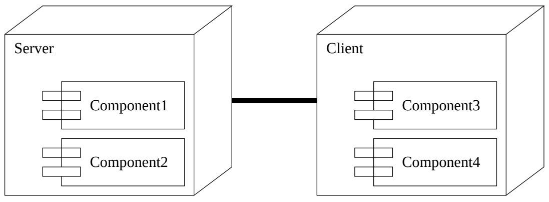


Figure 2.14: An example UML Deployment Diagram.

with no object names specified in front of the colon. Objects communicate with each other using messages, which may be an asynchronous message or a synchronous method call. Messages are tagged with sequence numbers, and may or may not have parameters. For example, the sequence of interaction is: the *Button* object sends a *digit* with a *code* parameter to the *Dialer* object, which in turn sends a *DisplayDigit* message to the *Display* object, and an *EmitTone* message to the *Speaker* object, with the same parameter *code*. After all digits have been input, the *Send* object sends a *Send*() message with no parameters to the *Dialer* object, which in turn sends a *Connect* message with parameter *pno* (phone number) to the *CellularRadio* object.

Sequence Diagram The Sequence Diagram is just another way to represent a collaboration diagram. Figure 2.12 shows the Sequence Diagram corresponding to the Collaboration Diagram in Figure 2.11. They represent exactly the same information, but have different emphasis. It is easy to see the event sequence on the timeline in the sequence diagram, while the user has to manually trace through the sequence numbers in the collaboration diagram. On the other hand, it is easier to see the interaction patterns and inter-relationships between objects in the collaboration diagram. Note the informal notation "*For each digit*" for the first scenario. A more formal variant of sequence diagram is *Message Sequence Charts* (MSC), commonly associated with the *Specification and Description Language* (SDL) [83] widely used in the telecommunications domain.

Activity Diagram The Activity Diagram describes the behavior of a set of objects. It has a formal semantics basis defined by Petri-Nets. Figure 2.13 shows an Activity Diagram for the order processing business process.

Deployment Diagram The Deployment Diagram is usually used at the later stages of system design to show allocation of software components on the execution platform. Figure 2.14 shows that *Component1* and *Component2* are allocated on the machine called *Server*; *Component3* and *Component4* are allocated on the machine called *Client*.

2.3.2 Adapting UML for Embedded Real-Time Systems

UML is a large and complex standard developed by a consortium of dozens of companies, each demanding the addition of their own features to the UML standard. In order to make UML more suitable for particular application domains, several extension and customization mechanisms have been provided, including *stereotypes*, *tagged values* and *constraints*. For example, a stereotype *Active Object* can be defined to denote that an object contains its own conceptual thread of execution. A UML *Profile* is a collection of stereotypes, tagged values and constraints that together provide a modeling environment customized to an application domain. Each UML tool vendor is free to define its own profile for its own tool. In order to promote inter-operability among different UML tools, several standard profiles have been defined, e.g., the UML Profile for Schedulability, Performance and Time [71] for analysis tasks related to schedulability and performance.

Despite the emergence of several tool vendors that specialize in real-time UML tools, such as ILogix [47], Artisan Software [6], IBM Rational [46], and Telelogic [89], UML has not met with as wide acceptance in the embedded software domain as it has in the enterprise software domain. Part of the reason for UML's failure in the embedded software domain lies in its generality, even though the profiling mechanism offers limited power of extensibility within the UML standard. As an alternative to UML, we have discussed the *Model-Integrated Computing* [88] (MIC) approach in Section 2.1, which advocates building *domain-specific* graphical modeling environments with built-in domain abstractions. Engineers can then use these customized modeling environments to express domain concepts directly, instead of encoding domain concepts in a general purpose language. The equivalent approach with UML would be to modify the UML meta-model, which will result in the model no longer conforming to the UML standard.

The UML+SDL Approach of Telelogic The CASE tool Telelogic Tau [89] advocates using UML in the early object-oriented analysis and design stages, and Specification and Description Language (SDL) [83], an International Telecommunications Union (ITU) standard typically used to describe communication protocols, in the latter stages of implementation. The approach is based on a translation technique that maps key concepts of UML into concepts of SDL. Some example mappings are:

- A UML *active object* is mapped into a SDL *process*.
- A UML *passive object* is mapped into a SDL *abstract data type*.

- A *communication channel* between active objects in UML is mapped into a SDL *channel for signal exchange*.
- A UML *statechart* is mapped into a SDL *state machine*.

Telelogic claims that SDL is more formal than UML, offering benefits such as executable models, functional simulation, automatic code generation and test generation with TTCN. However UML-RT [78] from IBM Rational seems to offer most of these benefits and obviates most of the need for using SDL.

Approach of Artisan Software

The Real-Time Studio CASE tool developed by Artisan Software [6] adds two new diagram types called *Context Diagram* and *System Hardware Diagram*. They are then used together with Class and Object Diagrams to model mapping of logical software components onto the architectural platform. By adding the two new diagram types, the tool is no longer 100% conformant to the UML standard.

Action Semantics

Action semantics is an OMG standard that aims to formally specify actions, for example at state transitions in Statecharts, to enable formal analysis and code generation. It applies to all UML-based modeling approaches, and is orthogonal to real-time modeling approaches. It is intended to replace the typical approach of using a programming language such as C++/Java to specify actions. The OMG Request For Proposal (RFP) [69] for action semantics says:

> UML currently uses uninterpreted strings to capture much of the description of the behavior of actions and operations. To provide for sharing of semantics of action and operation behavior between UML modelers and UML tools, there

> needs to be a way to define this behavior in a well-defined, inter-operable form. As such time as the Action Semantics requested in this proposal are mapped to a syntax, and are combined with UML, UML shall constitute a computationally complete language. This language is targeted at system analysis and behavior description, and is not envisioned to be a language suitable for system deployment.
>
> Action semantics is not a new programming language. Rather, it provides for the specification of systems in sufficient detail that they can be executed, and the it should provide just enough semantics to enable the specification of computation.

Embedded UML for Hardware/Software Codesign

Embedded UML [63] is a research project carried out by Cadence and University of California at Berkeley to define a UML profile suitable for hardware/software codesign. It represents synthesis of various ideas in the real-time UML community and concepts drawn from the codesign community. The following concepts from real-time UML are adopted:

- Specification of embedded systems as a collection of reusable communicating blocks using a functional decomposition.
- Class Diagrams for object type definition.
- Encapsulation of functions within a *block*, an extension of the *capsule* concept in UML-RT.
- Communications explicitly defined via ports, protocols and connectors.
- Use Cases and Sequence Diagrams to specify test-benches and test scenarios.

- Carefully defined state diagram semantics, combined with specified action semantics which can be used to drive code generation, optimization and synthesis.
- The concept of a refinement continuum, from non-executable specifications, to formal executable specifications, to implementation.

The following concepts are taken from the codesign world:

- A rigorously defined platform model in both hardware and software for the implementation architecture. This can be conceptually thought of as a collection of resources offering services, as in UML real-time extension profiles. The collection of platform services can be thought of as a system platform "API".
- Using *mapping* as the platform-dependent refinement paradigm for performance analysis, communication synthesis and optimized code synthesis/generation.
- A concept of *reactive objects* rather than *active objects*.

UML Profile for Schedulability, Performance and Time

As shown in Figure 2.15, the UML Profile for Schedulability, Performance, and Time (UML-SPT) [71] is partitioned into a number of *sub-profiles*, profile packages dedicated to specific aspects and model analysis techniques. At the core of the profile is the set of sub-profiles that represent the general resource modeling framework. These provide a common base for all the analysis sub-profiles in this specification. However, it is anticipated that future profiles dealing with other types of QoS (e.g., availability, fault-tolerance, security) may need to reuse only a portion of this core. Hence, the general resource model is itself partitioned into three separate parts. The innermost part is the resource modeling sub-profile, which introduces the basic concepts of resources and QoS. These are general

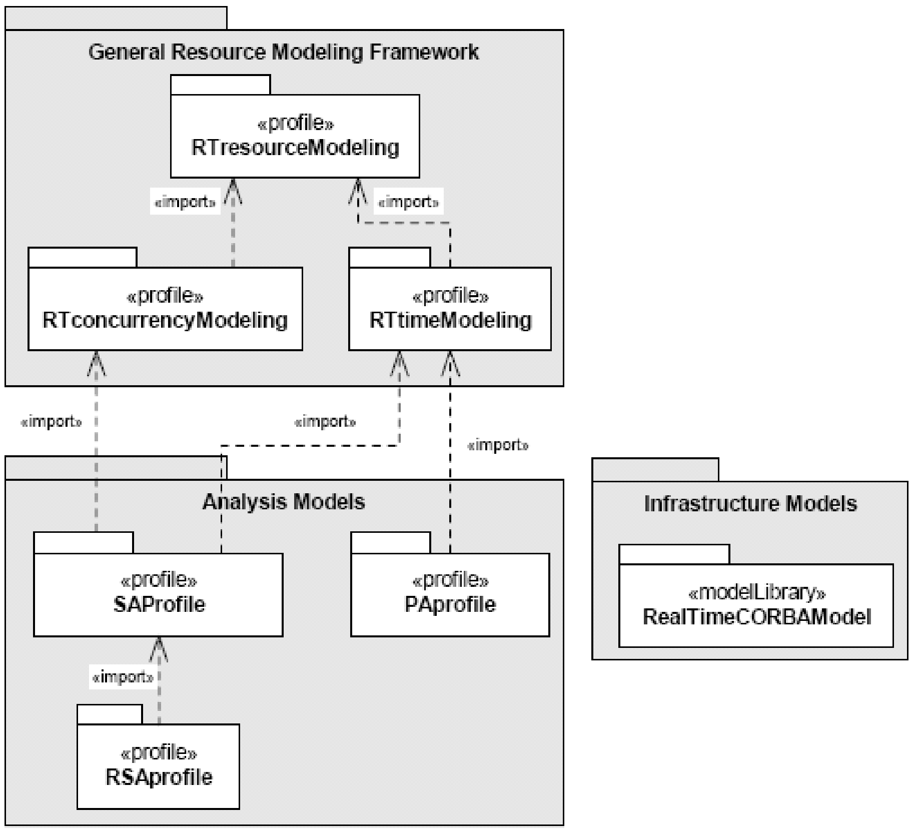


Figure 2.15: The structure of UML Profile for Schedulability, Performance and Time. This figure is taken from [71].

enough and independent of any concurrency and time-specific concepts. Since concurrency and time are at the core of the requirements behind this specification, they each have a separate sub-profile.

The three different model analysis profiles are all based on the general resource modeling framework. One sub-profile is dedicated to *performance analysis* and another is dedicated to *schedulability analysis*. In addition, there is a model library that contains a high-level UML model of Real-Time CORBA [76]. The modular structure shown in Figure 2.15 allows users to use only the subset of the profile that they need. This means choosing the particular profile package and the transitive closure of any profiles that it imports. For example, a user interested in performance analysis, would need the *PAProfile*, *RTTimeModeling*, and *RTResourceModeling* packages.

2.4 Real-Time Scheduling Theory

In this section, I provide a brief introduction to real-time scheduling theory. Some of the material here is excerpted from [96].

The problem of *scheduling* arises whenever multiple active entities, be it processes, threads or human beings, compete for shared resources. In an ERT system, the most frequently encountered scheduling problems are CPU scheduling and network packet scheduling in order to meet system-level timing constraints. There are generally two techniques for dealing with real-time problems today:

1. Keep the system lightly loaded so that it has a high safety margin.
2. Use real-time scheduling algorithm to analyze system schedulability before deployment in order to achieve high processor utilization.

The second approach is almost always better, since higher processor utilization translates into use of less powerful CPUs and networks, and savings on hardware costs.

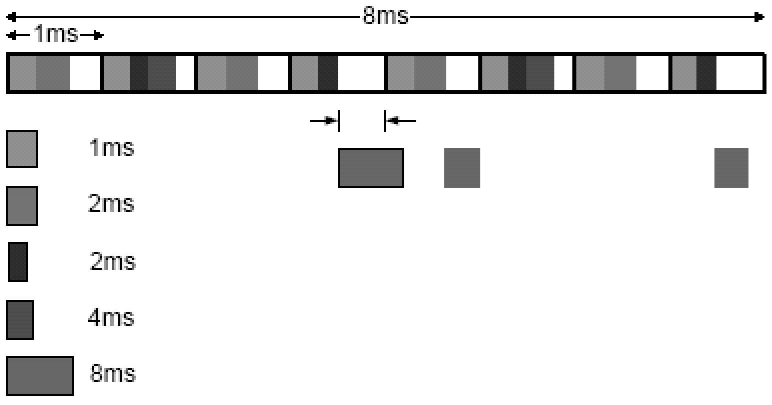


Figure 2.16: An example cyclic executive schedule.

An early approach to real-time scheduling is *cyclic executive scheduling*, where a static, time-driven schedule is created. Figure 2.16 shows an example cyclic executive

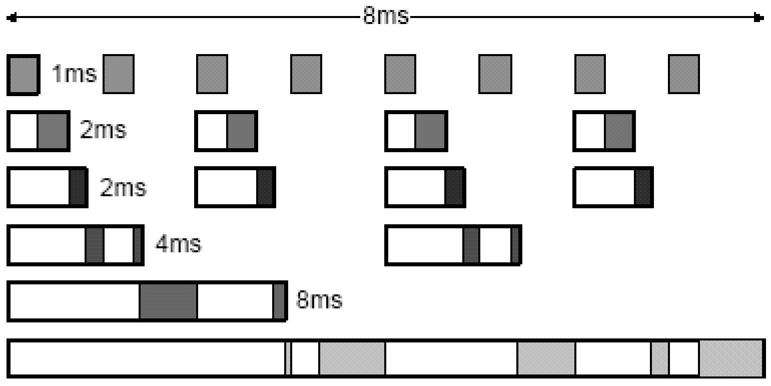


Figure 2.17: An example rate monotonic schedule.

schedule, containing 5 tasks with periods 1ms, 2ms, 2ms, 4ms and 8ms. The timeline consists of a *major cycle* of 8ms, and 8 *minor cycles* of 1ms each. The 1ms task has the highest rate, and executes within every minor cycle, and other slower tasks execute only in some of the minor cycles. The major cycle is typically the smallest common multiplier of all task periods.

Cyclic executive scheduling has been widely adopted in smaller embedded systems with tight resource constraints. It has a number of advantages such as high predictability, low runtime overhead both in terms of memory (only needs one execution stack) and in terms of time (scheduler is a simple table-lookup operation). However there are also drawbacks:

- When task periods are relatively prime, the major cycle can get very long, and storage of the static schedule can take up a lot of memory space.
- Aperiodic tasks must be handled with polling and cause waste of execution resources. For example, an infrequently occurring interrupt handler with a tight deadline of 2ms must be handled by polling the task occurrence every 1ms.
- It is difficult to maintain and evolve the system. Some activities do not fit within a single minor cycle and have to be broken up into pieces manually at the source

code level. An example is the 8ms task in Figure 2.16, which has to be broken into several pieces that communicate via global variables. Even small changes can result in major rework of the real-time schedule.

An alternative is to use a Real-Time Operating System (RTOS) and schedule the tasks based on priorities. Figure 2.17 shows the same taskset scheduled with *rate monotonic* priority assignment, that is, the task with smaller period is assigned higher priority. Using a RTOS remedies a lot of the problems with cyclic executive scheduling. There is no need to store a table of static schedules in memory; aperiodic tasks do not need to be handled with polling, hence no more wasted resources due to high polling rates; the system is much easier to maintain, since there is no need to manually split up a large task into smaller pieces in order to fit into a static schedule. However there are also drawbacks. Introduction of concurrency brings with it all the complexities such as semaphores, monitors and mailboxes. Everyone who has taken an introductory operating systems course knows the pain and perils of concurrency programming.

Task priority assignments can be static in the case of Figure 2.17, or dynamic, i.e., the priorities of tasks are dynamically adjusted during runtime. One example dynamic priority scheduling technique is Earliest Deadline First (EDF), where the task with the closest deadline is given the highest priority.

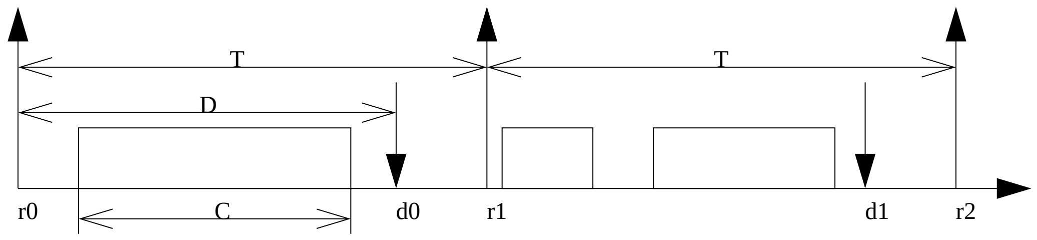


Figure 2.18: The task model showing attributes used for real-time scheduling analysis.

In order to introduce schedulability analysis techniques, first we need some notation. A task has the following attributes, as shown in :

- r: **release time**: the trigger time of the task execution request.
- C: **worst-case execution time** (WCET): the longest time that the process takes to finish when the process is fully allocated to it.
- D: **relative deadline**: the maximum acceptable delay for its processing.
- T: **period**(only for periodic tasks).
- d: **absolute deadline**: $d = r + D$.
- p: **priority**: larger number denotes higher priority.

Each time a task is ready, it releases a periodic request. The successive release times are at $r_k = r_0 + kT$, where r_0 is the first release, and r_k the $(k + 1)$th release.

For a set of independently executing periodic tasks with rate monotonic priority assignment[59], there are two approaches to performing schedulability analysis of a priority scheduled system:

- Utilization bound test. A sufficient schedulability condition is

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1)$$

  where n is the number of periodic tasks. The upper bound converges to 0.69 for high value of n. When the total utilization is less than this bound, the system is guaranteed to be schedulable. However, this is only a sufficient condition, not a necessary condition. For a taskset with higher utilization than this bound, we need to calculate the Worst-Case Response Time for each task and compare it to the deadline of the task to determine schedulability, as discussed below.

- Response time calculation based on busy period analysis. Task *i*'s Worst-Case Response Time (WCRT) R_i:

  $$R_i = C_i + B_i + \sum_{j, p_j > p_i} \lceil \frac{R_i}{T_j} \rceil \cdot C_j$$

  where the sum is over all tasks with higher priorities than task i. B_i stands for blocking time experienced by task i, caused by shared resources with lower priority tasks. This term can be bounded by priority inheritance or priority ceiling protocols [73]. After R_i is calculated, the task is schedulable if $R_i \leq D_i$. A taskset is schedulable If all tasks in the taskset are schedulable.

CHAPTER III

Model-Level Static Analysis

In this chapter, we discuss model-level static analysis techniques as implemented in the AIRES (Automatic Integration of Reusable Embedded Software) tool within the MoBIES tool-chain for the AMC Open Experimental Platform (OEP). Traditional static analysis techniques work at the level of programming languages, and study control and data flow relationships associated with functions and variables. Control dependency refers to flow of control through a sequential program, and data dependency refers to the locations of definitions and uses of the program variables. When we raise the level of abstraction to models, it is necessary to perform *model-level* static analysis; that is, dependency relationships between software components and ports at the level of system architecture, which often involve concurrency and distribution. This work is reported in [36, 29].

This chapter is structured as follows: Section 3.1 provides an introduction to the AMC software. Section 3.2 provides a general overview of the MoBIES tool-chain. Section 3.3 describes the AIRES tool for dependency and real-time analysis of the Embedded Systems Modeling Language (ESML) models. Section 3.4 demonstrates the usage of the tool-chain with an application example, and the chapter concludes with Section 3.7.

3.1 Avionics Mission Computing

In this section, we provide an introduction to the Avionics Mission Computing (AMC) software. Some of the material here is excerpted from Boeing documentation [81].

The AMC software is the embedded software aboard a military aircraft for controlling mission-critical functions, such as navigation, target tracking and identification, and weapon firing. It is modeled in Unified Modeling Language (UML), manually coded in C++, and runs on top of Real-Time CORBA Event Service [76]. Even though there exist UML models for the AMC software, they mainly serve in a documentation role that the engineer refers to while writing code manually. Therefore, the link between model and code is weak and easily broken in the process of system maintenance and evolution, where code is modified or enhanced without the corresponding changes at the model-level, or vice versa. Furthermore, UML has little support for analysis that is relevant for embedded systems, such as real-time properties like schedulability, safety properties like deadlock freedom, etc. The motivation for the MoBIES tool-chain is to have a more automated and integrated development process than the current one. The MoBIES tool-chain is a suite of tools for model-based design, static analysis and code generation for AMC software, and is a result of collaboration between the University of Michigan, Vanderbilt University and Southwest Research Institute, as well as engineers from Boeing.

The AMC software architecture is commonly referred to as the *BoldStroke* architecture. It is based on the publish/subscribe paradigm with Real-Time CORBA Event Service [76] as its underlying communications substrate. Event publishers push events through the event channel to event consumers, whose execution is triggered by the arrival of events.

Figure 3.1 shows a typical system architecture for an AMC System. Multiple CPUs are

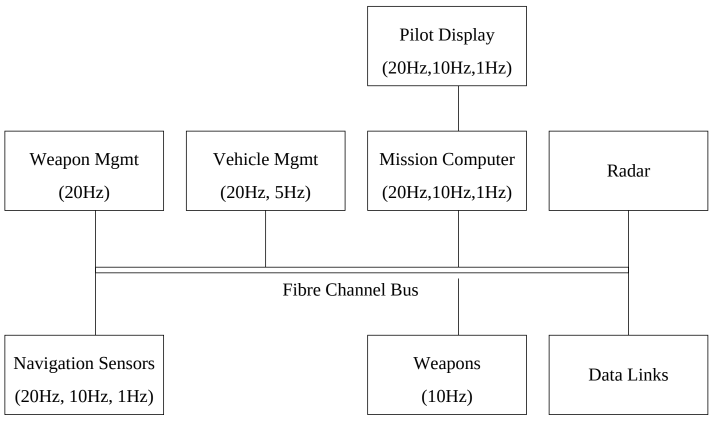


Figure 3.1: Typical system architecture of AMC system.

connected via a fibre channel bus. Our research focuses on the mission computer, where the main mission algorithms are located.

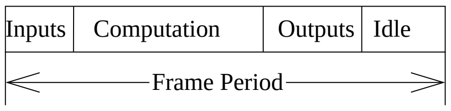


Figure 3.2: The four phases of an execution frame.

AMC systems are largely periodic. On each CPU, there are typically 5-6 execution rates such as 40Hz, 20Hz, 10Hz, 5Hz, and 1Hz. Rate monotonic scheduling (RMS) [53] is used to make real-time guarantees, i.e., the thread with higher execution rate has higher priority. Network communication is governed by predefined update intervals. Weapons control needs minimum update frequencies to meet release and accuracy constraints. The pilot interface requires a minimum update frequency for clear display. Pilot input commands requires a minimum sampling frequency to achieve responsiveness.

CPU time is partitioned into periodic frames. Figure 3.2 shows the four distinct phases of a frame:

- Polling inputs from remote devices through the network. That is, collect snapshot of overall system state.

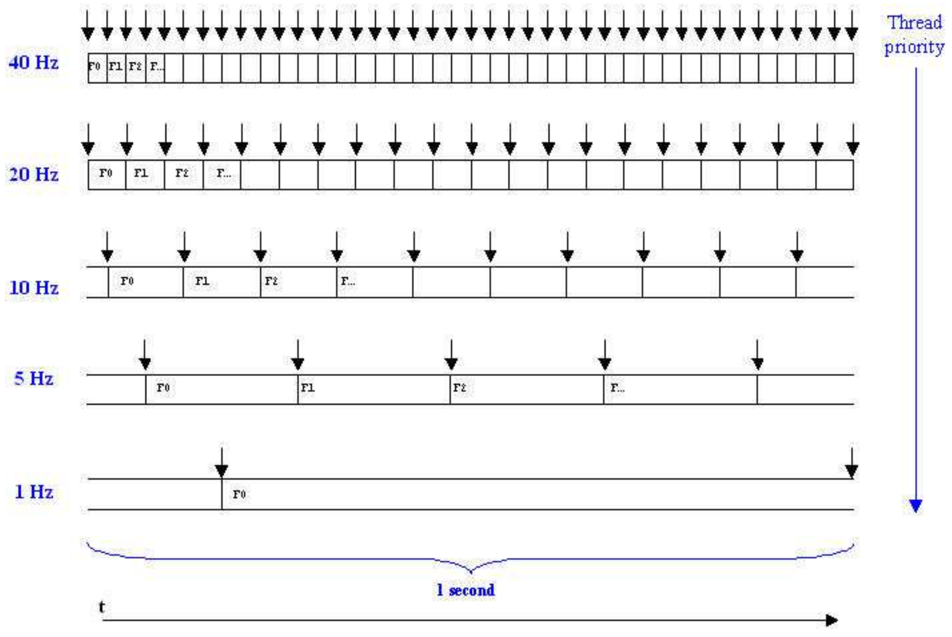


Figure 3.3: The execution frames within a 1 second time interval. Note that the frame start times are staggered to reduce peak load of the network.

- Execute the core algorithms to process data.
- Sending outputs to output devices through the network.
- Idle time. This time can be used for future enhancements and upgrades to the system.

The periods of frames are typically harmonic. An I/O frame manager executes at the highest rate interval timeout to control the start of frame for all threads. The frame manager is responsible for sending start-of-frame events to all rate groups, such that there is a full coherency of rate boundaries (i.e. - a single 1 Hz timeout = 20 20Hz timeouts). Additionally, because the avionics I/O bus is an atomic shared resource, executing I/O operations cannot be preempted like regular action processing. It is the frame manager's responsibility to interleave frames for optimal use of the I/O bus and CPU resources. In order to reduce bus contention, frames are scheduled such that no more than two frames are initiated on any given interval timeout of highest frequency , as shown in Figure 3.3.

3.2 Overview of the MoBIES Tool-Chain

The central repository of information in the MoBIES tool-chain is ESML [51], a domain-specific language for modeling the AMC software using Generic Modeling Environment (GME) [57]. The ESML meta-model defines a comprehensive modeling language that captures essential aspects of the embedded system, including software architecture, timing and resource constraints, execution threads, execution platform information (processors and network), allocation of components to threads/processors, etc. ESML is based on Real-Time Event Channel implemented in the TAO CORBA [76]. Components are composite objects with ports, which interact with one another, either through event triggers or procedure invocations. We adopt the CORBA Component Model (CCM) terminology, where each component can have the following types of ports.

- *Publish Port* to publish events.
- *Subscribe Port* to subscribe to events.
- *Receptacle* to issue method invocations.
- *Facet* to accept method invocations.

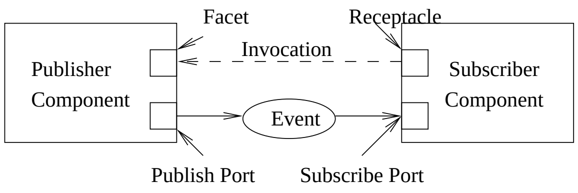


Figure 3.4: The control-push/data-pull style of interaction.

Component interactions typically (but not always) follow the *control-push data-pull* style as shown in Figure 3.4. First, the data producer component publishes a *DataAvailable* event from its publish port, indicating that it has fresh data; when the data consumer component receives the event from its subscribe port, it issues a *GetData()* call from its receptacle to the producer's facet to retrieve the data.

Each input port (subscribe port or facet) has an associated *action* that, in turn, triggers one or more output ports (publish port or receptacle) of the same component. This allows us to determine the intra-component trigger pathways used in the subsequent dependency analysis. Each *action* also has a WCET attribute used for real-time analysis. Each subscribe port can subscribe to multiple events, and has a *correlation* attribute, either *AND* or *OR*. For AND correlation, an input port inp is triggered, i.e., the action associated with inp is executed, only when *all* of the input events arrive at inp; for OR correlation, the port is triggered when *any* of the input events arrives.

Models described in ESML serve as the central repository of information for all analysis and code generation purposes. Several standardized interface formats such as Analysis Interface Format (AIF), Configuration Interface Format (CIF) and Instrumentation Interface Format (IIF) have been defined for use in conjunction with ESML to facilitate integration with third-party tools. They are described with UML-based meta-models, and translators between them can be written using APIs generated from the meta-models.

The workflow of the MoBIES tool-chain has the following steps, as shown in Figure 3.5.

1. The *input translation* step imports existing UML models in Rational Rose [46] into GME as component types in ESML. The designer then manually constructs ESML models of system architecture by instantiating and inter-connecting the components, and enhances the models with attributes specific to embedded systems such as timing and resource information.

2. The *analysis translation* step extracts information from the ESML models for anal-

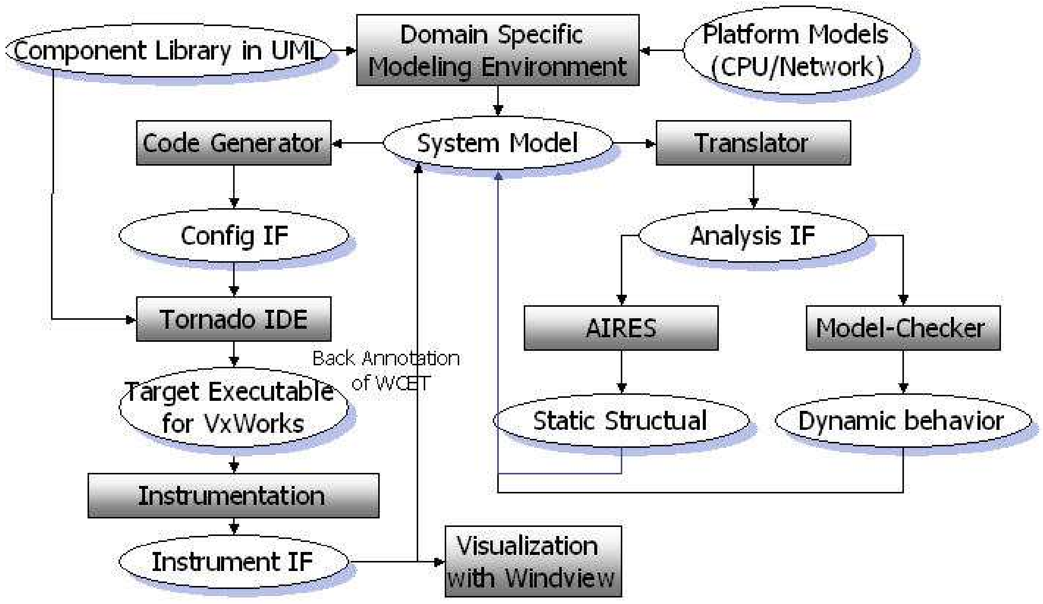


Figure 3.5: The MoBIES tool-chain for Avionics Mission Computing. Tools within the tool-chain inter-operate via standardized interface file formats in XML such as AIF, CIF and IIF. Translators, such as ESML2AIF, ESML2CIF and IIF2AIF, are command-line utility programs that transform between file formats.

ysis purposes in the form of *Analysis Interface Format* (AIF) XML files, which is essentially a subset of the ESML language that contains dependency and real-time information needed by the analysis tools. The analysis tool called AIRES performs various types of static analysis tasks on the AIF models and updates them with analysis results, which can be imported back into the ESML models, as shown by the bi-directional arrow between ESML and AIF models.

3. The *configuration translation* step generates system configuration file in the form of *Configuration Interface Format* (CIF), which is used to generate C++ header files used for initializing the component inter-connection topology at system startup. Together with the component library, the target application can then be built using the Tornado environment from WindRiver [104].

