SOLEIL: An Integrated Approach for Designing and Developing Component-based Real-time Java Systems

Ales Plsek

Département de formation doctorale en informatique
UFR IEEA

École Doctorale SPI Lille

SOLEIL: An Integrated Approach for Designing and Developing Component-based Real-time Java Systems

THÈSE

présentée et soutenue publiquement le 14 September 2009

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Aleš PLŠEK

Composition du jury :

Président du jury :	M. Pierre BOULET, Professeur	*Université de Lille I*
Rapporteurs :	M. František PLÁŠIL, Professeur	*Charles University*
	M. Laurent PAUTET, Professeur	*Telecom ParisTech*
Examinateurs :	M. Jean-Charles FABRE, Professeur	*INP Toulouse*
	M. François TERRIER, Professeur	*CEA LIST Saclay*
Directeurs de thèse :	M. Lionel SEINTURIER, Professeur	*Université de Lille I*
	M. Philippe MERLE, Docteur	*INRIA Nord Europe*

INRIA LILLE - NORD EUROPE - Laboratoire d'Informatique Fondamentale de Lille

Abstract

Over the last decade we witness a steady grow of complexities in real-time systems. Today, developers have to face real-time constraints in almost every software system, from embedded software to financial systems, internet services, and computer entertainment industry. To address this widespread challenge, the Real-Time Specification for Java (RTSJ) has been proposed. However, RTJS itself introduces many nonintuitive rules and restrictions that doom its programming model to be highly error-prone. Moreover, in contrast to the approaches for mainstream software development, the engineering technologies, tools, and frameworks for real-time systems are nowhere near as groundbreaking.

The vision behind this dissertation is to ultimately close the gap between real-time programming and today's software technology. Therefore, this dissertation investigates scalable software engineering techniques for RTSJ programming. Our fundamental philosophy is to introduce high-level abstractions of RTSJ concepts in order to leverage development of real-time Java systems.

As the first contribution of this thesis, we introduce *domain components* - an approach to unified expression and manipulation of domain-specific concerns along the software development lifecycle. We use the domain components to construct high-level abstractions of RTSJ specifics that ultimately allow developers to achieve full separation of functional and RTSJ-specific concerns in the development lifecycle. We thus allow developers to reuse and tailor the systems for variously constraining real-time requirements.

Second, we propose SOLEIL- a component framework for development of RTSJ systems, the framework introduces a development methodology mitigating the complexities of the RTSJ programming model. Furthermore, we introduce the HULOTTE toolset for automatic instantiation of developed applications. In this process, the functional implementation is separated from RTSJ-specific code which is automatically instantiated. In consequence, the development process is fully transparent, RTSJ complexities are hidden from the developers, and the process itself highly resembles to the standard Java development. Finally, the domain component concept and the RTSJ rules and restrictions are defined in the Alloy language which allows us to formally verify that the development process and outcoming software systems are compliant with RTSJ.

To validate the approach, we conduct several case studies challenging our proposal from different perspectives. First, performed benchmarks show that the overhead of the SOLEIL framework is minimal in comparison to manually written object-oriented applications while providing more extensive functionality. Second, considering the state-of-the-art RTSJ programming methods, we achieve better separation of functional and RTSJ concerns, thus increasing efficiency of the development process. Finally, we demonstrate universality of the domain component concept by showing its ability to address various domain-specific challenges.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

In the last decade we have witnessed an enormous boom in the field of *real-time systems* [IH92, Jen07]. The traditional areas of systems with real-time and embedded requirements - e.g. embedded or on-board software, experience exponential grow of software, doubling in size every 18-36 months, depending on the industry [McM04, Bou95]. However, new areas of software with various real-time constraints emerge as we speak, developers have to face the challenges of real-time programming in variety of systems - from financial software [SR08] to computer entertainment industry [RFP08]. A very near future will bring increased demand for large-scale, distributed, and reusable real-time systems with pressure on time-to-market and cost, e.g. [IBM07].

While we see this rapid growth of complexities at the level of real-time programming, the developments at the software engineering level are nowhere near as groundbreaking. There is still a little interest in real-time from the mainstream software engineering community. Today, real-time software developers find themselves working with tools (in the broadest sense of the word, including analysis and design techniques, middleware, programming languages, etc.) which have come of age in the era of the real-time embedded computing, not that of the software engineering for mainstream systems. To give a concrete example, in the world of programming languages for real-time software, usually assembler, Ada, or C/C++ languages are used, which fall behind twice to five times more in productivity and effectivity than the language of choice for most enterprise IT projects - Java [CDM+05, Nil04, Geo99]. Moreover, a steep learning curve of these languages makes hard to find and retain experienced developers.

As a result, there is a huge gap between the novel real-time software waiting to be developed and the software tools available to do so. A testament to this is the launch of a plethora of research initiatives over the past few years that target "software engineering for real-time programming" in one way or another. One of the dominant efforts is the *Real-time Specification for Java* (RTSJ) [BGB+00], where the goal is to use the Java language as an enabling technology to greatly reduce the efforts associated with developing and maintaining real-time software.

Today we can find the Java programming language in various types of real-time systems, from industrial control [RHN+07], audio processing [ABB+07b, JASB07], ship-board computing [IBM07], to avionics [ABC+07], and financial sector [BEA06]. Real-time Java is becoming ubiquitous. The progress in real-time Java technologies (real-time garbage collection [Daw08, Sie99, SUN08], ahead-of-time compilation [FS07], or operating system support [ABB+07a, MG07]) makes development of real-time applications in Java considerably easier.

However, despite the fact that RTSJ does not define any new language constructs and keywords the specification still introduces nontrivial concepts that in the end influence the programming style. We therefore believe that an ultimate success in this field can be achieved only by a framework that provides a unified approach to development of real-time Java-based systems and that leverages RTSJ concepts into higher levels of system design in order to mitigate the complexities introduced by RTSJ.

The vision behind our research is therefore to ultimately close the gap between real-time programming and today's software technology. Needless to say, this dissertation represents but a small step towards such an ambitious goal. In particular, we will focus on how to reconcile traditional software abstractions to facilitate development of real-time Java applications. More concretely, we employ advanced technologies of component-based software engineering (CBSE) [Cle02] and we extend them with new concepts in order to provide sufficient abstractions for RTSJ concerns. Ultimately, we propose a component framework supporting a full development lifecycle of RTSJ-based systems. Finally, we extensively explore the effects of our proposal on development of real-time Java systems, with stress on separation of concerns [Par72], extensive employment of generative programming, while increasing productivity and effectivity of programming with RTSJ.

In the remainder of this introductory chapter, we first highlight the problems to be tackled. Furthermore, we extensively discuss our research goals and formulate the statement of the thesis. We conclude the chapter with a preliminary overview of this dissertation's contributions and a roadmap to assist the reader in browsing the text.

1.1 Understanding the Problem

Despite the demonstrated utility of the Real-time Specification for Java, a number of open problems have limited its widespread acceptance. Briefly, from a software engineering perspective, these problems are:

- **RTSJ Memory Model** The memory model introduced by RTSJ brings many complexities to the development process. RTSJ defines three memory areas - Immortal, Scoped Memory, and Heap, each with its rules and restrictions. Therefore, for RTSJ programs to be correct, developers must deal with an added dimension: where a particular datum was allocated. Although many patterns and idioms were introduced [A. 05, BN03, PFHV04, BCC+03], the memory model makes RTSJ programming highly error prone. The controversial nature of the memory model further confirms the fierce discussions in the community [Lam05] where real-time garbage collection (RTGC) is often considered as a silver bullet. However, RTGC methods does not achieve performance needed for hard real-time systems. Therefore, we believe that before the RTGC will be matured enough, an effort should be made to facilitate RTSJ memory management with the methods of software engineering.

- **Programming Style and Development Process** Similarly as for the memory model, the RTSJ programming style defines additional rules and restrictions. Usually, for the same task different implementations must be provided depending of the real-time conditions under which the task will be executed. The specific real-time requirements not only influence application of particular code constructs but often propagate into the architecture. As a consequence, reuse of such a code between systems with different real-time requirements is almost imposable. However, with the growing complexity of real-time applications and time-to-market pressure we envisage that the reuse and adaptation of real-time applications will play a crucial role in the development process.

1.2 Research Goals

A complete process for designing of real-time and embedded applications based on RTSJ comprises many complexities, specially timing and schedulability analysis, which have to be included in a design procedure. The scope of our proposal is placed directly afterwards these stages, when real-time characteristics of the system are specified but the development process of such a system lies at its very beginning.

The goal of our work is to develop a component framework alleviating the RTSJ-related concerns during development of RTSJ-based real-time and embedded systems. Our motivation is to consider RTSJ-specific concerns as clearly identified software entities and clarify their manipulation through all the steps of software life cycle. The challenge is therefore to mitigate complexities of the RTSJ system development and offload the burden from users by providing appropriate technologies for management of RTSJ concerns.

Therefore, the goals of this dissertation are following:

- *Solution through Software Engineering Methods* – The proposal must be based only on standard RTSJ features and not depend on language extensions or on any special characteristics of particular VMs.
- *High-Level Abstractions* – We must introduce a higher level abstractions of RTSJ-concerns in order to represent them as first-class entities. This will allow developers to manipulate these concerns independently from the functional logic of the application. Consequently, we want to achieve full separation of concerns, hide the complexities of RTSJ, and allow programmers to develop RT applications as if using standard Java as much as possible. Ultimately, the goal is to provide a programming model easy to understand that facilitates the development of a majority of RT applications.
- *Formalization and Verification* – The proposed concerns must be formalized in order to support their validation. By this, developed applications will be validated to guarantee their compliance with RTSJ.
- *Component Framework* A component framework unifying development of RTSJ-based applications should be proposed. Furthermore, generative programming methods should be used to support automatic generation of RTSJ-related code. Finally, a verification process based on the introduced formalisms should be developed in order to validate conformance of developed applications to the formalized rules and restrictions.
- *Evaluation* – Finally, we want to evaluate the framework from both performance and software engineer perspectives. The framework should introduce only a minimal overhead, and a large case study should be conducted to demonstrate easy-of-use and software engineering benefits.

Thesis statement. An effective development process of RTSJ-compliant systems must consider RTSJ concerns at early stages of the system design and must provide their high level abstractions as the only way to avoid tedious and error-prone process when implementing them.

1.3 Contributions

It is always difficult to provide an overview of contributions in advance, with the problem statement only vaguely introduced and without the technical foundations required to support them. However, listing the contributions early on helps to sketch the context and subject domain of the dissertation. In short, this dissertation makes conceptual and technical contributions in the intersecting domains of real-time programming, component-based software engineering and real-time Java programming. The main contributions are summarized as follows and have been published as shown by the references:

- *RTSJ-specific Component Model and Domain Components* – We define a component model to address the specifics of RTSJ. The model introduces the concept of *Domain Component* that allows developers to represent the RTSJ concerns as first-class entities. We are thus able to easily manipulate with them during all stages of application development. [PMS08, PLMS08]

- SOLEIL *Framework* – We construct a component framework built on top of our RTSJ component model and we propose a methodology of RTSJ-based application development that fully separates functional and RTSJ-specific concerns. Furthermore, we formalize the component model defined and create the SOLEIL profile - a set of rules and guidelines for developers using the SOLEIL framework. Finally, we introduce an approach to validate conformance of developed applications to the profile. [PLMS08]

- HULOTTE *Framework* – We propose the HULOTTE framework that provides an approach to automatic instantiation of runtime platforms supporting execution of SOLEIL applications. The framework achieves a full separation of concerns and further employs methods of generative programming in order to take the advantages of CBSE also at runtime. Finally, different optimization heuristics are employed to reduce overhead of instantiated applications. [LMP+09]

- *Evaluation* – We evaluate our approach in several case studies, both from qualitative and quantitative perspective. First, we measure performance overhead of our approach and show that although introducing advanced CBSE technologies, we do not introduce any significant overhead. Furthermore, the approach is evaluated on a large case study identifying the potential of the approach to mitigate complexities of the RTSJ-oriented development process. Finally, we demonstrate framework extendability by applying it in the field of distributed and ambient computing. [PLMS08, PMS07, MPL+08]

1.4 Dissertation Roadmap

This dissertation is divided in four main parts. In the first part we conduct a state-of-the-art survey and precisely identify the goals of this dissertation. The second part presents our proposal. Consequently, the proposed ideas are validated in the third part and finally, we conclude and present perspectives in the fourth part. Below, we summarize each subsequent chapter in the dissertation.

Part I: State of the Art

Chapter 2: Real-Time Specification for Java In this chapter we first introduce Real-time Specification for Java, its key features and advantages. Furthermore, we present a motivation scenario of a RTSJ-based application and using this example we identify the complexities developers must face when programming with RTSJ. This motivation scenario will be revisited several times through the course of this dissertation to demonstrate various ideas of our proposal.

Chapter 3: Component-Based Software Engineering In this chapter we introduce Component-based Software Engineering and discuss how it can be applied to address the challenges of real-time programming. Consequently, we discuss and compare general purpose and RTSJ-dedicated component frameworks in order to identify their limitations. Furthermore, the FRACTAL component model is described in detail and we argue for employing FRACTAL as the enabling technology in this dissertation. Finally, we synthesize the facts stated in the state-of-the-art and show how the selected technologies contribute to meeting the goals of the dissertation, which are refined at the end of this chapter.

Part II: Proposal

Chapter 4: SOLEIL: A Framework for Java-based Real-Time Embedded Systems In this chapter we propose a RTSJ-specific component model and we also introduce the concept of domain components. Furthermore, we describe the SOLEIL framework providing methodology and validation for RTSJ-oriented development process. Finally, the ideas proposed in the chapter are demonstrated on the motivation scenario.

Chapter 5: HULOTTE: A Framework for the Construction of Domain-Specific Component Frameworks We propose the HULOTTE framework and show how it can be applied to leverage instantiation of applications developed in SOLEIL. Furthermore, we elaborate on implementation of the HULOTTE framework and show optimization heuristics used to reduce overhead of resulting applications. Furthermore, we discuss application of the HULOTTE framework in a more general case as a tool for instantiation of domain-specific component frameworks. Finally, the ideas proposed in the chapter are demonstrated on the motivation scenario.

Part III: Validation

Chapter 6: Case Studies This chapter applies the SOLEIL and HULOTTE frameworks in several case studies spanning different domains and challenges. The case studies serve both as a prove of concept and further evaluate our approach from various perspectives.

Part IV: Conclusion and Perspectives

Chapter 7: Conclusion and Perspectives In this chapter we summarize the contributions made in the dissertation. At that point we are able to evaluate the contributions of the dissertation with hindsight, naturally leading to a discussion on the limitations of this work and on possible directions for future research.

Part I

State of the Art

Chapter 2

Real-time Programming in Java

Contents

JAVA is a mature and widely accepted programming language; while Java has traditionally been relegated to non-safety-critical software, the acceptance of real-time and safety-critical Java technologies is increasing steadily. The Real-Time Specification for Java (RTSJ) [BGB+00] introduces the Java language into the world of real-time systems. To achieve this, the RTSJ was shaped by several guiding principles. Foremost among these is the principle to *hold predictable execution as first priority in all tradeoffs*. Another principle is that the RTSJ introduces no new keywords or other language constructs. Also, the RTSJ provides backward compatibility, meaning that existing Java programs run on RTSJ implementations. Despite this motivation for preserving all the principles of regular Java, RTSJ still introduces programming complexity that makes it difficult to build non-trivial applications.

In this chapter we provide a fundamental information about real-time systems and the challenges related to their development. Furthermore, we introduce RTSJ features and discuss their influence on the programming style. We however do not present the full list of RTSJ specifics [Dib08], we rather focus on those features that directly influence the programming style, showing on examples their benefits and flaws. The important motivation for this chapter is to expose the disadvantages of RTSJ, and thus explaining the immense difference between theoretical ideas proposed by RTSJ and its real-life experience.

Very soon after the release of RTSJ, the programmers realized the issues that are coming as a trade-off for a predictable Java-based program. Their concerns and remarks gave birth to the first efforts for research in RTSJ. Therefore, in the final part of this chapter we summarize almost a decade of research in RTSJ, focusing on the various approaches to RTSJ programming and

development. Our goal is to identify the current trends and evaluate their limitations in order to identify potential space for new contributions.

Contributions

The contributions of this chapter are:

- **Challenges in Real-time Programming.** We introduce the basic goals and challenges of the real-time programming domain.
- **Real-Time Specification for Java (RTSJ).** We introduce the basic principles of RTSJ. We discuss the features of RTSJ from the software developer's perspective and evaluate how they influence the programming style of developing real-time systems based on Java. Also, we discuss advantages and disadvantages of RTSJ and we highlight the crucial issues that must be faced when developing RTSJ-based systems.
- **Motivation Scenario.** We describe a SweetFactory case study in order to demonstrate RTSJ characteristics on a simple scenario. This case study will be revisited through the course of this dissertation in order to demonstrate various ideas discussed.
- **Beyond the Horizon of RTSJ.** Finally, we discuss the current research efforts focused on RTSJ, highlighting their advantages and limitations.

Structure of the Chapter

The rest of the chapter is organized as follows. In Section 2.1 we introduce basic challenges and issues of real-time programming. Furthermore, we discuss the programming languages usually employed for development of real-time applications. We introduce Real-time Specification for Java in Section 2.2. The section is giving an overview of the key RTSJ principles. Also, we highlight the advantages and disadvantages of RTSJ. Furthermore, we continue the discussion in Section 2.3 where we consider the related research projects with focus in RTSJ. Finally, a summary of this chapter is given in Section 2.4.

2.1 Real-time Programming

In this section we provide an overview of the real-time programming in general.

2.1.1 Real-time System Definition

Real-time system is a system in which its correctness depends not only on the logical result of the computations it performs but also on time factors [SR98]. The *real-time requirement* can be easily characterized by the sentence: *The right answer delivered too late becomes the wrong answer*. More precisely, the real-time performance requires predictable and efficient end-to-end control over system resources and imposes multiple quality of services (QoS) - predictability, throughput, scalability, dependability, security, etc. Meeting the requirements of a real-time end-to-end performance is generally considered as one of the hardest requirements in development of software systems.

In typical real-time software systems, each critical software part represents different requirements and tradeoffs. We distinguish two types of these requirements: hard and soft real-time [Jen07]. *Hard real-time* constraints are those for which an action performed at the wrong time will have zero or possibly negative value. The connotation of *hard real-time* is that compliance with all timing constraints is proven using theoretical static analysis techniques prior to deployment. *Soft real-time* constraints are those for which an action performed at the wrong time (either too early or too late) has some positive value even though it would have had greater value if performed

at the proper time. The expectation is that soft real-time systems use empirical (statistical) measurements and heuristic enforcement of resource budgets to improve the likelihood that software complies with timing constraints. Note that the difference between hard real-time and soft real-time doesn't depend on the time ranges specified for deadlines or periodic tasks. A soft real-time system might have a deadline of 100 microsecond, while a hard real-time system's deadline may be 3 seconds.

Based on these definitions, we consider software systems as systems that are composed of parts that are either hard-, soft- or non-real-time.

2.1.2 Developing Real-Time Applications

Due to the real-time nature of developed applications, the development process [But05] adds a number of aspects to the standard ones. Rather than developing systems with the stress on throughput as it is in the standard process, the developers must focus on system's predictability. Furthermore, with meeting the functional requirements of the system, the equal emphasis must be placed on meeting the real-time requirements. Already during the requirements analysis, the developer must formulate, together with the *functional requirements*, the description of the temporal behavior of the system - the *real-time requirements*.

Based on the real-time requirements established in the specification of the application, the developer defines the workloads, which are the basis of the *schedulability analysis*. The analysis allows designers to certify that, in the worst case, the activities scheduled in the application meet their real-time requirements. Usually, the Rate Monotonic Analysis [KRP+93] with the priority assignment process are employed. Based on the analysis, a *real-time model* is formed. The real-time model is a timing abstraction that holds all the qualitative and quantitative information needed to predict/evaluate the timing behavior of an application. It is used by designers to annotate timing requirements in the specification phase, to reason about the prospective architecture during design phases, and to guarantee its schedulability when the system is to be validated.

In modern systems, the schedulability analysis and the real-time model construction are performed with a tool support, between such tools we can list e.g. TIMES [AFM+02] and SYMTA/S [HHJ+05], or MAST [HGGM01].

A Real-time Scenario

To better illustrate the requirements of a real-time system, we introduce a typical real-time and embedded application scenario in Fig. 2.1. This example illustrates the basic concepts used to design a real-time application.

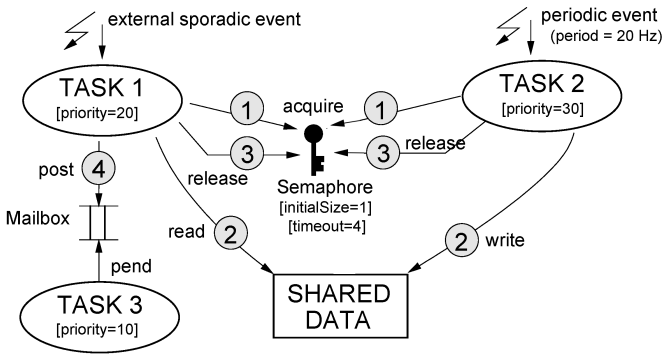


Figure 2.1: A Real-Time Application Example.

This example is composed of three tasks. The tasks `Task1` and `Task2` read and write a shared data which is protected by a binary semaphore. A task pending on the semaphore (via the `acquire` service) can not be blocked more than 4 time units (which corresponds to a timeout

specified by the semaphore). `Task2` is activated periodically by a timer, while `Task1` is activated in response to an external interrupt event. At the end of its execution cycle, `Task1` sends the content of the read data to the `Task3` using a mailbox.

2.1.3 Trends and Challenges

The future of distributed, real-time and embedded systems brings demand for large-scale, heterogeneous, dynamically highly adaptive systems with variously stringent QoS demands [BKT+06]. Moreover, the number of systems having some real-time requirements is rapidly increasing. As already showed by the RTSJ experience, there is a huge demand for systems composed of hard-, soft-, and non-realtime parts in the proportion illustrated in Fig. 2.2 [Vit08]. Such systems however require programming approaches that both allow development of a predictable code but also must provide an effective development of a non-real-time code. Here, a unified approach to development of such systems is crucial, since using different tools and approaches for programming real-time and non-real-time parts of systems is usually not possible due to their incompatibility.

Therefore, we summarize the current challenges in developing real-time systems.

- **Growing Complexity of RT Systems** The real-time systems are becoming pervasive. The real-time requirements are today present in almost any software, putting additional burdens on developers.
- **Market Pressure** Growing need for real-time systems has increased economic competition which has lead to the pressure on time-to-market delivery and cost reduction.
- **Development Process Challenges** Finally, new methodologies are needed to facilitate design, implementation and maintenance of real-time systems, while providing means to capitalize software development. The key features to achieve are:
  - *Increased Level of Abstraction.* The technologies must foster high-level design and development of systems in order to allow developers to easily manipulate even with advanced concepts which are inherently present in real-time software. This ultimately leads to software complexity reduction.
  - *Software Reuse.* A support for technologies of a reliable adaptation and software reuse is crucial since they boost software development and in the context of real-time mission-critical systems ease certification.
  - *Verification and Maintenance.* To provide a development approach allowing validation of applications along their development lifecycle. Such technologies allow early error detection and facilitate development by guiding the developers through the process while respecting the restrictions and limitations of the particular technology.

2.1.4 Real-Time Programming Languages

Real-time systems have historically been developed in hard-coded manner, e.g. with dedicated software written for specific types of hardware, using unstructured *spaghetti* designs and code. Usually not very productive or error-free programming languages have been used - e.g. assembler, or C/C++ where errors stem often from their memory management. Moreover, a steep learning curve of the real-time programming languages made hard to find and retain experienced developers, in case of the Ada language. This approach has yielded proprietary solutions that were tedious, error-prone and costly to develop, validate, and evolve.

To face these obstacles, the Java programming language seems to be a promising choice - mainly because of its simplicity, safety and for its cheap maintenance cost. The Java programming language has replaced C++ as the predominant programming language, largely because Java

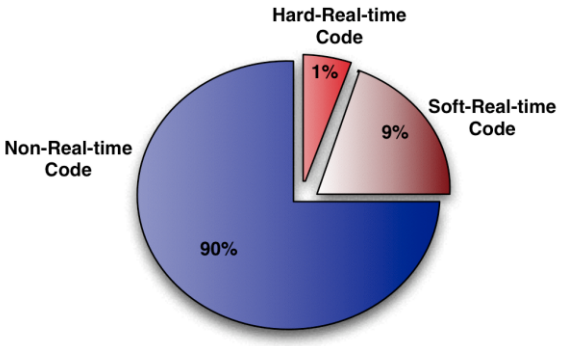


Figure 2.2: Proportions of the Source Code with Differently Stringent Real-Time Requirements

programmers are approximately twice as productive when developing new code and are five to 10 times as productive during maintenance of existing code [CDM+05, Nil04, Geo99].

However, conventional Java implementations are unsuitable for developing real-time embedded systems, mostly due to the lack of predictability. The main features contributing to a poor predictability of the Java language are: no scheduling control over threads, unpredictable synchronization delays, run-anytime garbage collection, coarse timer support, no event processing, and no safe asynchronous transfer of control.

The Real-time Specification for Java (RTSJ) [BGB+00], addresses these limitations through several areas of enhanced semantics. Moreover, it brings a higher-level view into the real-time and embedded world, which is desperately needed when avoiding accidental complexities and steep-learning curves.

2.2 Real-Time Specification for Java

In this section we describe the Real-Time Specification for Java [BGB+00] (RTSJ). However, the extensive discussion of all the features of RTSJ is beyond the scope of this dissertation, we will therefore focus on features of RTSJ that directly influence the programming styles of developers. For an exhaustive description of RTSJ we refer reader to [Dib08, BW01, Wel04].

RTSJ [BGB+00] is a comprehensive specification for development of predictable real-time Java-based applications. Between many constructs which mainly pose special requirements on underrunning JVM, two new programming concepts were introduced - real-time threads and special types of memory areas. These new concepts will be described in Section 2.2.1 and Section 2.2.2 respectively. Consequently, Section 2.2.3 introduces a motivation scenario to demonstrate application of RTSJ and we discuss the complexities related to RTSJ programming in Section 2.2.4. Finally, we conclude with a list of the most popular RTSJ implementations in Section 2.2.5.

2.2.1 Thread Types

In order to avoid critical tasks to lost their deadlines because of the garbage collector (GC), RTSJ makes distinction between three main kinds of tasks: (i) *Low-priority tasks* are tolerant with GC, (ii) *High-priority tasks* cannot tolerate unbounded preemption latencies by the GC, and (iii) *Critical tasks* cannot tolerate preemption latencies by the GC.

Low-priority tasks, or normal Java threads, are instances of the `java.lang.Thread` class and allocates objects within the heap. To implement the high-priority and critical tasks, RTSJ introduces two new types of threads that have precise scheduling semantics – `RealTimeThread` and `NoHeapRealTimeThread` (NHRT). We illustrate the new types of threads in Fig. 2.3. The

most important feature of these new threads is that they are scheduled preemptively so that the highest priority thread is always running.

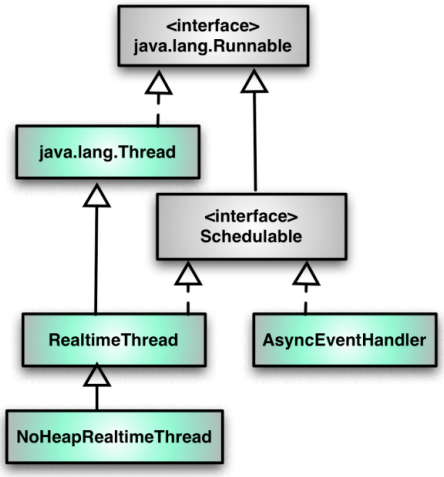


Figure 2.3: New types of threads introduced by RTSJ.