4. Once we have a running system on the target platform, we can collect information

from runtime instrumentation of the system in the form of *Instrumentation Interface Format* (IIF) and import it into the AIF model as timing annotations, or use the tool WindView from WindRiver to visualize the execution timeline. Instrumentation can also be used to reverse-engineer an existing application to create CIF models.

The MoBIES tool-chain contains a number of novel ideas:

- It uses domain-specific models specialized to the Avionics Mission Computing domain, which allow us to precisely capture domain concepts with domain-specific modeling constructs, as opposed to general purpose UML modeling tools that provides a fixed set of modeling constructs with limited extensibility.
- It covers the entire systems development life-cycle including modeling, analysis, code generation and runtime instrumentation, as opposed to *point solutions* that targets limited points in the system life-cycle.
- It is an integrated environment of multiple tools collaborating via standardized interface file format definitions in XML and the Open Tool Integration Framework (OTIF) [50], which allows easy plug-in of other third-party tools by adding a few lines of code that calls the OTIF API. This represents a significant advantage over closed, proprietary, monolithic tool architectures that offer limited or no extensibility.

3.3 Model-Level Static Analysis and the AIRES Tool

AIF is essentially a subset of the ESML language that contains the dependency and real-time information needed by the analysis tools. Given an AIF file, AIRES extracts system-level dependency information, including event- and invocation-dependencies, and constructs port- and component dependency graphs. Various analysis tasks are supported based on these graphs, such as checking for anomalies like dependency cycles, visual display of dependency graphs, as well as forward/backward slicing to isolate relevant components. It then assigns execution rates to component ports, and uses real-time scheduling theory to analyze the resulting system of real-time task-set. If the task set is not schedulable, the designer can add more processors and allocate components to them with the help of the automated allocation algorithm. AIRES can be viewed as a semantic translator from a logical, object-based to a runtime, task-based modeling paradigm. It traverses the model elements in the logical view, performs various analysis tasks, and feeds back results into the runtime view.

3.3.1 System-Level Dependency Analysis

Traditional software dependency analysis works at the code-level, and studies control and data flow relationships associated with functions and variables. Control dependency refers to flow of control through a sequential program, and data dependency refers to the locations of definitions and uses of the program variables. When we raise the level of abstraction from code to models, it is necessary to perform *model-level* dependency analysis; that is, structural relationships between software components and ports at the level of system architecture, which often involve concurrency and distribution.

We extract system dependency information from AIF models and construct a directed graph called *Port Dependency Graph* (PDG), where each node is a *port*, and each edge denotes dependencies between ports. Note that we use ports to refer to both event publish/subscribe ports, and invocation facet/receptacles.

Definition. A *Port Dependency Graph* (PDG) is a graph (V_p, E_p), where

- V_p is a set of *ports*, $\{p_i, 1 \leq i \leq N_p\}$. Each p_i can be one of 4 types: publish port p_{pub}, subscribe port p_{sub}, receptacle p_{recep} or facet p_{facet}.

- E_p is a set of directed, weighted *port connections*, $\{conn_i, i \leq i \leq N_{conn}\}$, and each $conn_i$ can be one of 2 types:

  Inter-component dependency: is either event-trigger dependency from the output port of the publisher component to the input port of the subscriber component, or invocation dependency from receptacle of the invoking component to facet of the invoked component.

  Intra-component dependency: describes the intra-component trigger pathways from input ports to output ports of the same component.

  The weight of an edge is equal to the execution rate of the ports that it connects multiplied by the size of data transferred at each execution cycle.

The PDG captures all the relevant dependency information in the ESML model, and serves as the backbone data structure for all subsequent analysis tasks. However, we define component dependency graphs (CDG) for purposes of convenient visual display as well as easy manipulation in certain analysis tasks. CDG captures dependency information at a higher level of abstraction — component-level instead of port-level — hiding all the intra-component dependencies. It can be derived directly from PDG.

We can use graph algorithms to analyze the dependency graphs, and identify such anomalies as:

- Dependency cycles. A cycle of event or invocation dependencies indicates a design error if it becomes an infinite loop at runtime. However, in the case of feedback loops, it is possible to have a legitimate dependency cycle if the component receiving the feedback has *AND* correlation for its inputs.

- Events published with no subscribers, or events subscribed to with no publishers.

- Component ports unreachable from any timers, hence unable to be assigned rates. This is elaborated on in Section 3.3.2.

In all cases AIRES provides warnings to the designer, but it is up to the designer to decide if it is an error or not.

We can also perform *forward/backward slicing* of the dependency graphs. Given a component or a port, we can answer user queries such as

- What downstream components/ports can this component or port potentially affect via event or invocation dependencies?

- What upstream components/ports can potentially affect this component or port?

This is achieved by traversing the dependency graphs forward or backward starting from a component (for CDG) or a port (for PDG). These queries are useful in software evolution, where a designer can assess the impact of changing or replacing a certain component, as well as for other purposes such as localizing faults, minimizing regression tests, reusing components, and system re-engineering.

Even though the current avionics software does not allow dynamic creation or destruction of components, both the inter- and intra-component dependencies can change at runtime due to *modal* behavior, that is, components can change mode to publish new events, stop publishing old events, or change its internal trigger pathways. For example, a modal component can have both active and inactive modes. When in the active mode, an input event triggers an output event; when in the inactive mode, an incoming event is simply ignored and dropped. Instead of a single PDG, we can view the system as having multiple pre-defined system-level modes, obtained by all combinations of component modes. Besides component modes, it is also possible for the system to have a system-wide *normal* mode and a *fault* mode. In the fault mode, one or more processors can fail, and certain *backup components* on the working processors are activated to replace components on the failed processors. We can construct a PDG for each system-level mode, and apply the analysis techniques to each mode separately.

3.3.2 Real-Time Analysis

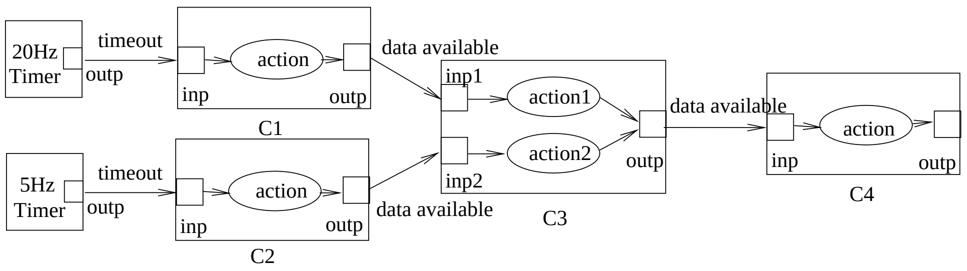


Figure 3.6: An example for rate assignment to ports. *inp* stands for *input port*, and *outp* stands for *output port*.

The runtime execution framework for Bold Stroke uses RT-CORBA Event Service [76] running on VxWorks [104], which supports a single address-space process on each processor with multiple threads. The mission computer interacts with sensors and actuators through periodic messages on one or more communication buses. Messages are triggered at harmonically-related execution rates such as 1Hz, 5Hz, and 10Hz. As a result, each processor has a number of *system threads,* also called *rate groups,* running at harmonically-related rates. This periodicity forces processing within a rate group to be divided into *execution frames,* where each frame represents the fixed execution period. For example, the execution frame for a 20Hz rate group has a period of 50ms. Triggered by the *Timeout* events generated by a periodic timer, the frame begins by polling input messages from the communications buses. After inputs are complete, a *DataAvailable* event is pushed to initiate a chain of actions along the dependency graph. When all actions within a given rate group complete (frame processing completes), an output message is sent to external devices on the communications bus. A frame failing to complete outputs prior to the start of the next frame is said to be in a *frame overrun* condition, meaning that it has missed its (relative) deadline which is equal to its period.

Each component/port pair is assigned an execution rate. All the ports assigned the same execution rate run within the context of the system thread with that rate. For example, all ports assigned a rate of 20Hz run within the context of the 20Hz thread. The WCET of the thread is thus the sum of WCETs of all the actions associated with input ports assigned to the thread. Note that rates are assigned to ports, not components, and therefore, a component may be *multi-rate* if it has multiple ports with different rates assigned.

The rate-assignment algorithm performs recursive depth-first search (DFS) [15] on the PDG starting from each timer's publish port, which publishes the *Timeout* event.[1] We consider both inter- and intra-component dependencies as port dependencies. All the ports reachable from a timer port is assigned the rate of the timer. If multiple input ports of a component are assigned different rates, and trigger the same output port, the output port is assigned the highest rate, i.e., the highest rate takes precedence and propagates through. If an input port of a component subscribes to multiple events with different rates, it is assigned the highest rate if its correlation attribute is *OR*, or the lowest rate if its correlation attribute is *AND*. For the latter case, the component *under-samples* the higher-rate inputs.

Figure 3.6 shows an example scenario to illustrate this algorithm. First, we traverse the dependency graph starting at the 20Hz timer, which is the highest-frequency timer. We assign 20Hz to all ports reachable from the 20Hz timer, which includes *C1.inp, C1.outp, C3.inp1, C3.outp, C4.inp, C4.outp*. All these ports are marked *visited* in the algorithm. The WCET of the 20Hz thread is the sum of all the WCETs of the chain of triggered actions, that is, *C1.action.wcet + C3.action1.wcet + C4.action.wcet*. Next, we start from

[1]Besides periodic threads, it is also possible to have aperiodic threads triggered by external hardware interrupts. The same algorithm can be used to traverse the PDG starting from a hardware interrupt source instead of from a timer, and assign execution rates as the maximum arrival rate of the interrupt.

the 10Hz timer, and assign 10Hz to *C2.inp, C2.outp, C3.inp2*. When the DFS algorithm encounters *C3.outp*, it backtracks since *C3.outp* has already been marked as *visited* by the previous traversal starting from the 20Hz timer. The WCET of the 10Hz thread is thus *C2.action.wcet* + *C3.action2.wcet*. If *C3* contains encapsulated data that are accessed by both threads, then we need to make the data access operation a *critical section*, and adopt synchronization protocols such as priority inheritance [73]. If this is the case, we need to add *blocking time* to the higher-priority thread, equal to the longest critical section of the lower-priority thread spent in the shared component.

After executing the rate assignment algorithm, we obtain a set of system threads or tasks (we use threads and tasks interchangeably). The Bold Stroke framework adopts a fixed-priority, rate monotonic, preemptive scheduling discipline, that is, the higher rate thread has a higher execution priority and can preempt lower-priority threads. This allows us to use mature Rate Monotonic Analysis (RMA) techniques [53] to calculate thread response times. If the hardware platform is composed of multiple processors, and a system-level thread crosses processor boundary, it becomes a distributed and precedence-constrained task chain. The end-to-end response time analysis technique [87] can be applied for schedulability analysis of such task chains.

Since the network utilization for a typical avionics system is low (10-20%), we assume network delays add a constant factor to the overall response time of a distributed task chain. This may not be a realistic assumption, especially since the avionics bus is a non-preemptively scheduled resource. It is part of our future work to incorporate detailed analysis of message scheduling delays on the avionics bus into the end-to-end analysis.

3.3.3 Automated Component Allocation

Given a component/port dependency graph and a multiple-processor hardware platform, we would like to allocate the components to processors in order to achieve certain objectives such as schedulability, load balancing, and minimized network communication. The typical allocation process works as follows: the designer manually allocates components to processors by modifying the ESML models in GME, and then invokes AIRES to assess system real-time properties. If the system is not schedulable, he goes back to the models, redoes the allocation or adds more processors, and iterates the process until schedulability is achieved. As a typical system contains thousands of components and complex interactions, it is highly desirable to provide tool support to automate this process.

At the most basic level, the designer can visually examine the dependency graphs to identify components with high or low-cohesion between them while making allocation decisions. We have also implemented simple allocation heuristics such as *first-fit* (to minimize the number of processors) and *best-fit* (to achieve load-balancing). Here we describe a heuristic algorithm based on [1] that attempts to minimize inter-processor communication costs while maintaining schedulability. No claims of optimality can be made due to its heuristic nature, and the designer can only view the allocation results as suggestions that help him make the final decisions. The current real-time analysis functionality is limited to static priority, rate monotonic analysis. We plan to support more dynamic scheduling disciplines such as EDF (earliest deadline first), MUF (maximum urgency first).

The algorithm is performed on CDG. First, we assign a *util* (utilization) attribute to each component, calculated from WCET and execution rate of its associated input ports. For example, an input port triggered at 20Hz and has an associated action with WCET of 5ms will contribute a utilization value of 5ms*20Hz/1000ms = 0.1. A component with two such input ports has utilization 0.2. The sum of utilizations of all components allocated to a processor must not exceed a certain upper bound *util_bound*, which is a customizable parameter. According to the classic scheduling theory [53], any processor with utilization under 0.69 is schedulable. Setting *util_bound* to a lower value puts more constraints on the allocation algorithm, and has the effect of balancing the workload across processors; setting it to a higher value makes it easier to find a feasible solution.[2] We perform a heuristic *k-way min-cut* algorithm [1] on the CDG, where k is the number of processors. That is, we cut the CDG into k clusters and allocate each component cluster to a processor, while minimizing the total weight of edges that are cut, subject to the constraint that the total utilization on each processor does not exceed *util_bound*. It is possible that the algorithm may fail to find a feasible allocation. In that case, the designer must redesign the system either by adding more processors, or increase *util_bound* for each processor.

This algorithm works quite fast in practice, and produces reasonable partitions of the CDG that serve as valuable first-cut suggestions in the early design stages. But it also has its limitations. This algorithm works under the assumption that the underlying platform is homogeneous, that is, all processors have the same processing power, and communication costs between processors are all the same. This is realistic for the tightly coupled avionics hardware architecture, which is PowerPC processors plugged into a VME backplane, but may not be applicable in the general case. More sophisticated optimization algorithms such as branch-and-bound or simulated annealing are needed to obtain accurate results, which have exponential complexity in the worst case, while the graph min-cut algorithm has polynomial complexity at the expense of optimality.

[2]In order to obtain more accurate results, real-time scheduling theory [53, 87] needs to be used as a subroutine in the allocation algorithm to assess system schedulability in place of the *util_bound* parameter. Since the allocation algorithm is heuristic in nature, we did not use more sophisticated schedulability checks during component allocation. However, the designer should use the tool capabilities described in Section 3.3.2 to access system schedulability after allocation.

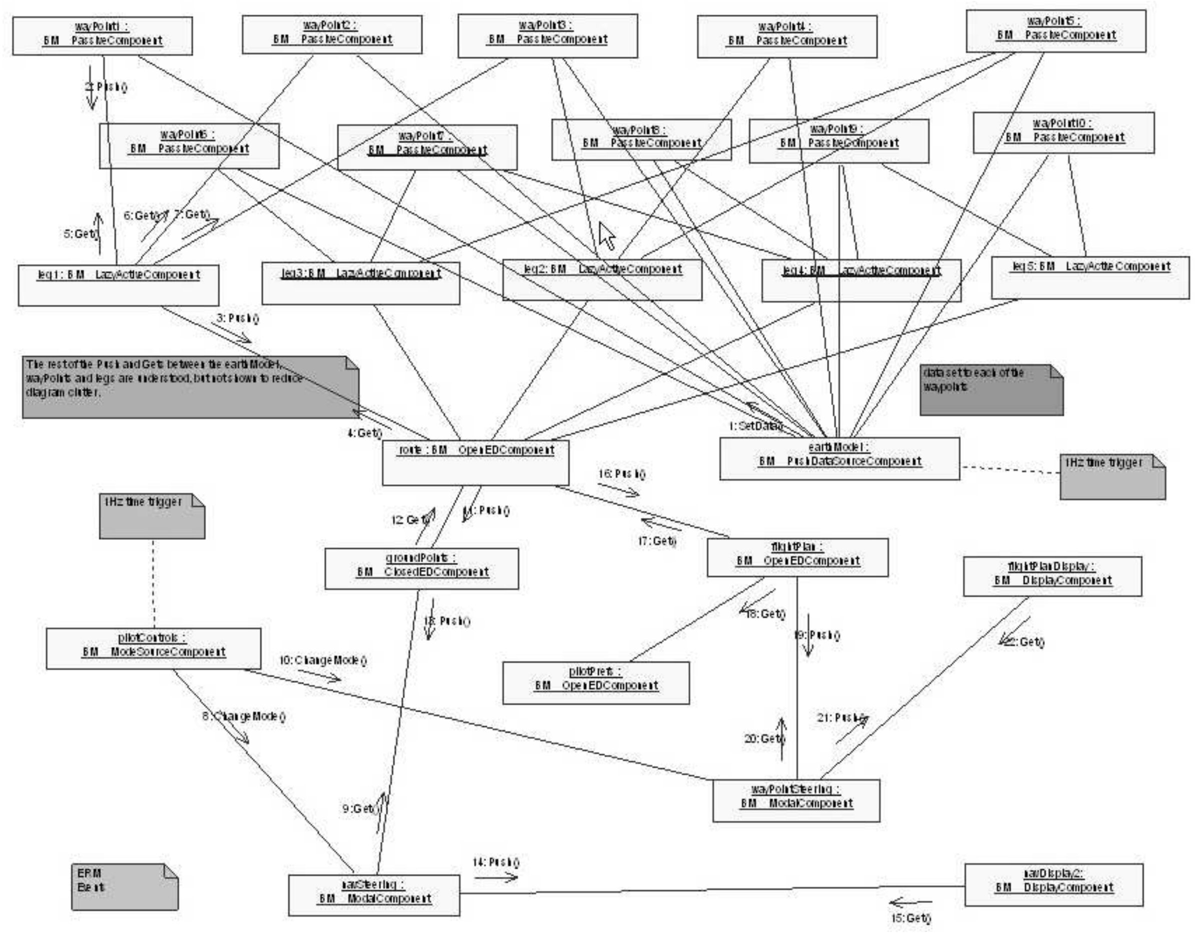

wayPoint1 : BM PassiveComponent
wayPoint2 : BM PassiveComponent
wayPoint3 : BM PassiveComponent
wayPoint4 : BM PassiveComponent
wayPoint5 : BM PassiveComponent
wayPoint6 : BM PassiveComponent
wayPoint7 : BM PassiveComponent
wayPoint8 : BM PassiveComponent
wayPoint9 : BM PassiveComponent
wayPoint10 : BM PassiveComponent
2: Push()
5: Get()
6: Get()
7: Get()
leg1 : BM LazyActiveComponent
leg3 : BM LazyActiveComponent
leg2 : BM LazyActiveComponent
leg4 : BM LazyActiveComponent
leg5 : BM LazyActiveComponent
3: Push()
The rest of the Push and Gets between the earthModel, wayPoints and legs are understood, but not shown to reduce diagram clutter.
data set to each of the waypoints
4: Get()
1: SetData()
route : BM OpenEDComponent
earthModel : BM PushDataSourceComponent
1Hz time trigger
16: Push()
17: Get()
12: Get()
11: Push()
1Hz time trigger
groundPoints : BM ClosedEDComponent
flightPlan : BM OpenEDComponent
flightPlanDisplay : BM DisplayComponent
13: Push()
18: Get()
19: Push()
22: Get()
pilotControls : BM ModeSourceComponent
10: ChangeMode()
pilotPrefs : BM OpenEDComponent
8: ChangeMode()
21: Push()
20: Get()
wayPointSteering : BM ModalComponent
9: Get()
14: Push()
ERM Bent
navSteering : BM ModalComponent
navDisplay2 : BM DisplayComponent
15: Get()

Figure 3.7: UML model for the 1Hz thread of medium single-processor scenario. This figure is taken from Boeing documentation [82].

3.4 An Example Application Scenario

We consider one of the application scenarios provided by our industrial partners, with UML Collaboration Diagrams shown in Figure 3.7 and Figure 3.8. Functionally, the scenario represents steering calculations needed to support various displays on the aircraft. There are two rate groups/system threads in the system: a 1Hz thread and a 20Hz thread. In the 1Hz thread, information from various *waypoints* — certain signposts along the designated route — is merged into route-based steering information, and is shown either in the navigation or flight plan display, depending on the pilot steering mode. In the 20Hz thread, inputs from track sensors are merged and fed into the tactical steering and HUD

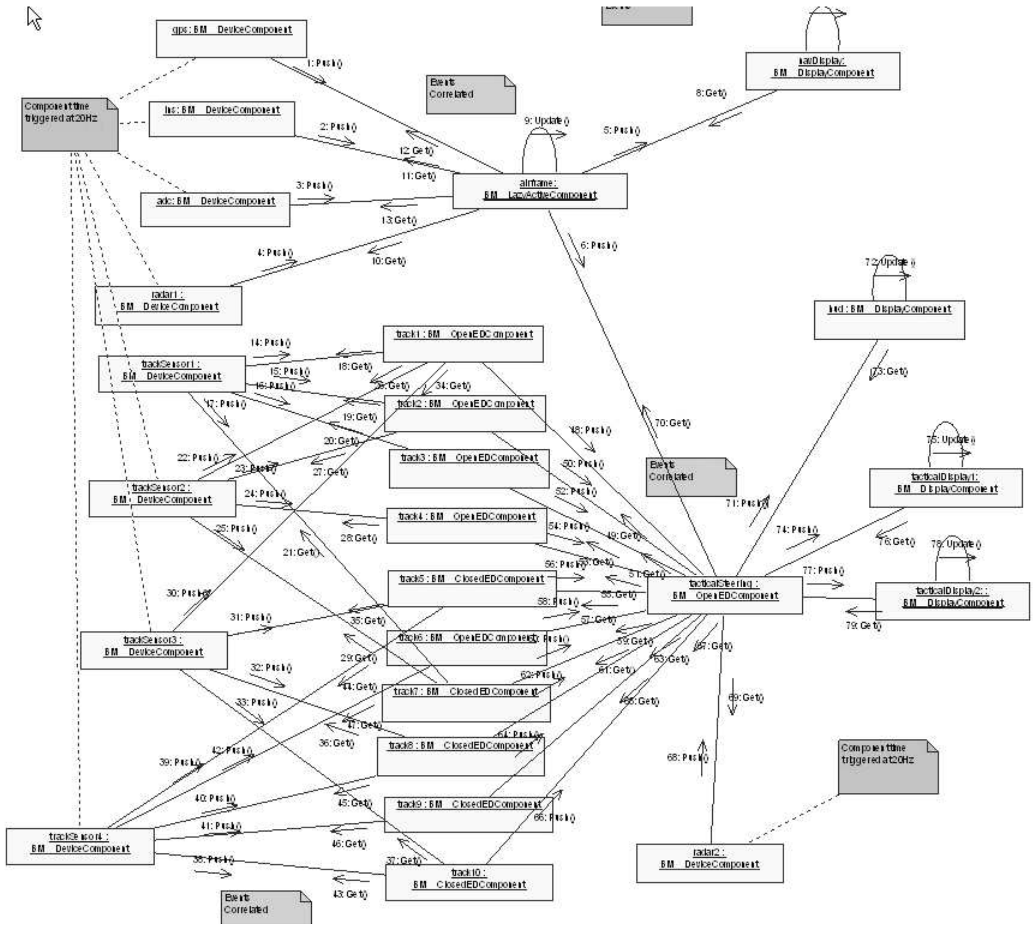
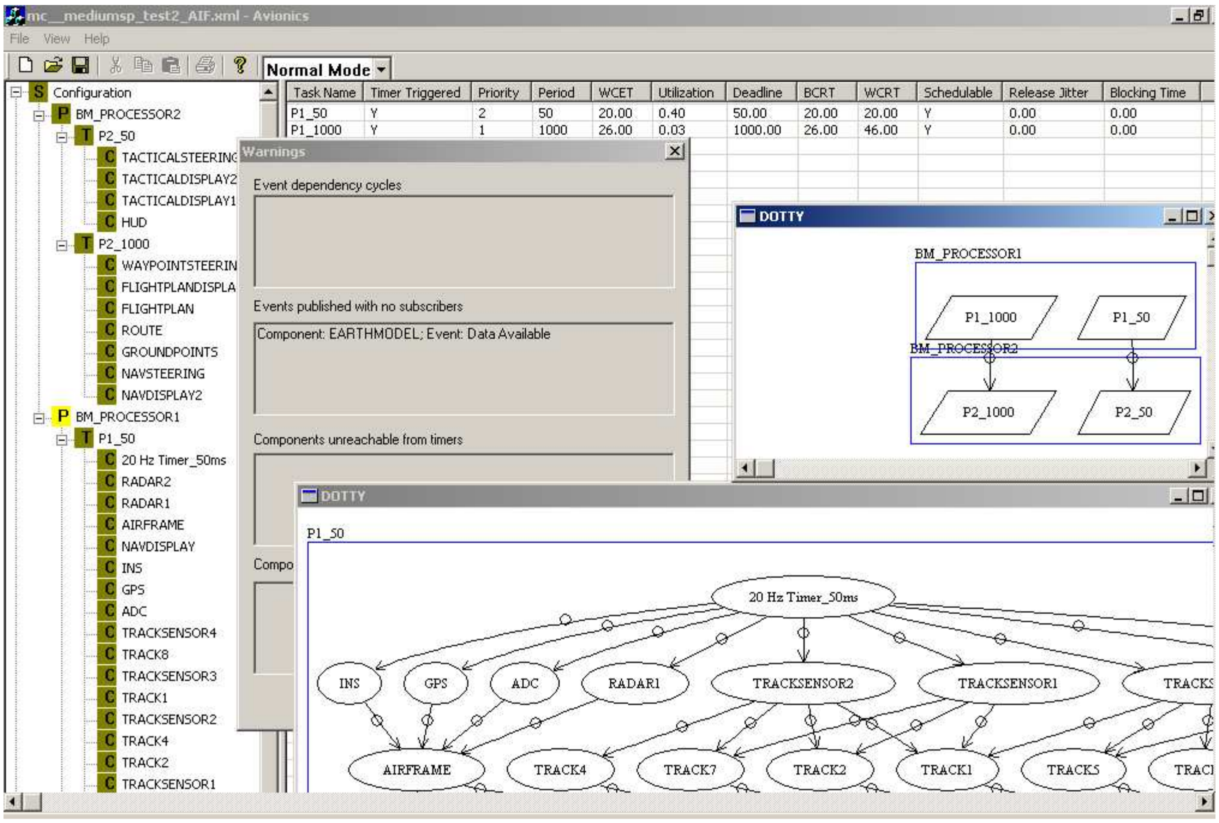

gps : BM DeviceComponent
ins : BM DeviceComponent
adc : BM DeviceComponent
radar1 : BM DeviceComponent
Component time triggered at 20Hz
Events Correlated
airframe : BM LazyActiveComponent
navDisplay : BM DisplayComponent
hud : BM DisplayComponent
trackSensor1 : BM DeviceComponent
trackSensor2 : BM DeviceComponent
trackSensor3 : BM DeviceComponent
trackSensor4 : BM DeviceComponent
track1 : BM OpenEDComponent
track2 : BM OpenEDComponent
track3 : BM OpenEDComponent
track4 : BM OpenEDComponent
track5 : BM ClosedEDComponent
track6 : BM OpenEDComponent
track7 : BM ClosedEDComponent
track8 : BM ClosedEDComponent
track9 : BM ClosedEDComponent
track10 : BM ClosedEDComponent
tacticalSteering : BM OpenEDComponent
tacticalDisplay1 : BM DisplayComponent
tacticalDisplay2 : BM DisplayComponent
radar2 : BM DeviceComponent

Figure 3.8: UML model for the 20Hz thread of medium single-processor scenario. This figure is taken from Boeing documentation [82].

(Heads-Up) display. This scenario is relatively simple from an analysis perspective since the dependency graphs for the two threads are disjunct from each other, but it serves as a good illustrating example. In order to demonstrate the end-to-end distributed real-time analysis functionality, we modify the original scenario, which runs on a single processor, to run on a 2-processor platform. Both threads crosses processor boundaries to become distributed tasks.

First, UML models are imported into GME as component types. Then, the engineer constructs a system model by instantiating component instances and connecting them together to form a system model in ESML. The CIF file is generated from ESML model,



Task Name	Timer Triggered	Priority	Period	WCET	Utilization	Deadline	BCRT	WCRT	Schedulable	Release Jitter	Blocking Time
P1_50	Y	2	50	20.00	0.40	50.00	20.00	20.00	Y	0.00	0.00
P1_1000	Y	1	1000	26.00	0.03	1000.00	26.00	46.00	Y	0.00	0.00

Component: EARTHMODEL; Event: Data Available

Figure 3.9: A screenshot of AIRES.

which can be used for building the system.

In Figure 3.9, the *normal mode* system configuration is analyzed and displayed. This scenario does not have a fault mode. The *Warnings Dialog* displays dependency anomalies such as event dependency cycles, events published with no subscribers, component/port unreachable from timers, hence unable to be assigned rates, etc. Shown on the lower right is the CDG, and the upper right is the task graph. The left pane tree view displays the processors, tasks and components organized hierarchically; the right pane list view displays different analysis results depending on the item selected on the left pane. In this figure, a *processor* node is selected in the tree view, and the list view displays tasks running on the processor, with attributes such as WCET, period, utilization, WCRT (worst-case response time), BCRT (best-case response time), system slacks (the maximum scale-up

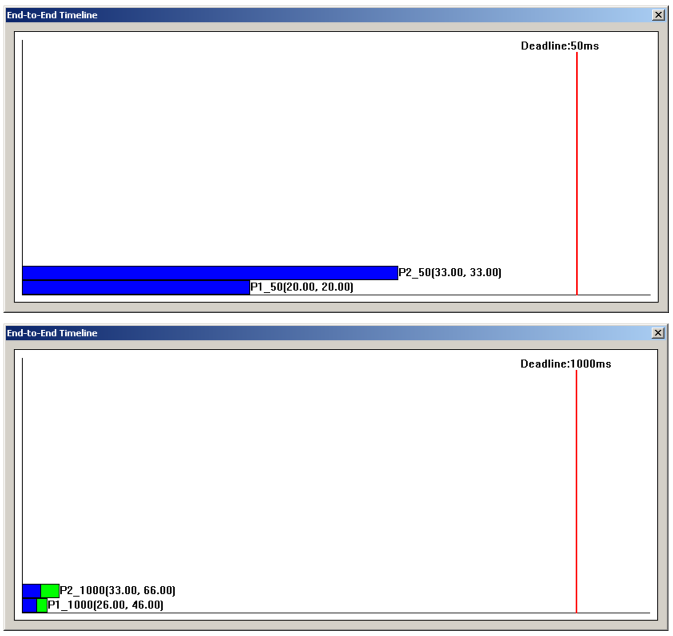


Figure 3.10: The end-to-end timeline of distributed system threads of 20Hz (left) and 1Hz (right). The numbers in square brackets denote response time interval [BCRT, WCRT] of the task starting from the timer trigger. The dark portion of the horizontal bar denotes BCRT, and the dark portion combined with light portion denotes WCRT.

factor while maintaining system schedulability), etc.

Figure 3.4 shows the end-to-end timeline for the two distributed system threads in the task graph. In this application scenario, the two system threads do not intersect at a component, so there is no blocking time due to contention for shared resources. On the left is the timeline for the 20Hz thread with period 50ms. *P1_50* is the timer-triggered task segment on processor *P1*, and *P2_50* is the subsequent task segment on *P2* triggered by the completion event of *P1_50*. The 20Hz thread is the highest priority thread in the system, and suffers neither preemption nor blocking delays. Therefore, its WCRT and BCRT are the same as its WCET. On the right is the timeline for the 1Hz thread with period 1000ms. It suffers preemption delays caused by the 20Hz thread. The first task segment *P1_1000* on *P1* has response time interval [26, 46]ms, and the next task segment *P2_1000* on *P2* has response time interval [33, 66]ms, both calculated relative to the 1Hz timer trigger on

P1. Both system threads are schedulable since they finish before their deadlines.

3.5 Experimental Evaluation

Here we discuss experimental evaluation of the MoBIES tool-chain. It is important to emphasize that the *entire MoBIES tool-chain* is being evaluated, not just our work. Some of the following discussions are excerpted from internal documents from Boeing.

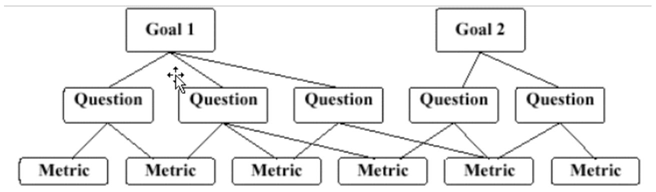


Figure 3.11: The Goal-Quality-Metric methodology. This figure is taken from Boeing documentation [77].

The evaluation process is based on the *Goal-Quality-Metric* (GQM) methodology [62], which is a framework for defining and evaluating a set of operational goals using measurements. It was developed to provide a goal-oriented evaluation approach that would support the measurement of processes and products in the software engineering domain. GQM utilizes a three-level measurement model as shown in Figure 3.11.

- *Conceptual Level (goals)*: A goal is defined for an object for a variety of reasons, with respect to various models of quality, from various points of view, and relative to a particular environment.
- *Operational Level (questions)*: A set of questions is used to define models of the object of study and then focus on that object to characterize the assessment of achievement of a specific goal.
- *Quantitative Level (metrics)*: A set of metrics, based on the models is associated with every question in order to answer it in a measurable way.

GQM goals follow a structured format to define essential characteristics. The following GQM template will be used for specifying MoBIES experimentation goals:

Analyze *object under investigation*

for the purpose of *hypothesis being tested*

from the point of view of *experimentation perspective*

in the context of *experimentation context*

To support evaluation of MoBIES technologies, two goals are defined. The first and primary goal is associated with the central MoBIES goal of improving embedded system development capabilities, and drives the MoBIES experimental evaluation:

G1: Analyze the MoBIES-enabled software component integration process for the purpose of identifying improvements as compared to current development processes from the point of view of the Product Specific Component Integrator in the context of production military distributed real-time embedded systems.

Some example questions related to the primary goal regarding the analysis capabilities of AIRES are:

- *G1.ANAL.Q1* How easy is it to obtain timing analysis results for different scheduling algorithms?
- *G1.ANAL.Q2* How accurate are timing predictions when compared to actual run-time measurements?
- *G1.ANAL.Q3* Does the addition of model-based analysis accurately identify situations which would otherwise require additional reconfigurations, including frame overruns, event dependency cycles and similarly unsatisfactory configurations?

- *G1.ANAL.Q4* Do the analysis algorithms for assigning rates and processors to components improve the slack time (across the whole system as well as in individual tasks)?

The secondary goal is associated with assessing the quality of MoBIES products themselves.

> *G2*: Analyze the MoBIES-developed toolset for the purpose of assessing the usability of the tools from the point of view of the Product Specific Component Integrator in the context of applying the MoBIES-enabled software component integration process to the experiments.

Some example questions related to the secondary goal regarding the analysis capabilities of AIRES are:

- G2.ANAL.Q1 How easy is it to update the model via the AIF with different analysis tools?
- G2.ANAL.Q2 Can you generate output from the model compatible with the AIF?
- G2.ANAL.Q3 Can the AIF be updated with runtime data via the Instrumentation Interface?

A *product scenario* is a fragment of the runtime execution scenario. For example, a *BasicSP* product scenario contains 3 software components running on a single processor within the same thread/rate group; a *MediumMP* scenario contains a dozen or so software components running on 2 processors with 2 different execution rates. Product scenarios vary in terms of size and complexity. The largest one provided by Boeing contains several hundreds of components on multiple processors, and serves as a good test for tool scalability.

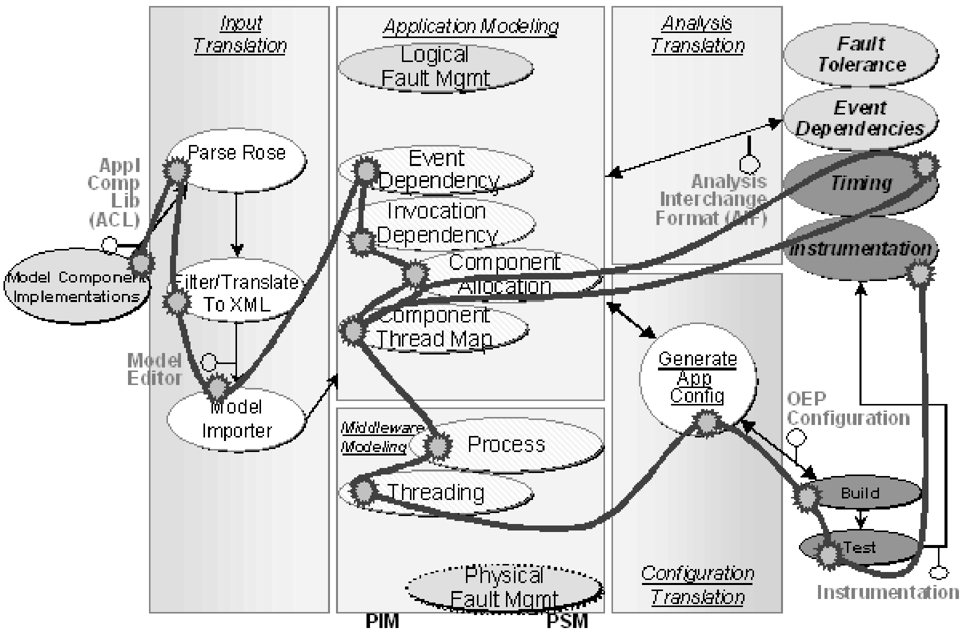


Figure 3.12: An example development scenario for GQM-based evaluation. This figure is taken from Principal Investigator (PI) meeting presentation by Boeing in 2002.

A *development scenario* defines a sequence of steps that a software engineer takes to construct and analyze a product scenario. Figure 3.12 shows an example development scenario. A set of different development scenarios are defined, differing in terms of steps taken. Some basic development scenarios do not perform timing and event dependency analysis, or feedback of instrumentation data to models, while other scenarios may include every conceivable step.

A software engineer conducts the experiments by first going through a particular development scenario with conventional development methods in order to establish a baseline for comparison, then using the MoBIES tool-chain to perform the same set of scenarios and measures the time savings as well as quality of development products. In the first-stage experiments, Boeing engineers performed the experiments and quantitative measurements regarding the software engineer metrics. In the second-stage experiments, we performed the experiments due to lack of human resources from Boeing. The Avionics OEP defined

11 product scenarios and 10 development scenarios. We participated in 8 of them related to event dependency and timing analysis, and received a *pass* grade for almost all of the metrics considered. Using the MoBIES tool-chain resulted in significant savings in overall time of software development compared to the baseline of conventional development methods.

3.6 Related Work

Ptolemy [12] is a modeling and simulation tool for embedded systems developed at University of California, Berkeley. Its main focus is on hierarchical modeling and simulation of different *Models of Computation* (MoC), also called *domains*, including communicating sequential processes (CSP), continuous time (CT), discrete events (DE), distributed discrete events (DDE), discrete time (DT), process networks (PN), synchronous dataflow (SDF), synchronous reactive (SR), etc. Different domains are composed hierarchically in order to enforce a uniform model of computation at a particular level of hierarchy. HyVisual [45] is a hybrid systems modeling tool built on top of Ptolemy, which further restricts the modeling semantics in order to target the domain of hybrid systems. Ptolemy advocates *actor-oriented design* as a natural extension of the concepts of *object-oriented design*, where actors are conceptually autonomous entities interacting through message passing, with the interaction dynamics defined through models of computation that governs the actors. The actor concept was first proposed by Gul Agha in the object-oriented programming languages community, and is embodied in many block-diagram-based modeling tools such as Simulink/Stateflow [65] and UML-RT [46]. The Ptolemy group formalized the notion of actors and made them *domain polymorphic*, that is, an actor can be embedded into different domains and function according to the MoC of the domains without modification of the actor itself. In order to facilitate actor-oriented programming, a domain-specific language called CAL [21] has been defined to replace low-level manual coding of actors in Ptolemy.

Giotto [52] is a time-triggered and platform independent programming language aimed at hard real-time applications. It consists of tasks interacting through ports, and modes that denote system-wide modal behavior. It defines a *time-triggered* model of computation as opposed to the common practice of priority-based scheduling in most commercial RTOSs. It adopts fixed-logical execution time (FLET) assumption in order to eliminate jitter caused by fluctuations in the actual execution time and scheduling algorithm. That is, the output of a task is always produced at the FLET time regardless of how long the task itself takes to complete its execution, provided it finishes within its FLET, which also serves as its deadline. This approach requires accurate estimation of WCET of tasks. If deadline violations occur at runtime, it is treated as a design error, and either the algorithmic part of the task must be redesigned to fit into its FLET, or the FLET must be prolonged to accommodate the task. A translation procedure has been defined to map from Simulink models into Giotto models, so that a seamless design process can be achieved starting from Simulink modeling tool down to Giotto-based runtime. This approach achieves a clean separation between functional aspects defined with Simulink and timing aspects defined with Giotto.

Rapide [60] is an Architecture Description Language (ADL) that uses partially ordered event sets (posets) as the formalism for component behavior description. The Rapide toolset includes a simulator used to simulate a Rapide specification, a visualizer used to visualize the resulting poset, and an animation viewer used to animate the poset. ESML can be viewed as another ADL that is specialized for the model of computation of the Bold Stroke framework, while Rapide is a more general ADL that is capable of modeling a wide variety of systems from enterprise transaction systems to communication protocols. Some concepts in ESML are missing in Rapide and vice versa. For example, Rapide does not have concepts of hardware platform or allocation of components to platforms. The main functionality of the Rapide toolset is dynamic simulation, while AIRES focuses on static dependency and real-time analysis.

Aladdin [84] is a tool for architecture-level dependency analysis of Rapide specifications. It introduced the concept of three types of *chains* – *affected-by*, *affects* and *related* chains – which is similar to our concepts of slicing of the dependency graphs. Similarly, Zhao [106] described a slicing-based approach to extracting reusable software architectures, which takes Wright [2] specifications as input. Their work does not focus on embedded real-time systems, and hence does not support any timing analysis.

MetaH [9] is an ADL and toolset for development of real-time, fault-tolerant, securely-partitioned, multi-processor software in the avionics domain. The toolset supports runtime executive code generation in Ada, real-time schedulability analysis, as well as reliability and security analysis. The AMC software framework differs from the MetaH framework in many ways. For example, Bold Stroke uses RT-CORBA Event Service as its underlying communication and execution substrate, which can be viewed as the counterpart of the MetaH runtime executive. The use of COTS (commercial off-the-shelf) software like TAO CORBA eliminates the need for generation of a customized executive for each application. Code generation from ESML models has a different meaning, and refers to generation of component configuration code in XML used at system initialization, instead of the runtime executive for MetaH.

TimeWeaver [17] is a real-time software composition framework and toolset, also developed within the MoBIES program. It enables a clean separation of functional and non-functional concerns in development of component-based real-time systems. Components are connected via couplers, designed to capture three types of interactions of couplings among components: data, control and timing. After designing a logical architecture from reusable software components, the designer can easily experiment with different physical design/deployment decisions from components to the physical platform by applying different couplers. The toolset supports code generation in Java or configuration generation in XML, as well as timing analysis by exporting information into the TimeWiz [94] tool.

VEST (Virginia Embedded Systems Toolkit) [85] is an integrated environment for constructing and analyzing component-based embedded systems. Designers can select or create passive components, compose them into a system, map them onto runtime structures such as processes or threads, map them onto hardware platform, and perform dependency checks and non-functional analysis along many dimensions such as real-time, performance and reliability.

3.7 Summary

In this chapter, we have discussed model-level static analysis techniques for object-oriented real-time software, using Avionics Mission Computing (AMC) as the main target application domain. Even though the AMC software is considered in this thesis as a major target application domain, our work has more general applicability to component-based embedded software, e.g., the *CORBA Component Model* (CCM) [70], a widely adopted industry standard from the Object Management Group [69]. In fact, the AMC software is currently being migrated to the CCM platform for its next generation.

The entire MoBIES tool-chain is a result of collaboration among multiple institutions, but we have mainly focused on the system-level dependency and real-time analysis techniques implemented in the AIRES tool, which include dependency anomaly detection, visual displays of dependency graphs, assignment of execution rates to component ports, timing and schedulability analysis, automated component allocation, etc. All the algorithms implemented in the tool are of polynomial complexity and scalable to large, realistically-sized systems. The analysis capabilities bring significant benefits to the component integrator, who is responsible for assembling together software components retrieved from the component repository, by providing valuable dependency and real-time information at the model-level. This information used to be buried in the source code underneath layers of abstraction and not readily accessible to the development engineer. For example, the engineer used to have to use a debugger to trace through hundreds of thousands of lines of code to track down a cyclic event dependency bug.

AIRES represents a significant improvement over the current software development practice, which relies heavily on time-consuming and expensive testing on the target platform, as it provides insight into non-functional aspects of models at design-level, and helps the engineer make high-level design decisions that have a large impact on the embedded software. It is complementary to tools in a typical Integrated Development Environment (IDE) such as Tornado that work at the code level, such as compilers, debuggers, runtime tracers and automated testers. As the model-based approach is becoming more mainstream, as evidenced by the Model-Driven Architecture [90] initiative, and the number of tool vendors in the embedded real-time domain that support it, tools that work at the model-level will become more prevalent.

CHAPTER IV

Application of Model-Checking to Avionics Mission Computing Software

AIRES mainly focuses on the *static structural* aspects while largely ignoring the *dynamic behavioral* aspects of embedded software. The dynamic behavior of software components is described in natural language documents, so it is not possible to perform automated analysis. In order to perform deeper semantic analysis, it would be valuable to construct *executable models*, which enables the use of *simulation* or *model-checking* to verify system correctness. One prominent example of executable models is Harel's Statechart [39]. Model-checking can be viewed as exhaustive simulation, which works by exhaustively exploring the system state space to prove certain correctness properties. In this chapter, we use a model-checker LTSA [61] to provide formal specification of dynamic behavioral of the AMC software, and prove properties such as safety and liveness. We also discuss several techniques to reduce state space ad improve scalability of model-checking by exploiting application-level domain semantics. This work is reported in [33].

This chapter is structured as follows: section 2.2.2 provides a brief introduction to the modeling formalism FSP and model-checker LTSA; section 4.1 discusses modeling of different component types; section 4.2 discusses modeling of component interactions resulting from instantiating and inter-connecting components from component types; section 4.3

discusses property specification for verification; section 4.4 describes two techniques for improving scalability of model-checking, and section 4.6 concludes the chapter.

4.1 Modeling of Component Types

The AMC software is component-based, with many different types of components, each with its unique functionality and interfaces, acting as basic building blocks of a complete system. The documentation provided by Boeing [82] describes the various component types in detail. However, these descriptions in prose are not formal and subject to misinterpretation or misunderstanding. We use FSP to provide an unambiguous, formal description for each component *type* based on the natural language descriptions, and instantiate each component *instance* to form a system architecture. In what follows, we describe each component by excerpting the corresponding descriptions from the Boeing documentation, and then presenting the FSP specification. Here are the naming conventions we use:

- `inEvt` denotes action of the subscriber component to receive an input event.
- `outEvt` denotes action of the publisher component to issue an output event.
- `issueGDCall` denotes action of the caller component to “issue GetData() call”.
- `receiveGDCall` denotes action of the callee component to “receive GetData() call”.
- `issueGDReply` denotes action of the callee component to “issue GetData() reply”
- `receiveGDReply` denotes action of the caller component to “receive GetData() reply”.

Similar naming conventions hold for the "SetData()" calls such as `issueSDCall`, `receiveSDCall`, `issueSDReply` and `receiveSDReply`.

DeviceComponent The DeviceComponent is used to simulate a device that generates its own data (as in sensor reports). Upon receiving a Push(), this component does a Push() if it is specified as an event supplier. In a typical scenario, this component is specified as an event consumer of interval timeouts, so as to simulate the device generating information on a periodic basis.

```
DeviceComp = (inEvt->outEvt->DeviceComp
|receiveGDCall->issueGDReply->DeviceComp).
```

DisplayComponent The DisplayComponent is used to display information to the console window. It is used to simulate any output device in a system. Upon receiving a Push(), this component does a Get() on each component specified in its receptacles. This component then displays the results on the console.

```
DisplayComp = (inEvt->issueGDCall->receiveGDReply
->display->DisplayComp).
```

ClosedEDComponent The ClosedEDComponent is closed in the sense that other components cannot alter its data via Set() operations. "ED" stands for *event driven*. Upon receiving a Push(), this component does a Get() on each component specified in its receptacles. This component then generates a Push() if it is specified as an event supplier

```
ClosedEDComp = (inEvt->issueGDCall->receiveGDReply->outEvt
->ClosedEDComp
|receiveGDCall->issueGDReply->ClosedEDComp).
```

OpenEDComponent The OpenEDComponent is open in the sense that other components can set its data. Upon receiving a Set(), this component does a Get() on each component specified in its receptacles. This component then generates a Push() if it is specified as an event supplier.

```
OpenEDComp = (inEvt->issueGDCall->receiveGDReply
->outEvt->OpenEDComp
|receiveGDCall->issueGDReply->OpenEDComp
|receiveSDCall->issueGDCall->receiveGDReply
->issueSDReply->outEvt->OpenEDComp).
```

LazyActiveComponent The LazyActiveComponent is used to simulate delayed response to acquiring data. As an optimization strategy, if a component is updated more than it is read, the Lazy Active pattern may be used to only update the data is a request is made. Upon receiving a Push(), this component flags its data as invalid. When this component's Get() is called this triggers a the LazyActiveComponent to call Get() on the components attached to the receptacles of the LazyActiveComponent.

```
LazyActiveComp = (inEvt->outEvt->DataStale
|receiveGDCall->issueGDReply->LazyActiveComp),

DataStale= (receiveGDCall->issueGDCall->receiveGDReply
->issueGDReply->LazyActiveComp).
```

ModalComponent The ModalComponent is used to alter the flow of events. The component can be enabled and disabled via the facet method ChangeMode(). When it is enabled, it will update and generate an event when it receives an event. When it is disabled, it will not update or generate an event.

```
ModalComp = Enabled,

Disabled = (enable->Enabled|disable->Disabled |inEvt->Disabled),

Enabled = (enable->Enabled|disable->Disabled
|inEvt->issueGDCall->receiveGDReply->outEvt->Enabled
|receiveGDCall->issueGDReply->Enabled).
```

PassiveComponent The PassiveComponent does not have an event consumer. When a Set() operation is called on the PassiveComponent, it updates and issues an event if it has a configured event supplier.

```
PassiveComp = (receiveSDCall->issueSDReply->outEvt->PassiveComp).
```

PushDataSourceComponent Upon receiving a Push(), the PushDataSourceComponent calls Set() on all of the components in the receptacles of the PushDataSourceComponent.

```
PushDataSrcComp =
(inEvt->issueSDCall->receiveSDReply->PushDataSrcComp).
```

4.2 Modeling of Component Interactions

4.2.1 Control-Push/Data-Pull

For the convenience of the reader, we duplicate discussion of the control-push/data-pull interaction style in Section 3.2.

ESML is based on Real-Time Event Channel implemented in the TAO CORBA [76]. Components are composite objects with ports, which interact with one another, either through event triggers or procedure invocations. The CORBA Component Model (CCM) terminology is adopted, where each component can have the following types of ports:

- *Publish Port* to publish events.
- *Subscribe Port* to subscribe to events.
- *Receptacle* to issue method invocations.
- *Facet* to accept method invocations.

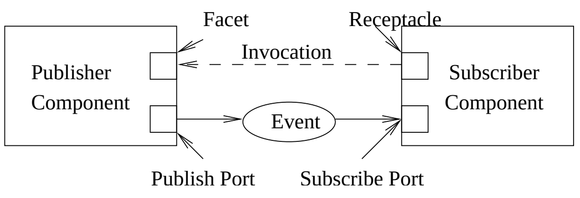


Figure 4.1: The control-push/data-pull style of interaction.

Component interaction typically (but not always) follows the *control-push data-pull* style as shown in Figure 4.1. First, the data producer component publishes a *DataAvailable* event from its publish port indicating that it has fresh data; when the data consumer component receives the event from its subscribe port, it issues a *GetData()* call from its receptacle to the producer's facet to retrieve the data. Below is FSP model for this interaction style:

```
Publisher = (outEvt->Publisher |
receiveCall->issueReply->Publisher).

Subscriber = (inEvt->issueCall->receiveReply ->Subscriber).

||ControlPushDataPull = (pub:Publisher||sub:Subscriber)
/{pub.outEvt/sub.inEvt, sub.issueCall/pub.receiveCall,
sub.receiveReply/pub.issueReply).
```

This approach treats the interaction between an event publisher and an event subscriber as *synchronous*, that is, the `outEvt` of the publisher synchronizes with the `inEvt` of the subscriber directly. Note that this is an abstraction of what actually happens in the real system, since we are hiding a lot of details such as message queuing and thread scheduling. We have to make a tradeoff between the level of abstraction and scalability. If we model everything in detail, then its not possible to achieve reasonable level of scalability. This level of abstraction turns out to be adequate for our purposes.

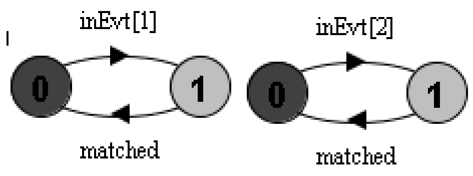


Figure 4.2: The correlator for two input events with AND synchronization.

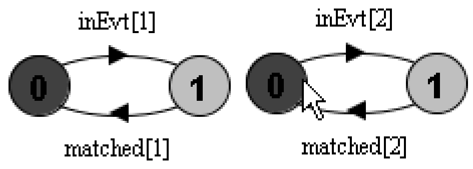


Figure 4.3: The correlator for two input events with OR synchronization.

4.2.2 Input Event Correlation

When a component subscribes to multiple events, there may be two types of synchronization patterns: `AND` synchronization means that the component must receive all input events to be triggered; `OR` synchronization means that the component only needs to receive one of the input events to be triggered. In order to model `AND` synchronization, we add a new process type called `InputANDCorrelator`, as shown below and in Figure 4.2:

```
Event(ID=1) = (inEvt[ID]->matched->Event).

||InputANDCorrelator(NumInputs=1) = (if(NumInputs>0) then (forall
[i:1..NumInputs] Event(i))).
```

It models parallel composition of `NumInputs` number of processes `Event`, which all synchronize on the same event `matched`. This ensures that the `matched` event is emitted only when all input events `inEvt[i..NumInputs]` occur. The `matched` event is in turn used to trigger the input event of the subscriber component.

`OR` synchronization can be modeled similarly, only replacing the shared action `matched` with different actions for each process `Event`,as shown below and in Figure 4.3.

```
Event(ID=1) = (inEvt[ID]->matched[ID]->Event).

||InputORCorrelator(NumInputs=1) = (if(NumInputs>0) then (forall
[i:1..NumInputs] Event(i))).
```

4.2.3 Real-Time Issues

The typical way to model real-time systems in FSP is to use an action `tick` that is shared among all the processes in the system to provide a system-wide heartbeat. This means that time needs to be discretized into uniform segments such as 1 millisecond or 1 second. The typical execution time of components in AMC is fairly small, in the range of microseconds, while the typical period of execution is fairly large, in the range of milliseconds or even seconds. If we model the timing aspects accurately by using a fine-grained partitioning of time on the microsecond scale, the system state space will quickly explode even for trivially small examples. Therefore we do not aim to model the timing aspects in a fully accurate way. Instead, we only ensure that the relative execution frequencies of different rate groups are correct, e.g., the 20Hz thread should execute 20 times more frequently than the 1Hz thread. Below is the way we model a system consisting of both a 20Hz timer and a 1Hz timer. We assume the event `timeout20Hz` is emitted at every clock tick, that is, there is a 50ms time interval in between every two clock ticks. Then the event `timeout1Hz` is emitted every 20 clock ticks.

```
Timer20Hz = (timeout20Hz->tick->Timer20Hz).
Timer1Hz = (timeout1Hz->Delay20[1]),
Delay20[t:1..20] = (when(t==20) tick->Timer1Hz
|when (t < 20) tick->Delay20[t+1]).
```

Since we do not model detailed timing information such as component execution times or scheduling policies, we cannot check for quantitative timing properties such as the worst-case response times and frame overruns. We assume these properties can be checked via other methods such as real-time scheduling theory or a real-time model-checker. Instead, we focus on the functional properties such as deadlock freedom or reachability. We assume that frame overruns do not occur, that is, all threads finish within their frame time, and encode this assumption in the model by adding an explicit synchronization event between the timer and the end of thread processing, typically a `display` event, as shown below:

```
Timer20Hz = (timeout20Hz->timer20HzDone->tick->Timer20Hz).

Thread20Hz = (...||tacDisplay:displayComp)
/{tacDisplay.display/timer20HzDone}.
```

This ensures that the next timeout event will not occur until all events belonging to the current frame have all been processed.

4.2.4 An Example Application Scenario

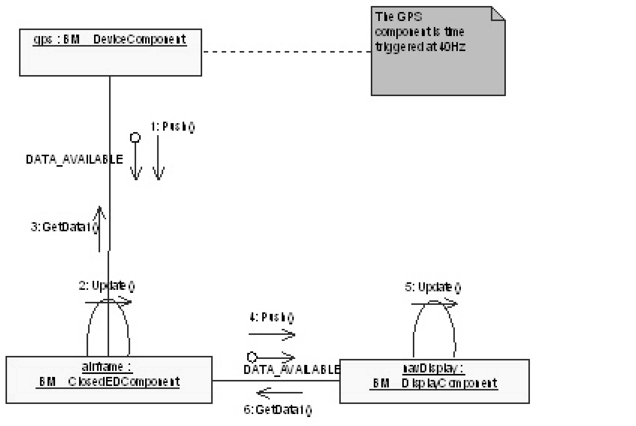

gps : BM_DeviceComponent
The GPS component is time triggered at 40Hz
1: Push()
DATA_AVAILABLE
3:GetData1()
2: Update()
5: Update()
4: Push()
DATA_AVAILABLE
6:GetData1()
airframe : BM_ClosedEDComponent
navDisplay : BM_DisplayComponent
cursorDevice : BM_DeviceComponent
The cursor component is time triggered at 20Hz
9:GetData1()
7: Push()
8: Update()
11: Update()
10: Push()
12:GetData1
selectedPoint : BM_ClosedEDComponent
tacticalDisplay : BM_DisplayComponent

Figure 4.4: The MultirateSP (Multi-Rate Single-Processor) scenario. This figure is taken from Boeing documentation [82].

As an illustrating example, we consider the application scenario in Figure 4.4, called *MultirateSP*, which stands for the *multi-rate single-processor* scenario. There are two execution rates in the system. With rate 40Hz, the system must update navigation displays with timely airframe position information using inputs from navigation sensors. Concurrently executing with rate 20Hz, there is also a device that captures the pilot's cursor position that is a point of interest for weapon release. When the position of the cursor updates, the position on the tactical display must be updated. In the 40Hz thread, the `gps` component pushes a `DataAvailable` event to the `airframe` component, which updates its state by getting data from `gps`. The `airframe` component then pushes a `DataAvailable` event to the `navDisplay` component, which then updates the display by getting data from `airframe`. A similar chain of three components runs in sequence in the 20Hz thread. Below is the complete FSP specification for this scenario:

```
Timer40Hz = (timeout40Hz->timer40HzDone->tick ->Timer40Hz).

Timer20Hz = (timeout20Hz->timer20HzDone->tick ->tick->Timer20Hz).

ClosedEDComp = (inEvt->issueGDCall->receiveGDReply->outEvt
->ClosedEDComp |receiveGDCall->issueGDReply->ClosedEDComp).

DisplayComp = (inEvt->issueGDCall->receiveGDReply->display
->DisplayComp).

DeviceComp = (inEvt->outEvt->DeviceComp |
receiveGDCall->issueGDReply->DeviceComp).

||Thread40Hz = (Timer40Hz||gps:DeviceComp
||airframe:ClosedEDComp||navDisplay:DisplayComp)
/{timeout40Hz/gps.inEvt, gps.outEvt/airframe.inEvt,
airframe.issueGDCall/gps.receiveGDCall,
airframe.receiveGDReply/gps.issueGDReply,
airframe.outEvt/navDisplay.inEvt,
navDisplay.issueGDCall/airframe.receiveGDCall,
navDisplay.receiveGDReply/airframe.issueGDReply,
navDisplay.display/timer40HzDone }.

||Thread20Hz = (Timer20Hz||cursorDevice:DeviceComp
||selPoint:ClosedEDComp ||tacDisplay:DisplayComp)
/{timeout20Hz/cursorDevice.inEvt,
cursorDevice.outEvt/selPoint.inEvt,
selPoint.issueGDCall/cursorDevice.receiveGDCall,
selPoint.receiveGDReply/cursorDevice.issueGDReply,
selPoint.outEvt/tacDisplay.inEvt,
tacDisplay.issueGDCall/selPoint.receiveGDCall,
tacDisplay.receiveGDReply/selPoint.issueGDReply,
tacDisplay.display/timer20HzDone}.

||SYSTEM = (Thread40Hz || Thread20Hz).
```

4.3 Specification of System Properties

There are two types of properties that can be checked within LTSA: *safety* and *liveness*. A safety property asserts that nothing bad happens, and a liveness property asserts that something good eventually happens. Example safety properties are mutual exclusion and deadlock freedom. An example liveness property is: if a process requests a resource, it must eventually get it, i.e., there is no starvation.

4.3.1 Deadlock Freedom

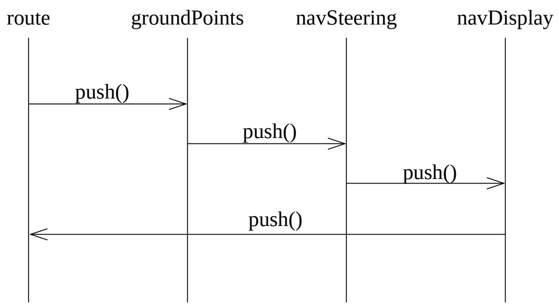


Figure 4.5: A deadlock situation caused by a dependency cycle.

The most obvious property to check is that the system is free of deadlocks. LTSA has a built-in capability to check for deadlocks. For illustration purposes, we artificially introduce a deadlock situation into a good application scenario by adding an extra event push from `navDisplay` to `route`, as shown in Figure 4.5, where all events are *synchronous*. Typically, events delivered by the CORBA Event Service are *asynchronous*, that is, the publisher hands off the event to Event Service and continues its own execution. The Event Service delivers the event to the subscriber later. When components execute locally on the same processor, it is possible to perform an optimization to bypass the Event Service and make events *synchronous*. That is, the sender waits for the event to be delivered to the receiver before resuming execution, just like a synchronous method call. Deadlock occurs when `navDisplay` pushes an event to `route`, which is still blocked waiting for the return of its previous push event to `groundPoints`. LTSA detects this situation and prints out an error trace, which allows us to construct the scenario in more user-friendly notations such as the UML Sequence Diagram. Note that there is no deadlock if we do not make the events synchronous, even when there is an event dependency cycle.