Real-time Thread

High priority tasks are instances of the `RealtimeThread` class, which extends the `Thread` class to support real-time tasks. Real-time threads can allocate objects within the heap, and within immortal and scoped regions (described in Section 2.2.2). Furthermore, parameters provided to the constructor of `RealtimeThread` allow the temporal and processor demands of the thread to be communicated to the system.

NoHeapRealTimeThread

`NoHeapRealTimeThread` (NHRT) extends `RealtimeThread` with the restriction that it is not allowed to allocate or even reference objects from the Java heap, and can thus safely execute in preference to the garbage collector. Such threads are the key to supporting hard real-time execution because they have implicit execution eligibility logically higher than any garbage collector. Since the NHRT can not access heap memory, it must operate in scoped or immortal memory.

Asynchrony

RTSJ defines mechanisms to bind the execution of program logic to the occurrence of internal and/or external events. In particular, RTSJ provides a way to associate an asynchronous event handler to some application-specific or external events. There are two types of asynchronous event handlers defined in RTSJ:

- The *AsyncEventHandler* class, with does not have a thread permanently bound to it – nor is it guaranteed that there will be a separate thread for each *AsyncEventHandler*. RTSJ simply requires that, after an event is fired, the execution of all its associated `AsyncEventHandlers` will be dispatched.
- The *BoundAsyncEventHandler* class, which has a real-time thread associated with it permanently. The associated real-time thread is used throughout its lifetime to handle event firings.

Event handlers can also be specified as no-heap, which means that the thread used to handle the event must be a `NoHeapRealtimeThread`.

RTSJ also introduces the concept of *Asynchronous Transfer of Control* (ATC), which allows a thread to asynchronously transfer the control from a locus of execution to another.

WaitFree Queues

In order to support communication between real-time and regular Java threads, RTSJ provides `WaitFreeQueues`. These queues provide a solution for sharing data between different NHRTs and also between NHRTs and heap-based threads. This strengthens the semantics of Java synchronization for use in real-time systems by mandating priority inversion [WHJ04] control. The wait-free queue classes provide protected, concurrent access to data shared between instances of `java.lang.Thread` and `NoHeapRealtimeThread`. Two queues are provided: `WaitFreeReadQueue` and `WaitFreeWriteQueue`.

`WaitFreeReadQueue` is a queue that can be non-blocking for consumers and is intended for single-reader multiple-writer communication, although it may also be used (with care) for multiple readers. A reader is generally an instance of `NoHeapRealtime- Thread`, and the writers are generally regular Java threads or heap-using real-time threads or schedulable objects. Communication is through a bounded buffer of objects that is managed first-in-first-out.

The `WaitFreeWriteQueue` class is intended for single-writer multiple-reader communication, although it may also be used (with care) for multiple writers. A writer is generally an instance of `NoHeapRealtimeThread`, and the readers are generally regular Java threads or heap-using real-time threads or schedulable objects. Communication is through a bounded buffer of objects that is managed first-in-first-out. The principal methods for this class are write and read.

2.2.2 Memory Management

RTSJ further distinguishes three memory regions: `ScopedMemory`, `ImmortalMemory`, and `HeapMemory`, where the first two are outside the scope of action of the garbage collector to ensure predictable memory access. Memory management is therefore bounded by a set of rules that govern access among scopes. RTSJ introduces these new types of memory areas, since the standard heap memory is heavily influenced by the unpredictable garbage collector. The need for scoped memory areas was argued in [BR02]. Readers interested in a discussion are encouraged to consults the paper [WP03]. We also discuss the characteristics and limitations of the real-time garbage collection at the end of this section.

`ImmortalMemory` is a single memory area that is shared among all threads. Objects allocated in the immortal memory live until the end of the application. In fact, unlike standard Java heap objects, immortal objects continue to exist even after there are no other references to them. Importantly, objects in immortal memory are never subject to garbage collection.

`ScopedMemory` is an abstract base class for memory areas having limited lifetimes. A scoped memory area is valid as long as there are real-time threads with access to it. A reference is created for each accessing thread when either a real-time thread is created with a `ScopedMemory` object as its memory area, or when a real-time thread runs the `enter()` method for the memory area. When the last reference to the object is removed, by exiting the thread or exiting the `enter()` method, finalizers are run for all objects in the memory area, and the area is emptied. Objects in scoped memory are never subject to garbage collection.

The RTSJ allows references across scopes. But as Java is a safe language it forbids the existence of dangling references. Therefore, every reference must always be a valid reference to a live object or null. To maintain safety, two rules are enforced:

- Because scoped memory areas can be shared, a reference counting technique is used to ensure that the objects in them are only reclaimed after all threads have finished using the memory area.

- Because a scoped memory area could be reclaimed at any time, it is not permitted for a memory area with a longer lifetime to hold a reference to an object allocated in a memory area with a shorter lifetime. This means that heap memory and immortal memory cannot hold references to objects allocated in scoped memory. Nor can one scoped memory area hold a reference to an object allocated in a lower (more deeply nested) memory area.

Conservatively speaking, these rules require that every memory access be checked to ensure that it does not violate the rules. Combined with the heap-access restrictions of no heap threads, this imposes some overhead at run-time.

Scoped memory areas may be nested, producing a scoping structure called a *scope stack*. Since multiple memory areas can be entered from an existing memory area, this scope stack can form a tree-like structure. One key relationship is as follows: if scope `B` is entered from scope `A`, then `A` is considered the parent of `B` and `B`, the child of `A`.

RTSJ introduces strict assignment rules, which can be expressed as follows:

- *"An object shall not reference any object whose lifetime could be shorter than its own"*, formulated by [HT08]
- Another important limitation is the *single parent rule*: *"A memory region can have only one parent, thereby preventing cycles in the scope stack"*, formulated by [Wel04]

The implication is that a single scope cannot have two or more threads from different parent scopes enter it. An important consequence of this restriction on scoping structure is that a real-time thread executing in a given region cannot access memory residing in a sibling region and vice versa.

We give an example of a RTSJ memory structure in Fig. 2.4 (inspired by [PFHV04]). It represents a valid scope structure composed of two memory scopes, immortal memory and heap memory. Notice the distinction between the instance of the `ScopedMemory` classes (Java objects) and the memory they denote. We show the `ScopedMemory` instance allocated within a parent scope holding a pointer to the start of the backing store used to allocate objects within that scope. The location of scoped memory instances is not directly related to their position in the scope hierarchy.

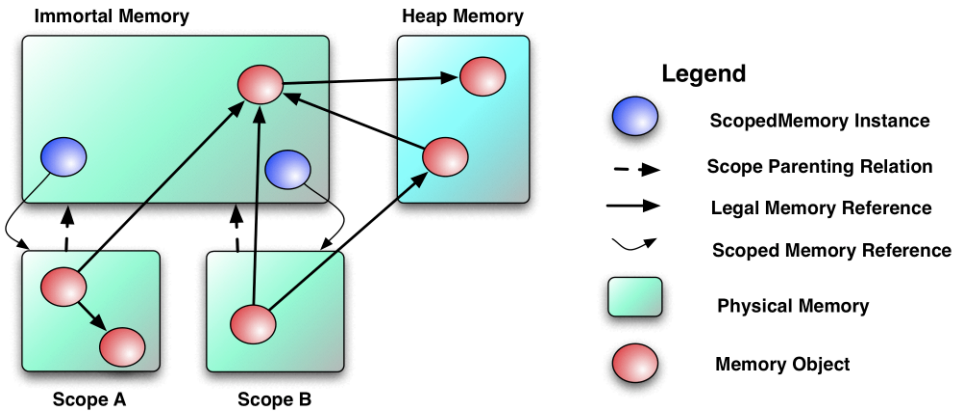


Figure 2.4: Memory Areas Defined by RTSJ : Two scoped memory areas parented in immortal memory. Heavy arrows represent allowed reference patterns. While any scope is allowed to refer into the heap, a `NoHeapRealtimeThread` is not allowed to read those references. These constraints are implemented by read/write barriers at runtime.

As we can see, the constraints and rules imposed by the memory management of RTSJ introduced many complexities into the development process. To resolve this, design patterns for

programming with scoped memory have been investigated by several projects [A. 05, BN03, PFHV04, BCC+03] . Rather than designing patterns, we must however comprehend these as implementation patterns providing guidelines how to implement different types of cross-scope communication.

Real-time Garbage Collection

A newly emerging direction in real-time Java programming is based on novel ways of memory management [PV08, PV06, Sie04]. Specially Real-Time Garbage Collection (RTGC) [Daw08, Sie99, SUN08] represents a promising approach. The main principle of RTGC is to perform garbage collection in pauses when the system is not working, this is usually enhanced by scheduling garbage collection predictably. However, RTGC still introduces some overhead, and although this overhead is bounded, this method is still far from being used in hard-real-time programming. Here, some applications have latency/throughput real-time requirements that cannot be met by current real-time garbage collection (GC) technology. Nevertheless, the RTGC is becoming a very successful technology in real-time systems, for example in the business sector [BEA06] or in large-scale, heterogeneous, dynamically highly adaptive systems [IBM07].

2.2.3 Sweet Factory - A Motivation Scenario

To better illustrate main specifics of RTSJ, we introduce an example scenario that will be revisited several times through the course of this dissertation. The goal of this motivation scenario, called *Sweet Factory*, is to implement an automation system controlling an output statistics from a production line in a sweet factory and report all anomalies. The example represents a classical scenario, inspired by [GHMS07], where both real-time and non-real-time concerns coexist in the same system.

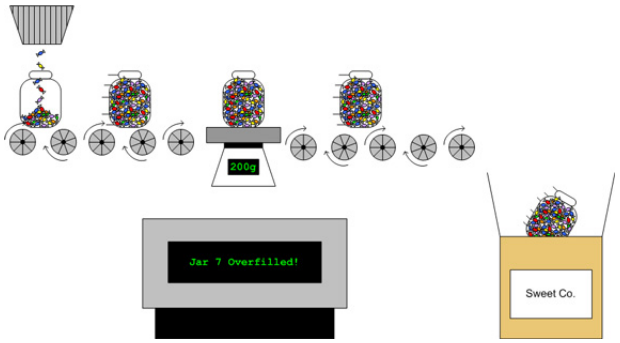


Figure 2.5: Sweet Factory Illustration

The system consists of a production line that periodically generates measurements, and of a monitoring system that evaluates them. Whenever abnormal values of measurements appear, a worker console is notified. The last part of the system is an auditing log where all the measurements are stored for auditing purposes. We illustrate the functionality of the system in Fig. 2.5. Since the production line operates in 10ms intervals and no deadline can be missed, the system must be designed to face under hard real-time conditions.

In Fig. 2.6 we show a class diagram of the application. Considering application of RTSJ, we witness several interesting issues. Although this simplified version of the Sweet Factory contains only four classes, it already contains many of the key features of RTSJ. The `MonitoringSystem` is the central class of the application, collecting the measurements produced by the `Production-Line` and processing them. Since `ProductionLine` is producing measurements in 10ms intervals and the `MonitorSystem` must proceed these measurements without dropping any of them,

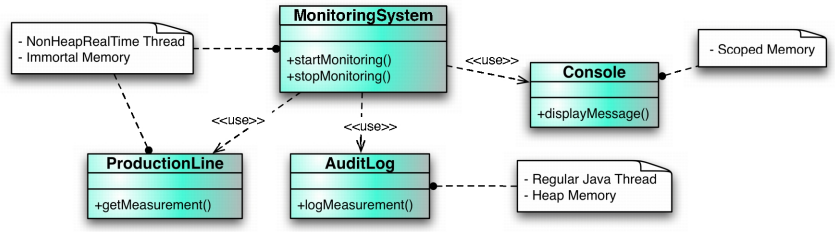


Figure 2.6: Sweet Factory Class Diagram

both classes must be therefore executed under strict hard real-time conditions. This leads to application of the `NonHeapRealTimeThread`. The task of the `ProductionLine` is to produce the measurements, therefore we will assign it the highest priority in the application. As the `MonitoringSystem` is in this sense dependant on the `ProductionLine`, it runs with the second highest priority.

```
1  public class MonitoringSystemImpl
2    implements MonitoringSystem, Runnable {
3
4    IProducer iProducer;
5    AuditLog log;
6    Console console;
7
8    private final ScopedMemory scope;
9
10   private final ReportRunnable reportRunnable
11       = new ReportRunnable(console);
12
13   public void initialization () {
14     runInArea(ImmortalMemory.instance(),
15       new Runnable(){
16         public void run() {
17           try {
18             console = new Console();
19           }
20           catch (IOException e) {
21             throw new RuntimeException(...);
22           }
23         });
24     }
25 }
26
27 class ReportRunnable implements Runnable {
28   Measurement measurement;
29   ServiceInterface iService;
30   void setM(Measurement m) {
31     measurement = m.deepCopy();
32   }
33
34   void run() {
35     console.reportError(measurement);
36   }
37 }
```

```
38   public void startMonitoring () {
39     Bootstrapper.runInArea(
40           ImmortalMemory.instance(),this);
41   }
42
43   public static void runInArea(
44             MemoryArea area, Runnable r){
45     RealtimeThread t = new RealtimeThread(new
46         PriorityParameters(
47           PriorityScheduler.MIN_PRIORITY)
48                 null, null, area, null, r);
49     t.start();
50     try {
51       t.join();
52     } catch (InterruptedException e) {
53       throw new RuntimeException(...);
54   }
55
56   public run() {
57     while (true) {
58           Measurement m =
59         iProducer.getMeasurement();
60       if (m.isWrong()) {
61         reportRunnable.setM{m};
62         scope.enter(reportRunnable);
63       }
64       log.write(m);
65       waitForNextPeriod();
66     }
67   }
68 }
```

Figure 2.7: `MonitoringSystem` Implementation

When looking at the remaining two classes - `Console` and `AuditLog`, they implement functionality which does not have any special real-time constraints. Therefore, `Console` class will be operating in a scoped memory, whereas the `AuditLog` logic is delegated to a regular Java thread benefiting from the garbage collected heap.

Continuing with the discussion, the communication between different classes of the system must be designed. According to RTSJ rules, `ProductionLine` and `MonitoringSystem` com-

municate asynchronously since they have different priorities. The same applies for communication between `MonitoringSystem` and `AuditLog`. For both, wait-free queues need to be used, as specified in RTSJ.

For illustration, in Fig 2.7 we show implementation of the `MonitoringSystem` class in RTSJ. Although the task of this class is simply to receive a measurement from the `ProductionLine`, compare it, report measurement in case of an error, and finally log the measurement, the actual implementation in RTSJ is quite complex. During the initialization, each object must be allocated in a dedicated memory area, this is illustrated by the initialization method, line 13, that allocates `Console` in immortal memory. Furthermore, when starting the computations, a switch to appropriate memory area must be performed, line 38. Finally, the computation starts, line 56, but the developers have to be still aware in which memory area is the code running and perform an explicit switch when necessary, as line 62 shows. The resulting code is thus mixed with the RTSJ incidental specifics and the functional logic can be hardly recognized.

2.2.4 Advantages and Disadvantages of RTSJ

The biggest advantage of RTSJ is its ability to support hard-, soft- and non-realtime tasks in the application at the same time and allowing them to interact between each other. Using RTSJ for hard real-time tasks means that developers can lean over the well known semantics of the Java world while respecting RTSJ defined restrictions. At the same time, the full scale of Java benefits (e.g. garbage collection, Java libraries) can be used when developing those tasks that do not bring any real-time constraints.

However, there is no silver bullet when introducing Java into the real-time world. Despite the effort to preserve semantics form the regular Java, RTSJ introduces a new programming style. This is caused mainly by the new memory model that on one hand achieves predictability, while on the other hand introduces a non-intuitive programming techniques. As extensively discussed in the literature, e.g. in [PV06, Nil06, ACG+07, ABG+08], the Scoped and Immortal memories introduce a new level of complexities for developers.

To better illustrate the complexities of RTSJ, we discuss them in the context of the motivation scenario introduced in the previous section. For illustration, in Fig. 2.6 we have highlighted by annotations the classes with the corresponding RTSJ concepts that help to implement their functionality. As we can see, real-time and non-realtime concerns are mixed together - whereas `MonitoringSystem` and `ProductionLine` are executed under hard real-time conditions, the `AuditLog` is a regular Java class. Therefore, in such system, identification of those parts that run under different real-time constrains is difficult. Hence the design of communication between them is clumsy and error-prone. RTSJ introduces rules and restrictions, e.g. on application of wait-free queues, that must be respected. However, reasoning about the implementation is hampered by the complexity of functional and RTSJ concepts. As a consequence, the developer has to face these issues at the implementation level which brings many accidental complexities.

Indeed, solving these issues during the implementation is an error-prone process. Following RTSJ rules is hard due to their non-intuitive nature. This is because for a RTSJ program to be correct, developers must deal with an added dimension: where a particular datum was allocated. Design patterns and idioms for programming effectively with scoped memory have been proposed [A. 05, BN03, PFHV04, BCC+03], but anecdotal evidence suggests that programmers have a hard time dealing with `NoHeapRealtimeThread` and that resulting programs are brittle.

Furthermore, respecting the RTSJ rules often leads to modification of architectural concepts - e.g. changing a communication style from a synchronous to asynchronous. Enforced by the rules of the RTSJ thread model, these modification potentially have a deep impact on the logic of the whole system. Finally, developing RTSJ in this ad-hoc manner fully prevents any reuse in different real-time conditions, since the RTSJ-specific concerns are tangled with the functional implementation and they even propagate into the system architecture. Consequently, components may work just fine when tested independently, but break when put in a particular scoped memory context.

Based on these shortcomings, many approaches to their mitigation have been proposed, we discuss them in Section 2.3.

2.2.5 Real-time Java Virtual Machines

In order to use RTSJ, a specially designed and implemented Real-time Virtual Machine (RT VM) is needed. The design and implementation of RTSJ virtual machines have been documented in several projects focused on different domains: e.g. RT VM development [Gre05, BCF+06, ABC+07, Ang02], ahead-of-time compilation [FS07], or memory management implementation [CC03, WSBR01, PV03, ZNV04]. Furthermore, RT OS enforcing predictability is the key requirement, experiences on developing such OS have been also reported e.g. in [ABB+07a, MG07].

Nowadays, many commercial RT VM are available, we provide the list of the most popular of them, (sorted by first release date)[1]:

- **TimeSys RTSJ Reference Implementation** Only licensed for non-commercial use. Runs on X86/Linux. Available at www.timesys.com.
- **Java RTS** Sun Java SE Real-time (Java RTS). Runs on Sparc/Solaris (Beta for SUSE Linux Enterprise Realtime 10, and Red Hat Enterprise MRG 1.0.). Available at http://java.sun.com/javase/technologies/realtime/rts/.
- **IBM WebSphere Real Time**. IBM WebSphere Real Time V2 for Real Time Linux, running on main-line real-time Linux distributions (Red Hat MRG and Novell SLERT). However, there are limitations on the hardware supported. The main reason for the hardware limitation is that the supported models have modified firmware to allow better control over SMI event prioritization, in order to achieve determinism of the underlying hardware. Available at www.ibm.com/software/webservers/realtime/.

Furthermore, other real-time, Java-like platforms have been developed, either as commercial or academic projects. Usually, these VMs are not fully compliant with RTSJ.

- **OVM** is an academia research project at Purdue University that implements RT VM. Available at http://www.ovmj.org
- **JamaicaVM** implements RTSJ and a deterministic garbage collector. Runs on various platforms. Available at http://www.aicas.com/jamaica.html.
- **jRate** (Java Real-Time Extension) is an extension of the GNU GCJ compiler front-end and runtime system which adds support for most of the features required by the Real-Time Specification for Java [Ang02]. Available at http://jrate.sourceforge.net/.
- **Aonix PERC** is a commercial project targeting many hard real-time, safety critical and embedded systems. The PERC platform is introducing many trade-offs and therefore is not fully compliant with RTSJ, however, allows developers to target a broader scope of applications rather than having strict limitations as it is the case of RTSJ. Available at www.aonix.com/perc.
- **aJ100** is a hardware coded JVM. Available at www.ajile.com.
- **JRockit Real-Time** JRockit Enterprise Java Runtime provided by ORACLE is a Java VM featuring real-time garbage collection suitable for soft real-time systems. Available at www.oracle.com/jrockit.
- **IBM/Apogee Aphelion**. Aphelion is comprised of reliable and high performance JREs for deploying Java applications on devices based on embedded systems. Supports many OS platforms, further details at http://www.apogee.com/

[1] The provided list of VMs corresponds to the status in June 2009.

During the course of this dissertation, we have been using **Java RTS** VM provided by SUN, running on Linux 2.6.24 kernel with RT-Preempt patch [MG07]. Furthermore, for debugging and testing purposes we have been using also the **JamaicaVM** running on a standard Windows XP platform. Although this configuration does not provide a sufficient level of predictability, it was suitable for simple tests and prototyping.

2.3 Beyond Real-Time Specification for Java

The Real-time Java programming language has become a viable platform for real-time systems. From the earlier experiments with RTSJ [BCC+03, NB03], to its first success - the ScanEagle Project [Boe05] where Real-Time Java performed better than C++ [ABC+07], RTSJ has gradually earned its credit. Today we can find RTSJ in various types of real-time systems, from industrial control [RHN+07], shipboard computing [IBM07], audio processing [ABB+07b, JASB07], to avionics [ABC+07], and financial sector [SR08, BEA06]. High performance real-time Java virtual machines are now available from multiple vendors. RTSJ has found its place in the world of real-time programming.

This recent significant increase of interest in real-time Java is reflected by an intensive research in the area. We notice movement from research of RTSJ compliant implementation patterns and idioms [A. 05, BN03, PFHV04] to more general approaches to RTSJ. However, despite before mentioned achievements of RTSJ, the issues of RTSJ discussed in Section 2.2.4 are yet to be addressed fully.

Nowadays, a significant effort is dedicated to introduction of component-based software engineering (CBSE) into the world of real-time Java [Nil07]. However, we leave the discussion of component frameworks for RTSJ to Chapter 3 where an exhaustive study of CBSE is conducted. In this section we discuss other relevant projects with focus in RTSJ.

Extensions to RTSJ

First, many extensions and modifications of the RTSJ memory model [HT08, BVGVEADK06] were proposed, these proposals however stay on the theoretical basis since they can not be evaluated on standard RTSJ implementations where they are not supported. In this dissertation we therefore do not consider them in order to stay in compliance with original RTSJ.

Enhancing the RTSJ Programming Model

On the other hand, new approaches were proposed to increase safety and enhance manipulation with the RTSJ code. Nilsen [Nil06] proposes a type system that enables programmers to develop code for which the byte code verifier is able to prove the absence of scoped memory protocol errors, thereby eliminating the need for run-time assignment checks. Benefits of the type system include improved software reliability, easier maintenance and integration of independently developed real-time software modules, and higher performance.

Similarly, the work introduced in [BV07] investigates fitness criteria of RTSJ in model-driven engineering process that includes automated code generation. The authors identify a basic set of requirements on code generation process. We further confront our approach with these requirements in Chapter 6.

New Programming Models for RTSJ

Apart from the efforts towards enhancement of RTSJ code implementation, new programming models for RTSJ have been proposed.

The STARS project [ACG+07] presents a new programming model for RTSJ based on aspect-oriented approach. Here, the real-time concerns are completely separated from applications base

code. Although, as shown in [SPDC06], aspect- and component-oriented approaches are complementary, the component-oriented approach offers higher-level perspective for system development and brings a more transparent way of managing non-functional properties with only slightly bigger overhead. Since the work has conducted a case study on the same application as in this project, we state further comparisons in Section 6.2.

Flexible Tasks Graphs [ABG+08] define a new restricted thread programming model for Java. As the biggest advantage of the model we see the static safety of memory operations. Runtime checks of these operations therefore does not have to be performed which brings a very good performance.

2.4 Summary

The goal of this chapter was to present the Real-time Specification for Java. First, we have presented RTSJ and its basic concepts. Our motivation was to consider RTSJ from a point of view of an application developer and we have discussed the principles of RTSJ that substantially influence the programming style used. Therefore, we have summarized the key advantages and disadvantages of RTSJ (Section 2.2.4) and illustrated their impact on RTSJ application in a motivation scenario (presented in Section 2.2.3). Second, we have considered the afore mentioned issues from a research perspective, discussing the current trend and technologies in the domain of real-time Java programming (Section 2.3).

Chapter 3

Component-Based Software Engineering

Contents

THE goal of this chapter is to provide a sufficient background about the technologies and fundamental principles employed in this dissertation. We introduce Component-based Software Engineering [CCL06] - the key technology that we employ when facing the challenges of real-time programming. Consequently, we discuss the state-of-the-art component frameworks focused on RTSJ, provide their comparison and identify the limitations. Furthermore, we chose the FRACTAL Component model [BCL+06] as the technological platform for our research. We elaborate more on the features of the model and we discuss the benefits that convinced us to use this platform.

At the end of this chapter we provide a synthesis of technologies presented in this and in the previous chapter and discuss how they complement to each other. Based on this discussion we restate and define more precisely the goals of this dissertation.

Contributions

The contributions of this chapter are:

- **Component-based Software Engineering.** We introduce Component-based Programming and present its key technologies contributing to effective development and deployment of

software systems. Also, we provide a brief overview of selected component models and extract their key advantages.

- **Component Frameworks for RTSJ.** We discuss the cutting edge technologies based on RTSJ and evaluate their limitations. The motivation is to consult the current research trends in RTSJ and evaluate how they address the issues that we have identified as the crucial disadvantages of RTSJ. Notably, we refer on the benefits that are achieved by using CBSE in the real-time Java domain.

- **FRACTAL Component Model.** Finally, we chose the FRACTAL Component Model as the key technological platform in this dissertation and we argue for this decision by presenting the key advantages of the FRACTAL component model.

- **State-of-the-Art Summary.** We synthesize the state-of-the-art and highlight the key challenges to be addressed in this dissertation. Particularly we consider presented technologies – Real-Time Systems, CBSE, and RTSJ, and show how they can be combined.

- **Goals Refinement.** Based on the state-of-the-art presented in this sections we refine more precisely the goals of the dissertation.

Structure of the Chapter

The rest of the chapter is organized as follows. We present Component-based Software Engineering in Section 3.1. Our motivation is to introduce the basic terminology and the key technologies used when developing component-based applications. Furthermore, in Section 3.2 we show application of CBSE in practice, conducting a survey of component frameworks from general purpose ones to RTSJ dedicated ones. In Section 3.3 we focus on the FRACTAL Component Model and present it in more details, discussing its key features and finally showing some FRACTAL research directions that are related to the goals of this dissertation. Finally, in Section 3.4 we summarize the presented technologies, based on this, we refine more precisely the goals of this dissertation in Section 3.5. Finally, a summary of this chapter is given in Section 3.6.

3.1 Component-based Software Engineering

Component-based Software Engineering (CBSE) [Cle02] has emerged as a technology for the rapid assembly of flexible software systems. The success of this technology has been proved by variety of its applications, from general component frameworks [BCL+06, BHP06, CBG+08] to domain specific component frameworks (DSCF) addressing a wide scale of challenges — embedded [vOvdLKM00] or real-time constraints [PLMS08, HACT04], dynamic adaptability [FSSC08, GER08], distribution support [SVB+08], and many others.

CBSE is a branch of software engineering that studies the design and construction of software systems as explicit compositions of software units (components). Various definitions of a software component have been proposed in the literature, among the most accepted one, we quote [Cle02]: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

As the main benefits of CBSE we consider:

- **Separation of Concerns.** CBSE easily achieves full separation of concerns in applications, since the functionality of the system is implemented in fine-grained component architectures reflecting the functional logic of the system. Furthermore, specific types of components can be used to implement non-functional services, achieving separation of functional and non-functional concerns.

- **Software Reuse.** CBSE fosters software reuse [Sch99] since a software component is considered as an independent entity with exactly specified interfaces and as such can be subject of reuse.
- **Development Process.** The development process of component-based systems [CCL06] is based on the principles of reuse and separation of concerns that increase product reliability and stability with shorter development time and reduced cost.
- **Static and Runtime Adaptation.** CBSE allows component-based systems to achieve a sufficient level of granularity in order to permit modification or update of specific subparts of the system both statically and dynamically at runtime (using e.g. reflection [LLC07]).

3.1.1 Component Frameworks

A *component framework* is composed of a component model and the tool support which permit assembling, deploying and executing component-based applications. Component frameworks simplify development of software systems. A proper component model represents cornerstone for each component framework, its extensiveness substantially influences the capabilities of a component model.

Development Roles The *component-based development process* [CCL06] introduces a methodology for development of software components and of systems based on software components. We distinguish two types of development roles involved in this process — *application developer* and *framework developer*. Application developer is responsible for development of functional components. The role of the framework developer is to develop the tool support and the runtime platform that will guarantee instantiation, deployment, and execution of the component application.

3.1.2 Advanced Technologies in CBSE

Since the basic principles of CBSE were presented, we can introduce a selection of more advanced concepts and recent research trends related to this disseratation.

Domain-Specific Services

Component applications often have to meet additional *domain-specific services* - in the literature [Mor06, DEM02, APPZ04, EPFD01] referred to also as *non-functional requirements/aspects/services/properties/concerns*. By these services we mean all the services required for a proper functioning of the component, for example logging, security, transaction support, persistence, or distributed communication support.

These domain-specific services are orthogonal to the functional logic of the component. Therefore their implementation tangled with the functional implementation is not desired for several reasons. Mainly, crosscutting of these services with the implementation hampers reuse of the component across domains where different domain-specific services can be required.

Therefore, Moreno [Mor06] argued that domain-specific services should be placed in a custom made containers and showed how generative programming technique, in this case using Aspect-Oriented Programming (AOP), can be employed to generate tailorable containers by composing different domain-specific features. We further refer to the technology of component containers.

Component Containers

Component-oriented container paradigm [Mic] defines that each component is wrapped by a controlling environment called *container* (also called *membrane* in [SPDC06]). Its task is to relieve the developer from dealing with various domain-specific services required by the component.

This approach is crucial for achieving the separation of concerns be implementing the functional part in the component while deploying the domain-specific services into the dedicated container where they are hidden from end users. As a consequence, the functional implementation is not tangled with domain-specific concerns and thus can be easily reused in a different domain using an according container.

The container of a component is implemented as an assembly of so-called *control components*. Additionally, special control components called *interceptors* can be deployed on component interfaces to arbitrate communication between the component and its environment, they are also integrated in the container.

Component Connectors

Apart from components, there is an emerging trend to view also the interaction among components as a first-class concept – modeled by an entity called *connector* [MDT03, BP04]. A connector is often used in design stage, where it represents an interaction of a set of components (i.e., it realizes a binding).

During the past few years, connectors found their position even at runtime, actually realizing inter-component communication [MDT03]. A connector at runtime is an inherently distributed entity that is typically responsible also for addressing distribution (by using a middleware) and for solving minor incompatibilities among components by employing adaptation.

Runtime Platform Construction

The execution machinery deployed at runtime to support execution of an instance of the component model is called *runtime platform* (in the literature also referred as *execution infrastructure*). Under this term we therefore refer to all the glue-code introduced by the framework itself to support instantiation of a component system and to support the system during the runtime. This therefore includes the instantiation and deployment code, implementations of component containers and connectors, and other related code. In the broader sense of this definition, we can also considers *middleware* as a part of the runtime platform. Such platform thus, apart from instantiation and runtime support, provides other non-functional services - multiple distribution models, concurrency support, various communication protocols, and many others.

Current trend in developing and implementing the runtime platform emphasizes a generative programming approach [CE00, ZPH08]. While this task can be seen only as an engineering challenge, the runtime platform plays a crucial role in deciding whether the component model itself will be successful in real-life applications, since characteristics of its implementation have a direct impact on the performance of a given application. Here, different optimizations, as discussed in [LP08], should be employed to mitigate notoriously known problem of CBSE systems — performance overhead (caused e.g. by inter-component communication).

Generative Programming in CBSE

As said, the recent trend in development of component frameworks is the application of generative methods [CE00, JKSS04] to achieve a generic approach to their instantiation. Moreover, having component containers or connectors also at runtime is feasible only when these entities can be automatically generated. Otherwise, a developer would be forced to implement each concept in an application separately, which is in overwhelming majority of cases impossible. The impracticableness of the manual approach is yet further magnified by the fact, that some of the properties (e.g. for connector configuration) are known only as late as at deployment. Therefore, generative programming technologies are recently proposed to instantiate the runtime platform supporting execution of the application, we can witness several interesting approaches. First, a general approach to generation of component connectors based on high-level specification [Bur06]. A generative approach to automate the instantiation process of a runtime platform for on-board

systems is proposed in [CCPS03]. Recently, Bures et al. [BHM09] propose a *meta-component system*, which provides a software product line for creating custom component systems. The authors summarize properties and requirements of current component-based frameworks and propose a generative method for generating runtime platforms and support tools (e.g. deployment tool, editors, monitoring tools) according to specified features reflecting demands of a target platform and a selected component model.

Furthermore, the generative programming approach can be also used when instantiating the runtime platforms and middleware, addressed e.g. in [ZPH08]. Usually, such approach is used to instantiate a middleware layer that directly fits the requirements of a particular application. The benefits of such approach are a complete performance gain and reduced footprint.

Verification of Software Components

The verification of system correctness becomes crucial when developing a distributed embedded application for mission-critical systems. Here, the ability to support the exhaustive verification of applications is a considerable attribute.

However, as we have shown in [PA08], the component-oriented approach is beneficial for formal verification of systems. Usually, complex systems can not be exhaustively verified by the methods of model checking [E. 00] since they generate an immense state space which can not be fully traversed. However, software components represent a considerably smaller state space. It is therefore natural to tackle the problem of software component verification.

3.2 State-of-the-Art of Component Frameworks

In this section we discuss the state-of-the-art of component frameworks. First, we focus on general purpose frameworks in order to identify how the advanced technologies of CBSE are supported in practice. Furthermore, we consult component frameworks for distributed, real-time and embedded systems, arguing on examples for the benefits of employing the CBSE technology in this domain. Finally, we discuss and compare component frameworks dedicated to RTSJ, evaluating their concepts and revealing their limitations.

3.2.1 General Purpose Component Frameworks

With the boom of component-based technology, a plethora of component models is emerging as we speak. Each component model has its specifics and particularities that reflect its focus and the application domain it is intended for. Interested reader can find inspiring discussions and comparisons of component models in numerous surveys, e.g. in [BHM09, CBG+08, LW07]. We however focus more in detail on selected component models to highlight interesting features and flaws.

Therefore, we first present two selected general purpose component models - FRACTAL and SOFA component model. Second, in the following section, we draw our attention to domain specific component models dedicated to distributed, real-time and embedded systems.

FRACTAL Component Model

FRACTAL [BCL+06] is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms or graphical user interfaces.

The FRACTAL initiative [BCS09] represents almost a decade of both academia and industry successful projects with many research outcomes and industry applications. The key features of FRACTAL are its openness and lightweight. FRACTAL provides an open approach to CBSE allowing developers to introduce various extensions wit minimal restrictions. Last but not least FRACTAL provides a rich tool support [OW209a].

The key governing benefit when using FRACTAL is its stress on separation of concerns. The provided model is hierarchical, supporting full separation of functional concerns by allowing developers to design appropriate components. Furthermore, separation of functional and non-functional concerns is achieved through a controlling layer that implements domain-specific services thus preventing them to crosscut the functional implementation.

Additionally, the research presented in [LLC07] focuses on reliable reconfiguration of FRACTAL applications. Although not directly solving reconfiguration of real-time systems, the work justifies the ability of FRACTAL to extensively support reliable ways of reconfiguration.

We describe the FRACTAL component model in more details in Section 3.3.

SOFA

SOFA [BHP06] is a distributed component model, the result of several years of experience in working on both SOFA and FRACTAL component models. In [BHP06], the main limitations of SOFA are identified as: (i) having a limited support for dynamic reconfigurations, (ii) lacking of a structure for the control part of a component, (iii) and having an unbalanced support for multiple communication styles.

SOFA 2.0 supports dynamic reconfiguration (i.e., adding and removing components at runtime, passing references to components, etc.). It proposes some reconfiguration patterns in order to avoid uncontrolled reconfigurations which lead to runtime errors. For structuring the control part of a component, SOFA 2.0 introduces microcomponents and control interfaces. Microcomponents are minimalist components: they are flat (there are no nested microcomponents); do not have any connectors; are not distributed. The parallel to FRACTAL would be FRACTAL 's controllers. Control interfaces are orthogonal to business interfaces in the sense that they focus on non-functional features of components. These interfaces are in direct relation to the control interfaces found in FRACTAL. In SOFA 2.0, multiple communication styles are supported thanks to classes of connectors. Additionally, an approach to automatic specialization of connectors to match the runtime properties is also supported.

3.2.2 Domain-Specific Component Frameworks

Typically, a domain-specific component framework (DSCF) is composed similarly as a general purpose component framework. In addition, it defines relevant concepts, called *domain-specific concepts*, according to the requirements of the targeted application domain (e.g. to address the distribution support or real-time constraints). In this section we focus on the domain of distributed, real-time and embedded Systems (DRE) [CL02] where many component models can be found, e.g. [SVB+08, vOvdLKM00, WRM+05a, PMPL08, dNBR06, RPV+06, WRM+05b]. We focus on the general characteristics of these component models and we present the discussion of component frameworks specific to RTSJ in the next section.

AADL

AADL [FLV06] is a textual and graphical language used to design and analyze both software and hardware architectures of real-time systems and their performance-critical characteristics. AADL permits engineers to represent embedded systems as component-based system architecture and model component interactions as flows, service calls, and shared access. Furthermore, it allows engineers to model task execution and communication with precise timing semantics.

AADL is part of a model-based engineering enterprise solution and, as shown in Fig. 3.1, AADL model development can be used in parallel with the software system's development. Then, the analysis views generated through AADL modeling can be compared to testing results during system implementation.

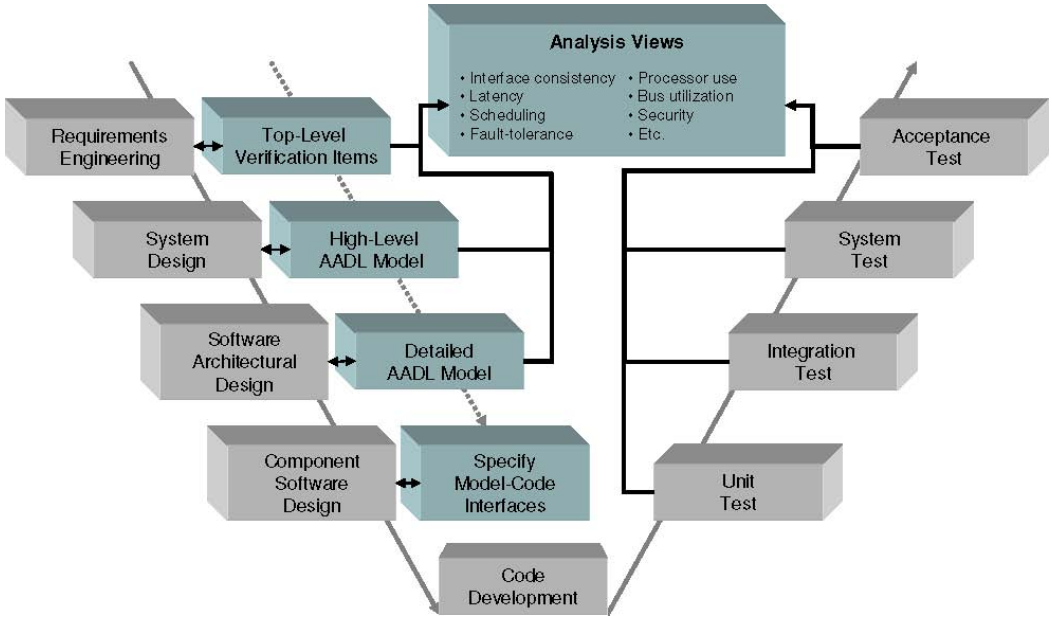


Figure 3.1: AADL Model Development

SOFA HI

SOFA HI [MWP+08] is a SOFA profile for high-integrity embedded systems based on the SOFA component model. The project was initialized by SciSys UK - an ESA [2] contractor. Currently, there is an effort to extend the SOFA HI towards the challenges of real-time systems [KPV+09].

This profile originates from the DiSCo project [PFP08], which addressed space missions where key challenges are hard real-time constraints for applications running in embedded environments, partitioning between applications having different levels of criticality, and distributed computing. The DiSCo Space-Oriented Middleware introduced a component model where each component provides a wide set of *component controllers* - a feature inspired by the FRACTAL component model.

SOFA HI provides a state-of-the-art hierarchical component model supporting expensively component containers and connectors. Furthermore, the model is supporting also more advanced technologies e.g. dynamic reconfiguration and formal verification. It also provides a development methodology providing an "activity" view that allows reasoning about activities (composed of tasks and mapped to chains of component operations), synchronization, execution times, deadlines, etc.

Ada-CCM

Ada-CCM [PMPL08] proposes a technology for the development of distributed real-time component-based applications, which takes advantage of the features that Ada language [TDB+07] offers for the development of applications with predictable temporal behavior.

The development process proposed by Ada-CCM is illustrated in Fig. 3.2. As we can see, it starts with description of real-time and functional requirements, the MAST modeler [HGGM01] is used to construct the real-time situation model, and generative programming techniques are extensively employed to facilitate the development. Furthermore, the approach adopts the component container technology in order to manage real-time properties of the component (e.g. scheduling parameters configuration).

[2] European Space Agency, www.esa.int/

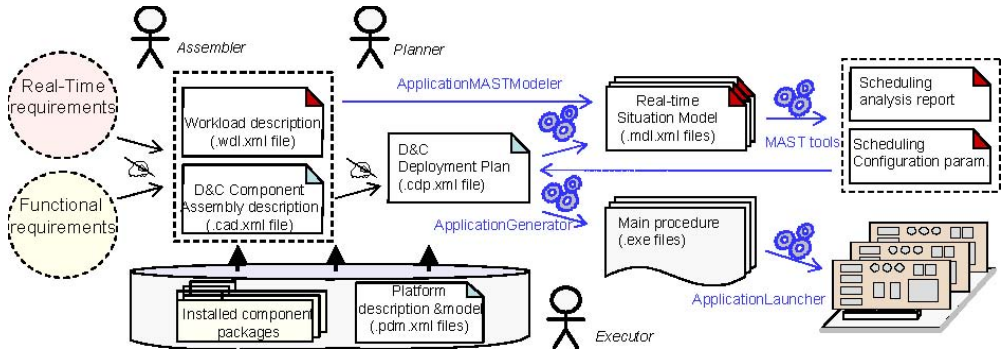


Figure 3.2: Component-based Application Development process in AdaCCM

Rohlik et al. –Reusable and Verifiable Software Components

Rohlik et. al. [RPV+06] propose a component framework for real-time applications. The notable feature of the approach is the separation of the treatment of functional and domain-specific (e.g. timing) requirements. These requirements are defined and modeled separately from each other and are only merged when the models are translated into code. The methodology proposed in the framework provides two views on the architecture: the functional view defining the framework from a functional point of view and the timing view defining hard real-time characteristics of the system. An approach to automatic instantiation of the runtime platform for this component model is described in [CCPS03].

OSGi-based framework for Real-time Systems

A recent popularity of the OSGi model [All09] is reflected also in the domain of real-time systems. To name at least on project, a hybrid real-time component model is proposed in [GDFSB08]. The authors propose a framework where real-time and non-real-time task coexist in symbiosis, whereas the non-real-time system is managed by the OSGi framework, the real-time part is implemented using RTAI [MDP].

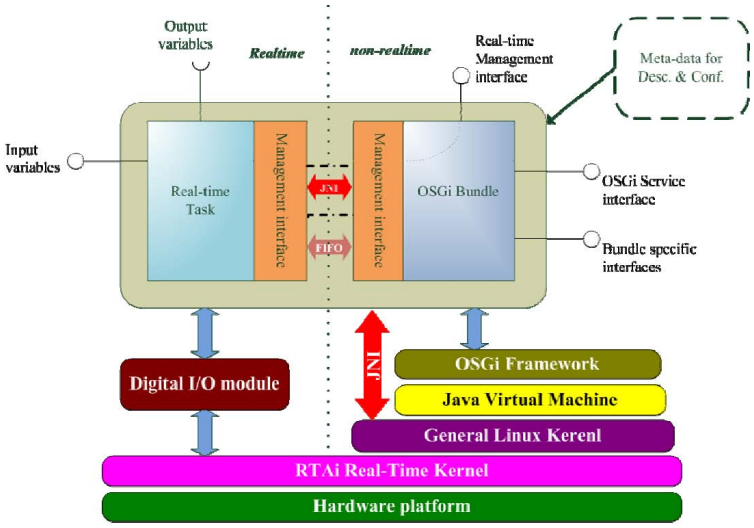


Figure 3.3: A Container Architecture Running the Hybrid Real-Time Component Model

We present the architecture of the solution in Fig. 3.3, the interesting aspect of this approach is support for cooperation between the real-time and non-realtime parts to the system. The responsibility of the non-realtime tasks is to provide component adaptation and management functions for the real-time tasks.

3.2.3 Component Frameworks for RTSJ

Already Dvorak et. al. [DR04] have argued for a component framework focused on Real-time Java. The authors distinguish two kinds of choices that have to be made during an application developed – *incidental* and *essential* choices. They claim that rather than solving *incidental* choices – e.g. language specific issues, the development process must focus on *essential* choices – meaning choices that reveal underlying requirements of the system - e.g. functional and real-time properties. Furthermore, a successful framework should allow developers to specify the essentials and let the tools automatically generate the platform-specific incidentals.

Moreover, with increasing complexity of RT Java systems, there is a pressure to incorporate CBSE as part of the strategy for reducing the total costs of developing and maintaining these systems by systematically enabling software reuse [Nil07].

Therefore, in this section we discuss component frameworks specifically created to address the challenges of RTSJ. We use the following characteristics of the frameworks as the evaluation criterions:

- *CBSE criterions*
  - *Component Model* The maturity of the component model is important to allow developers to achieve separation of concerns.
  - *Communication Model* The communication model must be rich enough to allow various interactions between components. This is important since RTSJ enforces both synchronous and asynchronous types of communication.
  - *Development Methodology* The framework must propose a development methodology guiding users through the development process in order to mitigate the complexities of RTSJ.
  - *Adaptation Support* Overall support for static and dynamic adaptation of developed applications.
- *RTSJ criterions*
  - *Memory Model* We evaluate framework's support for RTSJ memory model. We evaluate weather a sufficient support for handling cross-scope communication is provided. This concerns not only communication patterns and idioms, but a higher level approach allowing to face these obstacles already at design time.
  - *Thread Model* Supporting the notion of different types of schedulable entities is important in order to design communication between them, we therefore evaluate this aspect.
  - *Formalization and Validation Support* Finally, a formal approach embracing the framework should be proposed in order to support validation of developed applications.

Compadres Framework

Compadres [HGCK07, J. 06] proposes a component framework for distributed real-time embedded systems. The component model is hierarchical and supports solely event-oriented interactions between components. The complexities of the RTSJ threading model are not reflected by the framework. On the other hand, Compadres pays more attention to the RTSJ memory management. It defines that each component is allocated and executing either in a scoped or immortal memory. This restriction allows the framework to clearly address the challenge of cross-scope

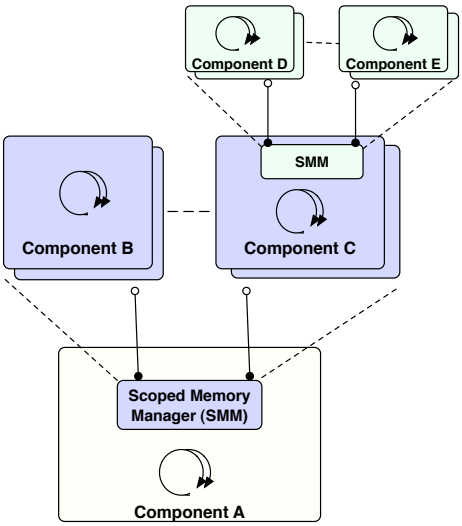


Figure 3.4: Compadress Memory model : Parent components communicate with their child components via scoped memory managers (SMMs)

communication. Compadres uses a set of communication patterns defined in [PFHV04] and further proposes *scoped memory managers* to provide communication between sibling components, as illustrated in Fig. 3.4.

However, in this approach, components can be allocated only in scoped or immortal memories, therefore communication with regular non-real-time parts of applications can not be expressed. And since the coexistence of real-time and non-real-time elements of an application is often considered as one of the biggest advantages of RTSJ, we believe that it should be addressed also by its component model and therefore we consider this memory model as too restrictive.

Compadres also proposes some notion of development methodology of real-time application development in order to separate development of functional and RTSJ concerns. However, a solution introducing systematically the real-time concerns into the functional architecture is not proposed, thus the complexities of designing real-time systems are not mitigated fully.

Etienne et al. – A Component Framework for RTSJ

Work introduced in [ECB06] defines a hierarchical component model for Real-Time Java. The classical concept of component model is here extended by introducing *active* and *passive* components. Whereas a passive component represents only e.g. libraries or shared entities, an active component is having its own thread of control and represents a real-time task. Additionally, properties describing periodicity, deadline and priority can be specified in order to express non-functional parameters of the active component. We consider this approach as highly appropriate for RTSJ systems since it allows developers to deal with the complexities of RTSJ thread model.

The model is also allowing to hierarchically design components into compositions, thus achieving a much coarse-grained architecture. Moreover, the developer can specify component *contracts*, which provide information about the structural, behavioral and temporal properties of the component. These contracts are verified during the assembly phase in order to check conformance of components employed in the system. The proposed component model is illustrated in Fig. 3.5.

The model employs a simple memory management model. Each component is allocated within a distinct scoped memory. This allocation policy gives advantages of controlling the lifetime of each component individually and an efficient mean of managing the memory footprint

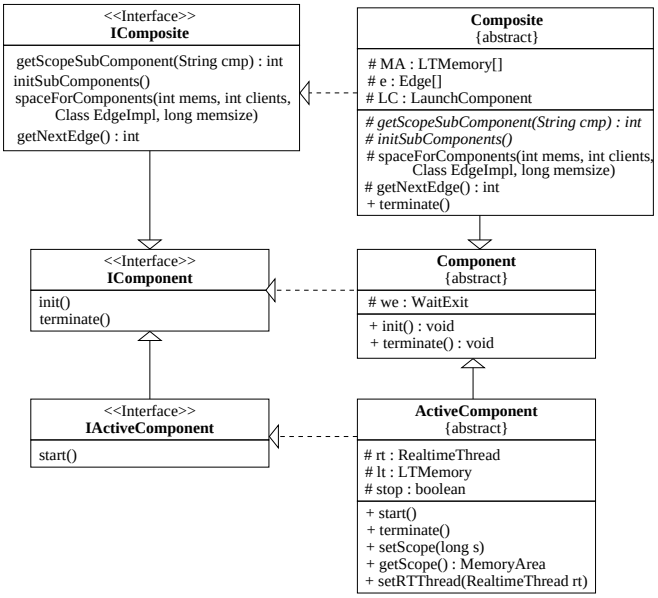


Figure 3.5: Etienne et. al. Component Model

of the application. The high execution cost of the scoped memory can be expected when the configuration of components spans over several hierarchy levels. Here, each service call performed by a client component requires one or more memory traversals, thus decreasing the performance of the system. To cope with this, subcomponents of a component can be allocated in a collective scoped memory and thus preventing from hierarchical traversals of memory. However, such a solution is limiting the flexibility of the system and can cause a memory leakage within the scope (old instances of subcomponents are remaining in the scope for the lifetime composite). The bottom line is that the developer has to find an exact balance between flexibility and performance of the system.

However, in the model, the real-time memory management concerns can not be expressed independently of the functional architecture, systems are thus developed already with real-time concerns. This not only puts additional burdens on designers but also hinders later reuse and modification. When designing the scoped memory structure, the model allows to assign one scope memory to more components, however this fact can not be expressed at the architectural level of applications. Moreover, only the deep-copy pattern is proposed to solve cross-scope communication. To summarize, we still see some space for improvement when considering the development of RTSJ applications in this framework.

Golden Gate Project

The project Golden Gate [DBC+04, BCC+03] evaluates suitability of RTSJ to be used in development of control software for onboard system. One of the milestones of the project was to design control loops for driving and steering a 6-wheel experimental Mars rover.

Driven by the specific requirements, the project introduced real-time components that encapsulate the functional code to support the RTSJ memory management. The main focus was laid on the memory management aspects of RTSJ, the usage of real-time threads together with their limitations is not addressed. However, the considerable results were achieved in benchmarking RTSJ [DR04], the authors conclude, as we have already mentioned, rather than facing language specific issues - the *incidentals*, the development process must focus on *essential* choices. The essentials are for example: data structures for inputs and outputs, pure functions that perform state

transformations, and required properties of sequencing, timing, concurrency, etc. These concepts relate directly to the problem domain and are neutral with respect to language, software architecture, and hardware architecture. Consequently, they leave options open late in the development and testing cycle, rather than making early (and sometimes regrettable) commitments that can only be changed at great cost.

As one of the outcomes of the project, we therefore consider a specification of a successful framework for RTSJ – such a framework should allow developers to specify the essentials and let the tools automatically generate the platform-specific incidentals that will satisfy the requirements.

Synthesis

We summarize and compare the characteristics of presented frameworks in the Table 3.1. Except from the frameworks presented above, the table also includes the SOFA HI component model, this servers as a comparisons in order to evaluate maturity of the presented frameworks.

Considering the CBSE criterions, SOFA HI offers the most of the features available in the domain of CBSE, from component containers supporting separation of concerns, to software connectors that provide a wide set of communication types. Development methodology and system reconfiguration are also addressed by this component model. Comparing with the component frameworks for RTSJ, we witness that these component models are still immature.

Considering the RTSJ criterions, the complexities of the RTSJ thread model are usually addressed by introducing active and passive components. To mitigate the problems of memory management, usually a restricted set of patterns [CS04, BN03, PFHV04] is defined. However, these technologies are not applied consistently and various frameworks provide support for them only to some limited extent. Furthermore, no high-level abstractions are employed to allow developers to face these obstacles before actually starting the implementation process. Finally, none of the presented frameworks stands on a formally defined ground, thus failing to achieve a demanded support for validation of developed applications.