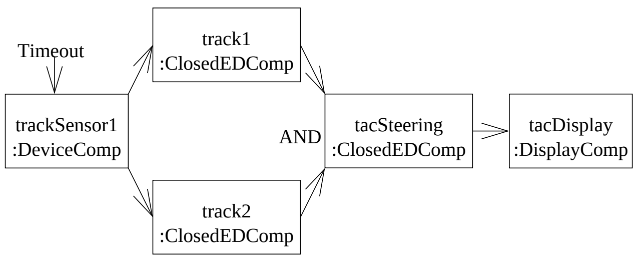


Figure 4.6: A fragment of the 20Hz thread in the MediumSP application Scenario.

Another possible cause of deadlock is when a component subscribes to multiple input events with `AND` synchronization, but for certain reasons not all of the input events are available. Figure 4.6 shows a hypothetical application scenario. Component `trackSensor` is triggered periodically by the 20Hz timeout, and issues an output event that is subscribed to by both `track1` and `track2`. Component `tacSteering` subscribes to output events of both `track1` and `track2` with `AND` synchronization, and issues an event to trigger `tacDisplay`. LTSA reveals no deadlocks in this scenario. Here is the FSP specification:

```
Timer1Hz = (timeout1Hz->timer1HzDone->Timer1Hz).

DeviceComp = (inEvt->outEvt->DeviceComp
|receiveGDCall->DeviceComp).

ClosedEDComp = (inEvt->issueGDCall->outEvt
->ClosedEDComp|receiveGDCall->ClosedEDComp).

DisplayComp = (inEvt->issueGDCall->display->DisplayComp).

Event(ID=1) = (inEvt[ID]->matched->Event).
||InputCorrelator(NumInputs=1)= (if(NumInputs>0) then
(forall[i:1..NumInputs] Event(i))).

||System = (Timer1Hz||trackSensor1:DeviceComp
||track1:ClosedEDComp||track2:ClosedEDComp
||correlatorTS:InputCorrelator(2) ||tacSteering:ClosedEDComp
||tacDisplay:DisplayComp) /{timeout1Hz/trackSensor1.inEvt,
trackSensor1.outEvt/track1.inEvt,
trackSensor1.outEvt/track2.inEvt,
track1.issueGDCall/trackSensor1.receiveGDCall,
track2.issueGDCall/trackSensor1.receiveGDCall,
track1.outEvt/correlatorTS.inEvt[1],
track2.outEvt/correlatorTS.inEvt[2],
correlatorTS.matched/tacSteering.inEvt,
tacSteering.issueGDCall/track1.receiveGDCall,
tacSteering.issueGDCall/track2.receiveGDCall,
tacSteering.outEvt/tacDisplay.inEvt,
tacDisplay.issueGDCall/tacSteering.receiveGDCall,
tacDisplay.display/timer1HzDone}.
```

For illustration purposes, let's make a change to the system by letting `trackSensor1` be a modal component that publish two types of events `outEvt1` and `outEvt2`, choosing non-deterministically which event to output at runtime depending on its mode. This could be used to model a modal component which outputs different events depending on its active mode, i.e., `OutEvt1` triggers `track1` and `outEvt2` triggers `track2`. This change results in a deadlock situation since `tacSteering` can only receive one of its two input events:

```
DeviceComp = (inEvt->outEvt1->DeviceComp|
inEvt->outEvt2->DeviceComp |receiveGDCall->DeviceComp).
...
||System = (Timer20Hz||trackSensor1:DeviceComp
||track1:ClosedEDComp||track2:ClosedEDComp
||correlatorTS:InputCorrelator(2) ||tacSteering:ClosedEDComp
||tacDisplay:DisplayComp)
/{timeout20Hz/trackSensor1.inEvt,
trackSensor1.outEvt1/track1.inEvt,
trackSensor1.outEvt2/track2.inEvt,
...}.
```

LTSA outputs this trace to deadlock:

```
Trace to DEADLOCK:
    timeout20Hz
    trackSensor1.outEvt1
    track1.issueGDCall
    track1.outEvt
```

4.3.2 Reachability

Each component should be triggered/invoked at least once during each execution cycle. Otherwise the component is redundant, which could signal a design error or inefficiency that wastes system resources. In order to check that a component `C` is indeed triggered, we introduce a property `NotReachable` stating that the event `e` that signifies the triggering or execution of component `C` never occurs. If this property holds, then indeed `e` is not reachable; otherwise, LTSA returns a counter example showing the path of execution leading to the event `e`. For example, we would like to check that the action `navDisplay.display` is executed/reachable:

```
property NotReachable = STOP+{reachable}.

||CheckReachability = (System || NotReachable)
/{navDisplay.display/reachable}.
```

Checking this property for the MultirateSP scenario yields this result:

```
Trace to property violation in NotReachable:
    timeout40Hz
    gps.outEvt
    airFrame.issueGDCall
    airFrame.receiveGDReply
    airFrame.outEvt
    navDisplay.issueGDCall
    navDisplay.receiveGDReply
    navDisplay.display
```

4.3.3 Sequencing Constraints

Certain events should happen in sequence. For example, the events in a linear chain of event triggers should happen in the order of precedence relation from the head to the tail of the chain. Below is the property specification used to check the correct ordering of events in the 40Hz thread:

```
property SeqConstraint = (evt1->evt2->evt3->evt4->SeqConstraint).
||CheckSeqConstraint = (SYSTEM||SeqConstraint)
/{timeout40Hz/evt1,
gps.outEvt/evt2, airframe.outEvt/evt3, navDisplay.display/evt4}.
```

LTSA reports no violations for this property. However if we change the property to:

```
property SeqConstraint = (evt1->evt2->evt3->evt4->SeqConstraint).
||CheckSeqConstraint=(SYSTEM||SeqConstraint)
/{timeout40Hz/evt1,
airframe.outEvt/evt2, gps.outEvt/evt3, navDisplay.display/evt4}.
```

Then LTSA reports a trace to property violation:

```
Trace to property violation in SeqConstraint:
    timeout40Hz
    gps.outEvt
```

If some events may happen in parallel, that is, the events form a general graph instead of a linear chain, then we can only specify those events that do form a linear chain, since LTSA does not allow non-determinism in property specifications. For demonstration purposes, assume that there is no precedence relation between `airframe.outEvt` and `gps.outEvt`, but both must follow `timeout40Hz` and precede `navDisplay.display`, then we should write the sequencing constraint as follows:

```
property SeqConstraint = (evt1->evt2->evt4->SeqConstraint).
||CheckSeqConstraint = (SYSTEM||SeqConstraint)
/{timeout40Hz/evt1, gps.outEvt/evt2, navDisplay.display/evt4}.

property SeqConstraint2 = (evt1->evt3->evt4->SeqConstraint2).
||CheckSeqConstraint2 = (SYSTEM||SeqConstraint2)
/{timeout40Hz/evt1, airframe.outEvt/evt3, navDisplay.display/evt4}.
```

LTSA reveals no violations of these constraint specifications.

4.3.4 Progress Property

The properties discussed so far are all *safety properties*, that is, they can be verified by detecting if a bad state is reached given a *finite* execution sequence. On the other hand, *liveness properties* can only be verified for an infinite execution sequence. A general treatment of liveness involves using a temporal logic to specify liveness properties. A restricted class of liveness properties is the *progress* property in the form of `progress P = (a1, a2, ..., an}`, which asserts that in an infinite execution of a system, at least one of the actions `a1, a2, ..., an` will be executed infinitely often. It is useful for verifying that a system does not contain starvation of certain actions. It is a stronger assertion than reachability, which only requires that certain actions are executed *at least once* during the system's lifetime.

For example, in order to check that `navDisplay.display` is executed infinitely often in any infinite execution of the MultirateSP scenario, we can add one line to the FSP model:

```
progress P1 = {navDisplay.display}.
```

LTSA reports that no progress violations are detected.

4.4 Scalability Improvements

Perhaps the single biggest impediment to adoption of model-checking in industry practice is state space explosion. We have constructed the FSP model for the MediumSP scenario, but checking any properties on the model results in an out-of-memory error in the model-checker. Since there is no synchronization between `Thread20Hz` and `Thread1Hz`, we focus on `Thread1Hz`. We have applied some techniques to improve scalability, as explained in the following sections. After applying these techniques, we were able to check the properties of this scenario, such as deadlock freedom, component sequencing and reachability, etc.

4.4.1 Exploiting Domain-Specific Constraints

We can take advantage of certain domain-specific constraints, such sa the control-push, data-pull interaction style, to simplify the model. Normally method calls are modeled with a two-way synchronization between the *caller* component and the *callee* component, as shown below in the FSP model:

```
Publisher = (outEvt->Publisher
|receiveGDCall->issueGDReply->Publisher).

Subscriber = (inEvt->issueGDCall->receiveGDReply->Subscriber).

||CtrlPushDataPull = (pub:Publisher||sub:Subscriber)
/{pub.outEvt/sub.inEvt, sub.issueGDCall/pub.receiveGDCall,
sub.receiveGDReply/pub.issueGDReply).
```

However, for all practical purposes we can treat the GetData() call and reply as an atomic operation, and omit the synchronization action on the method call reply:

```
Publisher = (outEvt->Publisher |receiveGDCall->Publisher).

Subscriber = (inEvt->issueGDCall->Subscriber).

||CtrlPushDataPull = (pub:Publisher||sub:Subscriber)
/{pub.outEvt/sub.inEvt, sub.issueGDCall/pub.receiveGDCall).
```

Note that this optimization may not be generally applicable to all method calls, only to the particular control-push, data-pull interaction style of the Avionics OEP, where there is no action in between the GetData() call and reply. This involves modifying definition of each component type. For example, the specification of `ClosedEDComp` is changed from

```
ClosedEDComp = (inEvt->issueGDCall->receiveGDReply->outEvt
->ClosedEDComp
|receiveGDCall->issueGDReply->ClosedEDComp).
```

to:

```
ClosedEDComp = (inEvt->issueGDCall->outEvt->ClosedEDComp
|receiveGDCall->ClosedEDComp).
```

After applying this optimization, the state space of MediumSP has been reduced significantly. However, this is still too large for LTSA to handle, even on a state-of-the-art UNIX workstation.

4.4.2 Compositional Analysis

Construction of the global state space of an application usually causes state space explosion. We can take advantage of inherent modularity within the application to compose and check the system hierarchically, instead of composing the entire system state space all at once. This is the typical divide-and-conquer approach. This approach is especially suitable for distributed systems, which are generally designed and constructed by decomposition into a hierarchy of simpler components.

LTSA has built-in capability of *Compositional Reachability Analysis* (CRA) [61]. After a set of components have been checked to be correct, we can abstract and reuse them in other contexts by hiding irrelevant events and only exposing those events that may be of interest to other surrounding components. For example, for the 20Hz rate group of the MultirateSP scenario, we can specify:

```
minimal ||AbstractSystem = (SYSTEM)
@{timeout40Hz, airframe.outEvt, gps.outEvt, navDisplay.display}.
```

After abstraction and minimization, the end result is the automaton in Figure 4.7. We can then reuse this automaton as a module in other contexts.

Looking at the 1HZ thread in Figure 3.7, we can see that there is a natural separation into three groups of components:

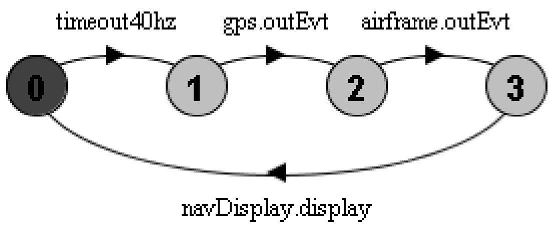


Figure 4.7: The abstracted system of the 40Hz rate group, hiding all other events except timeout40Hz, airframe.outEvt, gps.outEvt and navDisplay.display.

1. earthModel, wayPoint1, wayPoint2, wayPoint3, wayPoint4, wayPoint5, wayPoint6, wayPoint7, wayPoint8, wayPoint9, wayPoint10, leg1, leg2, leg3, leg4, leg5.
2. groundPoint, fltPlan, navSteering, waypointSteering, pilotPref, fltPlanDispl navDisplay, pilotControls.
3. timer1Hz.

We can compose Group1 and Group2 individually while minimizing each group by hiding events that are not relevant to the surrounding components, and finally composing them together with the 1Hz timer. For example, for Group1, we have:

```
minimal || Group1 = (Timer1Hz||earthModel:PushDataSourceComp
||wayPoint1:PassiveComp||wayPoint2:PassiveComp
||wayPoint3:PassiveComp||wayPoint4:PassiveComp
||wayPoint5:PassiveComp||wayPoint6:PassiveComp
||wayPoint7:PassiveComp||wayPoint8:PassiveComp
||wayPoint9:PassiveComp||wayPoint10:PassiveComp
||leg1:LazyActiveComp||leg2:LazyActiveComp
||leg3:LazyActiveComp||leg4:LazyActiveComp ||leg5:LazyActiveComp
||correlatorLeg1:InputCorrelator(3)
||correlatorLeg2:InputCorrelator(3)
||correlatorLeg3:InputCorrelator(3)
||correlatorLeg4:InputCorrelator(3)
||correlatorLeg5:InputCorrelator(2) ||route:OpenEDComp
||correlatorRoute:InputCorrelator(5))

/{...event equivalence specifications omitted}

//Only three interface events are exposed
@{earthModel.inEvt, route.outEvt, route.receiveGDCall}.
```

The minimized state machine is quite simple, consisting of only three states. Intuitively, the whole group of components behaves like a single component that accepts a timeout trigger that synchronizes with `earthModel.inEvt`, does some internal processing that is hidden from outside view, issues `route.outEvt`, and finally receives a GetData() call from its downstream component. The FSP specification for `Group2` is similar and is omitted here. The overall system specification is:

```
||Thread1Hz = (Timer1Hz||Group1||Group2)
/{timeout1Hz/earthModel.inEvt, timeout1Hz/pilotControl.inEvt,
route.outEvt/groundPoint.inEvt,
groundPoint.issueGDCall/route.receiveGDCall,
route.outEvt/fltPlan.inEvt,
fltPlan.issueGDCall/route.receiveGDCall,
navDisplay.display/timer1HzDone,
fltPlanDisplay.display/timer1HzDone }.
```

Given the system specification, we can check properties such as deadlock freedom, reachability, sequencing constraints, etc. For example, to check that *tacDisplay1.display* is indeed executed, that is, the `display()` method of the `tacDisplay` component is invoked, we can check for reachability constraint:

```
property NotReachable = STOP+{reachable}.

||CheckReachability = (System||NotReachable)
/{tacDisplay1.display/reachable}.
```

However, there is one drawback to the compositional analysis approach. Since internal events are hidden inside of each group of components, we cannot check for end-to-end sequencing constraints that span multiple groups and involves internal events from these groups. We can only check constraints that involve interface events that are exposed by the component group, or those that involve internal events of a single group. For example we can check sequencing constraints such as

```
property SeqConstraint = (evt1->evt2->evt3->SeqConstraint).
||CheckSeqConstraint=(Thread1Hz||SeqConstraint)
/{timeout1Hz/evt1, route.outEvt/evt2, tacDisplay1/evt3).
```

or:

```
property SeqConstraint =
(evt1->evt2->evt3->evt4->evt5->SeqConstraint).
||CheckSeqConstraint=(Group1||SeqConstraint)
/{timeout1Hz/evt1, earthModel.issueSDCall/evt2,
wayPoint1.outEvt/evt3, leg1.outEvt/evt4, route.issueGDCall/evt5).
```

but we cannot check for:

```
property SeqConstraint =
(evt1->evt2->evt3->evt4->evt5->evt6->SeqConstraint).
||CheckSeqConstraint=(Thread1Hz||SeqConstraint)
/{timeout1Hz/evt1, earthModel.issueSDCall/evt2,
wayPoint1.outEvt/evt3, leg1.outEvt/evt4, route.issueGDCall/evt5,
tacDisplay1/evt6).
```

The reason is that events `earthModel.issueSDCall`, `wayPoint1.outEvt`, `leg1.outEvt`, `route.issueGDCall` are not visible at the level of `Thread1Hz`.

4.5 Related Work

The publish/subscribe model of computation, as implemented in Real-Time CORBA Event Service [76], has been widely adopted in a variety of application domains, including both real-time embedded systems and enterprise distributed systems. Garlan [25] describes a model-checking framework for publish/subscribe systems. The key feature of this framework is a reusable, parameterized state machine model that captures runtime event management and dispatch policy. Generation of models for specific systems is then handled by a translation tool that accepts as input a set of component descriptions together with a set of properties, and maps them into the framework where they can be checked using the model-checker SMV [66]. Our modeling approach works at a higher level of abstraction and ignores a large amount of detail related to the internals of middleware such as queuing and dispatch policies. For example, we use a single synchronization primitive to represent the flow of an event from the publisher component to the subscriber component, which is a coarse approximation of the actual behavior. This is adequate for our purpose of verifying application-level logical constraints assuming that the middleware behaves correctly, and significantly reduces state space and allows us to check for much larger models. However, a direct comparison of scalability between our work and [25] is not practical since they use different model-checkers.

Karamanolis [49] uses LTSA to model and verify workflow schemas by mapping the Workflow Definition Language into FSP models. There are some similarities between the computational models of workflow schemas and AMC software. Both consist of (possibly hierarchical) components interacting with events sent and received from output and input ports. However, there are also important differences due to the vastly different application domains. For example, AMC software typically contains several rate groups executing periodically, while the life-cycle of a workflow schema only consists of one execution from start to finish. The relative execution frequencies of different rate groups cause the state space of an AMC application to be much larger than a workflow schema specification with similar complexity.

4.6 Summary

In this chapter, we have discussed application of model-checking to modeling and analysis of the AMC software. The documentation provided by our industrial partner describes the application components and scenarios with English prose, which is subject to misunderstanding and misinterpretation. Using the Finite State Processes (FSP) modeling language, we were able to formally model the application, and use the LTSA model-checker to verify properties such as deadlock freedom, component reachability and sequencing constraints. We have also discussed several techniques for coping with the state-space explosion problem. First, we exploit domain-specific properties to reduce the call-return two-way synchronization into a one-way synchronization, thus reducing the number of states of each component. Second, we take advantage of inherent modularity within the application scenario, and use the divide-and-conquer approach to compose the system hierarchically. These techniques showed significant effects in reducing system state space, and allowed us to check a relatively complex application scenario that used to be out of the reach of the model-checker.

Currently the FSP models discussed in this chapter are written manually. In order to integrate model-checking into the MoBIES tool-chain [36], we have done some preliminary work on developing an interpreter that translates an ESML model into its equivalent FSP specification, which consists of two parts. The first part is the fixed *preamble* with FSP specifications for all possible component types, as discussed in Section 4.1. The second part contains composition of components, which are instantiated from component types contained in the preamble. Interactions among components are represented with event equivalence, as discussed in Section 4.2. The interpreter is written in C++ using a set of APIs provided by GME. This corresponds to the *direct implementation* approach [57], where the designer writes C++ code to manually traverse the data structures contained within the source model, and write out the corresponding portions in the target model. As possible future work, it is conceivable to use the more recent *meta-generator* approach [48], which uses *Graph Grammars* and *Graph Rewriting* to develop a mathematically precise mapping from input meta-model to output meta-model.

CHAPTER V

Integrated Approach to Modeling and Analysis of Embedded Real-Time Systems

Embedded software is the software controlling everything around us from telephones and pagers to cars and airplanes. Its main task is to take over what mechanical and dedicated electronic systems used to do, that is, to engage and control the physical world, interacting directly with sensors and actuators. Therefore, embedded real-time systems typically perform information processing tightly coupled with physical processes. The boundary between physical and software processes are often blurred. However, modeling tools tend to focus on either one or the other. The Model-Integrated Computing [88] approach advocates integrated modeling:

> “Computers now control many critical systems in out lives...Such computers wed physical systems to software, tightly integrating the two and generating complex component interactions unknown in earlier systems. Thus, it is imperative that we construct software and its associated physical system so they can evolve together.”

Traditionally, the control engineer designs the control algorithms without consideration of controller platform issues, and then hand them over to the software engineer, who implements them on a minimum cost controller platform while guaranteeing system schedulability for a set of task execution frequency requirements. The authors in [79, 20] propose to break the rigid wall between controller design and software implementation, and adopt an integrated approach, thus opening up the possibility of applying a range of offline optimization and online adaptation techniques. For example, instead of treating each task as having a rigid minimum execution frequency requirement of 40Hz, we can relax it to an interval of [35Hz, 40Hz]. The controller suffers performance degradation with slower execution frequencies as long as it still maintains critical control objectives such as system stability. This enables the designer to perform cost-performance tradeoff analysis.

In this chapter, we propose an integrated approach for modeling and analysis of embedded real-time systems with tight coupling between *embedded* software and *embedding* physical environment, and analyze the real-time scheduling behavior of the software together with the physical system that the software is controlling within the same formal framework of *Timed Petri-Nets* [74]. By adopting this approach, we enable the designer to have an integrated view of the entire system when making design decisions, so she can clearly see the effect of making a change in embedded software on the rest of the system, or a change in the physical system on embedded software design. She can also perform optimization analysis such as maximizing total system utility given resource constraints, or minimizing total system cost given safety and liveness requirements.

We use TPN as the modeling language. We also describe an automated translation procedure from TPN models into TA models, thus enabling the use of mature model checkers for TA such as UPPAAL [8] for TPN analysis. Our approach allows the designer to model and analyze the embedded system in an integrated manner, including the physical system and the software controlling it, and use model-checking to determine schedulability of the software together with system-level timing constraints. We use the well-known Rail-Road

Crossing [41] problem as an application example. Using the model checker UPPAAL, we were able to check the system safety and liveness properties, as well as schedulability of controller software within the same framework. In case a system timing property is violated, UPPAAL provides an error trace leading to the violation state and allows us to gain more insight into the cause of the violation.

In order to gain wider acceptance in industry, it is important to provide highly-automated tool support instead of just algorithms described on paper. We use the Generic Modeling Environment (GME) [57] to provide the capabilities for modeling TPN and TA, as well as for implementation of translators from TPN to TA.

This work is reported in [31, 30]. This chapter is structured as follows: Section 5.1 provides a brief introduction to the two real-time formalisms used, TPN and TA. Section 5.2 presents TPN modeling of real-time scheduling, both non-preemptive and preemptive. Section 5.3 describes a simple algorithm for mapping TPN models into TA. Section 5.4 considers modeling and analysis of the railroad crossing problem. Section 5.6 describes related work. Section 5.5 provides further discussions on the integrated modeling approach, and the chapter concludes with Section 5.7.

5.1 Timed Petri-Nets and Timed Automata

Various timed extensions to Petri Nets (PN) has been proposed, including Ramchandani's *Timed Petri Net* [74], Merlin and Faber's *Time Petri Net* [67], Little and Ghafoor's *Timed Petri Net* [58] and Juan *et al.*'s *Delay Time Petri Net*, etc. We use the term *PN with Time* to refer to the various timed extensions to PN. Even though analysis techniques [86] exist that can perform certain types of timing analysis on certain variants of PN with Time, tool support is generally either not available, or only offers limited analysis capabilities, despite an abundance of tools for analysis of various *untimed* Petri-Nets. On the other hand, there are mature and scalable model checkers for Timed Automata (TA) [4], such as UPPAAL [8] and Kronos [105], that offer sophisticated analysis capabilities for temporal logic specifications of system property. We describe a simple translation algorithm from different variants of PN with Time to semantically equivalent TA models, so that we can leverage the TA model checkers as a universal analysis back-end, instead of having to construct separate tools for each different variant of PN with Time. In this section we describe translation algorithms for two most popular types of PN with time, Ramchandani's *Timed PN* , and Merlin and Faber's *Time PN* [67]. In our opinion, PN has certain advantages over TA in terms of usability, since it has constructs for modeling system structure as well as behavior. It is easy to add in structural and behavioral hierarchy [18, 102], which are absent in TA. However we can still map hierarchical PN models into TA models by flattening the hierarchy, at the expense of losing some clarity and understandability. Hence we propose to use Timed PN as front-end interface for the designer and TA model checkers as back-end analysis engine.

In this chapter, we use Ramchandani's *Timed Petri-Nets* [74](hence referred to as *TPN*) as the unified modeling framework, and translate it into TA. TPN is well-suited for modeling distributed event-triggered systems, since it is intuitive to map events to tokens, and execution of software components to transition firings. The translation procedure also gives a formal semantics for TPN in terms of TA, and clarifies a number of semantic ambiguities in the original TPN definition. In particular, the original definition does not force a transition to fire when it is enabled, while the semantics given by our TA-mapping forces a transition to fire as soon as it is enabled. This turns out to be convenient for modeling real-time scheduling. We also clearly define the semantics of *multiple-enabledness* of a transition as

- Freshly enabling a transition after each firing, which is intuitively the behavior of a task serving multiple queued execution requests.

- *Threshold-based* instead of *age-based*. See Section 5.6 for details.

We refer the reader to Sections 2.2.3 and 2.2.4 for more detailed discussions on TPN and TA.

5.2 Modeling of Real-Time Scheduling with TPN

This section focuses on CPU scheduling of a single-processor system. CPU is an inherently sequential resource; that is, only one task can execute on the CPU at one time. This leads to an interleaving notion of concurrency, and priorities are used for arbitration of competing requests for the shared resource. TPN has *maximum parallelism* semantics, that is, independent transition firings can take place concurrently as if the number of processors available is unlimited. In order to model CPU scheduling, it is necessary to introduce shared places in order to sequentialize the execution of concurrent transitions.

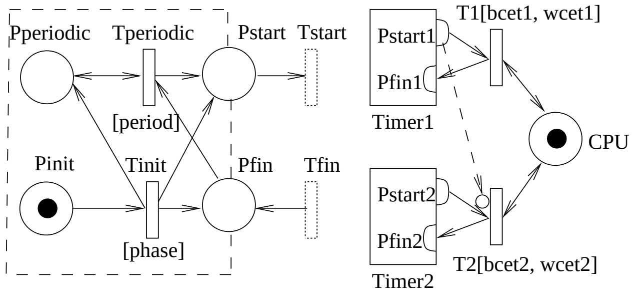


Figure 5.1: A periodic timer in TPN with *period* and initial *phase*. The notation $[phase]([period])$ denotes a delay interval of equal lower and upper bounds $[phase, phase]([period, period])$. A timer with jitter can be easily modeled by using different lower and upper delay bounds for phase and/or period. Note that the name of a place is used to also denote the number of tokens contained in the place.

Figure 5.1 shows the TPN model for a periodic timer. At time 0, T_{init} fires, and puts one token in each of $P_{periodic}$, P_{start} and P_{fin} after $phase$ time units, denoting the initial timer event at time $phase$. T_{start} fires at time $phase$, denoting initial task execution. $T_{periodic}$

fires and consumes tokens in $P_{periodic}$ and P_{fin} at time $phase$, and produces tokens in $P_{periodic}$ and P_{start} at time $phase + period$, signalling the start of another period of task execution. When task execution is finished, a token is deposited in P_{fin}. A *frame overrun* occurs if the task response time is greater than its period. In order to avoid frame overrun, the configuration $(P_{periodic} = 1, P_{fin} = 0)$ must not be reachable.

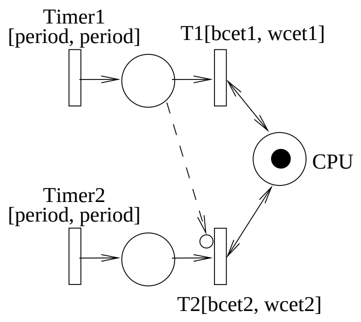


Figure 5.2: Static priority *non-preemptive* scheduling of two periodic tasks. The blocks marked $Timer_1$ and $Timer_2$ are a syntactical shorthand for the TPN model in Figure 5.1. The upper part represents high-priority task $Task_1$; the lower part represents low-priority task $Task_2$.

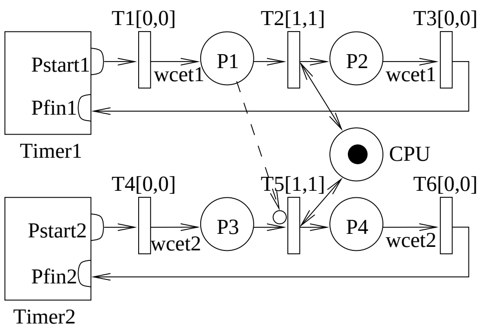


Figure 5.3: Static priority *preemptive* scheduling of two periodic tasks. The upper part represents high-priority task $Task_1$; the lower part represents low-priority task $Task_2$.

Figure 5.2 shows a TPN model for static priority, *non-preemptive* scheduling of two periodic tasks. The place CPU denotes the shared resource of a single CPU. A triggered task executes if CPU is available, i.e., place CPU contains a token; otherwise it waits until the CPU becomes idle. The inhibitor edge from P_{start1} to T_2 models the fact that $Task_1$

has priority over $Task_2$, since a non-empty P_{start1} prevents T_2 from firing.

Figure 5.3 shows a TPN model for static priority, *preemptive* scheduling of two periodic tasks. The edge connecting T_1 to P_1 has weight $wcet_1$, hence firing of T_1 puts $wcet_1$ tokens in P_1. If the CPU is idle, i.e., the place CPU contains a token, then T_2 fires immediately and puts a token in P_2 1 time unites later, meaning that $Task_1$ executes for 1 time unit. As long as P_1 is not empty, $Task_1$ has not finished execution, and continues competing for the CPU. When P_1 becomes empty and P_2 contains $wcet_1$ tokens, transition T_3 becomes enabled and fires immediately, denoting the end of $Task_1$'s execution. Again, the inhibitor edge from P_1 to T_5 models the fact that $Task_1$ has priority over $Task_2$. Note that this approach does not allow us to model execution time intervals for preemptive scheduling. That is, we can only model a single $wcet$ instead of a $[bcet, wcet]$ pair.

5.3 TPN to TA Translation

We formally define a translation algorithm for mapping a TPN model into a semantically equivalent TA model.

1. Declare a global urgent channel go. A transition with an urgent channel as its synchronization label is an urgent transition, and has to be taken as soon as it is enabled without delay.

2. Create an automaton named $Dummy$ with a single location, and a transition with synchronization label $go!$ starting and ending at that location, as in Figure 5.4.

3. For each TPN place $p \in P$, declare an integer global variable with the same name in the TA model.

4. Suppose a TPN transition $t \in T$ has an associated delay interval $[lb, ub]$, a pre-set of k input places $p_1^{in}, \ldots, p_k^{in}$, a post-set of m output places $p_1^{out}, \ldots, p_m^{out}$, and a set of

n inhibitor input places $p_1^{inh}, \ldots, p_n^{inh}$. Classify each $t \in T$ according to its number of input, output and inhibitor places. For example, all transitions with 1 input place, 2 output places, and 1 inhibitor place are put into the same class. For each transition class:

(a) Define a timed automaton template with two locations $disabled$ and $firing$, one local clock c, and $k + m + n$ integer parameters named $p_1^{in}, \ldots, p_k^{in}$, $p_1^{out}, \ldots, p_m^{out}$, $p_1^{inh}, \ldots, p_n^{inh}$.

(b) Add an invariant condition $c \leq ub$ at the location $firing$.