	Compadres	Etienne et al.	SOFA HI	Golden Gate
CBSE Criterions				
Component Model				
Communication Model				
Development Methodology				
Adaptation				
RTSJ Criterions				
Thread Model Support				
Memory Model Support				
Formalization and Validation				

Legend

Low Support — *High Support*

Table 3.1: Recapitulation and Comparison of Component Frameworks for RTSJ

3.2.4 Distributed and Embedded Computing in Real-time Java Systems

The area of distributed programming in the scope of real-time Java includes several research directions. The leading initiative is represented by an integration of Remote Method Invocation (RMI) into the RTSJ [WCJW02] and solving the task related issues such as handling real-time properties [BW03, WCJW02] or memory allocation [BVGVEA05, BW03]. The results of these projects are reflected in a status report of *JSR 50* [AJ06] which tries to cover all aspects of distribution (real-time properties handing, failure semantics, distributed threads and their scheduling). A similar approach proposes a profile for distributed hard real-time programming [TAdM07]. However, a framework addressing comprehensively the challenge of developing such a complex system still has not been proposed.

Another research area covers the Real-time CORBA specification [OMG] which can serve as a particular base for a requirements analysis of real-time distributed systems. Its main implementor in the RTSJ world is RTZen [RZP+05]. Although it is a middleware implementing almost all parts of the Real-time CORBA specification within the scope of RTSJ, it only focuses on a core of communication and does not provide any abstraction of RTSJ.

To summarize, all these projects focus only on low-level communication issues and their integration into the scope of RTSJ. They do not address any higher abstraction of the real-time communication. It could however be beneficial to reflect distribution in different stages of the application lifecycle (design, implementation, runtime).

3.3 FRACTAL Component Model

The FRACTAL component model [BCL+06] is a light weight component model, focused on programming language concepts. In contrast to other component models, such as EJB, .Net or CCM, it does not require the extra-machinery supporting its functionality. The model is built as a high level model and stresses on modularity and extensibility. Moreover it allows the definition, configuration, dynamic reconfiguration, and clear separation of functional and non-functional concerns.

The FRACTAL model is somewhat inspired from biological cells, where exchanges between the content of a cell and the environment are controlled by the membrane. By analogy, a FRACTAL component is a runtime entity, which offers server and client functional interfaces, as well as non-functional interfaces implemented as controller objects in the membrane. All interactions with the component are interactions with the membrane of the component, and this allows interception and intercession using interception objects positioned within the membrane. Moreover, all non-functional aspects are dealt within the membrane of a component, thus enforcing separation of concerns between functional and non-functional features.

The FRACTAL model is an open component model, and in that sense it allows for arbitrary classes of controllers and interceptor objects, including user-defined ones. FRACTAL is meant to be extensible; in this sense it leaves unspecified how communication takes place between components, how components are specified, and what its implementation is; even bindings can be components. Non-functional features of the component can also be customized.

In Fig. 3.6 we present an illustration example of a FRACTAL application architecture. Here, all the key concepts of the FRACTAL component model are presented, we further clarify them.

- **Composite Component** is a component composed of subcomponents
- **Primitive Component** is a component that does not have any subcomponents and is directly implemented in some programming language.
- **Shared Component** is a component that has more than one super-components.
- **Content** is one of the two parts of a component, the other one being its membrane. The content is an abstract entity controlled by a membrane. The content of a component is (recursively) made of sub components and bindings.

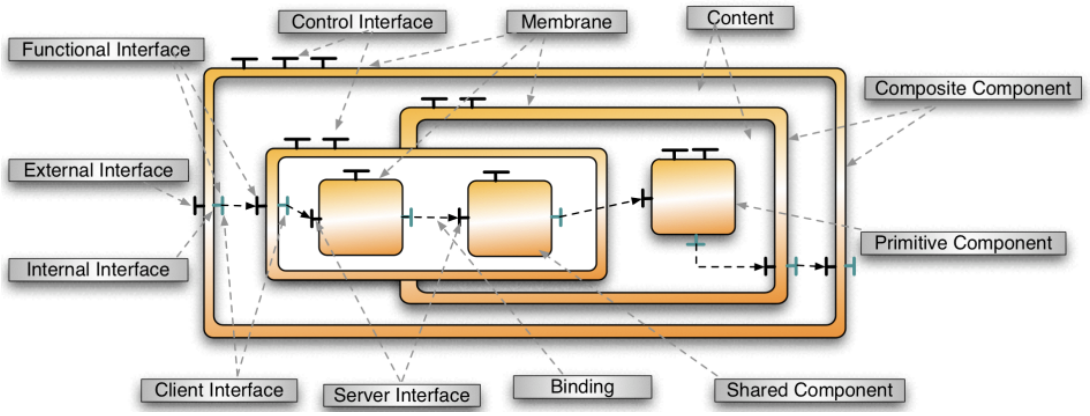


Figure 3.6: FRACTAL Concepts

- **Membrane** is one of the two parts of a component, the other one being its content. The membrane is an abstract entity that embodies the control behavior associated with a particular component. The membrane is composed by controllers.
- **Controller** exercises an arbitrary control over the content of the component it is part of (intercept incoming and outgoing operation invocations for instance).
- **Interface** is defined by a *name*, a *role* (client or server), a *cardinality* (singleton or collection), a contingency (mandatory or optional) and a signature (in Java, the fully qualified name of a Java interface). In the type system proposed by FRACTAL, the set of functional interfaces defines the type of a component. This can be further extended to take into account the non-functional interfaces as well. Moreover, there are external and internal interfaces. How these two relate is undefined in FRACTAL. This leaves freedom to define interceptors.
  - **Server Interface** is a component interface that receives invocations
  - **Client Interface** is a component interface that emits invocations
  - **Functional Interface** corresponds to a provided or required functionality of a component, as opposed to a control interface. In Fig. 3.6 they are depicted horizontally, directed towards left and right for server and client interfaces respectively.
  - **Control Interface:** is a component interface that manages non-functional properties of a component, such as introspection, configuration or reconfiguration, and so on. These are also called control interfaces. In Fig. 3.6 they are depicted vertically, directed towards top and down for server and client interfaces respectively.
- **Component Binding** represents a communication path between different interfaces. It can be realized by software connectors.

Component-based Control Membranes

The abilities of the FRACTAL component model are even more extended by a new feature introducing the component-based architecture for the control environment surrounding components [SPDC06, PMS07]. Similar to EJB's containers, the FRACTAL component model features a controlling environment, called *membrane*. This supports before mentioned domain-specific properties of components. However, in contrast with fixed structures of EJB containers, the control membrane of a component is implemented as an assembly of so-called control components and can dynamically evolve. The whole idea is depicted in Fig. 3.7.

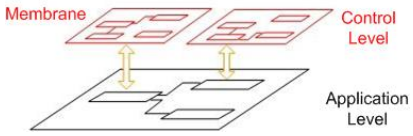


Figure 3.7: Component-based Control Membranes

Not only does this approach brings effective development in the sense of reusability and transparentness, but the main benefits lay in the ability to introspect and dynamically reconfigure the architecture of the control layers of each component. Moreover, the membranes can be designed individually thus precisely fitting the needs of specific components. This leads to a reflective component model, where both the functional layer and the control layer are implemented using components.

FRACTAL Implementations and Tool Support

There exist many implementations of the FRACTAL component model in various implementation languages, available at [OW209b]. Furthermore, apart from the research inspired by the FRACTAL initiative, many related projects have emerged around FRACTAL in order to leverage development of FRACTAL based systems. Between the most significant we list FRACTAL-ADL [LOQS07] – ADL specification, FRACLET [RM09] – annotation framework, F4E [Dav08] – an Eclipse plug-in, and FDF [FDDM08] – a deployment framework.

3.3.1 FAC: FRACTAL Aspect Model

The goal of the FRACTAL Aspect Model project (FAC) [NLLT08] is to allow developers to leverage the technique of aspect-oriented programming to the application design layer. The motivation is to provide an approach to manipulate the aspects as first-class entities at design time and at runtime.

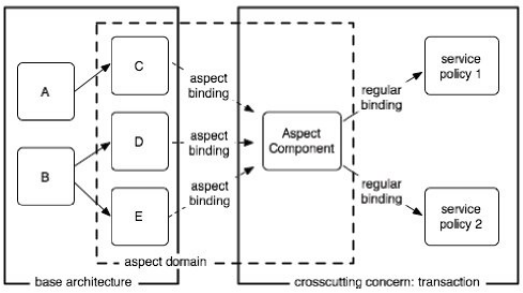


Figure 3.8: FAC: Functional and Aspect Components Example

To achieve this, a new type of a component is proposed - *aspect component*, which propose a general and symmetrical model for components and aspects. FAC decomposes a software system into regular components and aspect components which embody crosscutting concerns.

In Fig. 3.8 we illustrate the key idea of the solution. In the picture, the AspectComponent represents a certain aspect applied to components C,D and E. This relation is represented by the aspect binding connecting the standard components with the aspect component. Furthermore, the *aspect domain* is the reification of the components picked out by an aspect component. The goal of an aspect domain is to keep an overview of all the components affected by an aspect.

We highlight this project since it leverages domain-specific services, represented and implemented by AOP, to the application design layer. This allows developers to achieve full separation of functional and non-functional concerns along the whole development lifecycle.

3.3.2 Formalization of the FRACTAL Component Model

Recently, one of the research trends in CBSE is focused on formalization of component models. The key motivation is to formalize different component models and their semantics in order to evaluate and compare them. One of the popular languages used for this task is Alloy [Jac06]. In [KM08] authors evaluate application of Alloy in defining semantics of new modeling languages and state that such an approach yields a formally defined semantics of a language while considerably reducing invested effort. A formalization of a component model in Alloy was already conducted in [JS00] resulting in simplification and improved clarify of the model itself.

A formal specification of the FRACTAL component in the Alloy specification language is provided [MS08]. The specification develops a view of a FRACTAL component as a coalgebra, or, equivalently, as a form of generalized state machine. Although elementary, the specification identifies and removes certain ambiguities in the informal FRACTAL specification, generalizes it in places, and improves the programming-language-independent character of the component model specification.

Alloy Modeling Language

Alloy [Jac06] is a formal language to express software abstractions precisely and succinctly. A system is modeled in Alloy using a set of types called signatures. Each signature may have a number of fields. Constraints may be added as facts to a system to express additional properties. In terms of rigor Alloy rivals traditional formal methods. It represents an approach for the definition of the abstract syntax, the static semantics and the dynamic semantics of a system - a component model in our case. Alloy Analyzer [BB08] allows a fully automatic analysis of a specified system; it can expose flaws early and thus encourages incremental design.

```
1 sig Component {
2   interfaces: set Interface
3 }
4
5 sig Primitive extends Component{
6   content : one Class
7 }
8
9 sig Composite extend Component {
10  subComponent : set Component
11 }
```

```
1 sig Interface{
2   name : one String
3   boundTo: one Binding
4   owner : one Component
5 }
6
7 fact WellFormedContainment {
8   no c : Component | c in c.^subComponents
9 }
10
11 fact InterfaceHasOwner {
12   all i: Interface |
13     one c: Component |
14       i in c.interfaces && c in i.owner
15 }
```

Figure 3.9: Alloy: Basic Syntax and Semantics

We illustrate the application of Alloy by defining a simple component. The resulting Alloy model, given in Fig. 3.9, expresses both abstract syntax and static semantics of the metamodel. Each of the entities in the metamodel has a corresponding signature (denoted as `sig`) with a similar name in the Alloy model. Associations in the metamodel are usually represented by fields of a signature: thus the field `interface` in `Component` corresponds to the aggregation from the `Component` to the `Interface` in the metamodel. For each field, multiplicity markings can be present, specifying constraints on the field. The markings that we use are `one`, `lone` standing for "0 or 1", and `set`.

To express constraints over the model users can use facts - denoted as `fact`, which express the rules in the first order logic. For example, the `WellFormedContainment` – line 7, states that

there is no component that is a subcomponent of itself - expressed by "`c in c.^ subComponents`" where `^` is a transitive closure. The `InterfaceHasOwner` fact - line 11, states that for every interface exists one component that contains it.

Note that we have chosen a single Alloy model to express both the abstract syntax and static semantics of the metamodel since both structural properties and well-formedness rules are expressed by constraints in the Alloy model and there is no natural separation between the two. We also note that the Alloy Analyzer allows the graphical presentation of the meta-model specified by an Alloy model.

3.4 State-of-the-Art Synthesis

Our research lies within the intersection of three research domains: real-time system development, component-based software engineering (CBSE), and real-time Java programming (RTSJ), which were extensively described in Chapter 2 and Chapter 3. We employ these three technologies as cornerstones of the solution that we propose in this dissertation. In this section we therefore argue for synthesizing these domains by showing how the outcoming technology addresses the stated challenges.

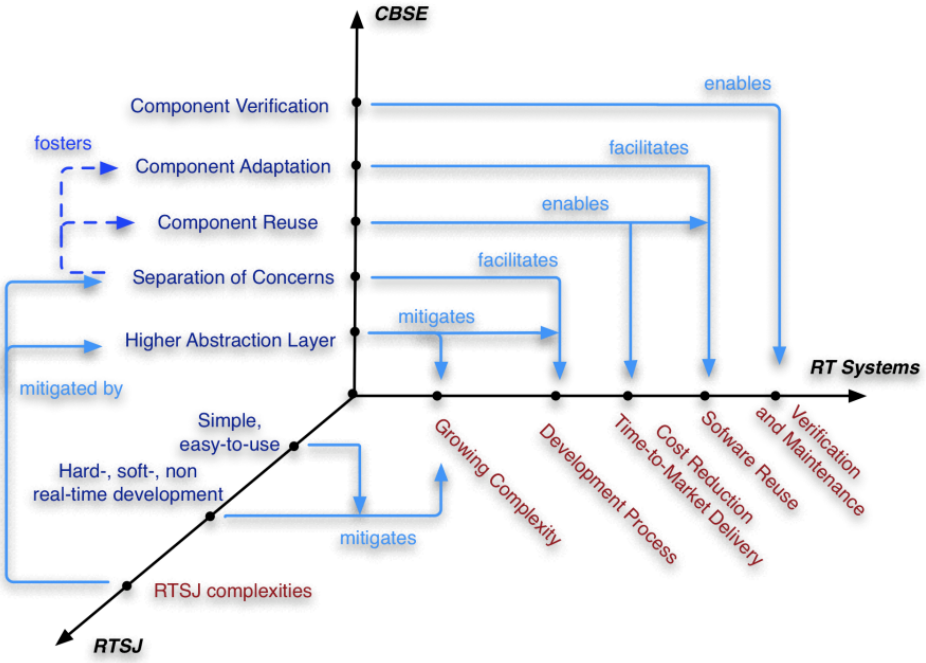


Figure 3.10: Synthesis of the Technologies Applied in the Dissertation

In order to synthesize the stated facts and propose an appropriate solution strategy, we illustrate these three technological domains in a diagram depicted in Fig 3.10. Each domain is represented by an axis - *RT Systems*, *CBSE*, and *RTSJ*. Along these axes, the essential characteristics of each technology are given. Furthermore, relations between the characteristics from different axes are presented in order to highlight how the aspects across these domains do supplement or influence each other.

For the domain of real-time systems – *RT Systems*, the predominant aspect is their growing complexity which puts further demands on development process. These issues lead to increased market demands, requiring better software reuse and employment of the state-of-the-art approaches for system verification and maintenance.

The second axis represents the real-time Java programming – *RTSJ*, as the key principle of RTSJ we highlight its simplicity, cheap adoption and easy accessibility. Furthermore, the most appreciated feature of RTSJ follows - the ability to implement applications embracing hard-, soft- and non real-time requirements. Finally, the benefits of RTSJ are counterbalanced by the complexities of the development process, specially when considering its memory model. We highlight it as the third characteristic in this axis.

The final axis represents the domain of component-oriented engineering - *CBSE*. As the key feature we consider the ability of CBSE to leverage the development process in the higher abstraction layers followed by the separation of concerns. These characteristics contribute to wide support of CBSE for component reuse and foster adaptation of software systems. Finally, we also mention ability of CBSE to support software verification.

The relations across different axes represent our fundamental applications in these domains towards our goal – a component framework for real-time Java systems. Starting with RTSJ axis, real-time Java mitigates complexities of RT systems, but, however, brings the challenge of memory management and other issues. By combining the features of the *CBSE axis* we mitigate the RTSJ complexities through the separation of concerns and introduction of high-level abstractions for RTSJ semantics. Continuing with addressing of the challenges in the *RT System axis*, we employ CBSE methods. Software reuse and development process are inherently addressed since CBSE fosters component reuse and adaptation through the separation of concerns. Similarly for the challenge of verification and maintenance.

3.5 Goals Revisited

In this section we refine more precisely the general goals of the dissertation based on the facts identified in the state-of-the-art survey.

Scope of the Dissertation

The process of developing a real-time system brings many challenges, specially constructing a real-time model that describes timing behavior of the system is a difficult task. Here, timing and schedulability analysis must be performed in order to create a model that reflects the real-time requirements of the system. The scope of our proposal is placed directly afterwards this stage, the timing behavior and tasks in the system are exactly specified (as seen in the illustration real-time scenario in Section 2.1.2) but the development of the system lies at its very beginning. Particularly, our motivation is to combine the state-of-the-art software engineering methods with RTSJ while still producing predictable software applications. Therefore, in this dissertation we focus solely on the specifics of RTSJ.

Refining the Goals

Based on the discussion conducted in this Chapter, we refine the goals of this dissertation as follows[3]:

- **G1** *RTSJ-specific Component Model.* As the first goal, a component model designed towards the specifics of RTSJ must be proposed. The model must provide:
  - **G1.1** *RTSJ Abstractions.* To propose an appropriate abstractions of RTSJ semantics in order to allow developers to manipulate with the RTSJ-related features as first-class entities.
  - **G1.2** *Separation of Concerns.* To clearly separate functional and RTSJ-specific concerns along the whole application lifecycle. To achieve this, the component model should support the component containers technology.

[3]We have label the goals and subgoals as **G1-G3** and in the remainder of this dissertation we will refer to them using these labels.

  - **G1.3** *Formalization and Validation Support.* To fully formalize the component model and its extensions towards RTSJ in order to support validation of applications during their development lifecycle. Provide an approach to validation of both design and implementation of applications in order to guarantee coherence with RTSJ.

- **G2** *RTSJ-specific Framework.* Based on the component model proposed as the goal **G1**, the dissertation must propose a full-fledged framework for development of RTSJ applications that will address following points:

  - **G2.1** *Development Methodology.* To employ the proposed features of the component model in order to improve and clarify development process of RTSJ applications.
  - **G2.2** *Runtime Platform Instantiation.* To provide transparent implementation of systems with comprehensive separation of concerns and extensive support of non-functional properties at design, implementation, and run-time. Furthermore, we must extensively employ methods of generative programming to automatically instantiate runtime platforms. Thus we will mitigate the complexities of developing the error-prone RTSJ code by hand.

- **G3** *Evaluation.* The final goal is to evaluate the framework from both performance and software engineer perspectives. In terms of quantitative evaluation, the framework must:

  - **G3.1** *Predictability.* Not introduce unpredictability.
  - **G3.2** *Performance.* Not add significantly more overhead than that associated to the equivalent application developed by other means (e.g. hand-coding from scratch).

  Since the potential of the system to avoid tedious and error-prone development is one of the key features when discussing its effectiveness and useability, in terms of qualitative evaluation the framework must:

  - **G3.3** *Effectivity.* Facilitate development of RTSJ applications as if using standard Java as much as possible.
  - **G3.4** *Simplicity.* Be easy to understand. The framework must maintain compliance with original RTSJ specification and do not introduce any RTSJ extensions that are not adopted by the RTSJ standard.

3.6 Summary

In this chapter we introduced CBSE and the technologies employed nowadays to leverage the development process of component-based systems. Furthermore, we have discussed its general benefits and argued for applying it in the field of RTSJ, by showing RTSJ-specific component frameworks and comparing their advantages.

More specifically, already in the previous chapter we have noticed movement from research of RTSJ compliant implementation patterns [A. 05, BN03, PFHV04] to research of frameworks alleviating the RTSJ complexities [DBC+04, ECB06, HGCK07]. Therefore, in this chapter we present several component-based frameworks addressing the challenges of RTSJ programming and evaluate them focusing on following aspects – richness of their component model, support for thread and memory models defined by RTSJ, the development methodology they provide for RTSJ, and their potential for automatic validation. As a result, we have identified that these frameworks address the RTSJ issues only partially, usually mitigating small and isolated problems without any unified strategy. Furthermore, these frameworks do not alleviate the RTSJ concepts into higher level of system development which prevents full mitigation of the accidental complexities caused by RTSJ issues.

As demonstrated, there is much to be done in the field of component based frameworks for RTSJ. The current solutions are still rigid, and mainly do not allow to express both real-time

and non-real-time concerns at a higher abstraction level. To meet all the challenges, an adequate component model allowing to fully describe RTSJ concerns independently of the functional logic needs to be proposed. Furthermore, there is a still a long road from technical solutions proposed and a development methodology that could embrace all the benefits of these specific approaches. A process leading developers systematically through design and development of RTSJ-based system while mitigating RTSJ complexities is therefore highly desired. Motivated by these observations, we restate more precisely the goals of this dissertation, Section 3.5.

Finally, after carrying out a survey of current component models both in academia and industry, we have decided to adopt the existing FRACTAL component model as the technological background for this dissertation. Based on the introduction of FRACTAL in Section 3.3, the justification for this decision is as follows:

- FRACTAL provides a state-of-the-art, open, and extensible component model, offering most of the features available in the domain of component-oriented software architectures.
- We specially appreciate the fact that FRACTAL preserves the component architecture entities during the whole system life cycle, also at runtime, with minimal performance overhead.
- FRACTAL represents almost a decade of CBSE experience and evolved significantly towards a realistic design and runtime platform, being a result of tens of man-years of research and numerous publications. Furthermore, FRACTAL has been implemented in different languages for various platforms, is being used in industry and provides an extensive tool support. All of these characteristics prove maturity of this component model.

In the following chapter we present SOLEIL framework – the first contribution of our dissertation.

Part II

Proposal

Chapter 4

SOLEIL: A Component Framework for Java-based Real-Time Embedded Systems

Contents

IN this chapter we introduce our component-based framework for development of real-time Java-based systems called SOLEIL. As the key philosophy for this framework we adopt the thesis statement – *an effective development process of RTSJ-compliant systems needs to consider RTSJ concerns at early stages of their design*. Following this philosophy, our goal is to leverage the concepts of RTSJ and mitigate complexities related to their implementation. Furthermore, by proposing a development methodology we want to provide a continuum between the design and implementation process where RTSJ concerns are manipulated in a consistent and transparent manner.

The important aspect of our approach is the motivation to provide appropriate abstractions of the RTSJ concepts in order to allow their manipulation in the development lifecycle. By following

this approach we want to achieve an effective development process of RTSJ-compliant systems that considers RTSJ concerns already at the design time. On the other hand, while looking for the appropriate level of abstraction, the separation of functional and RTSJ-specific concerns must be preserved in order to keep the complexity of the programming model at a reasonable level.

Therefore, as the cornerstone of our approach, we introduce the concept of *domain component* that allows developers to express RTSJ concerns - e.g. real-time threads or memory areas, as special software components in the system. By abstracting RTSJ we thus permit to manipulate with its concerns as first-class entities along the whole development lifecycle. This approach ultimately provides a full separation of functional and RTSJ-specific concerns. Furthermore, we propose a component model that embraces the concept of domain component. The component metamodel itself however is not sufficient to model RTSJ applications without formally specified semantics. Defining such semantics is important for reasoning about modeled entities, their composition, inter communication, and last but not least for providing tool support. Therefore, to formalize the model we use the Alloy language [Jac06], described in Section 3.3.2.

In the terms of the refined goals, presented in Section 3.5, in this chapter we first address the goal **G1** – proposing a RTSJ specific component model that introduces appropriate RTSJ abstractions (**G1.2**), respects the separation of functional and real-time concerns (**G1.2**), and is fully formalized to facilitate validation of developed instances(**G1.3**). Furthermore, in this chapter we partially address also the goal **G2** by proposing a framework based on the component model. In this framework we define a development methodology that fully benefits from the features of the component model (**G2.1**). Finally, the framework also extensively employs generative programming in order to automatically instantiate runtime platforms that implement framework glue code and RTSJ-specific code (**G2.2**). However, in this chapter we describe the runtime platform instantiation process from a user perspective and we address the issues related to its implementation in Chapter 5.

Contributions

The contributions of this chapter are:

- **A RTSJ-specific Component Model and Domain Components.** We propose *Domain Components* — a unified approach to specification of domain-specific requirements presented in custom containers. This allows application developers to easily manipulate domain specific requirements since they are represented as first-class entities and are separated from the functional concerns. Furthermore, we employ this concept to construct a RTSJ-specific component model allowing developers to manipulate with RTSJ concerns.
- **SOLEIL Profile and Verification** All the introduced concepts are formalized, using the Alloy language, in order to clarify exactly their semantics. Further, we define the SOLEIL profile - a set of rules and restrictions that must be respected by application developers to achieve RTSJ conformance.
- **SOLEIL Framework** Based on the component model and its formalization we construct a framework clarifying development methodology of RTSJ-based systems. The framework provides a continuum between the design and implementation process. The goal is to mitigate complexities of RTSJ-development by automatically generating runtime platforms where real-time concerns are transparently managed.

Structure of the Chapter

The remainder of this chapter is organized as follows. In Section 4.1 we define a core metamodel which serves as the cornerstone for further extensions towards RTSJ. In this core metamodel we introduce the key concept of our proposal — *Domain Component*. Section 4.2 we extend the metamodel towards the specifics of RTSJ, therefore, we introduce new domain components representing the RTSJ concepts and we demonstrate how to manipulate them. In Section 4.3 we

present SOLEIL framework. Here, a design process incorporating the model and its formalization are introduced - in Section 4.3.1. As an outcome of this process we obtain a real-time system architecture that can be used for implementation of the system. We benefit from separation of functional and non-functional concerns and design an implementation process that addresses these concepts separately - whereas functional concerns are developed manually by users, the code managing non-functional concerns is generated automatically. We elaborate on this implementation methodology in Section 4.3.2. In Section 4.4 we revisit our motivation scenario to illustrate applications of the concepts proposed in this chapter. Finally, a summary of this chapter is given in Section 6.6.

Throughout the chapter we present code snippets illustrating the formalization process in Alloy, the whole code is available in Appendix A.

4.1 A Generic Component Model

The cornerstone of our framework represents a component model that is specially designed in order to allow us to fully separate functional and non-functional concerns in all steps of system development. We define the core of the model in Section 4.1.1 – inspired by the FRACTAL component model [BCL+06] and enriched by the concepts of `Domain Component` and `Functional Component`. Consequently, we introduce these concepts in more details in Section 4.1.2 and Section 4.1.3.

4.1.1 Core Concepts

We present our component metamodel in Fig. 4.1. The metamodel is also formally defined in Fig. 4.2 in Alloy. Therefore, the UML diagram serves here as a graphical illustration since its semantics does not allow to fully express all the relations between the entities of the metamodel.

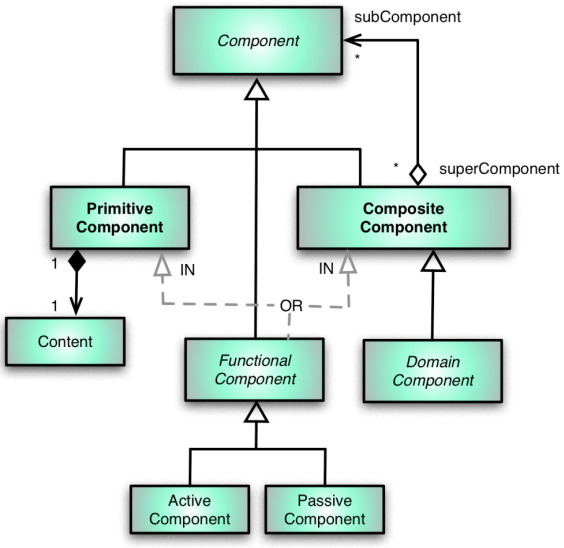


Figure 4.1: A Generic Component Model

The key element of the metamodel is the abstract entity `Component`, in Fig. 4.2 line 1, which is further extended by `Primitive Component` and `Composite Component` entities, lines 31-36. We thus introduce the notion of hierarchy into the model. Whereas the *primitive* component implements directly some functionality, expressed by the `Content` entity – line 36, the composite

component encapsulates a set of subcomponents, line 31. Furthermore, a notion of *supercomponent* enhances the metamodel with the concept of *sharing* - one component can be contained in more than one supercomponent, line 3. Finally, we define that each component in the system is either composite or primitive, line 40.

The other two key entities of the metamodel are `Functional Component` and `Domain Component`. The are introduced to enforce separation of functional and domain-specific concerns. We describe them in more details in Section 4.1.2 and Section 4.1.3.

```
1  abstract sig Component{
2    externalInterfaces : set Interface,
3    superComponents
   : set Composite
4  }
5
6  sig Interface {
7    owner     :     one Component,
8    boundTo   :     lone Interface,
9    type      :     one Type,
10   binding   :     one Binding
11 }{
12   boundTo != this
13 }
14
15 sig Attribute{}
16
17 sig Binding {
18   client            : one Interface,
19   server            : one Interface,
20   communicationType : one CommunicationType,
21   attributes        : set Attribute
22 }
23
24 sig CommunicationType, Type {}
25
26 one sig Asynchronous, Synchronous
27         extends CommunicationType {}
28
29 one sig Client, Server extends Type {}
30
31 sig Composite in Component {
32   internalInterfaces  : set Interface,
33   subComponents       : set Component,
34 }
35
36 sig Primitive in Component {
37   content : lone PrimitiveObject
38 }
39
40 fact ComponentIsCompositeOrPrimitive {
41   all c:Component |
42         (c in Composite or c in Primitive)
43     and (
44        c in Composite implies
45          c not in Primitive )
46     and (
47       c in Primitive implies
48          c not in Composite )
49 }
```

```
50 sig FunctionalComponent extends Component{}
51
52 abstract sig DomainComponent
53                 extends Composite {} {
54   no internalInterface
55   no externalInterface
56 }
57
58 fact NoDomainAsSubcomponentOfFunctional {
59    no d:DomainComponent |
60      some c:FunctionalComponent |
61         d in c.subComponents
62 }
63
64 sig Active extends FunctionalComponent{
65   periodicity : int
66 }
67
68 sig Passive extends FunctionalComponent {     }
69   isProtected : one Boolean
   }
70 }
71
72 sig Periodic, Sporadic extends Active {}
73
74
75 fact periodicANDsporadic {
76   all a:Active |
77     a in Periodic or a in Sporadic
78   if a in Periodic
79     #a.getServerInterfaces == 0
80   if a in Sporadic
81     all i:Interface |
82       if i in a.getServerInterfaces {
83          i.interfaceType = ASYNCHR
84       }
85 }
86
87 pred getServerInterfaces[a: Component]
88                           : set Interface {
89   all i: Interface | {
90      i in a.externalInterfaces
91      and
92      i.type == Server
93   }
94 }
```

Figure 4.2: Generic Component Model Formalization in Alloy

4.1.2 Functional Components

Functional components are basic building units of our model, representing functional concerns in the system. The motivation for the introduction of functional components is to exactly separate functional and domain-specific concerns of the applications in all steps of the software development lifecycle. In Fig. 4.2 we define `Functional Component` as an extension of `Component`, but we also specify that each functional component is either primitive or composite, line 40. This

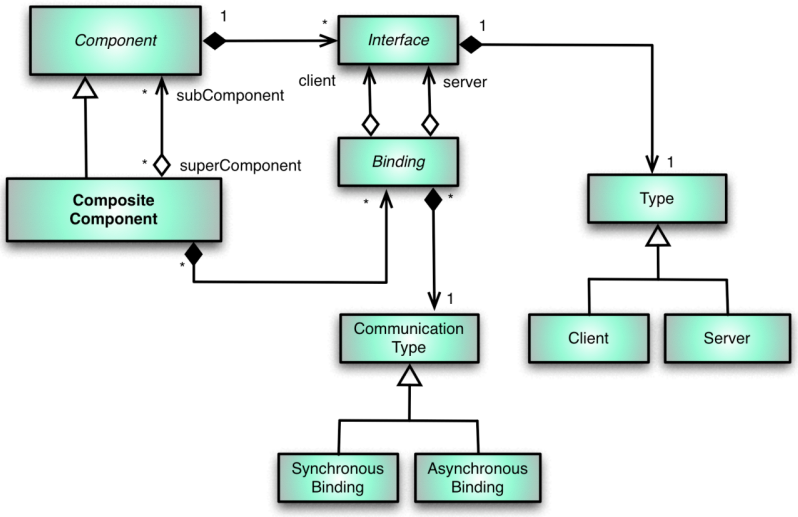


Figure 4.3: `Interface` and `Binding` Concepts of the Metamodel

can not be expressed in the UML language and we therefore only indicate this in the Fig. 4.1 by the slashed arrows. We distinguish two types of functional components - `Active` and `Passive` components.

Active Component

Active components contain their own threads of execution and properties regarding its periodicity, deadline, and priority. We can thus characterize the active component types by setting their attributes, to define e.g. a periodic/sporadic execution or event-based execution. By this way we will further define execution modes coherent with the RTSJ, see Section 4.2.1.

In Fig. 4.2 lines 64-68 we formalize the active and passive components. We further distinguish the active components as `periodic` and `sporadic`. Periodic can not have a provided interface, sporadic can but only with asynchronous communication. We define these restrictions in Fig. 4.2 line 75.

Passive Component

Passive component, in opposition to its active counterpart, is a standard component-oriented unit providing and requiring services. When being called upon, its execution is carried out in the context of the active component demanding its services. If a passive component is used in a concurrent environment, many strategies can be applied to guarantee coherence of its internal state, e.g. a mutual exclusion mechanisms. We call such a component a `protected component`, in Fig. 4.2 line 69. However, implementation of these strategies is in the competence of a framework developer, whereas the application developer only specifies the type of the applied strategy. We further refer to this in Section 5.2.

Binding and Composing Functional Components

In Fig. 4.3 we enrich the metamodel with the concepts of `Interface` and `Binding`, which are also formally defined in Fig. 4.2. These well-known concepts introduce notions of *client* and *server* interface. Furthermore, we define different types of Bindings: `Synchronous` and `Asynchronous`. This is motivated by the specific requirements on communication between Active and Passive components. We formalize this communication in Fig. 4.2 line 75.

4.1.3 Domain Components

A brand new concept that we introduce is `Domain Component`, inspired by [NLLT08]. The main purpose of domain components is to model domain-specific requirements in a unified way.

A domain component is a composite component that encapsulates functional components and by this relation we express that these functional components support the domain-specific requirements defined by the domain component. Therefore, a domain component represents a more general application of the concept of *aspect component*, presented in Section 3.3.1.

By deploying functional components into a certain domain component, the developer specifies that these subcomponents support the domain-specific property represented by the domain component. Moreover, a domain component contains a set of attributes parameterizing its semantics. The sharing paradigm allows developers to fully exploit this concept. A functional component is a part of a functional architecture of the application but at the same time can be contained in different domain components, which thus express domain-specific requirements of this component. We are thus able to express both functional and domain-specific concerns simultaneously in the architecture.

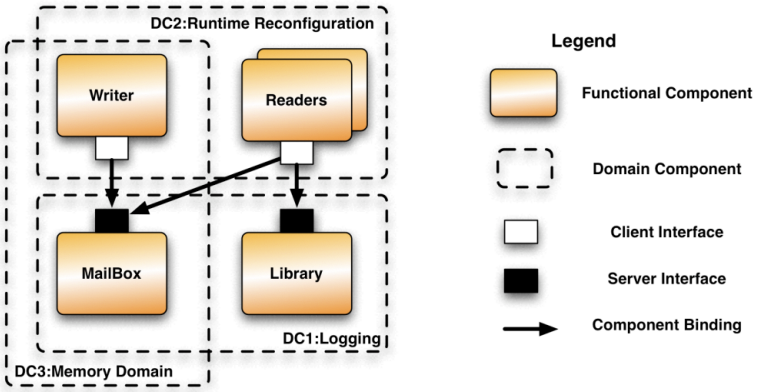


Figure 4.4: Domain Components Example

Therefore, a set of super components of a given component directly defines its functional and also its domain-specific roles in the system, the given component thus takes part both in the functional and domain-specific architecture of the system. Moreover, the domain-specific concerns are now represented as first-class entities and can be manipulated at all stages of component-software development lifecycle.

We illustrate the DomainComponent concept in Fig. 4.4. Components `Writer`, `Readers`, `MailBox`, `Library` and their bindings represent a business logic of the application. The domain component `DC1` encapsulates `MailBox` and `Library`, thus defining a domain-specific service (e.g. logging of every interface method invocation) provided by these two components. At the same time, component `DC2` represents a different service (e.g. runtime reconfiguration) and defines that this service will be supported by components `Writer` and `Readers`.

Within our model, domain components are reified as composite components. In Fig. 4.2 lines 50-68, we formalize the concept of Domain Component, we define domain components as exclusively composite, since they do not implement a functional behavior. Furthermore, we forbid nesting of domain components. Only functional components can be subcomponents of a domain component because domain components specify domain-specific properties that are shared by their sub-components.

The approach of modeling domain-specific aspects as components brings advantages commonly known in the component-based engineering world such as reusability, traceability of selected decisions or documentability of the solution itself. Also, by preserving a notion of a component, it is possible to reuse already invented methods (e.g. model verification) and tools (e.g.

graphical modeling tools) which were originally focused on functional components. If we go further and retain domain components at runtime then it is possible to reconfigure domain-specific properties represented by domain components on-the-fly.

4.2 A Real-Time Java Component Metamodel

When designing a component model for RTSJ, a sufficient level of abstraction from RTSJ complexities has to be provided. This will allow RTSJ concepts to be considered at early stages of the architecture design to achieve effective development process that mitigates all the complexities. Therefore, while keeping an appropriate level of abstraction, our goal is to define a proper representation of RTSJ concepts in the model. To achieve this, we extend the core model defined in Section 4.1.1 using the concept of domain component.

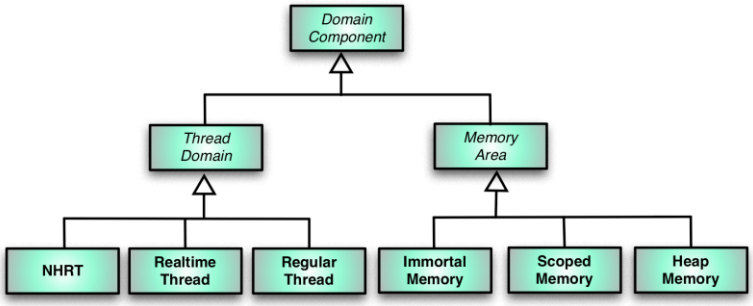


Figure 4.5: The RTSJ-specific Domain Components

In Fig. 4.5 we define a set of RTSJ compliant domain components. Their goal is to express RTSJ concerns as components and allow manipulation of these concerns as such. Two basic entities representing RTSJ concerns are defined: *ThreadDomain* and *MemoryArea*. This brings us the advantage of creating the most fitting architecture according to real-time requirements of the system. They are further described in Section 4.2.1 and 4.2.2. However, to be fully compliant with RTSJ, a set of composition and binding rules needs to be respected during the design of a real-time component system, we further elaborate on this in Sections 4.2.3 and 4.2.4.

4.2.1 ThreadDomain Component

ThreadDomain component represents `RealTimeThread`, `NoHeapRealTimeThread`, and `Regular-Thread` defined by RTSJ (see Section 2.2.1). Therefore, the model presented in Fig. 4.5 refines each thread type as a corresponding domain component. The goal of the *ThreadDomain* component is to manage threads that have the same properties (type, priority, etc.). Since in our model, each execution thread is dedicated to one active component, we deploy every active component as a subcomponent of an instance of the *ThreadDomain*. Consequently, the properties of the *ThreadDomain* are inherited by the active component precisely determining the execution characteristics of its thread of control. Therefore, each *ThreadDomain* component encapsulates all the active components containing threads of control with the same properties (thread-type, priority, etc.).

We precisely formalize the semantic of the *ThreadDomain* component in Fig. 4.6. `ThreadDomain` can not be arbitrarily nested, no RT-Thread can be defined as a descendant of another RT-Thread, line 6. Furthermore, an *active component* must always be nested in a unique `ThreadDomain`, line 12.

```
1  sig ThreadDomain extends DomainComponent {
2    priority       : int
3    memoryArea     : one MemoryArea
4  }
5
6  fact NoThreadDomainInDomainComp {
7      all t:ThreadDomain |
8        no c: DomainComponent |
9          c in t.superComponents
10 }
11
12 fact EveryActiveIsInThreadDomain {
13     all a:Active |
14       one tDom : ThreadDomain |
15         a in tDom.subComponents
16 }
17
18 sig RegularThread, RTThread, NHRT
19     extends ThreadDomain {}
20
21 sig MemoryArea
22         extends DomainComponent {
23   size: int
24 }
25
26 one sig HeapMemory, ImmortalMemory
27             extends MemoryArea {}
28
29 sig ScopedMemory extends MemoryArea {}
```

```
30 ct EveryComponentHasMemoryArea {
31   all fc: FunctionalComponent |
32     one ma: MemoryArea |
33       fc in ma.^subComponents
34 }
35
36 fact ScopedMemoryParent {
37   // all scopes are subcomponents
38   //     of the ImmortalMemory
39   all c:ScopedMemory |
40       one t:ImmortalMemory |
41         c in t.^subComponents
42   // only Immortal and Scoped memory
43   //     can be a parent
44   all c:ScopedMemory |
45     no f: (Component -
46               ImmortalMemory - ScopedMemory) |
47         c in f.subComponents
48   // scope is non-empty
49   all c:ScopedMemory |
50     some k:(Component - DomainComponent)|
51               k in c.subComponents
52 }
53
54 fact SingleParentRule {
55   all sc : ScopedMemory |
56     some parent : ScopedMemory |
57       sc in parent.subComponents or
58       sc in ImortalMemory.subComponents
59 }
```

Figure 4.6: ThreadDomain and Memory Area

Benefits of the ThreadDomain Component

The impact of the thread domain is three fold. First, a centralized management of active threads with the same properties is provided. Second, since communication between different scheduling entities in RTSJ is a crucial issue, we consider beneficial that thread domains allow designers to detect cross-thread communication. Based on this detection, corresponding patterns for its implementation can be applied, see Section 4.2.4. Finally, by introducing this entity, we are thus able to explicitly define those parts of a system that will be executed under real-time conditions. Therefore we exactly know which components have to be RTSJ-aware and we are able to enforce corresponding RTSJ rules. Moreover, communication between the system parts that are executed under different real-time or non-realtime conditions can be expressed at the architectural level. This brings an advantage of creating the most fitting architecture according to real-time concerns of the system.

4.2.2 MemoryArea Component

MemoryArea domain components represent the memory areas distinguished by RTSJ: `ImmortalMemory`, `ScopedMemory`, and `HeapMemory`. MemoryArea component thus encapsulates all functional subcomponents which have the same *allocation context*. By the allocation context we mean a memory that will be used to allocate data when executing a given component. Such specification of allocation context at the architectural level allows developers to detect communication between different memory areas (also known as *cross-scope communication*) and apply rules corresponding to RTSJ. Moreover, in combination with the ThreadDomain entity we can entirely model communication between different real-time and non-real-time parts of the system.

Although `MemoryArea` components can be nested, RTSJ specification defines several constraints to their application. We formalize these constraints in Fig. 4.6. First, we define that each functional component is deployed in a memory area, thus defining its allocation context, line 30. Furthermore, RTSJ defines a hierarchical memory model for memory areas. The immortal memory is defined as a parenting scope of all memory scopes, line 36.

Additionally, nested memory scopes can be easily modeled as subcomponents. However, dealing with Memory Scopes, the *single parent rule* - see Section 2.2.2, has to be respected. In the context of our hierarchical component model, parenting scope of each memory area can be easily identified, which considerably facilitates the scope management. This constraint is therefore formally specified in the model, the `SingleParentRule` line 54.

Instantiation Context

Apart from the *allocation context*, we distinguish the *instantiation context*. This context is used during the instantiation of the functional component and defines in which memory this allocation will be performed. In compliance with RTSJ, the instantiation memory must be parenting to the runtime allocation memory. Therefore, by default, we use the immortal memory as the instantiation context for every functional component.

4.2.3 Composing RTSJ Components

The restrictions introduced by RTSJ impose several rules on the composition process. These restrictions must be formally specified in order to construct component systems that adhere to the RTSJ. Moreover, the RTSJ-specific concerns are not only related to functional components, but also influence each other, these restrictions and dependencies has to be also formalized. We provide an excerpt of this formalization in Fig. A.3.

```
1  fact ThreadHasMemory{
2    all th:ThreadDomain | all a:Active |
3      if a in th.subComponents {
4        th.memoryArea = getMemoryArea[a]
5      }
6  }
7
8  pred Component.getMemoryArea[ a : Component]
9                : one MemoryArea {
10     one m : MemoryArea |
11       a in m.^subComponents
12 }
13
14 fact NHRTnotInHeap {
15   no c:FunctionalComponent |
16     some r:NHRT |
17       some h:HeapMemory |
18         c in r.^subComponents
19           and c in h.^subComponents
20 }
```

```
21 fact cross-thread-communication {
22    all a1,a2:Active |
23      all t1,t2:ThreadDomain {
24        a1 in t1.subComponents
25        a2 in t2:subComponents
26      }
27      implies {
28        assertOnlyAsynchrComm{a1,a2}
29      }
30 }
31
32 pred assertOnlyAsynchrComm
33            [a1,a2:ActiveComponent] {
34   all i1,i2: Interface {
35     i1 in a1.externalInterfaces
36     i2 in a2.externalInterfaces
37     i2 in i1.boundTo
38
39     a1 in t1.subComponents
40     a2 in t2.subComponents
41   }
42   implies {
43     i1.binding.communicationType
44                  = Asynchronous
45     i2.binding.communicationType
46                  = Asynchronous
47   }
48 }
```

Figure 4.7: Composition and Binding Rules for RTSJ Domain Components

First, a MemoryArea component has to be assigned to each ThreadDomain component, thus defining allocation context of each schedulable entity, according to RTSJ. However, in our model, for each active component we define a ThreadDomain – Fig. 4.6 line 12, and also, each active component has a MemoryArea – Fig. 4.6 line 30. We therefore implicitly specify a memory area for each Thread Domain. To make this explicit, we derive a new rule - Fig. A.3 line 1. An another example of RTSJ constraints between thread and memory model represents the `NoHeapRealTime-Thread` which is not allowed to be executed in the context of the Java `heap` memory. Within our design space, this constraint is translated by a `NHRTThreadDomain` which should not encapsu-

late any component encapsulated also by the `HeapMemoryArea`, see `NHRTNotInHeap` rule in Fig. A.3, line 14.

4.2.4 Binding RTSJ Components

Since components in our model are designed to represent RTSJ concepts, a special attention needs to be paid to their bindings. In our model, two types of bindings that cross real-time component boundaries can be found: *cross-thread communication* and *cross-scope communication*. We discuss them further.

Cross-Thread Communication

A communication between two threads of different priorities could introduce priority inversion [WHJ04] problem and therefore, according to RTSJ, must be properly managed. In our solution, such situation can be easily identified since it corresponds to a binding between active components residing in different thread domains. In Fig 4.7 we formalize the semantics of cross-thread communication and define exactly cases when it needs to be implemented, line 21. Line 32 further defines that cross-thread communications must be asynchronous.

Since adherence to RTSJ rules can be verified at the architectural level, the designer is able to decide which types of bindings can be used. This mitigates unnecessary complexities of the implementation phase where only the implementation of chosen binding concepts has to be performed.

Cross-Scope Communication

Our model additionally allows to clearly express cross-scope communication as a binding between two functional components both residing in different memory scopes. Here, many patterns introduced in [A. 05, BN03, PFHV04] can be used depending on designer's choice and a specific situation.

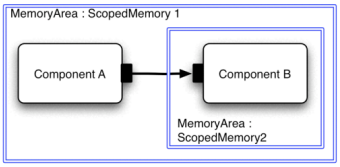


Figure 4.8: The Cross-Scope Pattern

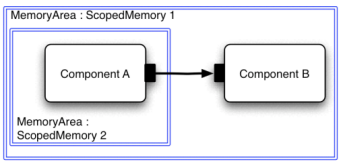


Figure 4.9: The Multi-Scope Pattern

Therefore, first we formalize the cases when a communication pattern needs to be used and derive which kind of pattern is suitable for each particular case. Moreover, we have selected the most used and widely known patterns [PFHV04], they are precisely formalized in this section, while their implementation is addressed in Section 5.2.5.

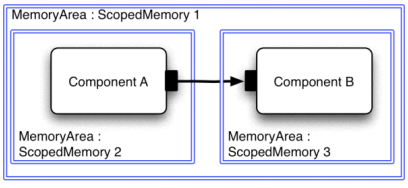


Figure 4.10: The Hand-off Pattern

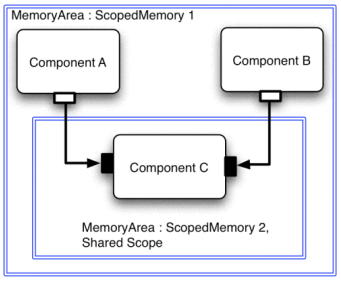


Figure 4.11: Shared Scope

```
1  fact cross-scope-pattern {
2    all c1,c2 : Component |
3      all sc1,sc2: Component {
4        c1 in sc1.subComponents
5          and c2 in sc2.subComponents
6              sc2 in sc1.subComponents
7        isClientTo[c1,c2]
8      }
9      implies
10       applyCrossScopePattern{c1,c2}
11 }
12
13 fact multi-scope-pattern {
14    all c1,c2 : Component |
15     all sc1,sc2: Component {
16       c1 in sc1.subComponents
17         and c2 in sc2.subComponents
18       sc2 in sc1.subComponents
19       isClientTo[c2,c1]
20     }
21     implies
22       applyMultiScopedPattern{c1,c2}
23 }
24
25 fact hand-off-pattern {
26   all c1,c2 : Component |
27     all sc1,sc2: Component {
28       c1 in sc1.subComponents
29         and c2 in sc2.subComponents
30       areSiblings[sc1,sc2]
31       areSiblings[c1,c2]
32     }
33     implies
34       applyHandOffPattern{c1,c2}
35 }
```

```
36 pred isClientTo[c1,c2] {
37   // returns True if c2 provides a server
38   // interface to c1
39 }
40
41 pred areSiblings[sc1,sc2] {
42   // returns true if exist a sc3
43   // that is a parent to sc1 and also to sc2
44 }
45
46 one sig CROSS_SCOPE_PATTERN,
47     MULTI_SCOPE_PATTERN,
48       HANDOFF_PATTERN in Attributes
49
50 fact applyCrossScopePattern {
51   one b: Binding | {
52     b.c in a.externalInterfaces
53     b.s in b.externalInterfaces
54     b.type = Asynchronous
55     CROSS_SCOPE_PATTERN in b.atributes
56   }
57 }
58
59 pred ApplyMultiScopePattern[c1,c2] {
60   ...
61   MULTI_SCOPE_PATTERN in b.atributes
62 }
63
64 pred ApplyHandOffPattern[c1,c2] {
65   ...
66   HANDOFF_PATTERN in b.atributes
67 }
```

Figure 4.12: Cross-scope Communication Patterns

The *cross-scope pattern*, illustrated in Fig 4.8, represents the basic communication pattern used when implementing communication between two scopes in a parent-child relation. The exact formalization of this patter is in Fig 4.12, line 1. This pattern implements entering a child scope from a parent scope and leaving the scope while deep-copying the resulted computation from a child scope to a parent scope.

The *multi-scope pattern*, in Fig. 4.9, represents a situation when we send some data from a child scope to a parent scope with the intention to store these computed data and keep them available after the child scope is reclaimed. The formalization of the rules defining when this pattern should be used is given in Fig. 4.12 line 13.

The *handoff pattern*, illustrated in Fig 4.10, is a more general application of the multi-scope pattern. This pattern is required for many real-time tasks which deal with filtering large amounts of data. Such tasks continuously process the data while retaining a small part of this data in a different scope, the main data area is reclaimed at the end of each computation. In this situation we therefore send data from one scope to another, while these scopes are not in a child-parent relation. Typically, we apply this pattern to a sibling scopes communication. The formalization of the rules defining when this pattern should be used is given in Fig. 4.12 line 13.

The *shared-scope pattern*, in Fig 4.11, corresponds to a shared scope concept defined by RTSJ. However, this pattern is rarely used in practice since it brings a high complexity into the implemented code. Developed code is hardly analyzable since the memory is reclaimed only when all the execution threads leave the scope.

In Fig. 4.12 lines 46–67 therefore show modification of the properties of the binding that is going to implement appropriate communication patterns. To achieve this, we however use concepts defined in Chapter 5, we refer reader to Section 5.2.5 for more details.

4.2.5 ADL Formalization

Although we are using a graphical notation to illustrate a real-time system architecture, we have also developed an Architecture Description Language (ADL) to precisely express system's architectures. The ADL is based on the FRACTAL-ADL [LOQS07], is extended towards the specifics of the SOLEIL metamodel and is defined in Alloy.

This allows us to easily express system's architecture and apply formalized rules and restrictions defined for our component model. In Fig. 4.13 we show a snippet of our ADL defined in Alloy. The set of predicates and functions is proposed to construct a system's architecture and also to define RTSJ-specific characteristics into the architecture, e.g. component – line 1, interface – line 5, binding – line 15 and many other properties – lines 12-31. Furthermore, such architecture description can be easily translated into XML format, allowing its further processing by third party tools.

```
1  pred Component.component [n : String] {
2   c : this.subComponents | c.name = n
3  }
4
5  pred Component.interface [n: Name, r : Type] {
6    one i : Interface | {
7      i in c.externalInterfaces
8      and i.name = n and i.role = r
9    }
10 }
11
12 pred Component.primitive {
13   this in Primitive
14 }
```

```
15 pred binding[clien : Interface
16         serv : Interface] {
17    one b : Binding |
18      b.client = clien
19      b.server = serv
20      b.communicationType = Synchronous
21    }
22 }
23
24 pred bindingAsynchronous[clien : Interface
25         serv : Interface] {
26    one b : Binding |
27      b.client = clien
28      b.server = serv
29      b.communicationType = Asynchronous
30   }
31 }
```

Figure 4.13: SOLEIL ADL defined in Alloy

4.3 SOLEIL Framework

Based on the RTSJ model defined in Section 4.2 we construct a framework supporting development of component based RTSJ-oriented systems - SOLEIL [Ale09], addressing goal **G2**. The framework embraces both design and implementation methodologies for RTSJ applications. Our motivation when proposing these methodologies is to fully benefit from the concepts introduced in our RTSJ metamodel.

The SOLEIL framework substantially influence the procedures of designing and implementing RTSJ-based applications. At the design time, we employ separation of concerns to design both functional and RTSJ-related architectures, consequently, we exploit the formalization of the metamodel to verify consistence of produced architectures. At the implementation time, we define implementation tasks for application developers while generating automatically the framework glue code and RTSJ-related glue code. Furthermore, we specify a SOLEIL profile defining a set of rules that must be respected during the implementation of functional components in order to achieve RTSJ compliant implementations. To validate conformance of implemented components to the profile, we translate the profile into the OCL [RG02] and propose the validation process based on this technology.