(c) Add an edge from $disabled$ to $firing$ with guard condition $p_1^{in} \geq B(p_1^{in}, t)$, …, $p_k^{in} \geq B(p_k^{in}, t), p_1^{inh} == 0, \ldots, p_n^{inh} == 0$, synchronization label go?, and assignment label $c := 0, p_1^{in} := p_1^{in} - B(p_1^{in}, t), \ldots, p_k^{in} := p_k^{in} - B(p_k^{in}, t)$.

(d) Add an edge from $firing$ to $disabled$ with guard condition $c \geq lb$, and assignment label $p_1^{out} := p_1^{out} + F(p_1^{out}, t), \ldots, p_m^{out} := p_m^{out} + F(p_m^{out}, t)$.

5. In the system configuration section, instantiate one automaton template for each TPN transition, with the appropriate global variables as parameters, representing the input, output and inhibitor places of that transition.

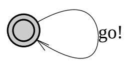


Figure 5.4: Automaton *Dummy* with an urgent transition go.

Figure 5.5 shows the mapping for a TPN transition t with 1 input place *in* and 1 output place *out*. The urgent channel *go* ensures that the automaton changes its state from

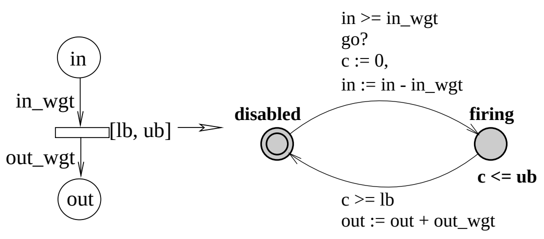


Figure 5.5: TA model of a TPN [74] transition t with 1 input place in, 1 output place out, and time bounds $[lb, ub]$. The process template has argument list (int in, out; const in_wgt, out_wgt; const lb, ub), and a local clock c.

disabled to *enabled* as soon as $in \geq in\_wgt$, that is, the input place *in* contains in_wgt or more tokens. The number of tokens *in* is reduced by in_wgt representing the consumption of tokens in the input place. The TPN transition's firing duration $[lb, ub]$ is modeled by the state *firing* in the TA model, which has an invariant condition $c \leq ub$, and a guard condition $c \geq lb$ on the state change from *firing* to *disabled* that represents the end of transition firing. The resulting semantics is that the automaton has to change its state from *firing* to *disabled* if it has been staying in state *firing* continuously for at least lb time units, and at most ub time units. If the input place *in* contains more than $2 * in\_wgt$ tokens, and the TPN transition is still enabled after one firing, then the urgent channel *go* will immediately force a state change back to *firing* from *disabled*, and the clock is reset to start counting the delay interval $[lb, ub]$ all over again. That is, a new transition is freshly enabled after each firing. Note that this is one of several possible semantics for firing of multiple-enabled TPN transitions [10], which is convenient for modeling a task serving multiple queued execution requests, as well as for modeling preemptive scheduling in Figure 5.3.

Figure 5.6 shows mapping from a TPN transition with 2 input places and 1 output place, and Figure 5.7 shows the mapping for a TPN transition with 1 input place, 1 output place and 1 inhibitor input place. Whenever the place *inh* is not empty, the transition is

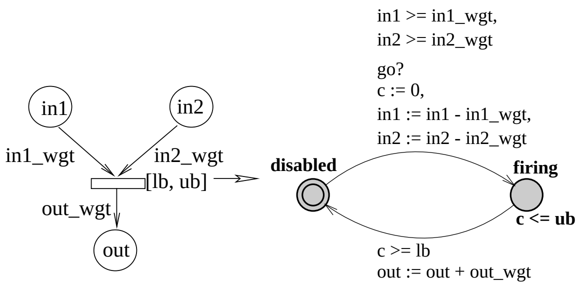


Figure 5.6: TA model of a TPN transition t with 2 input places $in1$, $in2$, 1 output place out, and time bounds $[lb, ub]$. The process template has argument list (int $in1$, $in2$, out; const $in1\_wgt$, $in2\_wgt$, out_wgt; const lb, ub), and a local clock c.

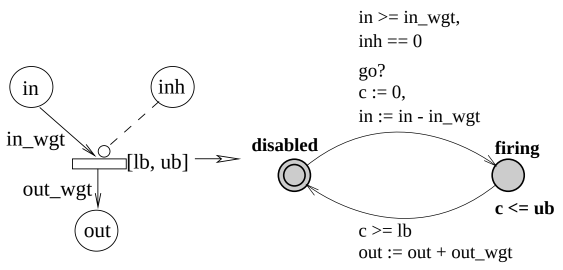


Figure 5.7: TA model of a TPN transition t with 1 input place $in1$, 1 inhibitor input place inh, 1 output place out, and time bounds $[lb, ub]$. The process template has argument list (int in, inh, out; const in_wgt, out_wgt; const lb, ub), and a local clock c.

disabled. Inhibitor arcs can be used to model priorities in resource arbitration, as shown in Figures 5.2 and 5.3.

Figure 5.8 shows mapping from a *Time PN* model, instead of a *Timed PN* model, to a TA model. This enables us to analyze different variants of PN with time in a unified framework by applying different translation rules into TA.

Figure 5.9 shows a simple example taken from [101][1]. In order to translate this TPN model into a TA model, it is simply a matter of instantiating the TA templates for TPN transitions with 1 input/1 output, and 2 input/1 output, which happen to be the only two types of transitions present, as shown in Figure 5.10.

[1]Note that the original example is *Time Petri Net*, while we model a *Timed Petri Net*, so the end-to-end timing properties are different from the original example.

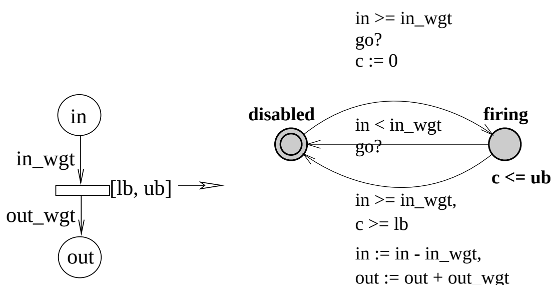


Figure 5.8: TA model of a *Time PN* [67] transition. Compare it to Figure 5.5 to see the difference in semantics between Merlin and Faber's *Time PN* [67] and Ramchandani's *Timed PN* [74]. See the end of Section 5.1 for details.

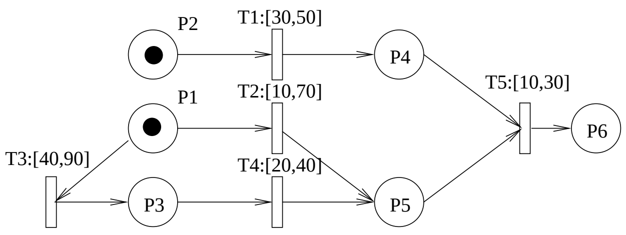


Figure 5.9: Simple TPN modeling concurrency, competition and synchronization.

```
int P1:=1, P2:=1, P3:=0, P4:=0, P5:=0, P6:=0;
urgent chan go;
T1 := T1in_1out(P2, P4, 1, 1, 30, 50);
T2 := T1in_1out(P1, P5, 1, 1, 10, 70);
T3 := T1in_1out(P1, P3, 1, 1, 40, 90);
T4 := T1in_1out(P3, P5, 1, 1, 20, 40);
T5 := T2in_1out(P4, P5, P6, 1, 1, 1, 10, 30);
System T1, T2, T3, T4, T5, Dummy;
```

Figure 5.10: The UPPAAL system definition section that instantiates the templates for the TA model that is translated from the TPN model in Figure 5.9.

UPPAAL does not directly support temporal logic queries for bounded liveness such as $\psi = \phi U_{<t} a$, i.e., property a must hold before t time units, and ϕ must hold until then. In order to check such properties, it is necessary to write an *observer automaton* [8]. This is similar to the approach in the model checker Spin [43], where LTL formulas are transformed into *never claims* that act as system observers that go into error state on detecting violation of the specified property. Figure 5.11 shows the observer automaton

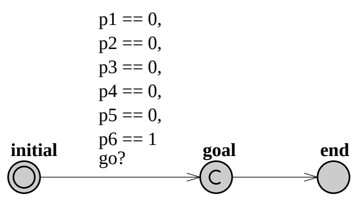


Figure 5.11: An observer automaton that keeps track of the time taken to reach the goal state. It has a local clock *c* that is set to 0 initially. The *goal* state is a *committed* location, meaning that it must be exited within 0 time, and no interleaving of other actions is allowed. Combined with the urgent transition from *initial* to *goal*, this allows us to record the moment when the TPN configuration $(0, 0, 0, 0, 0, 1)$ is first marked as the moment the *goal* state is reached and exited.

that records the time t it takes to reach the goal state $(0, 0, 0, 0, 0, 1)$ from the initial state $(1, 1, 0, 0, 0, 0)$, where the 6-tuple denotes the Petri Net marking $(P_1, P_2, P_3, P_4, P_5, P_6)$. Using the model checker UPPAAL we can prove that t falls within a time interval $[40, 160]$. In order to verify that this is a tight bound, it is necessary to perform three queries due to UPPAAL's lack of *parametric analysis* capability [42]:

1. A[] Observer.goal imply Observer.c $\geq$ 40 and Observer.c $\leq$ 160. This is checked to be true.

2. A[] Observer.goal imply Observer.c $\geq$ 41. This is checked to be false.

3. A[] Observer.goal imply Observer.c $\leq$ 159. This is checked to be false.

The properties that can be verified through transformation from TPN to TA can also be directly verified through state space exploration of the TPN model itself, so we do not claim to add any analytical power by the TPN-to-TA mapping. We are merely proposing to take advantage of mature tools for TA such as UPPAAL for analysis of TPN, as well as other variants of timed extensions of Petri-Nets. Also note that reachability analysis for TA with variables is in general undecidable, so the model-checking procedure is not guaranteed to terminate. This corresponds to the fact that reachability analysis for TPN is in general undecidable.

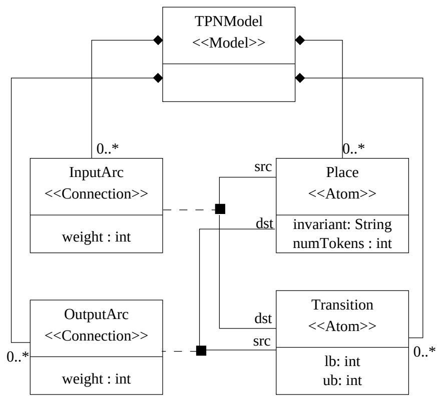


Figure 5.12: The UML-based meta-model for TPN.

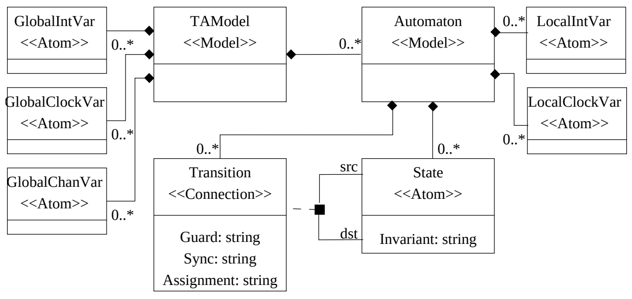


Figure 5.13: The UML-based meta-model for TA.

In order to implement automated tool support for the translation, we take advantage of the Generic Modeling Environment (GME) [57]. Figure 5.12 and 5.13 show the UML-based meta-models for TPN and TA, respectively. GME can generate domain-specific modeling environments for TPN and TA based on the meta-models. We have implemented an interpreter that performs TPN-to-TA translation by syntax-directed mapping of corresponding constructs in the two meta-models. For example, for each transition in the TPN model, an *Automaton* is generated for the TA model; for each place in the TPN model, a *GlobalIntVar* is generated for the TA model, etc. The input file format to UPPAAL is based on XML. Even though GME can export XML files, the DTD (Data Type Definition) files between GME-exported and UPPAAL-input XML files are obviously very different, so it is necessary to write another translator between these two file formats.

5.4 Railroad Crossing Problem

Although the Railroad Crossing (RC) problem is a standard textbook problem in real-time specification and verification, there has been little discussion about the real-time scheduling behavior of the controller computer. That is, it is generally assumed that the controller is dedicated to a single task with no interference from higher-priority tasks or operating systems activity, so there is no need for real-time scheduling theory. This may well be true for the simple controller we are considering here, but in general designers have been putting more and more functionality on a single micro-controller in order to reduce costs. Furthermore, there is also a tendency to take advantage of distributed, multi-processor platforms. In these kinds of complex embedded systems, the real-time scheduling problem is non-trivial to solve, and it is desirable to model the scheduling and runtime platform issues explicitly.

5.4.1 RC without Scheduling

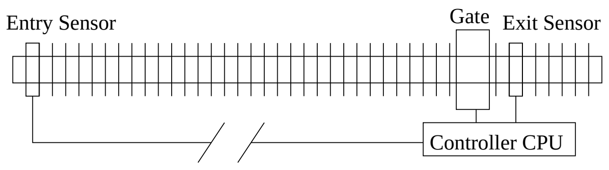
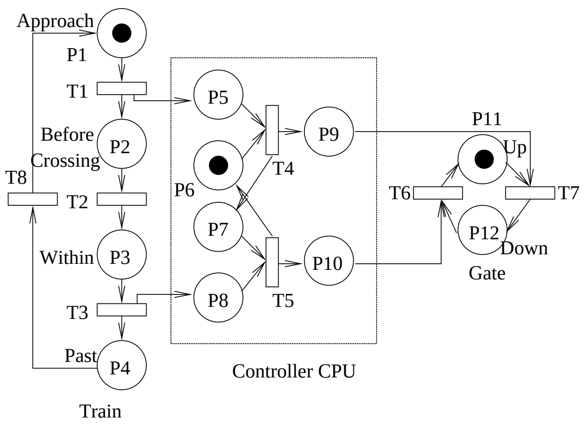


Figure 5.14: Railroad Crossing with a single controller CPU placed near the gate.

The RC problem describes a railroad crossing, whose physical layout is shown in Figure 5.14, and whose behavior is given by the TPN in Figure 5.15. Here we assume that the trains only travel from left to right. The system has to satisfy two properties:

- *safety*: Whenever the train is in the crossing, the gate has to be lowered.



Transition	T_1	T_2	T_3	T_4
[lb, ub]	[1,1]	[4,5]	[1,1]	[1,1]
Transition	T_5	T_6	T_7	T_8
[lb, ub]	[1,1]	[1,2]	[1,2]	[1,1]

Figure 5.15: TPN model of the RC system without consideration of real-time scheduling issues.

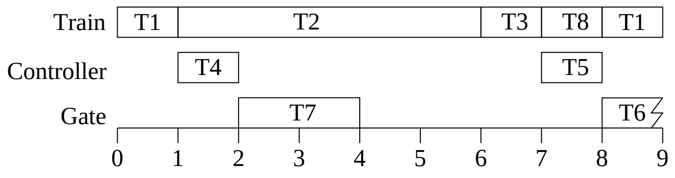


Figure 5.16: Execution trace of the TPN model in Figure 5.15.

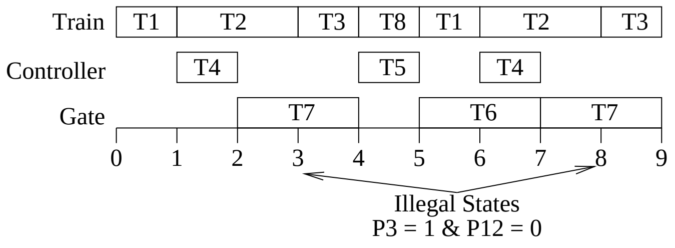


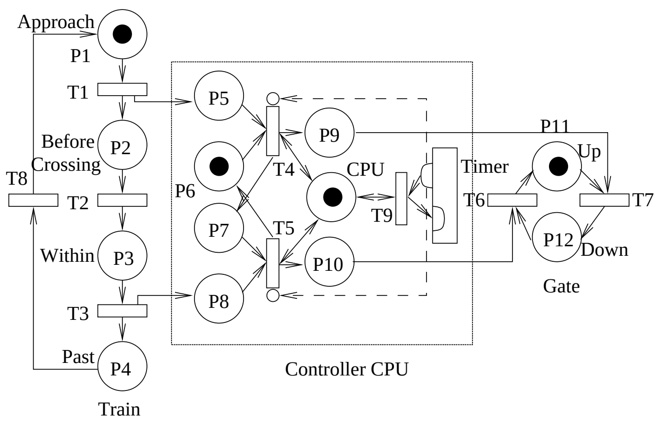
Figure 5.17: Execution trace of the TPN model in Figure 5.15, except T_2's delay interval is changed from [4,5] to [1,2]. That is, it takes shorter for the train to reach the crossing from entry sensor position.

- *bounded liveness*: Within a certain time limit δt after the train leaves the crossing, the gate has to be raised.

An entry sensor is placed some distance before the train reaches the crossing, and an exit sensor is placed a short distance after the train leaves the crossing. When the train crosses the position of the entry sensor (T_1 fires), a signal is sent from the sensor to the controller, which is typically placed near the gate. Upon receiving the signal, the controller sends a *lower-gate* command to the gate (T_4 fires). Upon receiving this command, the gate takes some time to lower itself (T_7 fires). Meanwhile, the train keeps going and enters the crossing (T_2 fires). In order to satisfy the safety requirement, the illegal state (P_3 = 1 & P_{12} = 0) should never be reached. That is, it should never be the case that the train is in the crossing and the gate is not lowered. After the train leaves the crossing (T_3 fires), the exit sensor sends a signal to the controller, which, in turn, sends a *raise-gate* command to the gate (T_5 fires). Upon receiving this command, the gate raises itself (T_6 fires). Note that we are not dealing with the *Generalized Railroad Crossing* [41] problem where multiple trains may be in the crossing at the same time. The TPN model in Figure 5.15 forces the gate to be raised and lowered once for each train going through the crossing.

Given the TPN specification of the RC system in Figure 5.15, we can map the TPN system into a TA model and use UPPAAL to check the system safety and liveness properties. For the safety property, it amounts to checking that (E<> P_3 = 1 and P_{12} = 0) is false. For the bounded liveness property, it is necessary to add an observer automaton similar to the one in Figure 5.11. The system specified in Figure 5.15 satisfies both properties if $\delta t = 3$, that is, the gate has to be raised within 3 time units after the train leaves the crossing. Figure 5.16 shows a possible execution trace. However, if we change T_2's delay interval from [4,5] to [2,3], then the safety property no longer holds. UPPAAL can provide us with an execution trace leading to the safety property violation, as shown in Figure 5.17. In this simple example it is trivial to construct the execution trace manually, but it may not be the case for more complex situations, and a model checker can be invaluable in aiding the designer in finding the execution scenario leading to an error.

5.4.2 RC with Single-Processor Scheduling



Transition	T_1	T_2	T_3	T_4	T_5
[lb, ub]	[1,1]	[4,5]	[1,1]	[1,1]	[1,1]
Transition	T_6	T_7	T_8	T_9	
[lb, ub]	[1,2]	[1,2]	[1,1]	[2,3]	

Figure 5.18: TPN model of the RC system with single CPU controller platform. A high-priority periodic task with period 10, execution time interval [2,3] and arbitrary release phase has been added. The timer block is a syntactical shorthand for the TPN model for a timer in Figure 5.1.

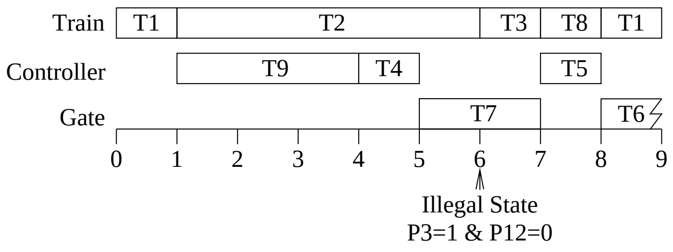


Figure 5.19: Execution trace of the TPN model in Figure 5.18.

In order to make the problem more interesting, we add a high-priority periodic task to the controller CPU. One can think of this task as a timer interrupt handler that demands immediate CPU processing. Figure 5.18 depicts the TPN model of the RC system for the single processor case. Note that we model non-preemptive scheduling for the sake of simplicity. Figure 5.19 shows that addition of the high-priority task results in the violation of safety property.

5.4.3 RC with Multi-Processor Scheduling

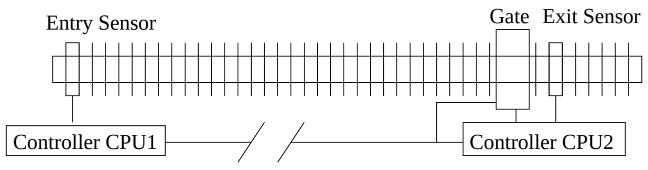
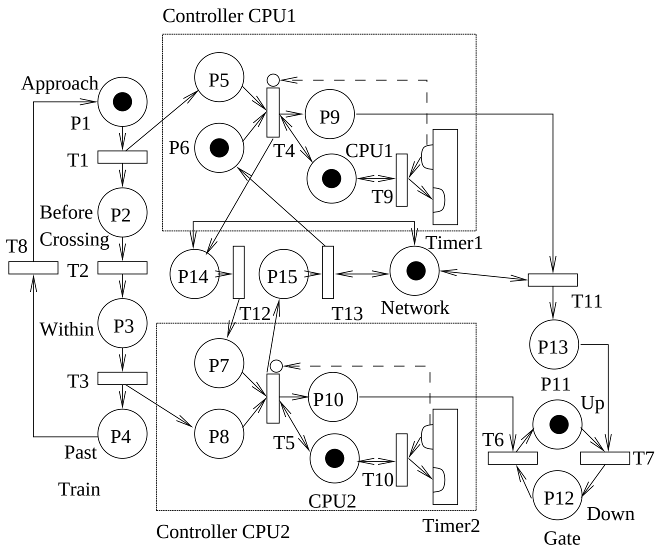


Figure 5.20: Railroad Crossing with a distributed multi-processor controller platform. CPU1 is placed near the entry sensor, and CPU2 is placed near the gate and exit sensor.



Transition	T_1	T_2	T_3	T_4	T_5
[lb, ub]	[1,1]	[4,5]	[1,1]	[1,1]	[1,1]
Transition	T_6	T_7	T_8	T_9	T_{10}
[lb, ub]	[1,2]	[1,2]	[1,1]	[1,1]	[1,1]
Transition	T_{11}	T_{12}	T_{13}		
[lb, ub]	[1,1]	[1,1]	[1,1]		

Figure 5.21: TPN model of the RC system with a multi-processor execution platform. Two high-priority periodic tasks with period 10, execution time interval [1,2] and arbitrary release phase are added, one on each of the two CPUs. The place $Network$ models the shared network connecting CPU_1 to CPU_2 and the gate. Here all the message transmission tasks on the network happen to have the same priority.

Figure 5.20 shows the layout for a multi-processor execution platform, where a controller is placed near the entry sensor that controls lowering of the gate, and another con-

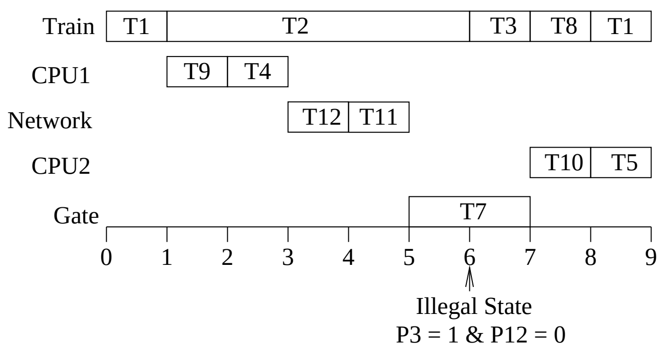


Figure 5.22: Execution trace of the TPN model in Figure 5.21.

troller placed near the gate that controls raising of the gate. Of course this example is only intended to illustrate our modeling techniques, not to model a realistic system design. Figure 5.21 depicts TPN model of the RC system for the multi-processor case with two high-priority periodic tasks added, as well as explicit modeling of the shared network resource. Model-checking reveals that the safety property is again violated. Figure 5.22 shows an execution trace leading to the violation.

The designer has a number of options to remove the safety violation:

- Switch to preemptive scheduling, and assign lower priorities to the two interfering periodic tasks T_9,T_{10} on the controllers, as well as the network task T_{12}.
- Switch to a faster execution platform, including the CPUs and network. For example, reduce the WCET of T_4 to be below 0.5, and the network message transmission latency T_{11} to be below 0.5.
- Impose a reduced speed limit on incoming trains once they reach the entry sensor position, so that the minimum time the train takes to reach the crossing from the entry sensor position is above 6.
- Switch to a more responsive gate so that the time it takes to raise or lower the gate is below 1.

Of course we can adopt a combination of any subset of the above options. In general, model-checking can be used to derive certain timing parameters, whether those of the software or the physical environment, given timing specification for the rest of the system, in order to satisfy system-level requirements. Ideally this requires parametric analysis capability such as that provided by Hytech [42], which is not present in UPPAAL. Still, we can use a trial and error approach, and perform a binary search on the possible intervals of variable values to find out the answer. The RC example is simple enough so that this analysis can be carried out manually, but it may not be the case for more complex situations.

5.5 Further Discussions

One may argue that the proposed integrated modeling approach is unnecessary, since we can use real-time scheduling theory such as Rate Monotonic Analysis (RMA) [53] to find out the worst-case response time (WCRT) of the software processes, and then use it for formal verification of system-level safety and liveness properties. Indeed this is the preferred approach for most embedded systems where software executes periodically with occasional interrupt-driven behavior. We can assume a worst-case minimum inter-arrival time (MIT) for the interrupts and apply rate monotonic analysis [53] to determine the WCRT.

There are a number of reasons for adopting the integrated approach:

- We can draw an analogy between the integrated approach and related work on integrated analysis of real-time scheduling and control system design [79, 20].

- In order to deal with external interrupts, RMA usually assumes a worst-case minimum inter-arrival time (MIT) for the interrupts. However, it may be pessimistic to perform analysis based on assumptions of MIT, since it may not be possible for the worst-case arrival behaviors of different interrupts to happen at the same time due to certain constraints of the physical environment. By explicitly modeling the environment, we can have a more accurate model of interrupt arrivals and reduce or eliminate the above-mentioned pessimism.

- Even though RMA is a mature technology that can deal with complex task systems, it has certain limitations compared to model-checking:
  - RMA requires severe assumptions that cause it either to yield pessimistic results, or not to be applicable to certain scheduling problems, e.g., ADA tasking models with rendezvous-style synchronization [14].
  - RMA analysis focuses almost exclusively on WCRT analysis, while sometimes it is also desirable to analyze the system's best-case response time(BCRT) when task jitter is important. For example, when analyzing an end-to-end distributed transaction, the upstream task's jitter is often a large contributing factor to the downstream task's response times. Due to lack of BCRT analysis techniques, RMA yields overly pessimistic results for these types of distributed systems, causing low system utilization and wastes system resources [11].
  - The parametric analysis capability of model-checkers like HyTech [42] or ACSR-VP [56, 55] enables us to perform reverse queries on the system, for example, how should we modify certain timing parameters in order to satisfy certain system-level timing requirements? This is not straightforward to achieve with RMA.

However, there are also some valid objections to adopting an integrated approach:

- By cleanly separating the control aspect from the software implementation aspect, we achieve separation of concerns and allow control engineers and software engineers to each focus on their area of expertise. This is an instance of the well-known divide-and-conquer approach. In fact this is the current industry practice, and is justified in most situations. The rationale is to achieve simplicity by sacrificing optimality to a certain degree.

- By explicitly modeling the scheduling behavior at the model level, we may further exacerbate the state-space explosion problem that plagues the model-checking approach.

In view of valid arguments on both sides, we believe it is up to the system designer to make informed and intelligent decisions on a case-to-case basis as to which approach to adopt based on various factors, including human resources, tool availability, project time budgets, system physical constraints, etc.

5.6 Related Work

Cortes [16] proposed a mapping algorithm from *PRES+* model, a variant of *Time Petri Nets* with additional data handling capabilities, into HyTech [42] models. Our mapping is simpler and more compositional because we take advantage of UPPAAL's capability of having guard conditions on urgent transitions, which is not present in Hytech. Cortes' mapping algorithm can only deal with 1-safe nets, where each place can contain at most one token, while our algorithm can deal with non-1-safe nets (each place can contain more than one token) and *multiple-enabledness* of transitions. Instead of assuming that the PN is 1-bounded, we can write temporal logic queries in UPPAAL to check for n-boundedness of any place or the entire PN. The ability to model non-1-safe PNs is required to model task queuing and preemptive scheduling, as shown in Figure 5.3. There are also other situations where it may be useful, e.g., Figure 5.23 shows a model used to detect server overload, taken from [10]. Note that our TA-defined semantics for a multiple-enabled transition is *threshold-based* instead of *age-based*, as defined in [10]. That is, in calculating the length of time that the transition has been enabled, we do not keep track of the age of each individual token; instead, we only require the total number of tokens to be above the enabling threshold in order for the transition to be enabled. It has been shown [10] that both semantics are useful in different situations, but the important point is that we have given a clear semantics to an otherwise semantically ambiguous modeling construct by our translation algorithm.

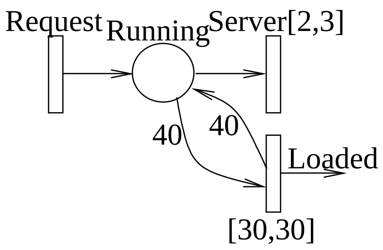


Figure 5.23: A model fragment used to detect server overload. The number of tokens in place *running* represents the number of outstanding requests to be processed at the server. If this number has been greater than 40 for more than 30 time units, an *overload* signal is generated by putting a token in the place *overload*. Note that this is a *Time PN* model [67] instead of a *Timed PN* model [74].

Naedele [68] presented an approach *delegated execution*, which allows modeling and simulation of both functional and scheduling aspects of real-time systems with High-Level Petri Nets (HLPN). The mine-drainage system [13] is used as an application example, which was conducted in CodeSign [22], a tool based on object-oriented real-time HLPN. Due to high expressive power of HLPN, his approach is scalable to larger models, and can model preemptive scheduling more elegantly than our approach. However the analysis technique is limited to simulation; formal analysis via model-checking is not supported.

Gannod [24] described a translation algorithm from untimed PN into PROMELA, the input language of the Spin Model Checker [43] in order to check for liveness and boundedness properties. They used the Domain Modeling Environment (DOME) from Honeywell Technologies to implement the translation algorithms. Our work deals with PN with Time, translates into the input language of UPPAAL instead of Spin, and use Generic Modeling Environment (GME) instead of DOME.

PARAGON [7] is a toolset for visual specification and formal verification of distributed real-time systems. Distinguishing features include the ability to specify resources and their usage by system components, and prioritized execution that allows to express different preemptive and non-preemptive scheduling policies. Unlike the real-time process algebra ACSR [26] underlying PARAGON, TPN lacks inherent notions of priority and preemption, so we have to use a number of *ad hoc* techniques to work around these limitations. First, we use inhibitor arcs to simulate priorities. This approach prevents us from modeling dynamic priority assignments such as Earliest Deadline First (EDF);it also becomes unwieldy when modeling a large number of tasks with distinct priorities. Second, although TPN is a dense-time formalism, unlike ACSR, which is discrete-time, we have to discretize the delay time of a TPN transition in order to model preemption, due to lack of a built-in notion of preemption like that in ACSR, or a stopwatch mechanism like that in Hybrid Automata [42]. Note that our focus is not on formal modeling of real-times scheduling; rather, we are proposing a general approach for integrated modeling and analysis of software and physical processes, which is independent of the underlying modeling formalism.