In the remainder of this section we therefore describe the SOLEIL framework. First, in Section 4.3.1 we introduce the design methodology based on our RTSJ component model. Second, in Section 4.3.2 we introduce the development methodology guiding the users through the process of developing functional components and automatically generating the nonfunctional concerns. The framework is described from the application developer perspective, whereas leaving the discussion about its implementation in Chapter 5. Finally, in Section 4.3.3 we present SOLEIL profile

- a set of rules and restrictions for development of RTSJ-compliant applications, and we present an approach to validation of this conformance in Section 4.3.4.

4.3.1 Design Methodology

The elevation of RTSJ concepts to the architectural level may hinder our task for an appropriate level of abstraction, since we are combining functional and real-time concerns. However, when developing real-time applications these concerns need to be considered at early stages of development, since they can influence the architecture of the whole system. Therefore, to avoid increased complexity of the design phase we propose a new design methodology with motivation to fully exploit the advantages of our component model at the design time.

The abstractions introduced in our model allow designers to gradually integrate RTSJ concerns into the architecture. Therefore, in the methodology we decouple the design process into several steps where each step focuses on different concepts of RTSJ. To clarify these steps, three design views are proposed: *Functional View*, *Thread Management View*, and *Memory Management View*. Whereas the functional view considers only functional aspects of the system, the two others stress on different aspects of RTSJ programming. Consequently, the views together with the RTSJ abstractions provided in our model allow developers to focuss on different concerns in the architecture and to design these aspects independently of the functional architecture.

In the rest of this section we describe these design views and their application in the design methodology.

Functional View

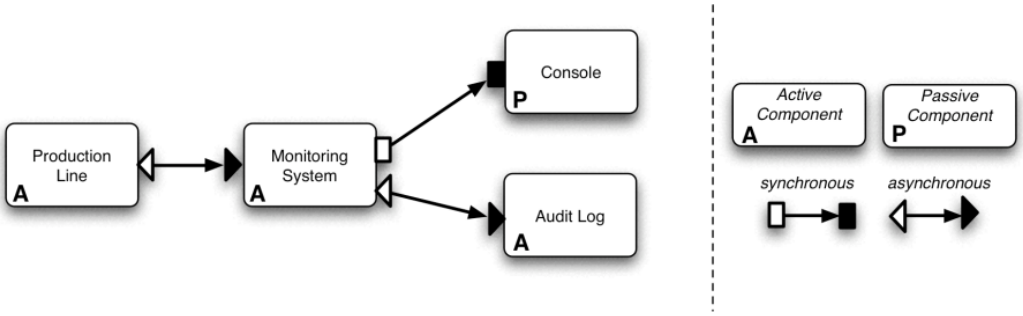


Figure 4.14: Functional View

The functional view deals only with the composition of functional – active/passive components. This helps developers to focus exclusively on designing functional aspects of the system. To illustrate the idea, we revisit our SweetFactory scenario presented in Section 2.2.3. The functional architecture of the system is constructed in Fig. 4.14, as we can see, we define each task in the SweetFactory as a self standing component and we clarify the communication between them, focusing solely on the business logic of the system.

Thread Management View

The thread management view considers only instances of *ThreadDomain* entities and active components. This allows designers to naturally filter out the passive components and the designer can focus on inter-thread communication represented by bindings between different active components.

At this point, reasoning about appropriate types of bindings between active components is simple, since bindings that cross boundaries of *ThreadDomain* components are clearly expressed.

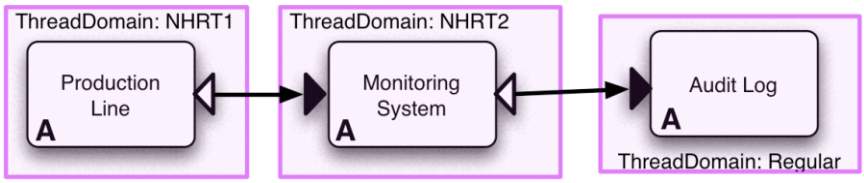


Figure 4.15: Thread Management View

We can thus easily architect the system in conformance to inter-thread communication restrictions introduced in Section 4.2.4. Additionally, we can smoothly change the execution characteristics of the system by designing several different compositions of *ThreadDomain* and *Active* components, which is beneficial when tailoring the system for different real-time conditions.

Looking at our scenario example depicted, using the thread management view, in Fig. 4.15, the *ProductionLine* and *MonitoringSystem* are deployed in the NHRT domain, since they must be executed in a highly predictable manner in order to meet the 10ms deadlines. Furthermore, both of them are in different instances of NHRT because they run with different priorities. Unlike the other components, the *AuditLog* component is not timing-critical, thus we can design it for use under regular Java.

Memory Management View

The memory management view allows developers to focus on managing different memory regions of the application. Active and passive components are deployed into instances of the *MemoryArea*, thus defining in which scope they are operating. Additionally, the bindings crossing different memory regions can be easily identified. This facilitates to appropriately deploy a glue code managing the cross-scope communication, introduced in Section 4.2.4. Similarly to the thread management view, different assemblies of components into memory regions tailored to fit various real-time conditions can be delivered.

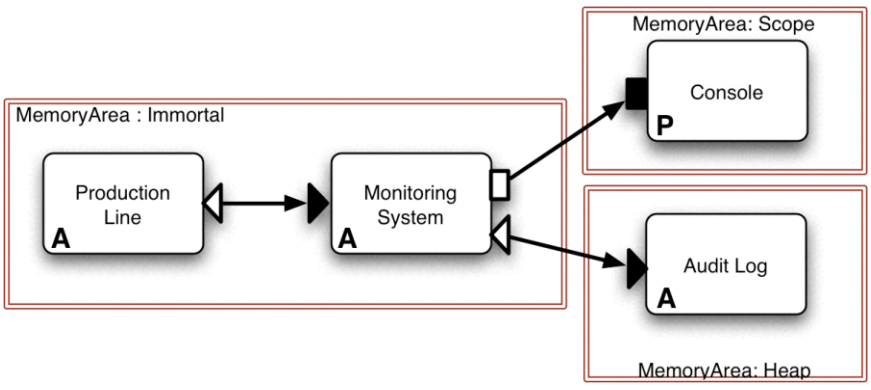


Figure 4.16: Memory Management View

Considering our motivation scenario, we deploy the memory management as follows. Since both threads executing *ProductionLine* and *Monitoring System* components run through the lifetime of the application, they can be allocated in the `ImmortalMemory` region. On the other hand, the *Console* component is accessed by the NHRT thread irregularly and therefore a `ScopedMemory` region is sufficient here. The *AuditLog* executed by a regular thread is allocated in a `HeapMemory` region. The final composition of the memory management can be seen in Fig. 4.16.

Applying the Design Methodology

The architecture design flow of our methodology is depicted in Fig. 4.17. First, we analyze system requirements which are divided into *functional* and *real-time requirements*. From the *functional requirements*, describing functional tasks of the system, we design interfaces, components, and consequently the *functional architecture*, in Fig. 4.17. So far, we follow the well-known concept of component-architecture design [CCL06] and the functional view can be employed.

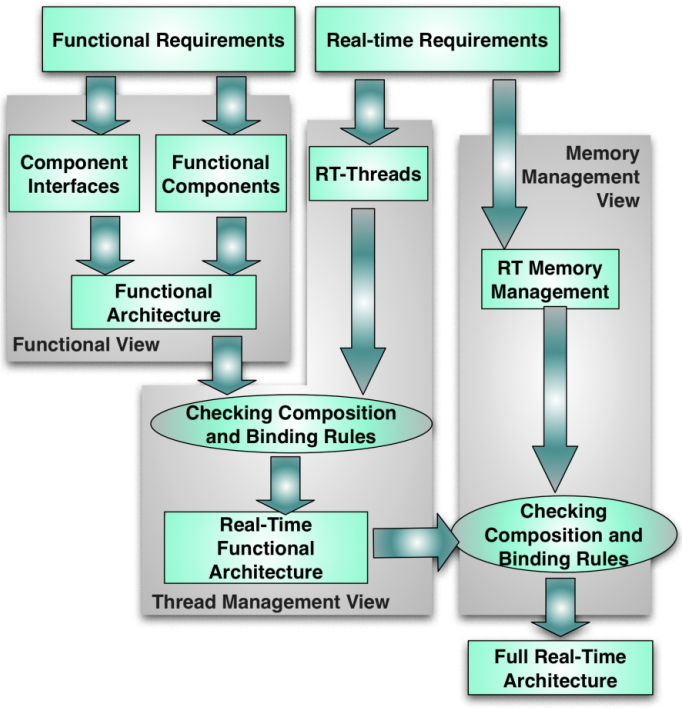


Figure 4.17: RealTime Component Architecture Design Flow

The *real-time requirements* describe real-time properties of the system, here we have to determine parts of the system that will be executed under real-time conditions – using the *Thread Management View*. Consequently, we deploy different parts of a functional system into various real-time and non-real-time components to obtain the *real-time functional Architecture*, as depicted in the figure. This can be easily achieved since our component model allows us to abstract different real-time units through `ThreadDomain` components. Then we extend the *Real-Time Functional Architecture* by an appropriate memory management – using the *Memory Management View*, thus achieving a complete and RTSJ compliant architecture of a real-time system.

We implement this design methodology in Alloy, by using the ADL defined in Alloy we are able to gradually design the application in the Alloy language and by using Alloy Analyzer [KM08] we immediately verify its conformance to the metamodel – thus implementing the *Checking Composition and Binding Rules* process defined in our methodology. We provide more details to this validation process later in this section.

4.3.2 Implementation Methodology

The design analysis described in the previous section yields in a *real-time system architecture* which is both RTSJ compliant and fully specifies the system together with its RTSJ related characteristics. Hence, it can be used as input for an implementation process where a high percentage of tasks

is accomplished automatically. Therefore, we adopt a generative-programming approach [CE00] where the non-functional code (e.g. the RTSJ-specific code) is generated automatically.

This approach allows developers to fully focus on implementation of functional properties of systems and entrust the management of non-functional concepts into the competence of the framework. Thus we eliminate accidental complexities of the implementation process. The separation of concerns is also adopted at the implementation level where functional and non-functional aspects are kept in clearly identified software entities - components.

We therefore introduce a new implementation methodology incorporating code generation technique, depicted in Fig. 4.18. The process guides the developers through the implementation phase, specifying the order of implementation of functional and RTSJ-specific concerns.

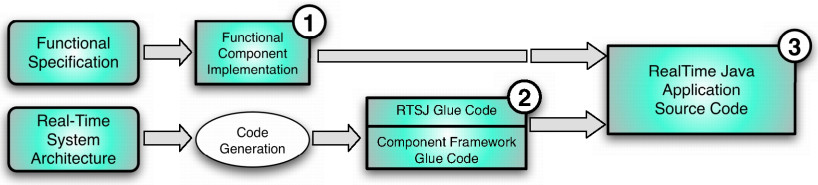


Figure 4.18: Runtime Platform Generation Flow

Implementing Functional Concerns of Applications

As the first step of the implementation flow, see Fig. 4.18 step 1, functional logic of the system is developed. Since we further follow component-oriented approach, the concept of separation on functional and non-functional code is preserved. The development process thus follows the classical approach where developers implement only component content classes. For illustration example see Section 4.4.2.

Infrastructure Generation Process

We further alleviate the implementation phase by employing the methods of generative programming to instantiation of the *runtime platform*. To fully exploit the advantages of the framework, we pose the following requirements on the generation process: (i) the functional and real-time concerns have to be deployed into clearly identified software entities; (ii) the generation process needs to be independent on the functional code. By meeting these requirements, we achieve transparency of the system implementation.

This instantiation of the runtime platform is the second step of our development process, in Fig. 4.18 step 2. In this step, we exploit the already designed *real-time system architecture*, created using the design methodology specified in the previous section, and we generate a glue code managing RTSJ-specific and domain-specific properties of the system. The generation process implements several tasks, we list them below.

- **RTSJ-related Glue Code**
  - *Realtime Threads and MemoryArea management* Real-time thread and memory areas management is the primary task of the generated code. Automatic initialization and management of these aspects in conformance to RTSJ substantially alleviates the implementation process for the developers.
  - *Cross-Scope Communication* Since the RT system architecture already specifies which cross-scope communication patterns will be used, their implementation can be moved under the responsibility of the code generation process.

  - *Initialization Procedures* The generated code has to be responsible also for bootstrapping procedures which will be triggered during the launch of the system. This is important since RTSJ itself introduces a high level of complexity into this process.

- **Framework Glue Code**

  - *Active Component Management* For active components, the framework manages their lifecycle - generating code that activates their functionality.
  - *Communication Concepts* Automatic support for synchronous/asynchronous communication mechanisms is important aspect offloading many burdens from developers.
  - *Additional Domain-Specific Support* Additionally, many other domain-specific properties can be injected by the framework: e.g. a support for introspection and reconfiguration of the system, lifecycle management, or component properties management.

The glue code is deployed into *containers* - encapsulating units of functional components. However, as already said, in this chapter we have described the challenges of the runtime platform generation process and we further address these challenges in Chapter 5.

Final Composition Process

Finally, by composing results of the functional component implementation and the infrastructure generation process we achieve a comprehensive and RTSJ-compliant source code of the system, in Fig. 4.18 step 3. Here, each functional component is wrapped by a layer managing its execution under real-time conditions. This approach respects our motivation for clear separation of functional and real-time concerns.

4.3.3 SOLEIL Profile

The methodology that we have introduced in the SOLEIL framework guides the developer through the development process while mitigating many complexities introduced by RTSJ. However, to effectively profit from the benefits of the framework, developers must follow the guidelines defined by the methodology. To clarify clearly all the constraints that must be respected, we introduce the SOLEIL profile. The profile summarizes the guidelines introduced either by the SOLEIL framework or by the SOLEIL component model into a consistent set of restrictions and rules.

When constructing the profile we have identified two kinds of constraints that developers must respect: (i) *architectural level constraints* are imposed over the architecture of applications and were extracted from the SOLEIL component model, and (ii) *implementation level constraints* are imposed on the implementation of the functional components and were extracted from the SOLEIL implementation methodology. We describe these two kinds of constraints in the remainder of this section.

Moreover, the profile is consequently used by the framework to implement a verification process that guarantees that developed architecture and implementation are coherent with the profile. Our motivation is to further enhance the development process by a verification approach that provides an immediate feedback, in order to facilitate effective development of applications.

Architectural Level Constraints

The architectural level constraints are based on the SOLEIL component model introduced in Section 4.1 and Section 4.2. These constraints have been specified as Alloy rules defining relations between entities of the metamodel. Since the SOLEIL methodology proposes to design application architectures using our ADL defined in Alloy, we reuse also the Alloy facts to define all the constraints at the architectural level.

Implementation Level Constraints

SOLEIL provides a set of high-level tools, methods and patterns allowing developers to generate runtime platforms in a generic way according to the designed architectural artifacts. The goal is to spare the application developer from dealing with the RTSJ related concerns at implementation level. However, to allow the automatically generated code and functional components to fit into each other, the application developers must not use RTSJ concepts at the level of functional components. We further list these restrictions.

- **Functional Component**
  - must not instantiate new threads, since the thread management is in competence of the SOLEIL framework
  - **Active Component**
    - ◇ *Periodic Active Component* must implement a unique task entry point which will be periodically called by the runtime platform (such as the Java `Runnable` interface).
  - **Passive Component**
    - ◇ for every *protected component*, a synchronization logic must be specified.
    - ◇ since the *protected component* services are executed as a critical section, the implementations of these services must not be based on internal synchronization mechanisms (such as the `synchronize` Java keyword) to avoid deadlocks.
- **Immortal Memory**, all the functional components contained in the Immortal Memory domain must:
  - not call the `new` statement outside of the instantiation section,
  - use only RTSJ compliant libraries specified in advance, e.g. Javolution [Dau07]. Any use of non-authorized classes or libraries will be reported.
- **Asynchronous Communication**
  - a method used to implement a provided asynchronous service must have `void` as the return type.
- **Cross-Scope Communication**
  - the methods specified within interfaces involved in a cross-scope communication must have parameters and return values of serializable types to ease their marshaling, and must not declare checked exceptions.

4.3.4 Validation Process

During the development process we implement verification of developed systems in order to guarantee RTSJ conformance. We verify the applications under development in two steps – at the design time to validate correctness of instances of our RTSJ component model, and at the implementation time - to validate conformance of functional component' implementations with the SOLEIL Profile.

Design Time Validation

The compliance of the application architecture with RTSJ is enforced during the design process. This is possible since the component model and its composition rules are fully formalized in Alloy. We are thus able to formally verify conformance of a designed architecture to our model and to the RTSJ rules. Therefore, the formally defined semantics of the model, specified in Section 4.1

and in Section 4.2, are verified by the Alloy Analyzer [KM08] and thus we can immediately observe impact of rules not only on the model but also on its instances.

The verification proves to be beneficial when functional and thread domain components are specified, since the memory domain components are proposed automatically by the tool in accordance to the model. This is facilitating the whole design process since the designer can choose from proposed instances of the model the one which suits his needs. Furthermore, such approach provides an immediate feedback and the designer can appropriately modify an architecture whenever it violates RTSJ. Moreover, the verification process of the architecture identifies the points where a glue code handling RTSJ concerns needs to be deployed, which substantially alleviates complexities of the implementation phase.

Also, this gives designers an opportunity to experiment with different configurations of thread and memory domains at the design time, while still being compatible with RTSJ. The execution characteristics of systems can be smoothly changed by designing several different assemblies of components into *ThreadDomain*s and *MemoryArea*s. This is beneficial when tailoring the same functional system for different real-time conditions.

Implementation Time Validation

To ensure correctness of the application implementation, the implementations of functional components must follow the rules specified in the SOLEIL Profile. However, these rules can not be expressed by Alloy, since they specify constraints upon the implementation of functional components. Therefore, we specify the SOLEIL profile in OCL [RG02], which consequently allows us to check implementation of functional components.

In order to perform checking, we use an approach [NL09] inspired by the HULOTTE framework proposed in Chapter 5. In this approach, we first translate our metamodel into its EMF [BSE+04] version. Consequently, we use Spoon [Paw06] tool suit that provides an EMF metamodel for a Java source code. We therefore obtain both our metamodel and functional component implementation as model instances in EMF. Finally, confront these metamodels with the OCL rules of the SOLEIL profile. The full approach is documented in [NL09] and for illustration we provide an extract of the OCL rules in Appendix B.

4.4 Motivation Scenario Revisited

At this place we revisit our SweetFactory motivation scenario introduced in Section 2.2.3, we illustrate the basic ideas presented and highlight some benefits of the SOLEIL framework.

4.4.1 Designing the Motivation Scenario

We have already demonstrated application of our design methodology on the SweetFactory in Section 4.3.1, here we therefore only shortly recapitulate the process and highlight interesting details.

We first use the functional view to obtain functional architecture and we can gradually integrate real-time concerns. After deploying all components into corresponding `ThreadDo- main` components, the composition and binding rules verification is conducted by the Alloy Analyzer. As a result, the bindings between components will be identified as a RTSJ violation – they express communication between threads of different types, according to the rules defined in Fig. 4.7. Consequently, possible solutions will be proposed, according to the binding rules specified in our model (Section 4.2.4). The final architecture of the system is shown in Fig. 5.26.

We further apply the formalization of our model to validate the application architecture. We have incrementally designed the system architecture using our formalized ADL from Section 4.3.1. We show an excerpt of the resulting specification in Fig. 4.20, moreover, we present the whole ADL specification also in XML format in Appendix C. Consequently, this final architecture is analyzed by the Alloy Analyzer in order to validate its conformance to the model.

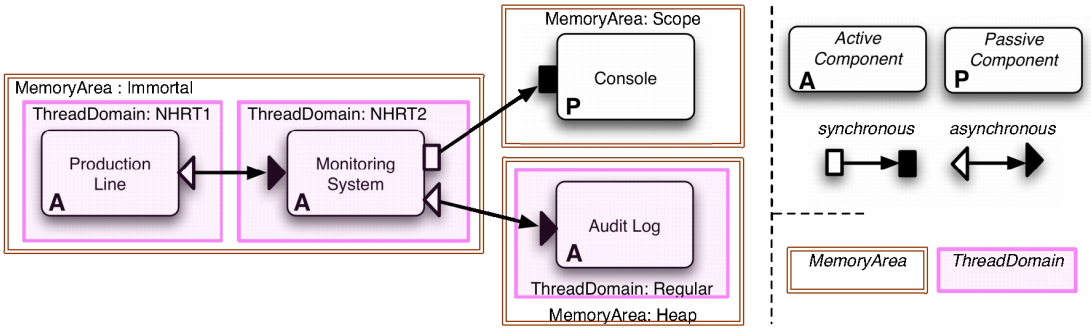


Figure 4.19: Sweet Factory: Real-time System Architecture

```
1  one sig SweetFactory
2              extends Composite {}{
3    component[ProductionLine]
4    component[MonitoringSystem]
5    component[AuditLog]
6    component[Console]
7
8    bindingAsynchronous[producerOut,
9           monitorIn]
10   binding[consoleOut,consoleIn]
11   bindingAsynchronous[auditLogOut,
12          auditLogIn]
13 }
14
15
16 one sig ProductionLine extends Active {} {
17     primitive
18     interface[producerOut,Client]
19 }
20
21 one sig MonitoringSystem extends Active {} {
22    primitive
23    interface[monitorIn,Server]
24    interface[consoleOut,Client]
25    interface[auditLogOut,Server]
26 }
27
28 one sig Console extends Active {} {
29    Console.primitive
30    interface[consoleIn,Server]
31 }
```

```
32 one sig AuditLog extends Passive{} {
33     primitive
34     interface[auditLogIn,Server]
35 }
36
37 one sig NHRT1 extends NHRT {} {
38    priority = 37
39    component[ProductionLine]
40 }
41
42 one sig NHRT2 extends NHRT {}{
43    priority = 36
44    component[MonitoringSystem]
45 }
46
47 one sig Scope extends ScopeMemory {
48    component[Console]
49 }
50
51
52 fact RegularThreadsSpecification {
53    AuditLog in RegularThread.subComponents
54    AuditLog in Heap
55 }
56
57 fact MemoryAllocations {
58    ProductionLine, MonitoringSystem
59          in ImmortalMemory
60 }
```

Figure 4.20: Sweet Factory Architecture: Formalization in Alloy

4.4.2 Implementing the Motivation Scenario

The process of implementing applications in our framework is largely influenced by the motivation to generate automatically the most of the source code and by the intention to mitigate complexities of RTSJ.

Functional Component Implementation

To illustrate the implementation process in the SOLEIL framework, we present source-code of the `MonitoringSystem` component. Since its implementation in plain RTSJ have been given in Section 2.7, we can easily compare benefits of both approaches.

The source code snippet of the `MonitoringSystem` component is presented in Fig. 4.21, line 5. The structure and conventions for the code are inspired by the Fractal implementation process introduced in [OW209b]. The `MonitoringSystem` class implements `IProducer` interface, defined on line 1, which corresponds to the provided interface of the component. Further,

the fields `auditlog` and `console`, lines 6– 7, represent required interfaces of the component. Since all of this information is given already at the architectural level, this source code can be automatically generated. We can therefore draw our attention to the `processMeasurement` method, line 9, which implements functional logic of the component. As we can see, the method implementation is straightforwardly following its logic and is not tangled with RTSJ-related code.

Focusing on the communication with the `AuditLog` and `Console` component, both residing in memory areas different than `MonitoringSystem`, developers does not have to face the complexities of memory context switching. The provided interfaces of the `AuditLog` and `Console` can be accessed transparently from the component's implementation, SOLEIL framework will guarantee correct switching of memory allocation contexts. Furthermore, although the binding with `IAuditLog` interface is implemented as asynchronous, using a `WaitFreeQueue`, this is transparent for the developers which can directly call its methods, the appropriate code handling the invocation will be automatically injected.

Furthermore, the developers does not have to deal with instantiation of the component, since these are automatically generated. Already at the architectural level we know in which allocation context the component will be instantiated and operating, therefore, component's instantiation process can be performed automatically. Similarly, initialization of the component's fields is performed automatically when instantiating `AuditLog` and `Console` components.

```
1  interface IConsumer {
2    public void processMeasurement(Measurement m){
3  }
4
5  class MonitoringSystemImpl implements IConsumer {
6    IConsole iConsole;
7    IAuditLog iAuditLog;
8
9    public void processMeasurement(Measurement m){
10     if (m.isWrong())
11       iConsole.reportError(m);
12     iAuditLog.log(m);
13   }
14 }
```

Figure 4.21: `MonitoringSystem` Component Implementation in SOLEIL

Implementing the Real-Time Concerns

The implementation of the RTSJ concerns is substantially alleviated, since the verification process verifies the architecture and determines injection of the appropriate glue-code. Moreover, as demonstrated in Fig 4.20, the system's architecture that we obtain provides the whole information needed to implement the runtime plaftorm described in Section 4.3.2, for example:

- the functional component `ProductionLine` is defined as a *periodic active* component,
- the binding between `MonitoringSystem` and `AuditLog` active components specifies an asynchronous communication.
- the non-functional components specify RTSJ-related attributes, such as a *memory type* and *size* of a `MemoryArea`, a *thread type* and a *priority* for a `ThreadDomain`.

As we have already said, Chapter 5 is dedicated to the process of designing and implementing the RTSJ concerns and therefore we provide full description of the RTSJ-glue code implementation process in this chapter.

4.5 Summary

In this chapter we present a component framework designed for development of real-time and embedded systems with the Real-Time Specification for Java (RTSJ). Our goal is to alleviate the development process by providing means to manipulate real-time concerns in a disciplined way during the development and execution life cycle of the system. Furthermore, we shield the developers from the complexities of the RTSJ-specific code implementation by separation of concerns and automatical generation of the runtime platform.

As the cornerstone of our approach we propose the `Domain Component` concept - a unified approach to model domain-specific concerns as first class entities (**G1.1**). This concept allows developers to manipulate these concerns easily during the full application development lifecycle.

Based on this concept, we define a component model tailored directly to the specifics of RTSJ (**G1**). Our contributions include separation of RTSJ-specific and functional concerns(**G1.2**), and the ability to express these concerns at the architectural level. Therefore, the model allows us to clearly define real-time concepts as software entities and to manipulate them through all the steps of the system development. Furthermore, we have formalized the metamodel semantics and RTSJ-related restrictions using the Alloy language (**G1.3**).

Consequently, we propose SOLEIL- a component model embracing the introduced concepts (**G.2**). As a part of the framework, we define a methodology for design and implementation of RTSJ applications (**G2.1**). The methodology clarifies manipulation of functional and RTSJ related concepts at design and implementation time, while still respecting full separation of these concerns. Moreover, using the formalized component model we can design applications and verify their conformance to RTSJ already at the design time. Finally, we alleviate the implementation phase by providing a process generating automatically RTSJ-related code and framework glue code based on a formalized real-time system architecture.

Our example scenario demonstrates that the presented solution allows to simultaneously design real-time and non-real-time parts of applications. This is important when trying to mitigate complexities of the RTSJ implementation phase. The model allows designers to easily introduce new assemblies of real-time components thus adapting the system for different real-time conditions.

The bottom line is that we are able to express different real-time concerns, and to integrate them seamlessly into the system's architecture. Moreover, by gradual integration of RTSJ aspects we mitigate the complexities of RTSJ development. Considering the implementation of each component, the designed architecture substantially simplifies this task. Functional and real-time concerns are strictly separated and a guidance for possible implementations of those interfaces that cross different concerns is proposed.

In the following chapter we address the process of implementing domain components and further we describe the approach towards automatic instantiation of the runtime platform (**G2.2**).

Chapter 5

HULOTTE: A Framework for the Construction of Domain-Specific Component Frameworks

Contents

IN Chapter 4 we have introduce domain components - a concept to expression of domain-specific concerns already at the architectural level. Furthermore, we have applied this context to the challenges of RTSJ and showed how they mitigate complexities of RTSJ programming and simplify application architectures through a consistent separation of concerns. Finally, we have incorporated this concept into the SOLEIL framework - a proposal solution to a fully fledged framework for development of real-time Java applications. The concept of domain components was described with the emphasis on their application and benefits, however, the challenge of domain component implementation was not addressed.

In this chapter, we focus specifically on domain components and propose an approach to their design and implementation. We have chosen a model-driven approach of step-wise refinement [BSR03, EM08] to gradually refine domain component architectures. To facilitate this, we identify a three general architectural patterns used during the refinement process. Furthermore, we introduce the HULOTTE framework - a set of tools that implements the refinement process. The framework allows to process architectures specified by application developers and execute the refinement process of domain components in order to provide their implementation. As the outcome of this process, a runtime platform providing support for execution of developed applications is instantiated. During the instantiation process we also introduce various heuristics to optimize applications' performance and memory footprint.

In the terms of the refined goals, presented in Section 3.5, we address the goal **G2.2** – HULOTTE framework provides a unified approach to instantiation of runtime platforms, automatically generating both framework glue code and RTSJ-specific code.

Contributions

The contributions of this chapter are:

- **HULOTTE Framework and Architecture Refinement Process.** The first contribution of this chapter is the HULOTTE framework which introduced concepts that are used to design and implement semantics of domain components. Consequently, we identify common patterns that are employed by developers when implementing semantics of domain components. Based on this, we introduce the *architecture refinement process* that specifies how to use architectural patterns to refine domain components, thus allowing their implementation.

- **Applying HULOTTE to SOLEIL.** Based on the architectural refinement proposed by the HULOTTE framework we refine and implement SOLEIL domain components.

- **HULOTTE Implementation.** We present the technology applied to implement the HULOTTE framework. We show how HULOTTE implements an approach to automatical refinement of domain components and generation of the SOLEIL glue-code. Furthermore, we discuss how this approach can be used in a more general perspective to leverage development of domain-specific component frameworks.

Structure of the Chapter

To remainder of this chapter is structured as follows. In Section 5.1 we extend the Real-time Component model from Chapter 4 and we introduce new concepts intended to refine semantics of domain components. Consequently, we propose architectural patterns defining how to implement these semantics. Section 5.2 shows application of the newly introduced concepts and patterns to implement the domain components of the SOLEIL framework proposed in the previous chapter. In Section 5.3 we describe our prototype implementation of the HULOTTE framework. In Section 5.4 we revisit our motivation scenario to illustrate applications of the concepts proposed in this chapter. Finally, a summary of this chapter is given in Section 5.5.

5.1 HULOTTE Framework

The goal of this section is to extend the real-time component model from chapter 4 to provide sufficient concepts to design and implementation of domain components. Therefore, in Sect. 5.1.1 we propose extensions to our model intended to refine semantics of domain components. The enriched model is divided into two levels: whereas the *core level* represents concepts addressed in the previous chapter, the *platform level* is dedicated to domain components and introduce concepts that will be used to their refinement. The key motivation when refining the model is to

provide sufficient abstractions while keeping simplicity of the concepts manipulated by functional component developers.

Second, in Section 5.1.2 we introduce the *architecture refinement process*. We define *architectural patterns* that are specifically defined to describe manipulation and application of the platform-level concepts. Consequently, the architecture refinement process defines how *framework developers* refines the application architecture through these *architectural patterns*.

5.1.1 Generic Component Model Extensions

We show the extended metamodel in Fig. 5.1. The metamodel is divided into two levels. The core level corresponds to the metamodel introduced in the SOLEIL chapter (Fig. 4.1), however, only the concepts relevant for this discussion are preserved. The platform level introduces the concepts which are used to refine domain components. The important aspect to highlight is that these concepts are hidden from functional component developers and final users of domain components. We further describe the concepts of the platform level in more details.

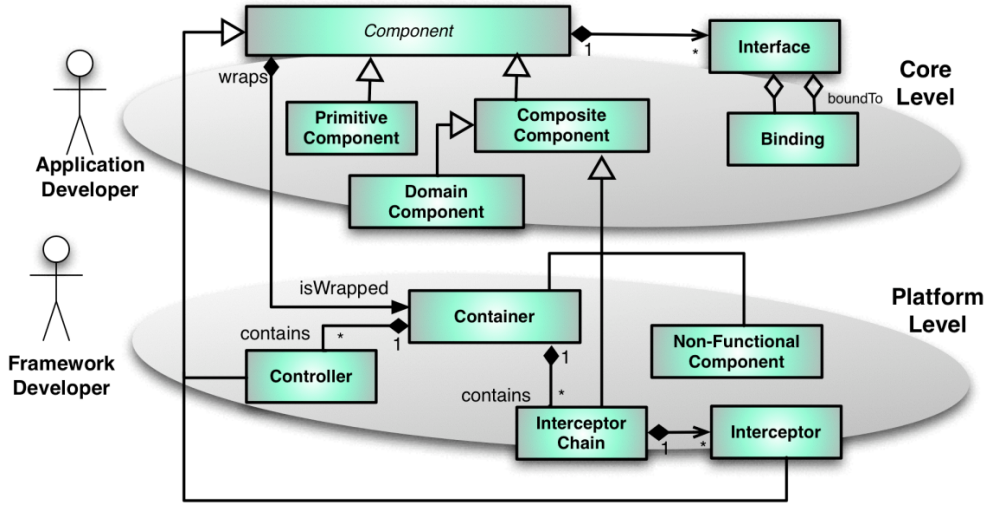


Figure 5.1: Component Metamodel and Domain Component

Component-Oriented Container

The container paradigm, introduced in Section 3.1.2, is the key concept used at the platform level. A container is a composite component wrapping every component in the application and is composed of a set of controllers and interceptors, it implements domain-specific services required by components. In Fig. 5.2, we first define Container, line 1, and assign one to every component, line 13.

We consequently develop our own set of containers for functional components which are specially designed to manage and implement domain-specific concerns in the application.

Functional and Control Interfaces

Furthermore, we distinguish functional and control interfaces, as depicted in Fig 5.3 and defined in Fig 5.2, line 38. Whereas *functional interfaces* are external access points to components, *control interfaces* are in charge of some domain-specific properties of the component, for instance its life-cycle management, or the management of its bindings with other components. *Control-interfaces* thus provide access points to container functionality that is hidden at the functional level to avoid confusion with the functional implementation.

```
1  sig Container extends Composite {
2    content              :  one Component
3    chains               :  set InterceptorChain
4    controllers          :  set Controller
5    controlInterface     :  set ControlInterface
6  } {
7    controlInterfaces in ExternalInterface
8    content, chains,
9        controllers in this.subComponents
10 }
11
12 sig Component {
13   container : one Container
14   ...
15 }
16
17 sig Controller extends Composite {
18   controlInterfaces :  set ControlInterface
19   content           :  lone PrimitiveObject
20   owner             :  one Container
21 }
22
23 sig Non-FunctionalComponent extends Composite{}
```

```
24 sig InterceptorChain extends Composite {
25   owner          :  one Component
26   inInterface    :  one FunctionalInterface
27   outInterface   :  one FunctionalInterface
28   trapInterface  :  one ControlInterface
29 } {
30   subComponents in Interceptor
31     inInterface, outInterface, trapInterface
32      in this.externalInterfaces
33 }
34
35 sig ControlInterface, FunctionalInterface
36       extends Interface {}
37
38 fact Interfaces {
39   Interface = FunctionalInterface
40               + ControlInterface
41 }
42
43 sig Interceptor extends Component {
44   owner :  one InterceptorChain
45 }
```

Figure 5.2: Platform Level Concepts Specified in Alloy

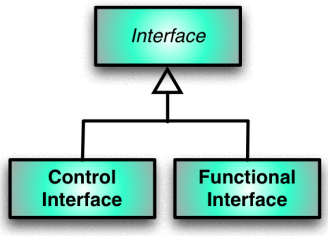


Figure 5.3: Functional and Control Interfaces

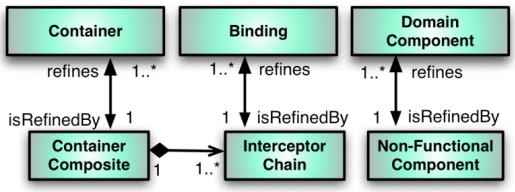


Figure 5.4: Architectural Patterns

Component Controller

Controllers are, together with interceptors, basic building units of containers, they implement domain-specific concerns and can be accessed by control interfaces. The control components incorporated in container can be divided into two groups. First, the controllers which are specific to the domain-specific needs of the component - e.g. asynchronous communication controller or RTSJ-related controllers. These components have to be present in the membrane since they implement non-functional logic directly influencing components' execution. The second group of controllers represent units which are optional and are not directly required by the component's functional code, e.g. Binding or Lifecycle controllers.

InterceptorChain

The `InterceptorChain`, in Fig 5.2 line 24, concept is defined in order to develop more advanced intercepting logic in interceptors while still achieving separation of concerns. Therefore, `InterceptorChain` is an element managing a chain of interceptors deployed on a specific functional interface. Each interceptor in the chain symbolizes one domain-specific concern which reflects communication *in situ* (e.g. monitoring) or modifies communication (e.g. adaptation of method call parameters). The presence and position of the interceptor in the chain is influenced by properties of the modeled binding and also by a presence of other interceptors.

This representation allows us to build bindings easily with different functionalities — by selecting relevant interceptors and their order in according to specified properties. The proper-

ties, according to which is the InterceptorChain component constructed, are the properties of the `Binding` entity, defined in the SOLEIL metamodel in Fig. 4.2. Furthermore, the division of the `InterceptorChain` architecture into separated interceptors permits handling real-time specific properties separately in dedicated interceptors.

Following this motivation, also each interceptor can be either `primitive` or `composite`. Furthermore, interceptor components can be interconnected by dedicated control interfaces called `TRAP`, thus allowing centralized management of strategies for interception mechanisms.

Development Roles

We have divided our model into two levels in order to clearly distinguish the defined concepts. Consequently, we also distinguish development roles that manipulate these concepts: *application developer* and *framework developer*.

The role of the *application developer* is to create and implement functional components and to specify domain-specific requirements by deploying functional components into domain components. While the application developer is aware of the semantics behind domain components, he does not provide their implementation and therefore can fully focus on functional concerns of the application. To give an example, a domain specific component `ThreadDomain` can specify execution context (an executing thread and its properties) of an active functional component, however, the application developer does not need to know how these properties are enforced at runtime.

The role of the *framework developer* is to define and implement semantics of domain components. First, his responsibility is to define domain components according to the needs of application developers and to define the rules constraining application of domain components at the functional level. Afterwards, the framework developer designs and implements semantics of domain components using the platform-level concepts — see Fig. 5.1, and the architectural patterns that we introduce in Sect. 5.1.2.

In the remainder of this chapter we focus on the tasks performed by the *framework developer*, while the role of the *application developer* was specified in the SOLEIL framework (Section 4.3).

5.1.2 Architecture Refinement of Domain Components

The key role of the framework developer is therefore to design and implement semantics of domain-specific components. When considering domain components and the functionality they express, they impact three core architectural concepts: *Functional Component*, *Binding*, and *Domain Component*. Meaning that architecture and consequent implementation need to be refined in order to implement corresponding domain-specific semantics. However, we want to achieve this while still off-loading complexities from application developers. Therefore, to implement domain components, we introduce a process where each domain component is refined by corresponding platform concepts. These architectural changes are transparent for the *application developer*. We further refer to this phenomenon as *architectural refinement* of core-level concepts through the platform-level concepts.

The main objectives of the *architectural refinement process* are: *(i)* to fully-evolve the high-level concepts of domain components into artifacts that will represent these concerns at the architectural level; *(ii)* to consequently compose these artifacts with the functional concepts into a fully-evolved view of the system called *platform architecture* which is suitable to be transformed into a desired target implementation (e.g. RTSJ).

Therefore, we distinguish the following three types of the architectural refinement process:

- **Functional Component Refinement.** The target of the refinement process is a functional component. To implement the domain-specific concerns according to domain component that encapsulate the functional one, we are obliged to modify the container architecture of this component. Usually, we introduce a set of controllers and interceptors that will implement these domain-specific concerns.

- **Binding Refinement.** The target of the refinement is the binding between functional components. This corresponds to a case when a domain-specific concern introduces special requirements on this binding (e.g. logging, broadcast communication management). To guarantee them, we refine the binding with container interceptors presented either at client, server, or on both sides of the binding.

- **Domain Component Refinement.** The domain component itself is refined to be present also in the refined architecture. This case is different from the previous two where domain components are represented in the containers of functional components. Here we use platform concepts to represent the domain component as a composite component in the refined architecture - `Non-FunctionalComponent`. A specific composite component with a dedicated container architecture is deployed to enforce domain-specific concerns over its subcomponents.

In the following section we therefore define three architectural patterns that address the challenge of the architectural refinement and allow framework developers to properly develop implementations of domain-specific concepts.

Architectural Patterns

The key purpose of the architectural patterns is to allow framework developers to implement semantics of domain components and thus to refine the application architecture in a systematic and programmatic way. The patterns are designed to implement any type of a domain-specific service that can potentially be reflected by a container. They therefore define architecture invariants, design and composition rules for the platform-level. Fig. 5.4 presents three architectural patterns: `ChainComposite`, `ContainerComposite`, and `Non-FunctionalComposite`, and we further clarify them.

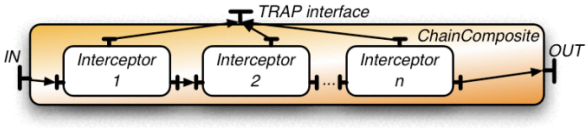
ChainComposite Pattern is defined as a composite component, the subcomponents of such a composite are *interceptors*. Within the `ChainComposite` pattern, the interceptor components are bounded via their incoming and outgoing interfaces in an acyclic list, as depicted in Fig. 5.5a. Here, the `IN` and `OUT` interface signatures of the interceptors are not necessarily identical; this allows developers to identify interceptors as *adaptors* of the intercepted execution flow. The interceptor itself could be a composite component allowing framework developer to implement complex intercepting mechanisms. The `ChainComposite` component at the platform level refines a binding specified at the functional level, thus the pattern is similar to the concept of *connector* [MDT03].

ContainerComposite Pattern, initially introduced in [SPDC06], is also specified as a composite component and reifies a container of a functional component. As defined in Fig. 5.2, it is composed of `ChainComposite` components and `Controller` components. The `ContainerComposite` pattern is applied on a primitive (see example in Fig. 5.5b) or composite functional component as follows:

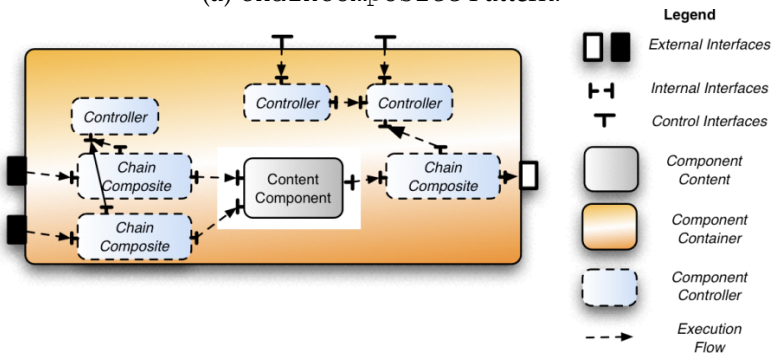
- A set of *Controller* components implementing various domain-specific services influencing the whole component (e.g. lifecycle management, reconfiguration management) is composed in the container. Moreover, the control interfaces are provided to allow an access to these services from outside of the component.

- For each interface of the functional component a `ChainComposite` pattern is used. `ChainComposite` components can be interconnected by `TRAP` interfaces with the controllers, thus allowing centralized management of strategies for interception mechanisms.

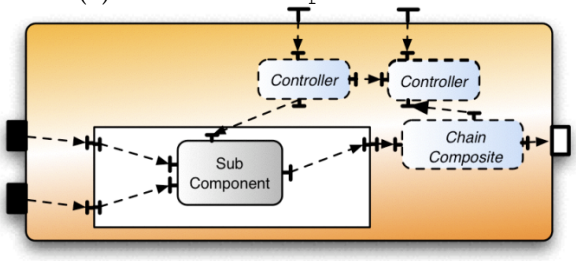
Non-functionalComposite Pattern is the final construct for manipulation of domain-specific concerns at the implementation layer. As illustrated in Fig. 5.4, the pattern is used to refine a domain component itself. By using this pattern, we deploy a composite component corresponding to the domain component defined at design time. Consequently, a container of such component contains controllers and interceptors, which superimpose domain-specific concerns over



(a) `ChainComposite` Pattern.



(b) `ContainerComposite` Pattern



(c) `Non-functionalComposite` Pattern

Figure 5.5: Architectural Patterns

the subcomponents of this component. Thus we manage domain-specific concerns of a group of functional components.

Therefore, the domain-specific services are not deployed into containers of functional components but we group them in a container of a special composite component present at runtime. We illustrate this idea in Fig. 5.5b.

Architectural Refinement Process

Once we specify the functional architecture containing domain components and also corresponding architectural patterns we employ the *architecture refinement process* – a process where the *core-level* architecture specified by the application developer is refined into an architecture where both functional architecture and runtime platform architecture are designed using the *platform-level* concepts. As a result of this process we obtain a *platform architecture* where both functional and domain-specific concerns are represented. The crucial point of the refinement process is therefore the propagation of domain-specific concerns into the architecture.

The important feature of the architecture refinement process is its variability and extensibility to allow employing different refinement strategies as well as support for new domain-specific components, validation and optimizations. All properties stated above are reflected in the implementation of the architecture refinement process called HULOTTE, described in Sect. 5.3.

5.2 Implementing SOLEIL with HULOTTE

In this section we apply the *architecture refinement process* to implement the basic concepts of the SOLEIL framework. Our motivation remains the same: While the functional code is implemented in the content part of the component, the real-time concerns are represented by the platform-level concepts in the containers.

For each component type of the SOLEIL framework we use the appropriate architectural patterns to design its container. Furthermore, we re-factor RTSJ-code and RTSJ-compliant patterns and we implement them in the controllers and interceptors of containers.

In this section we thus employ architectural patterns and formalizations in Alloy and show how we benefit from these technologies in the architectural refinement process. On the concrete examples of refining the SOLEIL framework concepts we show how we mitigate complexities of real-time Java development while still respecting the separation of concerns.

5.2.1 Active and Passive Components

Based on the definition of active component, Section 4.1.2, active component contains a thread of execution. Therefore, the goal of the container of the active component is to manage this thread. In Fig. 5.6 variant `a`, we present an architecture of such a container. The `ActivityController` is used to manage the thread of execution and instructs the `ActivityInterceptor` to enter the functional part of the component through the `Runnable` interface. Active Interceptors implements a run-to-completion execution model[4] for each incoming invocation from their server interfaces and are configured by the properties defined by the enclosing *ThreadDomain* component.

(a) Periodic Active Component

(b) Sporadic Active Component

Figure 5.6: Active Component Types

Note that although the `Runnable` interface is provided by the content part of the component, it is not exposed as an external interface. Its role is to provide an exact separation between functional execution of the component done in the content, and management of the execution context in the container.

Furthermore, we distinguish periodic component – Fig. 5.6 variant `a`, and sporadic component – Fig. 5.6 variant `b`. Whereas the management logic of periodic component is strictly given and implemented in the `ActivityController`, execution of the sporadic component is dependant upon external events or other activity components. Therefore, sporadic component can provide an interface, as illustrated in the figure. However, only asynchronous communication is provided, we discuss specifics of implementing asynchronous communication in Section 5.2.4.

In Fig. 5.7 we present a simple implementation of the `ActiveInterceptor` for a periodic component, as we can see, the `Runnable` interface is used here to provide a unique entry point for the component (as specified by the SOLEIL profile, in Section 4.3.3).

In Fig. 5.8 we present an implementation of the `ActiveInterceptor` for a sporadic component. In this case, an event is absorbed on the incoming interface `IService` of the interceptor, the active interceptor switches the execution context into the context of its own thread, line 9), and calls the `IService` interface provided by the implementation of the component. In the example,

[4]This execution model precludes preemption for *active components*.

```
1 public class ActiveInterceptor {
2   Runnable iActiveRunnable;
3   void execute() {
4     while(true) {
5       iActiveRunnable.start();
6       waitForNextPeriod();
7     }
8   }
9 }
```

Figure 5.7: `ActiveInterceptor` Implementation for Periodic Active Component

```
1 public class ActiveInterceptor
2           implements IService {
3   Service iService
4
5   void service(Data d) {
6     runnable.setData(d);
7     thread.start();
8   }
9   Thread thread = new Thread(..., runnable);
10
11  Runnable runnable = new Runnable {
12    Data d;
13    void run() {
14      iService.service(m);
15    }
16    void setData(Data data) {
17      d = data;
18    }
19  }
20 }
```

Figure 5.8: `ActiveInterceptor` Implementation for Sporadic Active Component

the interceptor's thread is a regular `Thread` for illustration, in a general case the interceptor can be generated with the type of the thread that is required.

Protected Component

In general, to implement passive components, no architectural patterns are required. However, we distinguish a specific type of a passive component - `Protected Component`. Such component provides concurrently more interfaces and guarantees that concurrent requests on these interfaces will be proceeded in accordance to a pre-defined synchronization logic.

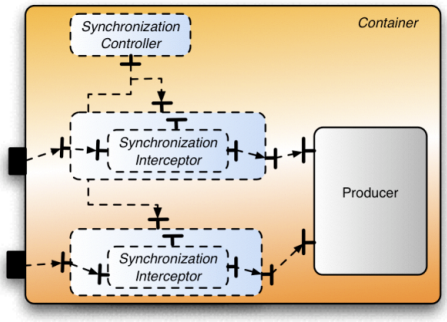


Figure 5.9: Container Architecture of Protected Component

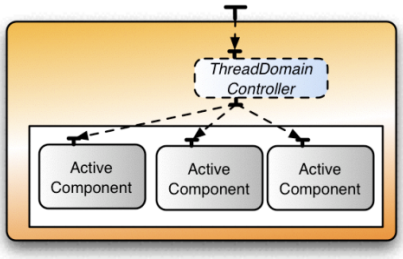


Figure 5.10: ThreadDomain Refinement

To meet this requirement, we introduce `SynchronizationController` and `SynchronizationInterceptor` depicted in Fig. 5.9. Their role is to implement the synchronization logic specified by the application developers. At runtime, each incoming call is intercepted and reported through the TRAP interface to the controller. Depending on the implemented logic, the call will be either authorized to continue with its execution in the content part or will be blocked until the controller decides otherwise. Therefore, many strategies and synchronization approaches can be implemented in the controller depending on users' requirements. However, when using this concept, no synchronization block statement can be used inside the content of the component since this could lead to a deadlock, as restricted in Section 4.3.3.

The introduction of the ProtectedComponent is motivated by the implementation of the Shared

Scope pattern that we describe in Section 5.2.5.

5.2.2 ThreadDomain Refinement

We refine the ThreadDomain component as a non-functional composite component (using the Non-functional Composite pattern) in order to manage running threads in the system during the runtime. We illustrate this idea in Fig. 5.10, the dedicated container contains a `ThreadDomain-Controller` that is connected through a control interface to every `ActiveController` of each active component that is a subcomponent of the ThreadDomain component.

Furthermore, instances of the ThreadDomain component take responsibility for bootstrapping of all threads in the system. We centralize the management of the bootstrapping procedures since this process is nontrivial in the RTSJ applications. The complexities of bootstrapping process bring the task of initializing threads with various types and execution contexts, moreover, the allocation context of each thread has to be also respected during its instantiation. However, thanks to the domain components in the architecture we can exactly identify all the properties needed to bootstrap a thread and implement them without cross-cutting the functional code

5.2.3 Immortal Memory

We refine the immortal memory as a composite component `ImmortalMemory`. As a consequence, we are able to refine the services provided by an immortal memory as control components which are accessible through provided interfaces. We illustrate the architecture of its component container in Fig. 5.11. The two services provided by the immortal memory are `ObjectPool` and `WaitFree Queues`, we further address their implementation in more details.

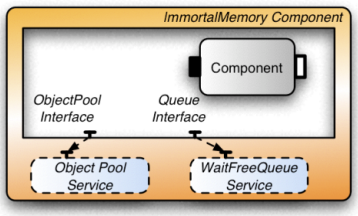


Figure 5.11: Immortal Memory Container

```
1  public interface ObjectPoolInterface {
2    Object get(int poolID, Object obj);
3    void put(int poolID, Object obj);
4    int createObjectPool(...);
5  }
6
7  public interface QueueInterface {
8    Object get();
9    void put(Object obj);
10   Queue createQueue(...)
11 }
```

Figure 5.12: Immortal Services API

Since the resources of the immortal memory are limited, the only solution how to effectively use them is to reuse objects. This approach is indeed present in many RTSJ-based systems[BCC+03]. To provide an easy-of-use and safe access to shared objects, we refine shared objects as a service `ObjectPool` provided by the `ImmortalMemory` component. The `ObjectPool` service implemented in the container is depicted in Fig. 5.11. A simple API for the ObjectPool service is defined in Fig. 5.12 line 1.

The second service provided by the ImmortalMemory component is *WaitFreeQueue*. The goal is to maintain and manage *WaitFree Queues*, which are then used to implement asynchronous communication between real-time threads of different types and priorities. The service provides an access to WaitFreeWriteQueue and WaitFreeReadQueue, as defined in Section 2.2.1. The `WaitFreeQueue` service is implemented in the container and is depicted in Fig. 5.11. API for the WaitFreeQueue service is defined in Fig. 5.12 line 7.