There are a number of other approaches to formal modeling and analysis of real-time scheduling. In the TIMES tool [5], a discrete transition in an extended TA denotes an event releasing a task, and the guard on the transition specifies all the possible arriving times of the event. The schedulability problem can be then transformed into a reachability problem for TA. Other work includes Corbett's work on timing analysis of Ada tasking programs [14], and Lee's work on real-time process algebra ACSR-VP [56]. None of them has the concept of integrated modeling and analysis.

Wang [101] described a reachability analysis algorithm for TPN that enables computation of end-to-end system timing properties. Our TPN-to-TA translation algorithm can perform verification of more complex system properties in the form of temporal logic specifications, not just reachability. Furthermore, tool support for the algorithms described in [101] is not available.

Many authors have recognized the significance of tight integration of embedded software with its physical environment, and the need for an integrated analysis framework. Seto [79] proposed an integrated approach to controller design and task scheduling, where task frequencies are allowed to vary within a certain range as long as such a change does not affect critical control functions such as maintenance of system stability. An algorithm was proposed that optimizes the overall system control performance while maintaining schedulability by adjusting task frequencies. Similarly, Eker [20] presented a Matlab toolbox for simulation of a real-time kernel in parallel with continuous plant dynamics. The toolbox allows the user to study the interactions between the control tasks and the scheduler, making it possible to experiment with more flexible approaches to real-time control systems, such as feedback scheduling. This body of work deals with controller performance with traditional metrics in control theory such as signal rise time, stability, etc., while our focus is on modeling of system real-time behavior, and static, offline verification through model-checking, although it is possible to take advantage of existing TPN tools or UPPAAL for integrated simulation.

5.7 Summary

In this chapter, we have discussed an integrated approach to modeling and analysis of embedded real-time systems with tight coupling between *embedded* software and *embedding* physical environment, where the physical system and the software artifacts are modeled within the same formal framework. We have also described a translation procedure from TPN models into TA models, thus enabling the use of mature model checkers for TA such as UPPAAL for TPN analysis. We describe implementation of automated tool support within the GME environment. Our approach allows the designer to model and analyze the embedded system in an integrated manner, including the physical system and the software controlling it, and use model-checking to determine schedulability of the software together with system-level timing constraints.

Although we have used a specific modeling formalism (TPN), this approach is general and can be applied together with other real-time and hybrid modeling formalisms that are capable of modeling both the embedded software and the physical environment. For example, we discuss in [32] the application of Hybrid Automata and its associated model-checker HyTech [42] to the Railroad-Crossing problem.

Just like using model-checking for functional verification in Chapter IV, lack of scalability is the most important limiting factor in using model-checking for real-time verification. Even though real-time and hybrid model-checkers such as UPPAAL and HyTech have been in existence a long time ago, their adoption in industry has been slow due to the scalability problem. In order to circumvent this problem, we have done some preliminary investigations [35] into using *simulation* with the widely-used hybrid systems modeling tool Matlab/Simulink [65] for verification, as an alternative to using model-checking with UPPAAL. Simulation is much more scalable than model-checking, and is already widely used in current industry practice. More details are discussed in Chapter VII as part of the future work.

CHAPTER VI

Implementation Synthesis from UML-RT Models

A UML profile is a set of notations that extend or specialize standard UML with mechanisms such as *stereotypes, tagged values,* and *constraints*. UML-RT is a UML profile widely used to develop embedded software in the telecommunications industry. It is essentially an ADL with concepts of components, ports and connectors, embodied in the CASE tool Rational Rose Real-Time (RoseRT) [46]. Despite its name, UML-RT itself does not provide any support for schedulability analysis in order to generate a real-time implementation that meets timing constraints. The commercial code generator provided by RoseRT generates functional code in C++ or Java, but largely ignores real-time issues. It is the job of the engineer to manually map the logical model to RTOS threads and choose a scheduling discipline to satisfy timing constraints. In this chapter, we summarize existing approaches for implementation synthesis of UML-RT models, and describe customized schedulability analysis techniques for the native runtime model of UML-RT, in order to facilitate design-space exploration in finding the most appropriate runtime model for any particular application. This work is reported in [34].

This chapter is structured as follows: Section 6.1 provides a brief introduction to UML-RT and issues related to implementation synthesis. Section 6.2 describes different alternatives for mapping a UML-RT model into a multi-threaded executable. Section 6.3

describes schedulability analysis techniques for the native runtime model of RoseRT. Section 6.4 uses the elevator control system as an application example to illustrate the analysis techniques, and Section 6.5 concludes this chapter.

6.1 Introduction to UML-RT

UML-RT can be characterized as an Architectural Description Language based on UML. It adds architectural concepts to UML such as components, ports, connections, but not quantitative timing information useful for schedulability analysis. As shown in Figure 6.1, UML-RT has the following key concepts:

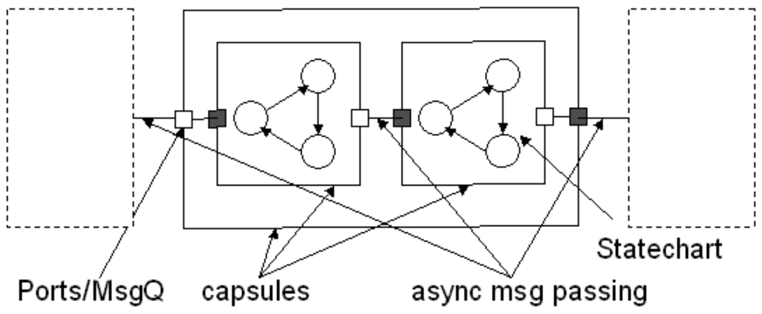


Figure 6.1: The key concepts of UML-RT.

- A *capsule* is an active object with its own logical thread of control, representing an active unit of computation. A capsule typically has a behavior description in the form of an object-oriented version of Statechart called *ROOMChart*, which differs from conventional Statechart as defined by Harel [39] by removing certain features, like instantaneous broadcast of data among parallel state machines, that are difficult to implement in an distributed, asynchronous framework. Instead, parallel state machines are modeled as separate capsules communicating with buffered asynchronous message passing. A capsule may contain other capsules to form a structural hierarchy.

- Explicit representation of *ports*, *protocols* and *connectors* enables construction of architectural models from a collection of capsules.

- A target runtime framework called *TargetRTS* (Target Run-Time System) that serves as a virtual machine to support the UML-RT runtime model. It runs on top of a RTOS to hide the vendor-specific details of execution platform and present a uniform set of APIs to the engineer.

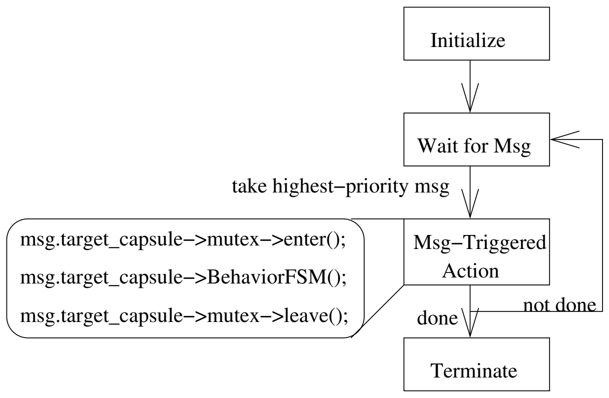

Initialize
Wait for Msg
take highest−priority msg
msg.target_capsule−>mutex−>enter();
msg.target_capsule−>BehaviorFSM();
msg.target_capsule−>mutex−>leave();
Msg−Triggered Action
done
not done
Terminate

Figure 6.2: The runtime behavior of an OS thread that acts as a dispatcher of incoming messages to their destination capsules assigned to this thread.

The model of computation of UML-RT follows the Run-To-Completion (RTC) semantics for each capsule. Once triggered by a message at its input port, the capsule must execute the triggered action to completion before processing the next message. Messages can be assigned priorities and queued in priority order instead of FIFO order. This means that each capsule is a mutually exclusive shared resource, and scheduled with *priority-based non-preemptive* scheduling discipline. Therefore, we have to take into *blocking* time while doing schedulability analysis. One or more capsules can be grouped together and assigned to an operating system thread. As can be seen from Figure 6.2, an OS thread processes incoming messages in a non-preemptive manner, consistent with the RTC semantics of capsules assigned to it. However, there can be preemptions between different threads/tasks in a multi-threaded system. (We use the words *thread* and *task* interchangeably.) A capsule executing in the context of a higher-priority thread can preempt another capsule executing in a lower-priority thread.

Besides capsules, there are also *passive objects* used to encapsulate shared data that do not have their own thread of control, but rather execute in the context of the thread of capsules that invoke methods on them. Ideally we should try to minimize use of passive objects since they introduce concurrency problems associated with shared data, and maximize use of capsules that communicate with other capsules through message passing.

The main target application domain of UML-RT is telecommunication systems, which are generally soft real-time in nature. Perhaps due to this reason, the designers of UML-RT have not put much emphasis on real-time issues when implementing a UML-RT model on the target platform. The default execution model is single-threaded, that is, all capsules are mapped into the same thread of execution. Messages are queued and scheduled non-preemptively in priority-order. It is desirable to introduce more parallelism and concurrency into the system to improve predictability by adopting a multi-threaded execution architecture. It is important to distinguish between the concepts of design-level concurrency and implementation-level concurrency [75]. At the design level, each capsule conceptually contains its own thread of execution, but it does not necessarily have to be mapped into an OS thread at the implementation level. Although it is possible for each capsule to have its own OS thread, it may incur too much context-switching overhead if there are a large number of capsules. A number of alternatives have been proposed for mapping a UML-RT design model into a multi-threaded executable. In this chapter, we discuss these alternatives, and our own approach to schedulability analysis of the native runtime model of UML-RT. The interaction style of active objects communicating through asynchronous message passing is very prevalent in real-time software, for example, the Quantum Framework [93] advocates this programming style without using expensive CASE tools. It simply makes a lot of sense in terms of good software engineering principles such as modularity, encapsulation, decoupling of interactions, etc. Therefore, the issues discussed in this chapter has much wider applicability than just UML-RT and the RoseRT CASE tool.

6.2 Implementation Alternatives for UML-RT Models

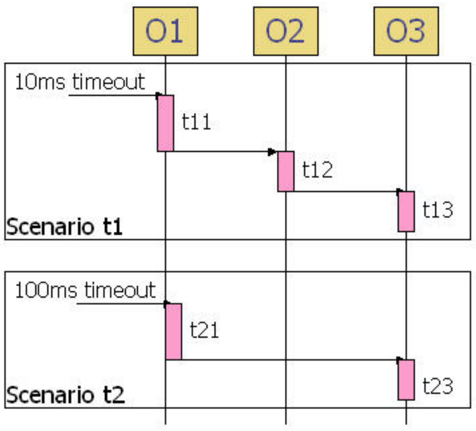


Figure 6.3: An example application scenario in UML-RT.

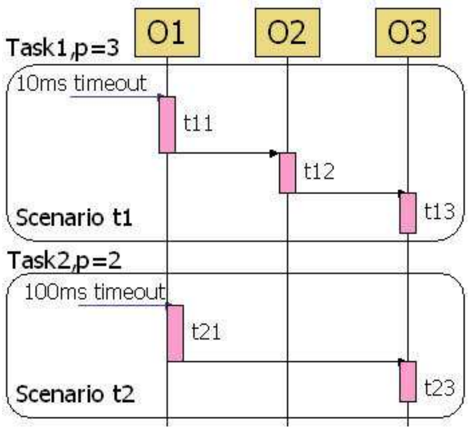


Figure 6.4: Scenario-Based Multi-Threading, Scenario-Based Priority-Assignment (SMSP) for implementation of UML-RT models.

Suppose we have a logical UML-RT model as shown in Figure 6.3, consisting of three

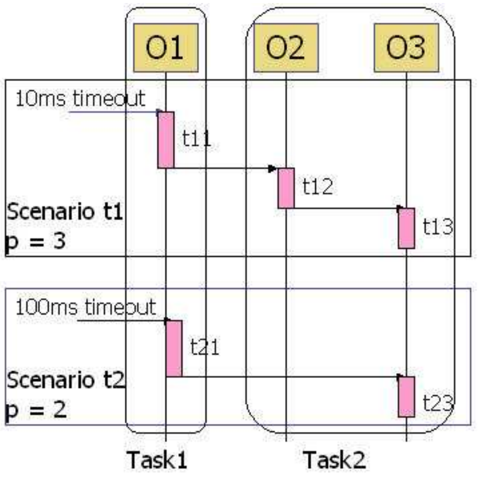


Figure 6.5: Capsule-Based Multi-threading, Scenario-Based Priority-Assignment (CMSP) for implementation of UML-RT models.

capsules O_1, O_2, O_3 and two application scenarios t_1, t_2. Scenario t_1 is initially triggered by a periodic timeout message with period 10m that triggers an action t_{11} in capsule O_1, which in turn sends a message to capsule O_2 and triggers action t_{12} in O_2. Finally, O_2 sends a message to O_3 and triggers action t_{13}. We can view this scenario as a logical end-to-end task t_1 consisting of three precedence-constrained subtasks t_{11}, t_{12} and t_{13}. Similarly, the scenario t_2 is an end-to-end task consisting of two subtasks t_{21} and t_{23} triggered by a 100ms periodic timeout message. There are multiple ways of implementing this model on a multi-tasking RTOS:

6.2.1 Scenario-Based Multi-Threading, Scenario-Based Priority-Assignment

This is proposed by Saehwa Kim in [64]. As shown in Figure 6.4, each application scenario is mapped into a separate thread with uniform priority. Priorities are associated with the end-to-end threads, with statically-assigned priorities. This eliminates the need for dynamic priority adjustments, but creates problems with shared data and necessitates error-prone concurrency control mechanisms. We call this approach *Scenario-based Multi-threading, Scenario-based Priority-assignment* (SMSP).

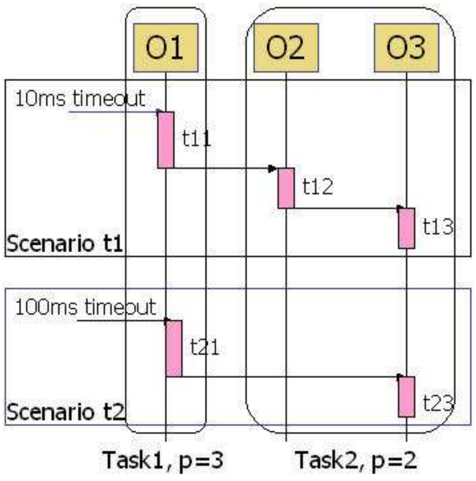


Figure 6.6: Capsule-Based Multi-threading, Capsule-Based Priority-Assignment (CMCP) for implementation of UML-RT models.

6.2.2 Capsule-Based Multi-threading, Scenario-Based Priority-Assignment

This is proposed by Saksena in [75]. As shown in Figure 6.5, one or more capsules are grouped into the same thread. Fixed priorities are associated with the end-to-end scenarios, and thread priority is adjusted dynamically to maintain a uniform priority across each application scenario. We call this approach *Capsule-based Multi-threading, Scenario-based Priority-assignment* (CMSP).

6.2.3 Capsule-Based Multi-threading, Capsule-Based Priority-Assignment

This is the default runtime model of UML-RT as implemented in the RoseRT CASE tool [46]. As shown in Figure 6.6, one or more capsules are grouped into a thread with uniform priority. The figure only shows one of many possibilities for grouping capsules into threads. Two extreme cases are mapping all capsules into a single thread, or mapping each capsule into its own thread. We call this approach *Capsule-based Multi-threading, Capsule-based Priority-assignment* (CMCP). [1] Since each end-to-end periodic scenario does not have uniform priority, the classic rate monotonic analysis technique [53] is not

[1]In addition to application threads, additional *system threads* may be used to implement framework services such as a periodic timer. We do not consider system threads here.

applicable, and the modified HKL algorithm described in this chapter is needed to perform schedulability analysis.

6.2.4 Discussions

Depending on application characteristics, it may be appropriate to adopt different implementation alternatives. The objective is to form threads in such a way to minimize inter-thread interactions and multi-threading overhead.

Scenario-Based Multi-Threading is more appropriate if

- there is little interaction among different application scenarios, or,
- the capsules are fine-grained, so assigning a thread to each capsule may incur too much overhead.

This is the case for Avionics Mission Computing software discussed in Chapter III, which adopts the runtime model of Scenario-Based Multi-Threading and Scenario-Based Priority Assignment.

Capsule-Based Multi-Threading is more appropriate if

- there is intensive interaction and sharing of capsules among different application scenarios, in order to avoid excessive locking and unlocking of shared capsules, or,
- the capsules are coarse-grained, so the overhead associated with assigning a thread to each capsule is acceptable.

Both the CMSP approach, shown in Figure 6.5, and the SMSP approach, shown in Figure 6.4, associate static priorities with end-to-end application scenarios instead of capsules, so that classic RMA techniques can be applied to analyze schedulability. However, they both have some shortcomings:

- The CMSP approach requires the engineer to stick to a programming discipline of dynamically adjusting capsule priorities to reflect the priority of the end-to-end application scenario that is currently executing. This hurts the encapsulation of capsules by mixing system-level concerns (scenarios) with component-level concerns (capsules). It also involves runtime system-call overheads that may not be acceptable to certain resource-constrained embedded systems. Certain small RTOSes may not even provide APIs to dynamically change thread priorities.

- The SMSP approach creates shared data when multiple scenarios cut through the same capsule, and necessitates the use of error-prone concurrency control mechanisms to protect shared data, such as mutexes, semaphores and monitors. This breaks a key advantage of UML-RT, which eliminates the need for such concurrency control mechanisms, by using asynchronous message passing as the main communication mechanism among capsules instead of shared data. It also involves modifying the UML-RT runtime kernel as implemented in RoseRT. Unless the modifications are incorporated into RoseRT, it is unlikely to be acceptable to the average engineers.

In comparison, the CMCP approach has a number of advantages from a software engineering perspective. There is no need for dynamic priority change, or error-prone mutual exclusion mechanisms due to Run-To-Completion semantics of capsules. This makes programming considerably easier. It is also the native runtime model implemented in the RoseRT CASE tool, so a lot of legacy applications are already using this approach. However, classic RMA techniques do not apply, since each end-to-end scenario consists of multiple segments with varying priorities.

The decision to choose a runtime model should be based upon multiple factors, including application characteristics, software engineering benefits, and schedulability. The design space for an application consists of choice of a runtime model, as well as priority assignments to threads. In order to fully explore the design-space, we have adapted the scheduling algorithm developed by Harbour, Klein, Lehoczky, commonly called the HKL algorithm [38], to fit the CMCP runtime model as shown in Figure 6.6. This allows the engineer to determine schedulability of UML-RT models conforming to the CMCP runtime model, and make informed decisions when choosing an appropriate runtime model for his application.

For the CMCP approach, we need to carefully manage the number of threads in order to strike a balance between context-switching overheads due to a large number of threads and blocking time due to insufficient parallelism. We do not deal with the issue of grouping capsules into threads or assigning priorities to threads in this chapter; rather, we focus on the problem of schedulability analysis given a set of capsule-to-thread groupings and priority assignments.

6.3 Schedulability Analysis Technique for CMCP

Consider a UML-RT model consisting of m capsules or active objects $O_1, O_2, \ldots, O_m$, and n end-to-end scenarios or transactions, where each scenario is mapped into an end-to-end *virtual thread*, forming the task set $\tau_1, \tau_2, \ldots, \tau_n$. Each end-to-end virtual thread $\tau_i, i = 1, \ldots, n$ cuts through one or more capsules, and triggers an action within each capsule, forming a chain of subtasks $\tau_{i1}, \ldots, \tau_{im(i)}$. We use $O(\tau_{ij})$ to denote the capsule that the subtask τ_{ij} belongs to, and $PO(\tau_{ij})$ to denote the set of passive objects that τ_{ij} accesses. Each subtask τ_{ij} is actually an event-triggered action within a capsule $O(\tau_{ij})$. We use the word *virtual thread* because each transaction actually consists of multiple segments of event/action pairs distributed over different operating system threads. Due to run-to-completion semantics, a subtask may suffer a blocking time equal to the largest execution time of other subtasks sharing the same capsule. A capsule may also be involved in multiple sub-tasks within one end-to-end virtual thread. We do not explicitly model passive objects, since they execute within the context of the capsules that invoke methods on them, but we do take into account blocking time introduced by them.

Each subtask τ_{ij} is characterized by a set of parameters (C_{ij}, D_{ij}, P_{ij}), where

- C_{ij} is the worst-case execution time.
- D_{ij} is deadline of τ_{ij} relative to the arrival time of task τ_i, taking 0 to be its arrival time.
- P_{ij} is the fixed priority level of τ_{ij}, equal to the priority level of the capsule that τ_{ij} belongs to.

Each end-to-end thread τ_i is characterized by a set of parameters (C_i, D_i, P_i), where C_i is the sum of all the execution times for its subtasks; the deadlines satisfy $0 \leq D_{i1} \leq \ldots \leq D_{im(i)} = D_i$. In most cases subtasks do not have separate deadlines assigned, and there is only one end-to-end deadline. The following assumptions are made:

- Subtasks executing at a given priority level can be preempted by any subtask of higher priority, except when they access the same capsule.
- Threads do not suspend themselves at any instant between their activation and their completion.
- The $(k+1)$th job of τ_i will not execute until the kth job of τ_i has been completed. Furthermore, any subtask τ_{ij} is not ready for execution until subtasks $\tau_{ir}, 1 \leq r < j$ have been completed.

The task model is very similar to the end-to-end tasks with subtasks with varying priority as described by Harbour, Klein, Lehoczky in [38]. We call the schedulability analysis algorithm introduced in [38] the HKL algorithm. But we have to take into account extra blocking time caused by RTC semantics and shared data objects. We first briefly describe the HKL algorithm. Some of the materials here are excerpted from [38].

The *canonical form* of a task τ_i is a new task τ'_i with the same sequence of subtasks as τ_i, but with strictly increasing priorities. τ'_i is obtained by applying the following algorithm, where P'_{ij} denotes the priority of subtask τ'_{ij}.

```
$P'_{im(i)} = P_{im(i)}$;
for $l = m(i)$ downto $2$
 if $P'_{il} < P_{il-1}$ then $P_{il-1} = P'_{il}$
 else $P'_{il-1} = P_{il-1}$
end;
```

One example transformation is a task-chain consisting of subtasks with priority sequence (8, 2, 5, 4, 3). The canonical form of this task-chain consists of priority sequence (2, 2, 3, 3, 3). It was proven in [38] that transforming a task into its canonical form does not affect its schedulability. This result allows one to check whether the canonical form of τ_i is schedulable instead of τ_i itself, which simplifies the analysis considerably.

Now define $Pmin_i$ to be the minimum priority of all subtasks of τ_i. The next step is to classify all tasks $\tau_j, j \neq i$ according to their relative priority levels with respect to $Pmin_i$. For example, if the canonical form of τ_i consists of a single segment of priority 18, and τ_j consists of priority sequence (19, 10, 19, 10, 25, 10), or, (H, L, H, L, H, L), where H stands for "higher or equal", and L stands for "strictly lower". There are three types of tasks [97]:

- Type 1, or H^+, tasks, with all subtask priorities higher or equal to 18. These tasks can preempt task τ_i multiple times.

- Type 2, or $(H^+L^+)^+$, tasks. The first subtask has higher priority than τ_i, but it can only preempt τ_i once, since it is followed by subtasks of lower priority. Multiple tasks of this type may preempt τ_i, but only for the first segment. The non-first priority segments cause a *blocking* effect as with type 4 tasks.

- Type 3, or $((HL)^+H)$, tasks. They differ from type 2 since they end with a high priority segment. We omit the discussion of type 3 tasks, which is quite involved, since they do not appear in the example we consider in this chapter.

- Type 4, or $(L^+H^+)^+L^+$, tasks. The first subtask has lower priority than τ_i. Any one of the following subtask segments can have a *blocking* effect on τ_i, but only one such segment among all tasks of type 4 can have such a blocking effect.

- Type 5, or L^+, tasks. They have no effect on completion time of τ_i, and can be ignored.

Suppose we are calculating response time of task t_i. To simplify discussions, let's assume the canonical form of t_i consists of subtasks of uniform priority P_i. If this assumption does not hold, the general principles discussed here still apply to each individual subtask segment. Define $H_1(i), H_2(i), H_4(i)$ to be the indices of all tasks of type 1, 2, 4, respectively.

For each $j \in H_2(i)$, let $B_2(i,j)$ be the execution time of the *first* H^+ segment of task τ_j. $B_2(i,j)$ denotes the *preemption* time caused by τ_j to τ_i. Then the total preemption time suffered by τ_i is:

$$B_2(i) = \sum_{j \in H_2(i)} B_2(i,j)$$

For each $j \in H_2(i) \cup H_4(i)$, let $B_4(i,j)$ be the *blocking* time suffered by τ_i, caused by all H^+ segments of task τ_j of type 4, and all *non-first* H^+ segments of task τ_j of type 2.

Then the total blocking time suffered by τ_i is:

$$B_4(i) = max(B_4(i,j)|j \in H_4(i) \cup H_2(i))$$

For a Type 2 task, only the first higher priority segment should be counted in $B_2(i)$, while the remaining segments should be counted in $B_4(i)$. Since multiple tasks of Type 2 can use their *first* segments to preempt t_i, therefore $B_2(i)$ is a *sum* of them; while only one task of Type 2 or 3 can use its *non-first* segment to preempt t_i, therefore $B_4(i)$ is a *max* of them.

In order to adapt the HKL algorithm to the UML-RT model, we need to take into account additional blocking time $B(i)$ caused by the RTC semantics of capsules and mutually exclusive access to passive objects.

$$B(i) = \sum_{k,l,j,k!=i,P_{kl}<P_{ij},O(\tau_{kl})=O(\tau_{ij})} C_{kl} + \sum_{m,n,j,m!=i,P_{mn}<P_{ij},PO(\tau_{mn})\cap PO(\tau_{ij})\neq\phi} C_{mn}$$

The first blocking term is due to lower priority application scenarios sharing the same capsule and the RTC semantics. The second term is due to sharing of passive objects.

The equation for calculating the Worst-Case Response Time (WCRT) of task τ_i is:

$$\text{WCRT}(i) = \text{WCET}(i) + B_2(i) + B_4(i) + B(i) + \sum_{j\in H_1(i)} \lceil\frac{\text{WCRT}(i)}{\text{Period}(j)}\rceil \cdot \text{WCET}(j) \quad (6.1)$$

where WCET(j) is the total Worst-Case Execution Time of τ_j, and Period(j) is the execution period of τ_j, if it is a periodic task, or the minimum inter-arrival time of execution triggers for τ_j, if it is a sporadic task.

This is the classic RMA equation [53] with added blocking time terms $B_2(i)$ $B_4(i)$ and $B(i)$. If WCRT(i) is less than deadline of τ_i, then τ_i is schedulable.

6.4 The Elevator Control Application Example

We use the elevator control system as an application example, taken from [28][2]. Figure 6.7 shows the 8 capsules and 1 data object involved in a single-processor implementation. According to the *Capsule-based Multi-threading*, *Capsule-based Priority-assignment* approach, each capsule is assigned a fixed priority. There are three end-to-end scenarios consisting of subtasks of varying priorities:

1. **Stop Elevator at Floor**. The elevator is equipped with arrival sensors that trigger an interrupt to the capsule *arrival sensors interface* when the elevator approaches a floor, which in turn sends a message *approaching floor* to the capsule *elevator controller*. The *elevator controller* invokes a synchronous method call on the passive data object *elevator status and plan* object to determine whether the elevator should stop or not. We do not model method invocations to passive data objects as separate subtasks, since the passive object inherits the thread and priority from the invoking capsule, and can be viewed as an extension of the invoking capsule. But we do need to take into account blocking time caused by sharing of passive objects by multiple threads.

2. **Select Destination**. The user presses a button in the elevator to choose his/her destination, which triggers an interrupt to the capsule *elevator buttons interface*, which in turn sends a message *elevator request* to the capsule *elevator manager*. The *elevator manager* receives the message and records destination in the passive object *elevator status and plan*, which is a shared object protected by the priority ceiling protocol, and causes blocking time to the higher priority subtask.

[2]The analysis technique described in [28] is not entirely accurate, since it makes the pessimistic assumption on the number of preemptions caused by the higher priority task on the lower priority task. Also the original example is not based on UML-RT, but the concepts are similar enough to be viewed as a UML-RT model.

3. **Request Elevator**. The user presses the up or down button at a floor, which triggers an interrupt to the capsule *floor buttons interface*, which in turn sends a message *service request* to the capsule *scheduler*. The capsule *scheduler* receives message and interrogates the passive object *elevator status and plan* to determine if an elevator is on its way to this floor. If not, the *scheduler* selects an elevator and sends a message *elevator request* to the capsule *elevator manager*. The rest of the sequence is identical to the **select destination** scenario.

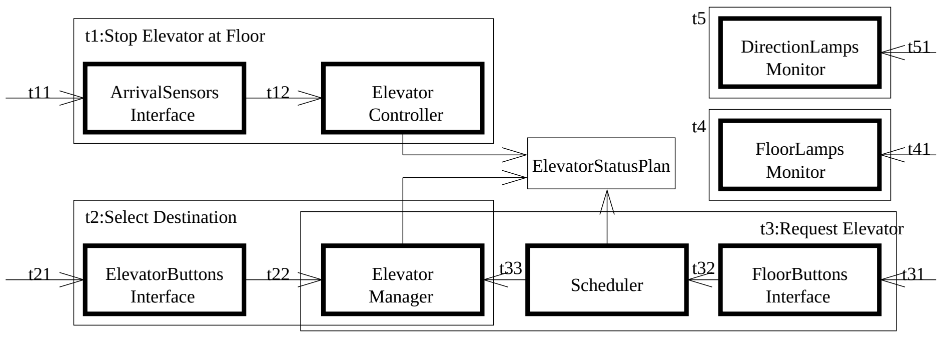


Figure 6.7: The event sequence diagram for the single-processor elevator control system.

Task	Period	WCET	Priority	WCRT
t_1**: Stop elevator at floor**				
t_{11}: Arrival Sensors Interface	50	2	9	-
t_{12}: Elevator Controller	50	5	6	34
t_2**: Select Destination**				
t_{21}: Elevator Buttons Interface	100	3	8	-
t_{22}: Elevator Manager	100	6	5	40
t_3**: Request Elevator**				
t_{31}: Floor Buttons Interface	200	4	7	-
t_{32}: Scheduler	200	20	4	-
t_{33}: Elevator Manager	200	6	5	46
t_4**,** t_5**: Other Tasks**				
t_{41}: Floor Lamps Monitor	500	5	3	58
t_{51}: Direction Lamps Monitor	500	5	2	63

Table 6.1: The task set of the single-processor elevator control system. Note that it is a common practice to assign a higher priority to the interrupt handler tasks [53], i.e., the *Interface* subtasks here, in order to avoid losing any interrupts.

Consider a building with 10 floors and 3 elevators. All end-to-end tasks are interrupt driven, not periodic. In order to perform schedulability analysis, we estimate the worst-case arrival rate of the interrupts and use them as approximations for periods assigned to each task. For example, the **Request Elevator** scenario is assigned a period of 200 ms by assuming that all 18 floor buttons (up and down buttons for each floor, except the top and bottom floors) are pressed within 3.6 seconds, which is likely to be the worst-case arrival rate.

Let's consider the end-to-end task t_1 **Stop Elevator at Floor**, which consists of two subtasks with execution time 2 and 5, priorities 9 and 6, respectively. Its canonical form is a single task with execution time 7 and priority 6. Other tasks can be classified as follows:

- t_2 and t_3 are type 2 tasks, with a higher-priority segment followed by a lower-priority segment.

- t_4 and t_5 are type 5 tasks, with all segments having priorities lower than 6. They have no effect on the WCRT of t_1.

Preemption time $B_2(1)$ caused by type 2 tasks t_2 and t_3 is $t_{21} + t_{31} = 3 + 4 = 7$; blocking time $B_4(i)$ caused by type 2 and 4 tasks is 0, since there are no non-first higher-priority segments of type 2 tasks, and no type 4 tasks at all; blocking time B_i caused by the *scheduler* object accessing its critical section is 20ms. Therefore,

$$\text{WCRT}(1) = \text{WCET}(1) + B_2(1) + B_4(i) + B(i) = (2 + 5) + (3 + 4) + 20 = 34$$

which is less than the period 50, therefore t_1 is schedulable.

Next, let's consider the end-to-end task t_2 **Select Destination**, which consists of two subtasks with execution time 3 and 6, priorities 8 and 5, respectively. Its canonical form is a single task with execution time 11 and priority 5. Other tasks can be classified as follows:

- t_1 is a type 1 task, with a single higher-priority segment with WCET 7.
- t_3 is a type 2 task, with a higher-priority segment t_{31} followed by lower-priority segments t_{32} and t_{33}.
- t_4 and t_5 are type 5 tasks, with all segments having priorities lower than 5.

Preemption time $B_2(2)$ caused by type 2 tasks is $t_{31} = 4$. There are no type 4 tasks. Again, blocking time B_i caused by the *scheduler* object accessing its critical section is 20ms. We use Equation 6.1 to get:

$$\text{WCRT}(2) = 9 + 4 + 20 + \lceil \frac{\text{WCRT}(2)}{50} \rceil \cdot 7 = 40$$

We can calculate WCRT for the other end-to-end tasks, as shown in the WCRT column of Table 6.1. Note that we associate the WCRT of the end-to-end task with the last segment of the task in the table. No deadlines are missed, and the system is schedulable.

In this example, the native runtime model of UML-RT, i.e., SMSP, actually performs worse than CMCP and CMSP in terms of task response times. However, since all three approaches result in a schedulable system, the engineer may choose SMSP due to its software engineering benefits, as discussed in Section 6.2.4.

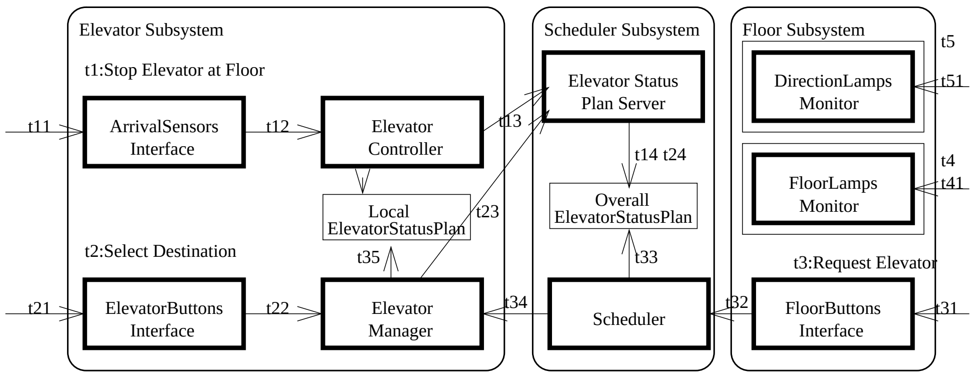


Figure 6.8: The event sequence diagram for the distributed elevator control system.

Task	Period	WCET	Priority	Processor
t_1**: Stop elevator at floor**				
t_{11}: Arrival Sensors Interface	50	2	9	ElevatorCPU
t_{12}: Elevator Controller	50	5	6	ElevatorCPU
t_{13}: Send Message *arrived*	50	2	6	CAN bus
t_{14}: Elevator Status Plan Server	50	3	9	SchedulerCPU
t_2**: Select Destination**				
t_{21}: Elevator Buttons Interface	100	3	8	ElevatorCPU
t_{22}: Elevator Manager	100	6	5	ElevatorCPU
t_{23}: Send Message *elevator commitment*	100	2	5	CAN bus
t_{24}: Elevator Status Plan Server	100	3	9	SchedulerCPU
t_3**: Request Elevator**				
t_{31}: Floor Buttons Interface	200	5	7	FloorCPU
t_{32}: Send Message *service request*	200	2	4	CAN bus
t_{33}: Elevator Scheduler	200	20	8	SchedulerCPU
t_{34}: Send Message *scheduler request*	200	2	4	CAN bus
t_{35}: Elevator Manager	200	6	4	ElevatorCPU
t_{36}: Send Message *elevator commitment*	200	2	4	CAN bus
t_{37}: Elevator Status Plan Server	200	3	9	SchedulerCPU
t_4**,** t_5**: Other Tasks**				
t_{41}: Floor Lamps Monitor	500	5	3	FloorCPU
t_{51}: Direction Lamps Monitor	500	5	2	FloorCPU

Table 6.2: The task set of the multi-processor elevator control system.

The single-processor system may become overloaded when more floors and more elevators are involved. In order to be scalable to a large number of floors and elevators, the system needs to be redesigned to take advantage of multiple processors connected via a network, possibly the Controller Area Network (CAN bus). Figure 6.8 shows the system architecture. There is one *ElevatorCPU* for each elevator, and one *FloorCPU* for each floor. There is only one *SchedulerCPU* that is a central decision point for scheduling elevator requests, consisting of the capsule *scheduler* as well as another capsule *elevator status and plan server* for handling updates and queries from the capsules from the *ElevatorCPU* and *FloorCPU*. Table 6.2 shows the task set of the multi-processor elevator control system for a system consisting of 12 elevators and 40 floors. Each scenario spans multiple processors, and we need to take into account delays caused by scheduling of network packets.

We can use the holistic schedulability analysis technique [95], with our enhanced HKL algorithm as a subroutine, to calculate the end-to-end WCRT of distributed tasks. We omit the details of this calculation due to space limitations, but the analysis results show that all tasks meet their deadlines.

6.5 Summary

Most UML CASE tools provide code generators for generating functional code in programming languages such as C/C++ or Java, but they generally ignore timing and schedulability issues. For example, the CASE tool RoseRT [46] maps all capsules into a single thread by default. Messages targeted to this thread is queued and processed in priority order non-preemptively. We describe a technique for analyzing schedulability of the native runtime model of UML-RT, Capsule-based Multi-threading, Capsule-based Priority Assignment(CMCP), by modifying the HKL algorithm [38] to add blocking time caused by the Run-To-Completion semantics of UML-RT. This technique can be used as the inner loop subroutine for determining system schedulability during design-space exploration to synthesize a real-time implementation from a logical UML-RT model. We believe this work bridges the gap between a logical UML-RT model and its real-time implementation on the target platform by giving the engineer powerful analysis techniques for assessing real-time properties of different ways of mapping capsules into threads. It focuses on the nonfunctional and real-time aspects of implementation synthesis, and is complementary to the existing code generators, which focuses on the functional aspects. The logical next step is to implement the schedulability analysis algorithms discussed in this chapter, perhaps as a plug-in to RoseRT that exchanges data with RoseRT through the XMI interface.

In this chapter, we have considered schedulability analysis techniques given a system configuration of capsule-to-thread grouping and thread priority assignment, but it is an open issue as to how to arrive at such a configuration. Exhaustive search is not feasible in general because the size of design space grows exponentially with the number of capsules or priorities. There may be some guidelines to follow, such as “assign higher priority to interrupt service routines to avoid losing interrupts”, but not for the system in general. We plan to investigate applicability of optimization techniques such as branch-and-bound, simulated annealing and genetic algorithms to design space exploration in order to achieve a close-to-optimal design in terms of objectives such as minimized number of threads or minimized response time for critical application scenarios.

CHAPTER VII

Conclusions and Future Work

In this thesis, we have developed a set of techniques and tools for model-based design and analysis of Embedded Real-Time (ERT) software, mainly focusing on issues related to analysis and implementation synthesis.

First, we have developed model-level static analysis techniques and implemented them in a software tool Automatic Integration of Reusable Embedded Software (AIRES), as part of the MoBIES tool-chain. Traditional static analysis techniques works at level of programming languages, with abstractions such as statements, variables and procedures, while AIRES works at the level of models, with abstractions such as components, ports and connectors. It has been evaluated with the Goal-Quality-Metric methodology and received positive feedback from our industrial partner. This work is discussed in Chapter III.

AIRES mainly focuses on the static structural aspects of the AMC software while ignoring the dynamics of component interactions. The AMC software is component-based with many different types of components, each with its unique functionality and interfaces, acting as basic building blocks of a complete system. The documentation provided by our industrial partner describes the dynamic behavior of various component types in detail. However, descriptions in natural language are not formal and subject to misinterpretation or misunderstanding. In order to perform deeper semantic analysis, we use the model-checker LTSA to provide a formal model of the AMC software in order to check for dynamic behavioral properties such as safety and liveness. We also describe several techniques to improve scalability of model-checking by exploiting application-level domain-semantics. This work is discussed in Chapter IV.

ERT systems typically consist of embedded software and embedding physical environment with tight coupling and interaction in-between. To extend the scope of modeling and analysis from *software* to *system* level, we describe an integrated approach to modeling and analysis of ERT systems, where the embedded software and the embedding physical environment are modeled and analyzed within the same modeling formalism. This enables the engineer to have a holistic view of the entire system and perform integrated tradeoff analysis. We use Timed Petri-Nets (TPN) [74] as the modeling formalism to illustrate this methodology, but the general concept is not restricted to TPN. We also describe a syntax-directed, automated transformation technique from TPN to Timed Automata (TA) in order to take advantage of the TA model-checker UPPAAL for verification purposes. This work is discussed in Chapter V.

To bridge the gap between models and implementation on the target platform, many modeling tools come with automatic code generators to translate models into code in a programming language. However, current code generation technology focuses on functional issues while largely ignoring non-functional and real-time issues. In order to facilitate synthesis of real-time implementations from models, we have developed schedulability analysis algorithms for the native implementation model of UML-RT [78], suitable for applications with tight interaction between different rate groups. This algorithm can be used in the inner loop of state-space exploration to find a suitable implementation architecture for a logical UML-RT model that satisfies timing constraints. This work is discussed in Chapter VI.

How would an embedded software engineer benefit from my thesis? Even though we have used AMC as a main target application domain, our work has more general applicability to component-based embedded software, e.g., the *CORBA Component Model* (CCM) [70], a widely adopted industry standard from Object Management Group (OMG) [69]. In fact the AMC software is currently being migrated to the CCM platform for its next generation. For more general large-scale embedded software, the event-based publish/subscribe architectural style of AMC is fairly common, due to its nice property of decoupling between publishers and subscribers. The concepts of static analysis and model-checking are applicable to most such applications, with some adaptation of the tools themselves, since the current tools have a lot of AMC-specific details.

It used to be the case that embedded software engineers cannot afford to use Object-Oriented (OO) concepts due to concerns with their overhead. But as CPUs become faster and cheaper, OO is becoming more and more popular for real-time embedded software development, with UML as the most representative example. In this case, implementation synthesis techniques discussed in Chapter VI are useful for helping the engineer make design decisions when generating a real-time implementation from logical models in UML.

This thesis mainly focuses on the aspects of algorithms and tools, but has largely ignored other important methodological and organizational issues involved in adopting a model-driven approach in embedded real-time software development, which may well prove to be the real hurdle to industry adoption of MDD. The Rational Unified Process (RUP) [72] is a popular methodology for design and analysis of object-oriented enterprise applications, but there is not a corresponding dominant methodology in the embedded software domain. We plan to develop a RUP-like software development process and a workflow tool supporting it, customized to specific domains such as automotive embedded systems. However, it may be difficult to explore these issues in a university environment, hence calling for close collaboration between industry and academia.

The model-based approach has been gaining acceptance in the development of automotive embedded control systems, as evidenced by the popularity of modeling and simulation tools such as Matlab/Simulink from Mathworks [65], Statemate Magnum from ILogix [47], ASCET-SD from ETAS [23]. As part of the MoBIES program and in collaboration with our industrial and academic partners, we have developed a set of techniques and tools for modeling and analysis of automotive applications [54], conceptually similar to the AIRES tool for Avionics Mission Computing [36]. In this section I briefly touch upon the possibilities for our future work in model-based approaches for embedded systems in the automotive domain.

Integrated Modeling and Simulation of Networked Control Systems The design and analysis techniques discussed in this thesis have generally ignored the issues related to distributed systems such as scheduling of network packets. CAN bus is the most popular network protocol used in distributed embedded control systems in the car today. It specifies the datalink and physical layers of the OSI network stack, and is characterized as Carrier Sense Multiple Access/Collision Aviodance (CSMA/CA) with Non-Destructive Priority Arbitration. Unlike other popular networking protocols such as Ethernet, the CAN protocol is deterministic, and can be analyzed with the Rate Monotonic Analysis (RMA) [53] technique. There are also tools on the market today for simulation of CAN networks, such as those from Vector CANTech [100].

Control performance of a distributed control system deteriorates with longer network delay and delay jitter. When the delay or jitter gets large enough, the control system may become unstable. It is desirable to explicitly quantify and model network-induced delays and their effects on control performance such as rise time, overshoot and steady-state error.

One way of doing this is to perform response time analysis of the taskset, and insert delay blocks into the Simulink model with delay equal to the Worst-Case Response Times of the particular tasks [35]. This approach does not take into account the effects of delay jitters on control performance. A better approach is to implement a simulator for CAN bus within Simulink using S-functions, and simulate the network packet delays together with the control algorithm and plant dynamics. This would allow us to observe the effects of packet transmission delay, jitter or loss on control performance. This idea is a natural extension of Chapter V, where the emphasis is on integrated modeling and analysis of the embedded software together with the embedding physical environment. In Chapter V, we discussed the state space explosion problem that plagues the model-checking approach. Here we plan to take an alternative approach of using simulation for verification, which is much more scalable than model-checking, and is already widely used in industry practice.

Code Generation from UML-SPT to OSEK API In Chapter VI, we discussed implementation synthesis from UML-RT models to a multi-threaded implementation, focusing on timing and scheduling issues. However, there is another aspect to implementation synthesis, that is, generation of functional code in a programming language such as C. Most commercial UML tools come with code generators to generate functional code, e.g., from Class Diagrams or State Transition Diagrams. UML has not been widely accepted in the automotive embedded software domain, partly due to lack of support by UML tool vendors for automotive-specific features. Here I discuss mapping rules from a UML real-time profile into a popular Real-Time Operating System (RTOS) standard in the automotive domain, in the hope that it will improve acceptance of UML by the automotive industry.

The UML Profile for Schedulability, Performance and Time (UML-SPT) [71], introduced in Section 2.3.2, is a UML profile designed to enhance UML with real-time modeling notations and facilitate development and integration of schedulability and performance analysis tools that work on the UML models. OSEK [92] is a popular RTOS standard used in automotive embedded systems, defined with stringent timing and resource constraints in mind. It consists of three parts: *OSEK-OS* specifies the operating system, *OSEK-COM* specifies the communications mechanism, and *OSEK-NM* specifies the network management system. We mainly focus on OSEK-OS here. It describes a static RTOS where all kernel objects such as tasks, counters, alarms, events, messages and resources are created at compile time. The OSEK Implementation Language (OIL) file is used to describe the kernel objects, and construct a customized kernel for the application, ensuring that only the necessary kernel objects and mechanisms are included in the kernel build. This minimizes the size and overhead of the RTOS as compared to the approach where all RTOS mechanisms are included regardless of whether the application needs them.

We have done some preliminary investigations [37] into generation of C code that conform to the OSEK API from UML-SPT. Table 7.1 shows some mapping rules, and Appendix B shows an example application scenario and OSEK code generated from it based on the mapping rules. The logical next step is to implement the code generator within an open-source UML tool.

RT-UML	OSEK
≪SAResource≫	Resource
≪SAResource≫. SAAccessControl	Priority Ceiling Protocol. OSEK does not implement other protocols specified in RT-UML such as Priority Inheritance
≪GRMAquire≫ ≪GRMRelease≫	API calls GetResource() and ReleaseResource()
≪SASchedulable≫	Basic Task. The OSEK concept of an extended task is not present in RT-UML
≪SATrigger≫	Alarm. Declared in the OIL file, and set with API calls SetRelAlarm() and SetAbsAlarm(), to set an alarm with either relative or absolute expiration time.
≪SAResponse≫. SAAbsDeadline ≪SAAction≫. SARelDeadline ≪SAResponse≫. RTduration	No direct mapping. OSEK does not support specifying absolute deadlines for aperiodic tasks, or relative deadlines for intermediate tasks, or task's worst-case execution time. However, we can always use *ad hoc* annotations to specify these attributes.

Table 7.1: Mapping rules from UML-SPT entities to OSEK entities. This is intended to be a small sample rather than a comprehensive definition.

APPENDICES

APPENDIX A

FSP Model for the MediumSP Scenario

This is the FSP model for the MediumSP (Medium Single-Processor) scenario as shown in Figures 3.7 and 3.8.

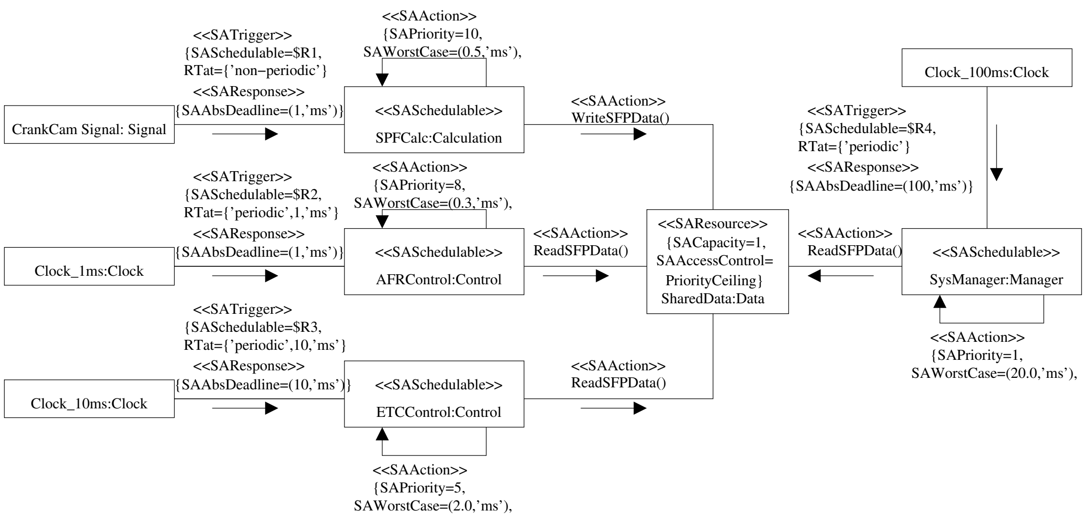
```
Timer20hz = (timeout20hz->timer20hzDone->tick->Timer20hz).
Timer1hz = (timeout1hz->timer1hzDone->Delay20[1]),
Delay20[t:1..20] = (when (t==20) tick->Timer1hz
|when (t < 20) tick->Delay20[t+1]).

ClosedEDComp =
(inEvt->issueGDCall->receiveGDReply->outEvt->ClosedEDComp
|receiveGDCall->issueGDReply->ClosedEDComp).

OpenEDComp =
(inEvt->issueGDCall->receiveGDReply->outEvt->OpenEDComp
|receiveGDCall->issueGDReply->OpenEDComp
|receiveSDCall->issueSDReply->outEvt->OpenEDComp).

DisplayComp =
(inEvt->issueGDCall->receiveGDReply->display->DisplayComp).

DeviceComp = (inEvt->outEvt->DeviceComp
|receiveGDCall->issueGDReply->DeviceComp).

LazyActiveComp = (inEvt->outEvt->DataStale
|receiveGDCall->issueGDReply->LazyActiveComp),
DataStale = (receiveGDCall->issueGDCall->receiveGDReply
->issueGDReply->LazyActiveComp).

ModeSourceComp = (inEvt->ModeSourceComp
|inEvt->enable1->ModeSourceComp
|inEvt->disable1->ModeSourceComp
|inEvt->enable2->ModeSourceComp
```

```
|inEvt->disable2->ModeSourceComp).

ModalComp = Enabled,
Disabled = (enable->Enabled|disable->Disabled
|inEvt->Disabled), Enabled = (enable->Enabled
|disable->Disabled
|inEvt->issueGDCall->receiveGDReply->outEvt->Enabled
|receiveGDCall->issueGDReply->Enabled).

PushDataSourceComp =
(inEvt->issueSDCall->receiveSDReply->PushDataSourceComp).

PassiveComp =
(receiveSDCall->issueSDReply->outEvt->PassiveComp).

Event(ID=1)=(inEvt[ID]->matched->Event).
||InputCorrelator(NumInputs=1)= ( if(NumInputs>0) then
(forall[i:1..NumInputs] Event(i)) ).

||Thread1hz = (Timer1hz ||earthModel:PushDataSourceComp
||wayPoint1:PassiveComp ||wayPoint2:PassiveComp
||wayPoint3:PassiveComp ||wayPoint4:PassiveComp
||wayPoint5:PassiveComp ||wayPoint6:PassiveComp
||wayPoint7:PassiveComp ||wayPoint8:PassiveComp
||wayPoint9:PassiveComp ||wayPoint10:PassiveComp
||leg1:LazyActiveComp ||leg2:LazyActiveComp ||leg3:LazyActiveComp
||leg4:LazyActiveComp ||leg5:LazyActiveComp
||correlatorLeg1:InputCorrelator(3)
||correlatorLeg2:InputCorrelator(3)
||correlatorLeg3:InputCorrelator(3)
||correlatorLeg4:InputCorrelator(3)
||correlatorLeg5:InputCorrelator(2) ||route::OpenEDComp
||correlatorRoute:InputCorrelator(5) ||groundPoint:ClosedEDComp
||flightPlan:ClosedEDComp ||navSteering:ModalComp
||waypointSteering:ModalComp ||pilotPref:OpenEDComp
||flightPlanDisplay:DisplayComp ||navDisplay:DisplayComp
||pilotControls:ModeSourceComp)
/{timeout1hz/earthModel.inEvt,
earthModel.issueSDCall/wayPoint1.receiveSDCall,
earthModel.issueSDCall/wayPoint2.receiveSDCall,
earthModel.issueSDCall/wayPoint3.receiveSDCall,
earthModel.issueSDCall/wayPoint4.receiveSDCall,
earthModel.issueSDCall/wayPoint5.receiveSDCall,
earthModel.issueSDCall/wayPoint6.receiveSDCall,
earthModel.issueSDCall/wayPoint7.receiveSDCall,
earthModel.issueSDCall/wayPoint8.receiveSDCall,
earthModel.issueSDCall/wayPoint9.receiveSDCall,
earthModel.issueSDCall/wayPoint10.receiveSDCall,

earthModel.receiveSDReply/wayPoint1.issueSDReply,
earthModel.receiveSDReply/wayPoint2.issueSDReply,
earthModel.receiveSDReply/wayPoint3.issueSDReply,
earthModel.receiveSDReply/wayPoint4.issueSDReply,
earthModel.receiveSDReply/wayPoint5.issueSDReply,
earthModel.receiveSDReply/wayPoint6.issueSDReply,
earthModel.receiveSDReply/wayPoint7.issueSDReply,
earthModel.receiveSDReply/wayPoint8.issueSDReply,
earthModel.receiveSDReply/wayPoint9.issueSDReply,
earthModel.receiveSDReply/wayPoint10.issueSDReply,

wayPoint1.outEvt/correlatorLeg1.inEvt[1],
wayPoint2.outEvt/correlatorLeg1.inEvt[2],
wayPoint3.outEvt/correlatorLeg1.inEvt[3],
correlatorLeg1.matched/leg1.inEvt,

wayPoint3.outEvt/correlatorLeg2.inEvt[1],
wayPoint4.outEvt/correlatorLeg2.inEvt[2],
wayPoint5.outEvt/correlatorLeg2.inEvt[3],
correlatorLeg2.matched/leg2.inEvt,

wayPoint5.outEvt/correlatorLeg3.inEvt[1],
wayPoint6.outEvt/correlatorLeg3.inEvt[2],
wayPoint7.outEvt/correlatorLeg3.inEvt[3],
correlatorLeg3.matched/leg3.inEvt,

wayPoint7.outEvt/correlatorLeg4.inEvt[1],
wayPoint8.outEvt/correlatorLeg4.inEvt[2],
wayPoint9.outEvt/correlatorLeg4.inEvt[3],
correlatorLeg4.matched/leg4.inEvt,


wayPoint9.outEvt/correlatorLeg5.inEvt[1],
wayPoint10.outEvt/correlatorLeg5.inEvt[2],
correlatorLeg5.matched/leg5.inEvt,

leg1.issueGDCall/wayPoint1.receiveGDCall,
leg1.issueGDCall/wayPoint2.receiveGDCall,
leg1.issueGDCall/wayPoint3.receiveGDCall,
leg1.receiveGDReply/wayPoint1.issueGDReply,
leg1.receiveGDReply/wayPoint2.issueGDReply,
leg1.receiveGDReply/wayPoint3.issueGDReply,

leg2.issueGDCall/wayPoint3.receiveGDCall,
leg2.issueGDCall/wayPoint4.receiveGDCall,
leg2.issueGDCall/wayPoint5.receiveGDCall,
leg2.receiveGDReply/wayPoint3.issueGDReply,
leg2.receiveGDReply/wayPoint4.issueGDReply,
leg2.receiveGDReply/wayPoint5.issueGDReply,

leg3.issueGDCall/wayPoint5.receiveGDCall,
leg3.issueGDCall/wayPoint6.receiveGDCall,
leg3.issueGDCall/wayPoint7.receiveGDCall,
leg3.receiveGDReply/wayPoint5.issueGDReply,
leg3.receiveGDReply/wayPoint6.issueGDReply,
leg3.receiveGDReply/wayPoint7.issueGDReply,

leg4.issueGDCall/wayPoint7.receiveGDCall,
leg4.issueGDCall/wayPoint8.receiveGDCall,
leg4.issueGDCall/wayPoint9.receiveGDCall,
leg4.receiveGDReply/wayPoint7.issueGDReply,
leg4.receiveGDReply/wayPoint8.issueGDReply,
leg4.receiveGDReply/wayPoint9.issueGDReply,

leg5.issueGDCall/wayPoint9.receiveGDCall,
leg5.issueGDCall/wayPoint10.receiveGDCall,
leg5.receiveGDReply/wayPoint9.issueGDReply,
leg5.receiveGDReply/wayPoint10.issueGDReply,

leg1.outEvt/correlatorRoute.inEvt[1],
leg2.outEvt/correlatorRoute.inEvt[2],
leg3.outEvt/correlatorRoute.inEvt[3],
leg4.outEvt/correlatorRoute.inEvt[4],
leg5.outEvt/correlatorRoute.inEvt[5],
correlatorRoute.matched/route.inEvt,

route.issueGDCall/leg1.receiveGDCall,
route.issueGDCall/leg2.receiveGDCall,
route.issueGDCall/leg3.receiveGDCall,
route.issueGDCall/leg4.receiveGDCall,
route.issueGDCall/leg5.receiveGDCall,
route.receiveGDReply/leg1.issueGDReply,
route.receiveGDReply/leg2.issueGDReply,
route.receiveGDReply/leg3.issueGDReply,
route.receiveGDReply/leg4.issueGDReply,
route.receiveGDReply/leg5.issueGDReply,

route.outEvt/groundPoint.inEvt,
groundPoint.issueGDCall/route.receiveGDCall,
groundPoint.receiveGDReply/route.issueGDReply,

route.outEvt/flightPlan.inEvt,
flightPlan.issueGDCall/route.receiveGDCall,
flightPlan.receiveGDReply/route.issueGDReply,

groundPoint.outEvt/navSteering.inEvt,
navSteering.issueGDCall/groundPoint.receiveGDCall,
navSteering.receiveGDReply/groundPoint.issueGDReply,

navSteering.outEvt/navDisplay.inEvt,
navDisplay.issueGDCall/navSteering.receiveGDCall,
navDisplay.receiveGDReply/navSteering.issueGDReply,

flightPlan.outEvt/waypointSteering.inEvt,
waypointSteering.issueGDCall/flightPlan.receiveGDCall,
waypointSteering.receiveGDReply/flightPlan.issueGDReply,

waypointSteering.outEvt/flightPlanDisplay.inEvt,
flightPlanDisplay.issueGDCall/waypointSteering.receiveGDCall,
flightPlanDisplay.receiveGDReply/waypointSteering.issueGDReply,

flightPlan.outEvt/pilotPref.inEvt,
pilotPref.issueGDCall/flightPlan.receiveGDCall,
pilotPref.receiveGDReply/flightPlan.issueGDReply,

timeout1hz/pilotControls.inEvt,
pilotControls.enable1/navSteering.enable,
pilotControls.enable2/waypointSteering.enable,
pilotControls.disable1/navSteering.disable,
pilotControls.disable2/waypointSteering.disable,

navDisplay.display/timer1hzDone,
flightPlanDisplay.display/timer1hzDone }.

||Thread20hz = (Timer20hz ||gps:DeviceComp ||ins:DeviceComp
||adc:DeviceComp ||airframe:LazyActiveComp
||correlatorAirframe:InputCorrelator(4) ||navDisplay:DisplayComp
||radar1:DeviceComp ||radar2:DeviceComp ||trackSensor1:DeviceComp
||trackSensor2:DeviceComp ||trackSensor3:DeviceComp
||trackSensor4:DeviceComp ||track1:ClosedEDComp
||track2:ClosedEDComp ||track3:ClosedEDComp ||track4:ClosedEDComp
||track5:ClosedEDComp ||track6:ClosedEDComp ||track7:ClosedEDComp
||track8:ClosedEDComp ||track9:ClosedEDComp ||track10:ClosedEDComp
||tacticalSteering:OpenEDComp
||correlatorTacticalSteering:InputCorrelator(12)
||correlatorTrack1:InputCorrelator(3)
||correlatorTrack2:InputCorrelator(2)
||correlatorTrack5:InputCorrelator(2)
||correlatorTrack7:InputCorrelator(2)
```

```
||correlatorTrack8:InputCorrelator(2)
||correlatorTrack10:InputCorrelator(2)
||tacticalDisplay1:DisplayComp ||tacticalDisplay2:DisplayComp
||hudDisplay:DisplayComp)
/{timeout20hz/gps.inEvt,
timeout20hz/ins.inEvt, timeout20hz/adc.inEvt,
timeout20hz/radar1.inEvt, timeout20hz/radar2.inEvt,
timeout20hz/trackSensor1.inEvt, timeout20hz/trackSensor2.inEvt,
timeout20hz/trackSensor3.inEvt, timeout20hz/trackSensor4.inEvt,

gps.outEvt/correlatorAirframe.inEvt[1],
ins.outEvt/correlatorAirframe.inEvt[2],
adc.outEvt/correlatorAirframe.inEvt[3],
radar1.outEvt/correlatorAirframe.inEvt[4],
correlatorAirframe.matched/airframe.inEvt,

airframe.issueGDCall/gps.receiveGDCall,
airframe.receiveGDReply/gps.issueGDReply,
airframe.issueGDCall/ins.receiveGDCall,
airframe.receiveGDReply/ins.issueGDReply,
airframe.issueGDCall/adc.receiveGDCall,
airframe.receiveGDReply/adc.issueGDReply,
airframe.issueGDCall/radar1.receiveGDCall,
airframe.receiveGDReply/radar1.issueGDReply,

airframe.outEvt/navDisplay.inEvt,
navDisplay.issueGDCall/airframe.receiveGDCall,
navDisplay.receiveGDReply/airframe.issueGDReply,

trackSensor1.outEvt/correlatorTrack1.inEvt[1],
trackSensor2.outEvt/correlatorTrack1.inEvt[2],
trackSensor3.outEvt/correlatorTrack1.inEvt[3],
correlatorTrack1.matched/track1.inEvt,
track1.issueGDCall/trackSensor1.receiveGDCall,
track1.receiveGDReply/trackSensor1.issueGDReply,
track1.issueGDCall/trackSensor2.receiveGDCall,
track1.receiveGDReply/trackSensor2.issueGDReply,
track1.issueGDCall/trackSensor3.receiveGDCall,
track1.receiveGDReply/trackSensor3.issueGDReply,

trackSensor1.outEvt/correlatorTrack2.inEvt[1],
trackSensor2.outEvt/correlatorTrack2.inEvt[2],
correlatorTrack2.matched/track2.inEvt,
track2.issueGDCall/trackSensor1.receiveGDCall,
track2.receiveGDReply/trackSensor1.issueGDReply,
track2.issueGDCall/trackSensor2.receiveGDCall,
track2.receiveGDReply/trackSensor2.issueGDReply,
trackSensor1.outEvt/track3.inEvt,
track3.issueGDCall/trackSensor1.receiveGDCall,
track3.receiveGDReply/trackSensor1.issueGDReply,

trackSensor2.outEvt/track4.inEvt,
track4.issueGDCall/trackSensor2.receiveGDCall,
track4.receiveGDReply/trackSensor2.issueGDReply,

trackSensor3.outEvt/correlatorTrack5.inEvt[1],
trackSensor4.outEvt/correlatorTrack5.inEvt[2],
correlatorTrack5.matched/track5.inEvt,
track5.issueGDCall/trackSensor3.receiveGDCall,
track5.receiveGDReply/trackSensor3.issueGDReply,
track5.issueGDCall/trackSensor4.receiveGDCall,
track5.receiveGDReply/trackSensor4.issueGDReply,

trackSensor4.outEvt/track6.inEvt,
track6.issueGDCall/trackSensor4.receiveGDCall,
track6.receiveGDReply/trackSensor4.issueGDReply,

trackSensor1.outEvt/correlatorTrack7.inEvt[1],
trackSensor2.outEvt/correlatorTrack7.inEvt[2],
correlatorTrack7.matched/track7.inEvt,
track7.issueGDCall/trackSensor1.receiveGDCall,
track7.receiveGDReply/trackSensor1.issueGDReply,
track7.issueGDCall/trackSensor2.receiveGDCall,
track7.receiveGDReply/trackSensor2.issueGDReply,

trackSensor3.outEvt/correlatorTrack8.inEvt[1],
trackSensor4.outEvt/correlatorTrack8.inEvt[2],
correlatorTrack8.matched/track8.inEvt,
track8.issueGDCall/trackSensor3.receiveGDCall,
track8.receiveGDReply/trackSensor3.issueGDReply,
track8.issueGDCall/trackSensor4.receiveGDCall,
track8.receiveGDReply/trackSensor4.issueGDReply,

trackSensor4.outEvt/track9.inEvt,
track9.issueGDCall/trackSensor4.receiveGDCall,
track9.receiveGDReply/trackSensor4.issueGDReply,

trackSensor3.outEvt/correlatorTrack10.inEvt[1],
trackSensor4.outEvt/correlatorTrack10.inEvt[2],
correlatorTrack10.matched/track10.inEvt,
track10.issueGDCall/trackSensor3.receiveGDCall,
track10.receiveGDReply/trackSensor3.issueGDReply,
track10.issueGDCall/trackSensor4.receiveGDCall,
track10.receiveGDReply/trackSensor4.issueGDReply,

airframe.outEvt/correlatorTacticalSteering.inEvt[1],
radar2.outEvt/correlatorTacticalSteering.inEvt[2],
track1.outEvt/correlatorTacticalSteering.inEvt[3],
track2.outEvt/correlatorTacticalSteering.inEvt[4],
track3.outEvt/correlatorTacticalSteering.inEvt[5],
track4.outEvt/correlatorTacticalSteering.inEvt[6],
track5.outEvt/correlatorTacticalSteering.inEvt[7],
track6.outEvt/correlatorTacticalSteering.inEvt[8],
track7.outEvt/correlatorTacticalSteering.inEvt[9],
track8.outEvt/correlatorTacticalSteering.inEvt[10],
track9.outEvt/correlatorTacticalSteering.inEvt[11],
track10.outEvt/correlatorTacticalSteering.inEvt[12],
correlatorTacticalSteering.matched/tacticalSteering.inEvt,

tacticalSteering.issueGDCall/airframe.receiveGDCall,
tacticalSteering.receiveGDReply/airframe.issueGDReply,
tacticalSteering.issueGDCall/radar2.receiveGDCall,
tacticalSteering.receiveGDReply/radar2.issueGDReply,
tacticalSteering.issueGDCall/track1.receiveGDCall,
tacticalSteering.receiveGDReply/track1.issueGDReply,
tacticalSteering.issueGDCall/track2.receiveGDCall,
tacticalSteering.receiveGDReply/track2.issueGDReply,
tacticalSteering.issueGDCall/track3.receiveGDCall,
tacticalSteering.receiveGDReply/track3.issueGDReply,

tacticalSteering.outEvt/hudDisplay.inEvt,
hudDisplay.issueGDCall/tacticalSteering.receiveGDCall,
hudDisplay.receiveGDReply/tacticalSteering.issueGDReply,

tacticalSteering.outEvt/tacticalDisplay1.inEvt,
tacticalDisplay1.issueGDCall/tacticalSteering.receiveGDCall,
tacticalDisplay1.receiveGDReply/tacticalSteering.issueGDReply,

tacticalSteering.outEvt/tacticalDisplay2.inEvt,
tacticalDisplay2.issueGDCall/tacticalSteering.receiveGDCall,
tacticalDisplay2.receiveGDReply/tacticalSteering.issueGDReply,

navDisplay.display/timer20hzDone,
hudDisplay.display/timer20hzDone,
tacticalDisplay1.display/timer20hzDone,
tacticalDisplay2.display/timer20hzDone}.

||SYSTEM = (Thread1hz || Thread20hz).
```

APPENDIX B

An Application Scenario in UML-SPT and Corresponding OSEK Code



<<SATrigger>> {SASchedulable=$R1, RTat={'non−periodic'}
<<SAResponse>> {SAAbsDeadline=(1,'ms')}
CrankCam Signal: Signal
<<SAAction>> {SAPriority=10, SAWorstCase=(0.5,'ms'),
<<SASchedulable>> SPFCalc:Calculation
<<SAAction>> WriteSFPData()
<<SATrigger>> {SASchedulable=$R2, RTat={'periodic',1,'ms'}
<<SAResponse>> {SAAbsDeadline=(1,'ms')}
Clock_1ms:Clock
<<SAAction>> {SAPriority=8, SAWorstCase=(0.3,'ms'),
<<SASchedulable>> AFRControl:Control
<<SAAction>> ReadSFPData()
<<SAResource>> {SACapacity=1, SAAccessControl= PriorityCeiling} SharedData:Data
<<SAAction>> ReadSFPData()
<<SASchedulable>> SysManager:Manager
Clock_100ms:Clock
<<SATrigger>> {SASchedulable=$R4, RTat={'periodic'}
<<SAResponse>> {SAAbsDeadline=(100,'ms')}
<<SAAction>> {SAPriority=1, SAWorstCase=(20.0,'ms'),
<<SATrigger>> {SASchedulable=$R3, RTat={'periodic',10,'ms'}
<<SAResponse>> {SAAbsDeadline=(10,'ms')}
Clock_10ms:Clock
<<SASchedulable>> ETCControl:Control
<<SAAction>> ReadSFPData()
<<SAAction>> {SAPriority=5, SAWorstCase=(2.0,'ms'),

Figure B.1: An application scenario in automotive engine control modeled with UML-SPT.

Figure B.1 shows an application scenario in automotive engine control provided by the MoBIES project [98]. The system consists of 3 periodic tasks and 1 interrupt-triggered task accessing a shared data area protected by Priority Ceiling Protocol. It consists of both Electronic Throttle Control (ETC) and Air-Fuel Ratio (AFR) Control. With ETC, the usual mechanical linkage between the gas pedal and the throttle plate is eliminated, and the throttle is actuated by a DC motor. AFR controls the fuel injector timing so that the ratio of fuel to air must not deviate more that 0.1 from the stoichiometric air-to-fuel ratio of 14.64.

Fuel is injected into the intake port area of the engine cylinders once every thermodynamic cycle (once every two engine revolutions). The injectors are set up to deliver a pulse of fuel whose duration corresponds to a fuel amount (mass) which achieves a desired air to fuel ratio of the charge in the cylinder. The timing of the pulse with respect to the cylinder intake valve opening is also important for reasons of proper mixing of fuel and air so that ignition of the charge is reliable.

Both ETC and AFR are triggered periodically, with different periods dictated by application characteristics. Both tasks read from a shared data area called *scaled fuel parameters* that gets updated by an SFPCalculation task, which is triggered by the crankshaft and camshaft pulse signals aperiodically. The task calculates the appropriate duration of each injector pulse as well as the engine angle to start each injector pulse.

The corresponding OSEK OIL file is shown as follows. For space limitations, we do not show the detailed C code for each task. The OIL file defines the alarms used to trigger tasks, but does not specify the period for a periodic alarm. The way to have a task triggered periodically is to explicitly set alarms by using the *SetRelAlarm* API call in the C code. For example, in order to set the *RunAFRControlAlarm* to trigger *AFRControl* task every 1 ms, we need to call

```
SetRelAlarm(RunAFRControlAlarm /*AlarmID*/,
 0/*Offset*/,1000/*Period*/);
```

When the system starts up, the *InitAlarms* TASK is first invoked which sets up alarms for all the periodic tasks. At runtime, the interrupt service routine *CrankCamSignalISR* gets triggered by external interrupts from crankshaft and samshaft, and sets the alarm *RunSPFCalcAlarm*, which in turn triggers the aperiodic task *SFPCalcTask*.

```
CPU ETCAFR {
   /**************************************/
   /*              Tasks                 */
   /**************************************/
```

```
TASK SFPCalcTask {
     TYPE = BASIC;
     SCHEDULE = NON;
     PRIORITY = 10;
     ACTIVATION = 1;
     AUTOSTART = FALSE;
     STACKSIZE = 4096;
     SCHEDULE_CALL = FALSE;
};

 TASK AFRControlTask {
     TYPE = BASIC;
     SCHEDULE = NON;
     PRIORITY = 8;
     ACTIVATION = 1;
     AUTOSTART = FALSE;
     STACKSIZE = 4096;
     SCHEDULE_CALL = FALSE;
};

TASK ETCControlTask {
     TYPE = BASIC;
     SCHEDULE = NON;
     PRIORITY = 5;
     ACTIVATION = 1;
     AUTOSTART = FALSE;
     STACKSIZE = 4096;
     SCHEDULE_CALL = FALSE;
};

 TASK SysManagerTask {
     TYPE = BASIC;
     SCHEDULE = NON;
     PRIORITY = 1;
     ACTIVATION = 1;
     AUTOSTART = FALSE;
     STACKSIZE = 4096;
     SCHEDULE_CALL = FALSE;
};

/*************************************/
/* Must be highest priority in the system.*/
/*************************************/
TASK InitAlarms {
        TYPE = BASIC;
        SCHEDULE = FULL;
        PRIORITY = 16;
        ACTIVATION = 1;
        AUTOSTART = TRUE;
        STACKSIZE = 128;
        SCHEDULE_CALL = FALSE;
};

/*************************************/
/*             Alarms                */
/*************************************/
ALARM RunSPFCalcAlarm {
     COUNTER = SYSTEM_COUNTER;
     TASK = SFPCalcTask;
     ACTION = ACTIVATETASK;
};

ALARM RunAFRControlAlarm {
     COUNTER = SYSTEM_COUNTER;
     TASK = AFRControlTask;
     ACTION = ACTIVATETASK;
};

ALARM RunETCControlAlarm {
     COUNTER = SYSTEM_COUNTER;
     TASK = ETCControlTask;
     ACTION = ACTIVATETASK;
};

ALARM RunSysManagerAlarm {
     COUNTER = SYSTEM_COUNTER;
     TASK = SysManagerTask;
     ACTION = ACTIVATETASK;
};

/*************************************/
/*             Resources             */
/*************************************/
RESOURCE SharedData {
     /*Put application-specific attributes here. */
}

/*************************************/
/*             Counters              */
/*************************************/
COUNTER SYSTEM_COUNTER {
     MAXALLOWEDVALUE = 65535;
     TICKSPERBASE = 1;
     MINCYCLE = 1;
```

```
   };

   /************************************/
   /*             ISRs                 */
   /************************************/
   ISR CrankCamSignalISR {
        CATEGORY = 2;
   }


   /************************************/
   /*             O/S                  */
   /************************************/
   OS OSEK_OS {
        CC = AUTO;
        STATUS = STANDARD;
        SCHEDULE = AUTO;
        SYSTEMSTACKSIZE = 16000;
        StartupHook = TRUE;
        ErrorHook = FALSE;
        ShutdownHook = FALSE;
        PreTaskHook = FALSE;
        PostTaskHook = FALSE;
        WINDVIEW_SUPPORT = FALSE;
        RTA_SUPPORT = FALSE;
        STACK_FILL_DIAGNOSTIC = FALSE;
};
```

APPENDIX C

Acronyms

ADL Architecture Description Language

AFR Air-Fuel Ratio

AIRES Automatic Integration of Reusable Embedded Software

AIF Analysis Interface Format

AMC Avionics Mission Computing

BCET Best-Case Execution Time

BCRT Best-Case Response Time

CASE Computer-Aided Software Engineering

CDG Component-Dependency Graph

CIF Configuration Interface Format

CMCP Capsule-based Multi-threading, Capsule-based Priority Assignment

CMSP Capsule-based Multi-threading, Scenario-based Priority Assignment

CORBA	Common Object Request Broker
COTS	Commercial Off-The-Shelf
DARPA	Defense Advanced Research Projects Agency
ERT	Embedded Real-Time
ESML	Embedded Systems Modeling Language
ETC	Electronic Throttle Control
FSP	Finite State Processes
GME	Generic Modeling Environment
GPS	Global Positioning System
GQM	Goal-Quality-Metric
HKL	Harbour, Klein, Lehoczky
IIF	Instrumentation Interface Format
LTSA	Labelled Transition Systems Analyzer
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MIC	Model-Integrated Computing
MoBIES	Model-Based Integration of Embedded Software
OEP	Open Experimental Platform

OIL	OSEK Implementation Language
OSEK	Open Systems and the Corresponding Interfaces for Automotive Electronics (German Acronym)
OO	Object-Oriented
OMG	Object Management Group
OTIF	Open Tool Integration Platform
PDG	Port-Dependency Graph
QoS	Quality of Service
RC	Railroad Crossing
RMA	Rate Monotonic Analysis
RMS	Rate Monotonic Scheduling
ROOM	Real-Time Object-Oriented Modeling
RTC	Run-To-Completion
RTOS	Real-Time Operating System
SFP	Scaled-Fuel Parameters
SMSP	Scenario-based Multi-threading, Scenario-based Priority Assignment
SMV	Symbolic Model Verifier
TA	Timed Automata
TPN	Time Petri-Nets

TargetRTS	Target Runtime System
UML	Unified Modeling Language
UML-RT	UML Real-Time
UML-SPT	UML Profile for Schedulability, Performance and Time
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] T. F. Abdelzaher and K. G. Shin, “Period-based load partitioning and assignment for large real-time applications,” *IEEE Trans. Comput.*, vol. 49, no. 1, pp. 81–87, 2000.

[2] R. B. Allen and D. Garlan, “A formal approach to software architectures,” in *IFIP Congress, Vol. 1*, 1992, pp. 134–141.

[3] R. Alur, C. Courcoubetis, and D. L. Dill, “Model-checking for real-time systems,” in *Proc. Annual Symposium on Logic in Computer Science*, 1990, pp. 414–425.

[4] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[5] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, “Times: a tool for schedulability analysis and code generation of real-time systems,” in *Proc. International Workshop on Formal Modeling and Analysis of Timed Systems*, Sept. 2003.

[6] (2004) The Artisan Software website. [Online]. Available: http://www.artisansw.com

[7] H. Ben-Abdallah, D. Clarke, I. Lee, and O. Sokolsky, “Paragon: A paradigm for the specification, verification, and testing of real-time systems,” in *Proc. IEEE Aerospace Conference*, Feb 1997, pp. 469–488.

[8] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, “UPPAAL - a tool suite for automatic verification of real-time systems,” in *Proc. Workshop on Verification and Control of Hybrid Systems*, October 1995, pp. 232–243.

[9] P. Binns, M. Englehart, M. Jackson, and S. Vestal, “Domain-specific software architectures for guidance, navigation and control,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 2, pp. 201–227, June 1996.

[10] M. Boyer and M. Diaz, “Multiple enabledness of transtions in petri nets with time,” in *Proc. International Workshop on Petri Nets and Performance Modeling*, 2001, pp. 219–228.

[11] S. Bradley, W. Henderson, and D. Kendall, “Reducing conservatism in response time analysis of distributed systems,” in *Proc. IEE Colloquium on Real-Time Systems*, 1999, pp. 71–74.

[12] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, “Ptolemy: A framework for simulating and prototyping heterogeneous systems,” *The Computer Journal*, vol. 4, pp. 155–182, 1994.

[13] A. Burns and A. J. Wellings, “HRT-HOOD: A structured design method for real-time systems,” *Real-Time Systems Journal*, vol. 6, pp. 73–114, 1994.

[14] J. C. Corbett, “Timing analysis of Ada tasking programs,” *IEEE Trans. Software Eng.*, vol. 22, no. 7, pp. 461–483, July 1996.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. McGraw Hill Publishers, 2001.

[16] L. Cortes, P. Eles, and Z. Peng, “Verification of embedded systems using a petri net based representation,” in *Proc. IEEE International Symposium on System Synthesis*, 2000, pp. 149–155.

[17] D. de Niz and R. Rajkumar, “Geodesic - a reusable component framework for embedded real-time systems,” Carnegie Mellon University, Tech. Rep., 2002.

[18] Y. Deng, S. Lu, and M. Evangelist, “A formal approach for architectural modeling and prototyping of distributed real-time systems,” in *Proc. IEEE Hawaii International Conference on System Sciences*, 1997, pp. 481–490.

[19] B. P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*. Addison Wesley, 1999.

[20] J. Eker and A. Cervin, “A matlab toolbox for real-time and control systems co-design,” in *Proc. International Conference on Real-Time Computing Systems and Applications*, 1999, pp. 320 –327.

[21] J. Eker and J. Janneck, “Caltrop—language report (draft),” Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA 94720, USA, Technical Memorandum, 2002, http://www.gigascale.org/caltrop.

[22] R. Esser, J. W. Janneck, and M. Naedele, “Applying an object-oriented petri net language to heterogeneous systems design,” in *Proc. Petri-Nets in System Engineering*, 1997, pp. 1–10.

[23] (2004) The ETAS website. [Online]. Available: http://www.etas.de

[24] G. C. Gannod and S. Gupta, “An automated tool for analyzing petri nets using spin,” in *Proc. IEEE International Conference on Automated Software Engineering*, 2001, pp. 404–407.

[25] D. Garlan, S. Khersonsky, and J. S. Kim, “Model checking publish-subscribe systems,” in *Proc. International SPIN Workshop on Model Checking of Software*, 2003, pp. 166–180.

[26] R. Gerber and I. Lee, “A layered approach to automating the verification of real-time systems,” *IEEE Trans. Software Eng.*, vol. 18, no. 9, pp. 768–784, 1992.

[27] D. P. Gluch, R. Obenza, and C. B. Weinstock, “Model-based verification of software and firmware,” in *Proc. Embedded Systems Conference*, 2000.

[28] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000.

[29] Z. Gu, S. Kodase, S. Wang, and K. G. Shin, “A model-based approach to system-level dependency and real-time analysis of embedded software,” in *Proc. IEEE Real-Time Technology and Applications Symposium*, 2003, pp. 78–85.

[30] Z. Gu and K. G. Shin, “Analysis of event-driven real-time systems with Time Petri-Nets,” in *Proc. IFIP World Computer Congress, Stream on Distributed and Parallel Embedded Systems*, 2002, pp. 141–148.

[31] ——, “An integrated approach to modeling and analysis of embedded real-time systems based on Timed Petri-Nets,” in *Proc. IEEE International Conference on Distributed Computing Systems*, 2003, pp. 350 –359.

[32] ——, “Integrated modeling and analysis of computer-based embedded control systems,” in *Proc. IEEE International Conference on Engineering of Computer-Based Systems*, 2003, pp. 141–148.

[33] ——, “Application of model-checking to component-based real-time embedded software,” in *Submitted to IEEE Real-Time Systems Symposium*, 2004.

[34] ——, “Synthesis of real-time implementation from UML-RT models,” in *Submitted to IEEE RTAS Workshop on Model-Driven Embedded Systems*, 2004.

[35] Z. Gu, S. Wang, J. C. Kim, and K. G. Shin, “Integrated modeling and analysis of automotive embedded control systems with real-time scheduling,” in *Proc. Society of Automotive Engineers Congress, Paper Number 2004-01-0279*, 2004.

[36] 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 *Proc. IEEE Real-Time Systems Symposium*, 2003, pp. 78–81.

[37] Z. Gu, S. Wang, and K. G. Shin, “Issues in mapping from UML Real-Time Profile to OSEK API,” in *Proc. UML Workshop on Specification and Validation of UML models for Real Time and Embedded Systems*, October 2003. [Online]. Available: http://www-verimag.imag.fr/EVENTS/2003/SVERTS/

[38] M. Harbour, M. H. Klein, and J. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. Software Eng.*, vol. 20, no. 2, pp. 13–28, 1994.

[39] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[40] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad, "Cadena: An integrated development, analysis, and verification environment for component-based systems," in *Proc. IEEE International Conference on Software Engineering*, 2003, pp. 160–172.

[41] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "A benchmark for comparing different approaches for specifying and verifying real-time systems," in *Proc. IEEE International Workshop on Real-Time Operating Systems and Software*, May 1993.

[42] T. Henzinger, P. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," *Software Tools for Technology Transfer, special issue on timed and hybrid systems*, pp. 110–112, 1997.

[43] G. Holzman, "The spin model checker," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[44] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 22, pp. 26–60, 1990.

[45] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Zheng, "Hyvisual: A hybrid system visual modeler," University of California, Berkeley," Technical Memorandum, 2003.

[46] (2004) The IBM Rational website. [Online]. Available: http://www-306.ibm.com/software/rational/

[47] (2004) The ILogix website. [Online]. Available: http://www.ilogix.com

[48] J. S. J., A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai, "Domain translation using graph transformations," in *Proc. IEEE International Conference on Engineering of Computer-Based Systems*, April 2003, pp. 159–168.

[49] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. M. Wheater, "Model checking of workflow schemas," in *Proc. IEEE International Enterprise Distributed Object Computing Conference*, 2000, pp. 170–179.

[50] G. Karsai, A. Lang, and S. Neema, "Tool integration patterns," in *Proc. Workshop on Tool Integration in System Developement, ESEC/FSE*, September 2003, pp. 33–38.

[51] G. Karsai, S. Neema, A. Bakay, A. Ledeczi, F. Shi, and A. Gokhale, “A model-based front-end to TAO/ACE,” in *Proc. Second Workshop on the ACE TAO*, 2002.

[52] C. M. Kirsch, T. A. Henzinger, and B. Horowitz, “Giotto: A time-triggered language for embedded programming,” in *Proc. ACM International Conference on Embedded Software, LNCS 2211*, 2001, pp. 166–184.

[53] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza, *A Practitioner’s Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[54] S. Kodase, S. Wang, Z. Gu, and K. G. Shin, “Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers,” in *Proc. IEEE Real-Time Technology and Applications Symposium*, 2003, pp. 181–188.

[55] H.-H. Kwak, I. Lee, and O. Sokolsky, “Parametric approach to the specification and analysis of real-time scheduling based on acsr-vp,” *Science of Computer Programming*, vol. 42, no. 1, pp. 49–60, Jan 2002.

[56] H.-H. Kwak, I. Lee, A. Philippou, J.-Y. Choi, and O. Sokolsky, “Symbolic schedulability analysis of real-time systems,” in *Proc. IEEE Real-Time Systems Symposium*, 1998, pp. 409–418.

[57] A. Ledeczi, M. Maroti, G. Karsai, and G. Nordstrom, “Metaprogrammable toolkit for model-integrated computing,” in *Proc. IEEE International Conference on Engineering of Computer-Based Systems*, March 1999, pp. 311–317.

[58] T. Little and A. Ghafoor, “Spatial-temporal composition of distributed multimedia objects for value-added networks,” *IEEE Computer*, vol. 24, no. 10, pp. 42–50, 1991.

[59] C. Liu and J. W. Layland, “Scheduling algorithms for multi-programming in a hard real-time environment,” *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[60] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and analysis of system architecture using rapide,” *IEEE Trans. Software Eng.*, vol. 21, no. 4, pp. 336–355, 1995.

[61] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. Wiley, 2000.

[62] J. J. Marciniak, *Encyclopedia of Software Engineering*. Wiley-Interscience, 1994.

[63] G. Martin, L. Lavagno, and J. Louis-Guerin, “Embedded uml: a merger of real-time uml and co-design,” in *Proc. IEEE International Symposium on Software/Hardware Codesign*, 2001, pp. 23–28.

[64] J. Masse, S. Kim, and S. Hong, “Tool set implementation for scenario-based multithreading of uml-rt models and experimental validation,” in *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003, pp. 70–77.

[65] (2004) The Mathworks website. [Online]. Available: http://www.mathworks.com

[66] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishing, 1993.

[67] P. Merlin and D. Farber, “Recoverability of communication protocols – implications of a theoretical study,” *IEEE Trans. Commun.*, vol. 24, pp. 1036–1043, Sept. 1976.

[68] M. Naedele, “Modeling and simulating functional and timing aspects of real-time systems by delegated execution,” in *Proc. IEEE International Conference on the Engineering of Computer-Based Systems*, 2000, pp. 64–72.

[69] (2004) The Object Management Group website. [Online]. Available: http://www.omg.org

[70] OMG, “CORBA Component Model, v3.0,” Object Management Group, Tech. Rep., 2002.

[71] ——, “UML profile for schedulability,performance, and time specification,” Object Management Group, Tech. Rep., 2003.

[72] T. Quatrani, *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.

[73] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[74] C. Ramchandani, “Analysis of asynchronous concurrent systems by Timed Petri-Nets,” Massachussets Institute of Technology, Tech. Rep. TR-120, February 1974.

[75] M. Saksena and P. Karvelas, “Designing for schedulability: integrating schedulability analysis with object-oriented design,” in *Proc. IEEE Euro-Micro Conference on Real-Time Systems*, 2000, pp. 101–108.

[76] D. Schmidt, D. Levine, and T. Harrison, “The design and performance of a real-time CORBA object event service,” in *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1997, pp. 434–445.

[77] M. Schulte, “Experimentation plan for the model-based integration of embedded software weapon system open experimental platform,” The Boeing Company, Tech. Rep., 2003.

[78] B. Selic, G. Gullekson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. Addison Wesley, 1994.

[79] D. Seto, J. Lehoczky, L. Sha, and K. Shin, “On task schedulability in real-time control systems,” in *Proc. IEEE Real-Time Systems Symposium*, 1996, pp. 13–21.

[80] D. Sharp, “Object-oriented real-time computing for reusable avionics software,” in *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2001, pp. 185–192.

[81] ——, “Execution framework description for the model-based integration of embedded software, weapon system open experimental platform,” Boeing, Tech. Rep., 2002.

[82] ——, “Execution scenario description for the model-based integration of embedded software, Weapon System Open Experimental Platform,” Boeing, Tech. Rep., 2002.

[83] (2004) The SDL Forum website. [Online]. Available: http://www.sdl-forum.com

[84] J. A. Stafford and A. L. Wolf, “Architecture-level dependence analysis for software systems,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 4, pp. 431–451, 2001.

[85] J. Stankovic, “Vest: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems,” University of Virginia, Tech. Rep., 2000.

[86] H. Storrle, “A evaluation of high-end tools for Petri-Nets,” University of Munich, Tech. Rep., 1998.

[87] J. Sun and J. W. Liu, “Bounding the end-to-end response times of tasks in a distributed real-time system using the direct synchronization protocol,” University of Illinois, Tech. Rep., 1996.

[88] J. Sztipanovits and G. Karsai, “Model-integrated computing,” *IEEE Computer*, vol. 30, no. 4, pp. 110–111, April 1997.

[89] (2004) The Telelogic website. [Online]. Available: http://www.telelogic.com

[90] (2004) The OMG MDA website. [Online]. Available: http://www.omg.org/mda

[91] (2004) The UML website. [Online]. Available: http://www.omg.org/uml

[92] (2004) The OSEK-VDX website. [Online]. Available: http://www.osek-vdx.com

[93] (2004) The Quantum Framework website. [Online]. Available: http://www.quantum-leaps.com/qf.htm

[94] (2004) The TimeSys website. [Online]. Available: http://www.timesys.com

[95] 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)*, vol. 40, pp. 117–134, 1994.

[96] K. Tindell, “Embedded systems in the automotive industry,” in *Proc. Embedded Systems Conference, Paper Number 407*, March 1999.

[97] S. Tripakis, “Description and schedulability analysis of the software architecture of an automated vehicle control system,” in *Proc. ACM International Conference on Embedded Software, LNCS 2491*, 2002, pp. 123–137.

[98] (2004) The MoBIES Homepage at University of California, Berkeley. [Online]. Available: http://vehicle.me.berkeley.edu/mobies/

[99] (2004) Uppaal2k: Small tutorial. [Online]. Available: http://www.uppaal.com

[100] (2004) The Vector CANTech website. [Online]. Available: http://www.vector-cantech.com

[101] J. Wang and Y. Deng, "Reachability analysis of real-time systems using time petri nets," *IEEE Trans. Syst., Man, Cybern.*, vol. 30, no. 5, pp. 725–736, 2000.

[102] J. Wang, Y. Deng, and M. Zhou, "Compositional time petri nets and reduction rules," *IEEE Trans. Syst., Man, Cybern.*, vol. 30, no. 4, pp. 562–572, 2000.

[103] (2004) The Webopedia website. [Online]. Available: http://www.webopedia.com

[104] (2004) The WindRiver website. [Online]. Available: http://www.wr.com

[105] S. Yovine, "Kronos: A verification tool for real-time systems," *Software Tools for Technology Transfer*, vol. 1, pp. 123–133, 1997.

[106] J. Zhao, "A slicing-based approach to extracting reusable software architectures," in *Proc. European Conference on Software Maintenance and Reengineering*, 2000, pp. 215–223.