5.2.4 Cross-Thread Communication

When modeling a RTSJ compliant binding between active components from different ThreadDomain components, *queue communication* and *Scope Sharing* concepts can be used, as restricted by

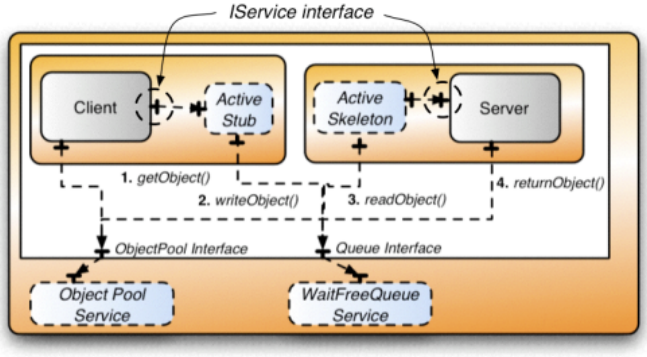


Figure 5.13: Cross-Thread Communication

```
fact WaitFreeQueues_ApplicationRules {
  all a,b : Active {
    if isAsynchronousBinding(a,b)  {
      if getThreadType(a) == getThreadType(b){
        if getPriority(a) > getPriority(b)
          setWriteFreeQueue(a,b)
        if getPriority(a) < getPriority(b)
          setReadFreeQueue(a,b)
        if getPriority(a) == getPriority(b)
          // can not happen here
      }
      if isBiggerType(a,b)
        setWriteFreeQueue(a,b)
      if isBiggerType(b,a)
        setReadFreeQueue(a,b)
    }
  }
}

// from a to b
pred isAsynchronousBinding
       (a,b: ActiveComponent) {
  one b: Binding | {
    b.client in a.externalInterfaces
    b.server in b.externalInterfaces
    b.type = Asynchronous
  }
}
```

```
fact DefiningPoolPatternApplication {
  all a,b : Active {
    if isAsynchronousBinding(a,b)  {
      if a,b in ImmortalMemory.^subComponents
        useObjectPoolPattern[a,b]
      if getMemoryArea(a) != getMemoryArea(b)
        useDeepCopy(a,b)
    }
  }
}

pred useDeepCopy[a,b : ActiveComponent] {
  one b: Binding | {
    b.client in a.externalInterfaces
    b.server in b.externalInterfaces
    b.type = Asynchronous
    DEEP_COPY in b.atributes
  }
}

// see Appendinx A for
//    the remaining predicates
```

Figure 5.14: WaitFreeQueue and ObjectPool Formalization

```
class ActiveStub implements IService {
   Queue queue = ImmortalMemory.getQueue(ID);
  public void service(Object data) {
    queue.write(data);
  }
}
```

```
class ActiveSkeleton implements IService {
  Queue queue = ImmortalMemory.getQueue(ID);
  boolean run = true;
  IService iService;
  public void service() {
    while (run) {
      queue.waitForData();
      Object data = queue.read();
      iService.service(data);
    }
  }
}
```

Figure 5.15: Active Interceptors Implementations

RTSJ and specified in Fig 4.7. In this section we will focus on queue communication and we will address the scope-sharing concept in the next section.

In Fig. 5.13 we illustrate the container architectures of two active components having non-equal types and priorities. We have already defined that such communication must by *asynchronous*, in Fig. 4.7. Furthermore, in Fig. 5.14 we formalize how to implement this communication according to the RTSJ rules defined in Section 2.2.1. Depending on the properties of each active component we define if the *WaitFreeWriteQueue* or *WaitFreeReadQueue* will be used, line 1. Consequently, as the queues are going to pass objects between the active components, we have to formalize this communication since it could violate RTSJ rules in case that we are crossing memory scopes, line 29. Therefore, when communicating withing the immortal memory, the object pool should be used, line 33, otherwise we recommend the deep-copy pattern, line 35.

To use the communication queues properly and to shield developers of functional components from dealing with the *WaitFreeQueues*, we hide the appropriate implementation in the containers of each active component. We deploy special stubs and skeletons implemented in container interceptors. From these interceptors we connect to the *WaitFreeQueue* service to obtain references to each side of the communication queue. We demonstrate implementation of these interceptors in Fig. 5.15.

5.2.5 Cross-Scope Communication

In this section we implement the cross-scope communication rules introduced in Section 4.2.4. Our simple goal is to leave the functional code unmodified and to implement the specific cross-scope patterns in the interceptors. These interceptors are deployed on the bindings that cross different memory scopes, their implementation depends on the design procedure choosing one of many RTSJ memory patterns [CS04, BN03, PFHV04].

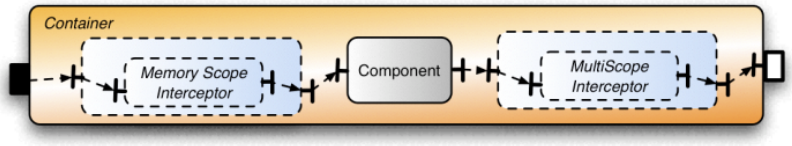


Figure 5.16: Memory Scope Component - Interceptors

```
1  class InterceptorRunnable implements Runnable{
2    Data input, result;
3    IService iService;
4    void setData(Data data) {
5      input = data.deepCopy();
6    }
7
8    void run() {
9      result = iService.service(input);
10   }
11
12   Data getData() {
13     return result;
14   }
15 }
16
17 class MemoryScopeInterceptor
18         implements IService  {
19   ScopeMemory scope;
20   InterceptorRunnable intRunnable;
21   public Result service(Data input) {
22     intRunnable.setData(input);
23     scope.enter(intRunnable);
24     return intRunnable.getResult();
25   }
26   ....
27 }
```

Figure 5.17: Memory Scope Interceptor Implementation

```
1  class MultiObject implements Runnable{
2    Data input;
3    IService iService;
4    void run() {
5      input = input.deepCopy();
6      iService.service(input);
7    }
8  }
9
10 class MultiScopeInterceptor implements
11     IService {
12   ScopeMemory parentScope;
13   Data input;
14
15   MultiObject multiObject;
16
17   public MultiScopeInterceptor () {
18     ...
19   }
20
21   public void service(Data input) {
22     parentScope.execInArea(
23         multiObject.setData(input));
24   }
25 }
```

Figure 5.18: MultiScope Interceptor Implementation

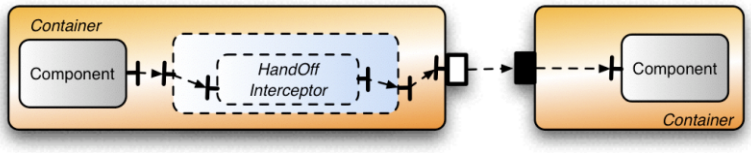


Figure 5.19: HandOff Pattern Implementation Schema

```
1  class HandOff implements Runnable {
2    IService iService;
3    Data input;
4    MemoryScope mem_b;
5    void run() {
6      iService.service(input)
7    }
8  }
9
10 class Bridge implements Runnable {
11   MemoryScope mem_b;
12   HandOff handOff_;
13   void run() {
14     memb_b.enter(handOff_)
15   }
16 }
```

```
17 class HandoffInterceptor implements
18     IService {
19   ScopeMemory parentScope;
20   Bridge bridge;
21   Handoff handoff;
22   ServiceInterface iService;
23
24   public HandOffInterceptor () {
25     ...
26   }
27
28   public void service(Data input) {
29     handOff.setData(input);
30     parentScope.executeInArea(bridge);
31   }
32 }
```

Figure 5.20: HandOff Interceptor Implementation

The *cross-scope pattern* is the basic pattern for a cross-scope communication. This pattern targets cases when a communication from a parent to a child scope is needed. To implement this pattern, we introduce `MemoryScopeInterceptor` (see Fig. 5.16) that is deployed on every binding crossing different memory areas and is contained at the server side. A code snippet from Fig. 5.17 shows implementation of the interceptor – line 17, that manages entering and leaving of the memory scope and uses a simple deep-copy pattern for returning results from the scope, line 13.

The goal of the *multiscope pattern* is to guarantee a transition of the data from a child scope to a parent scope, therefore a correct switch of the execution context between two scopes is the crucial task. In Fig. 5.18 line 10 we therefore show implementation of the pattern in our approach. The implementation shows that flow of the intercepted call is first changed to a corresponding parenting scope - line 22, and then from a proper scope we call the functional `iService.service` method - line 6.

The *handoff pattern* – a more sophisticated solution of cross-scope communication handling is demonstrated by the `Handoff Interceptor` from Fig. 5.19. We employ the *handOff pattern* [PFHV04] to store data in a scope while still operating in the original one. Fig. 5.20 shows a code snippet of the `HandOffInterceptor` implementation. A direct communication between two sibling nodes `a` and `b` is not possible because of the *single parent rule*. The hand-off algorithm therefore switches first from the scope `a` into the parent scope - line 30, and from this scope enters the scope `b` - executing in class `Bridge`, line 14. Finally, having the allocation context of the scope `b`, we performe a deep copy of the data and run the requested method in the class `Handoff`, line 6.

Scope Sharing

Scope sharing can be applied only in the RTSJ compliant cases. A shared scope must not violate the *single parent rule*, therefore the communicating components need to be allocated in the same scoped memory. Thanks to the RTSJ concepts in the component model, this condition can be easily verified from the architectural model of the system.

Such sharing is illustrated in Fig. 4.11. Here, we are able to verify at design-time whether this sharing is compliant with RTSJ. Furthermore, we show a container design of a component in the

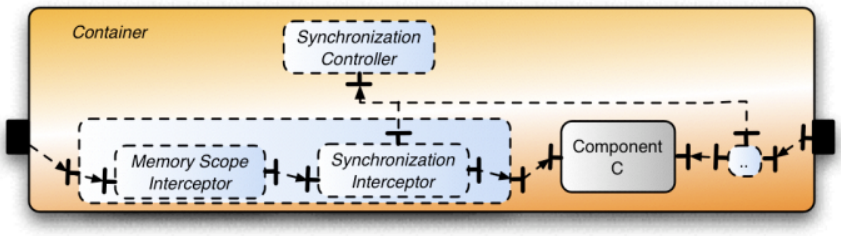


Figure 5.21: Shared Scope Component

shared scope, Fig. 5.21. To implement it, we use the cross-scope pattern designed previously and also we need to use the concept of `protected component`.

5.2.6 Fractal Control Layer

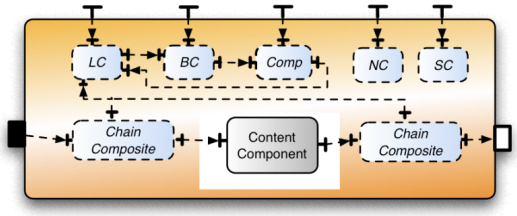


Figure 5.22: Fractal Control Layer

Apart from the controllers and interceptors presented above, we introduce a basic set of controllers that is provided for every functional component in the system. These components are inspired by the Fractal component model [BCL+06]: five controllers are provided, for managing the lifecycle - `LifecycleController` (LC), the bindings - `BindingController` (BC), the component name - `NameController` (NC), the super components - `SuperController` (SC) and the component - `ComponentController` (Comp).

5.3 HULOTTE Framework Implementation

This section is divided into two parts. First we describe implementation of the HULOTTE framework [Ale09, LMP+09, NL09] — an extensible tool-set that we have developed to implement the architecture refinement process. However, rather than to implement the whole process in a single transformation step that can be error-prone and hard to extend, we employ a step-wise refinement process [BSR03] in order to refine the high-level concepts in our architecture gradually in several stages. This technology allows framework developers to easily modify and extend this process with new domain-component definitions and semantics. Consequently, we employ methods of generative programming to compose functional code implemented by the application developer with the runtime platform implementation.

Second, in Section 5.3.6, we discuss the framework in a more general perspective envisaging its application as a metaframework used to develop domain-specific component frameworks.

5.3.1 HULOTTE Architecture

To develop the framework, we have applied the technology for development of extensible toolsets introduced in [LOQS07]. HULOTTE is thus developed purely using CBSE paradigm allowing framework developers seamless extensions towards different refinement strategies. The HULOTTE framework, depicted in Fig. 5.23, consists of three main units — *front-end* processing a description of a functional architecture stored in ADL, *middle-end* responsible for a step-by-step architecture refinement, and *backend* which serves as a target domain specific implementation generator.

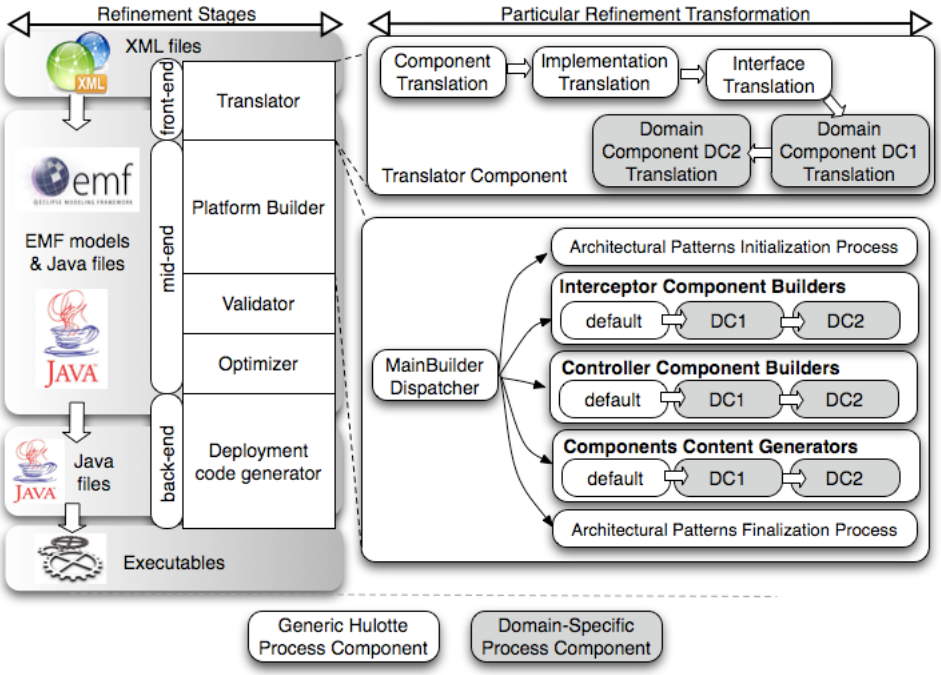


Figure 5.23: Overview of the Internal HULOTTE Implementation Structure

The motivation for decomposition of the process into three independent units is to separate responsibilities and concerns between the transformation steps. The front-end allows us to process architectures represented by different notations (e.g. FRACTAL-ADL [LOQS07], UML, ACME [GMW97]) and to transform them into an independent internal representation. Consequently, the middle-end, executing the architecture refinement process, is independent from the architecture description format. Finally, the back-end permits generation of different types of target implementations according to deployment requirements (in a more general sense this approach is not limited to RTSJ, but can be applied for e.g. C for embedded devices or Java for enterprise applications). In the remainder of this section we highlight interesting issues of each part of the HULOTTE framework.

5.3.2 Front-end

Front-end implements the *translation layer* that proceeds an architecture description — in our case given in an extended FRACTAL-ADL (we illustrate such architecture in Appendix C), and transforms it into an internal EMF-model [BSE+04] based representation.

The translation process gradually proceeds ADL artifacts (component, interface, domain component, binding) and for each applies a dedicated translation component responsible for extracting the information and building an appropriate representation in the internal model. The trans-

lation process can be extended by appending a new translator component. The new translator typically reflects a domain-specific extension of ADL (e.g. in Sect. 4.2.5).

5.3.3 Middle-end

Middle-end is the central part of the HULOTTE framework and implements the refinement process. Its task is to process the architecture description in the form of the EMF model produced by the front-end, apply defined architecture refinements — creating, connecting, or merging model elements according to employed transformations. Internally, the middle-end is composed of three processing units —*PlatformBuilder*, *Validator*, and *Optimizator*.

PlatformBuilder is responsible for the model refinement and consists of a chain of component builders (for implementations of interceptors, controllers, and components), illustrated on the right side of the Fig. 5.23. where each chain participates in the refinement process. From the builders the runtime platform components are instantiated either by loading definitions from an off-the-shelf component library or programmatically, via the high-level API provided by the framework. The selection and execution order of chains is controlled by *MainBuilder Dispatcher* that recursively explores the platform architecture and applies appropriate builder chains. Moreover, refining the internal structure as a chain of ComponentBuilders encourages extensibility of the whole process, since a new domain-specific builder can be easily introduced. As we can see in Fig. 5.23, except the default builders we can define e.g. `DC1`, `DC2` builders that correspond to specific domain components.

Validator verifies that resulting platform architectures are in conformance to the architectural constraints and invariants of domain components. The task is not only to verify whether the architectural patterns were applied correctly but also to assert that domain components were specified with respect to their constraints (e.g. to arbitrarily apply two different domain components over the same functional component is sometimes not meaningful, see the Limitations of the Approach in Section 6.4).

Optimizator introduces optimization heuristics in order to mitigate the common overhead of component-based applications. The heuristics focus on reducing interceptions in inter-component communication which usually causes performance overhead, and on merging architecture elements in order to decrease memory footprint. Moreover, since a complete architecture of the system is available at this stage, additional architecture optimizations, identified in [LP08], can be introduced while still being independent from the target domain.

5.3.4 Back-end

Back-end part of the framework is also highly configurable in order to reflect current target domain and chosen implementation language. In the case of our implementation of HULOTTE, the back-end is a collection of Java code generators generating Java classes from particular model elements. As well as the rest of our approach, the back-end is also extensible to employ specific code optimizations.

Taking into account all the constraints for real-time and embedded systems, we can conclude that there are several reasons to perform optimizations at development time rather than runtime [CL02]: This allows composition tools to e.g. generate a monolithic firmware for the device from the component-based design and by this achieve better performance and better predictability of the system behavior. It also enables *global optimizations*: e.g., in a static component composition known at design time, connections between components could be translated into direct function calls instead of using dynamic event notifications. Finally, verification and prediction of system requirements can be done statically from the given component properties.

5.3.5 Soleil - Runtime Platform Instantiation

We further apply the HULOTTE approach to the SOLEIL framework in order to generate runtime platform corresponding to the real-time architecture specified by the designer and refined by the

architecture refinement process. The goal of this back-end process is to generate Java source code including container source code, a framework glue code, and a bootstrapping code.

Moreover, our tool offers different generation modes corresponding to various levels of functionality, optimization, and code compactness:

1. **`SOLEIL`** This default mode generates a full componentization of the runtime platform. The RTSJ interceptors and the reconfigurability management code are therefore implemented as controllers and interceptors, within the *containers*. The structure of the latter is also reified at runtime, as well as the `ThreadDomain` and `MemoryArea` composite components. This generation mode provides a complete introspection and reconfiguration capabilities of the component framework at the functional and at the membrane level.

2. **`MERGE-ALL`** In this generation mode the implementation of functional component code and its associated membrane are merged into a single Java class. Therefore, it generates one class per each functional component defined by the developer. Since the number of Java objects in the resulting infrastructure is considerably decreased, this mode achieves also memory footprint reduction. In comparison with the `SOLEIL` mode, it corresponds to a first optimization level where several indirections introduced by the container architecture are replaced by direct method calls. As component container structures are not preserved at the runtime, the `MERGE-ALL` mode does not provide reconfiguration capabilities of the container level. However, these capabilities are still provided at the functional level. The source-to-source optimizations performed by the generation process are based on Spoon [Paw06], a Java program processor, which allows fine-grained source code transformations.

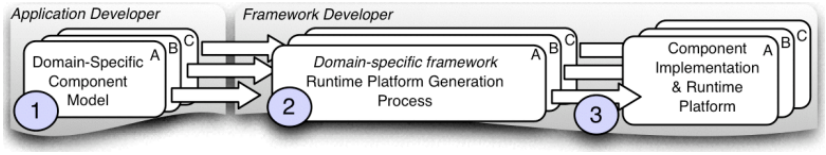
3. **`ULTRA-MERGE`** The most optimized mode achieves that the whole resulting source code fits into one unique class. Moreover, the generated code does not preserve the reconfiguration capabilities anymore. The resulting infrastructure is therefore purely static. It exclusively embeds the functional implementations merged to the code which takes into account the component activations, the asynchronous communications, and the RTSJ dedicated code.

5.3.6 HULOTTE as a Meta-Framework

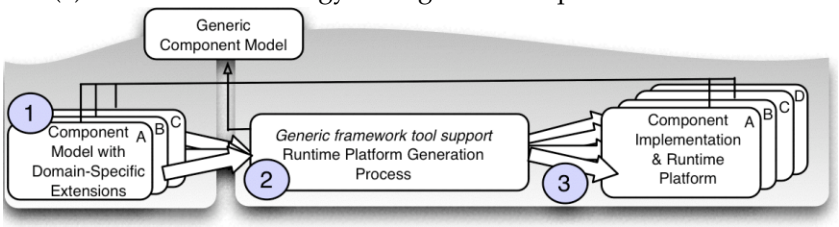
In this section, we discuss application of the framework in a more general perspective envisaging the framework as a meta-framework used to develop Domain-Specific Component Frameworks.

Domain specific component framework (DSCF) is composed of a domain-specific component model and the tool support which permit assembling, deploying and executing demanded applications [BHM09]. The main goal is to allow developers to address domain-specific challenges by using appropriate abstractions available already at the component-model level. To achieve separation of concerns, domain-specific services required by the target application domain (such as dedicated memory ares, tasks parameters, security, distribution support or real-time constraints), in the literature [DEM02] referred to also as *non-functional requirements*, are usually deployed in the *runtime platform* composed of a set of custom made containers (used e.g. in [Mor06, Mic, JHA+05]). Today, a plethora of DSCFs emerges, addressing a wide scale of challenges — embedded [vOvdLKM00] or real-time constraints [PLMS08, HACT04], dynamic adaptability [FSSC08, GER08], distribution support [SVB+08, MPL+08], and many others.

One of the main benefits expected in using the component paradigm is reuse [Cle02]. However, it has been argued [DHT01] that the vast, and increasing number of proposals to address these domain-specific requirements does not encourage reuse, while sharing common concepts and tooling support. Although the current trend emphasizes *generative programming methods* [CE00] as cornerstone of software development, generative methods are usually tailored to specific domains and applied in a costly, ad-hoc fashion. This prevents from any reuse or amelioration of solutions to a framework construction. We however believe that DSCFs share the same concepts and patterns to their construction and application.



(a) Current Methodology: Using Domain-Specific Frameworks



(b) Our Proposal: Using a Generic Framework

Figure 5.24: Development Methodologies of Domain-specific Component Application

Based on this observation, we envisage a new contribution for the proposed HULOTTE framework – a prototype framework for specification and implementation of arbitrary domain-specific concerns in a unified way, which can be easily extended towards different application domains.

Domain-Specific Component Frameworks and their Application

Typically, DSCF defines the relevant architectural concepts, called *domain-specific concepts* , according to the requirements of the targeted application domain (e.g. to address the distribution support or real-time constraints). A recognized methodology of developing DSCF [CL02] is composed of several steps as it is illustrated on Fig. 5.24a. In this case, each component model (step 1) is used to develop functional concerns of the application — *functional components*. Typically, functional components encapsulate a business logic of an application.

Afterwards, the framework tool-support is employed to create a *runtime platform*, in Fig. 5.24a step 2. The runtime platform is composed of a set of *containers* [Mor06] that encapsulate functional components, and its goal is to relieve the developer from dealing with *domain-specific requirements* and to implement the execution support. Current trend in developing the runtime platform emphasizes a generative programming approach. Here, different optimizations should be employed to mitigate notoriously known problem of CBSE system — performance overhead (caused e.g. by intercomponent communication). Finally, functional components and the runtime platform are assembled together to form the resulting application, Fig. 5.24a step 3.

Similarly as in the HULOTTE framework, we distinguish two types of development roles involved in this process — *application developer* and *framework developer*. Application developer is responsible for development of functional components and specification of domain-specific requirements — in Fig. 5.24a step 1. The role of the framework developer is to design and implement the runtime platform generation process, and the domain-specific requirements defined by the application developer — in Fig. 5.24a step 2 and 3.

Developing Domain-Specific Component Frameworks with HULOTTE

Considering the presented process, we can notice that for each domain, a different process is used. However, the steps 2 and 3 share many similar concepts. Moreover, they are usually implemented in an ad-hoc manner without any reuse. We therefore propose a new development process presented in Fig. 5.24b.

As the cornerstone we use a generic component model that is easily extendable towards different application domains, in Fig. 5.24b step 1. Consequently, since all domain specific models

share the same concept, a unified approach to runtime platform generation can be employed in steps 2 and 3. These steps correspond to the Front-, Middle-, and Back-end parts of the Hulotte framework implementation presented in Section 5.3.1.

Therefore, the Hulotte framework can be also comprehended as a framework, in the literature also refereed as *meta-framework* [BHM09], composed of high-level tools, methods, and patterns allowing *framework developers* to generate runtime platforms in a generic way according to concerns captured by Domain Components. Within our approach, the platform is built using component assemblies and is based on our generic component model. Moreover, since we are able to reason about the whole system (functional and platform concerns) using common concepts (components, assemblies), various architecture optimizations independent from the target domains can be introduced, which contributes to better performance of resulting applications.

5.4 Motivation Example Revisited

To illustrate the ideas presented in this chapter, we revisit the SweetFactory motivation scenario introduced in Section 2.2.3. In Chapter 4 we have discussed this scenario from the perspective of the application developer, showing how to develop application architecture and implement functional components. In this section we will continue with this example by addressing the responsibilities of the framework developer in the HULOTTE framework - implementing semantics of domain components and automatic generation of the framework glue code.

Therefore, we apply the architecture refinement process to refine the artifacts of the architecture with the platform level concepts. Based on the architecture defined in Section 4.4, we will focus on functional components and refine architectures of their containers according to domain components encapsulating them, using the domain component semantics defined in Section 5.2.

`ProductionLine` and `MonitoringSystem` Components

As the first step we will refine components `ProductionLine` and `MonitoringSystem`, they are both active components communicating together and the immortal memory is their allocation context. The complete architecture is illustrated in Fig. 5.25. In the picture we can see active components `ProductionLine` and `MonitoringSystem` encapsulated by their containers, these compositions are then deployed in the non-functional components `NHRT1`, `NHRT2`, instances of a `ThreadDomain` entity representing a `NoHeapRealtimeThread`s with different parameters.

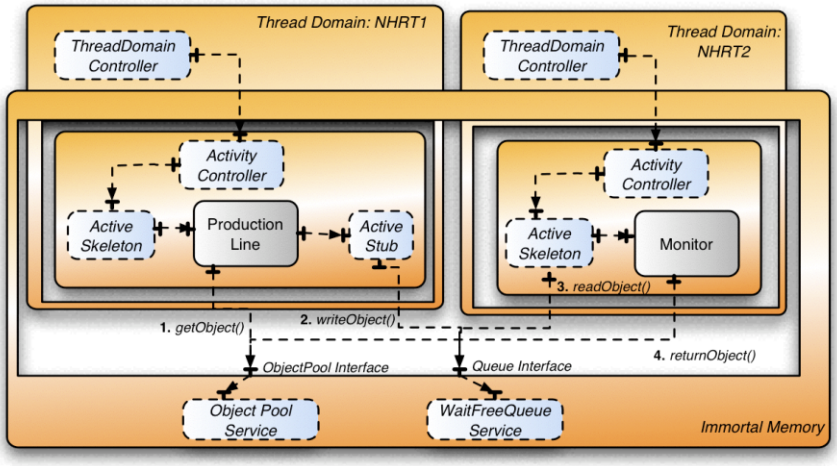


Figure 5.25: `ProductionLine` and `Monitor` Architecture Refinement

Inside the `ProductionLine` and `MonitoringSystem` containers, various controllers and interceptors are present. *ActivityController* and *ActivityInterceptor* implement execution model of an active component. Both of them represent non-functional concepts specific to the `MonitoringSystem` component. The `NHRT1` and `NHRT2` components contain both a `ThreadDomain` controller that implements logic for management of `NoHeapRealtimeThread` subcomponents. Furthermore, since the allocation context of these components is the immortal memory, both of these components are deployed also into an instance of the `ImmortalMemory` component - a non-functional component providing various services.

Consequently, we need to refine an asynchronous communication between these two components. According to the rules defined in Fig. 5.14, the asynchronous communication will be implemented using a *WaitFreeQueue* provided by the `ImmortalMemory` component. However, the components are operating in immortal memory and therefore there is no need for deep-copying the objects, we follow the rule from Fig. 5.14 line 29 defining when to use object pools to effectively recycle the memory. This situation is illustrated in Fig. 5.25 where the *ActiveStub* and *ActiveSkeleton* are both connected to the *WaitFreeQueue* and *ObjectPool* services, and implemented as shown in Fig. 5.15.

Finally, the `NHRT1` and `NHRT2` components contain both a `ThreadDomain` controller that implements logic for management of `NoHeapRealtimeThread` subcomponents.

`Console` and `AuditLog` Components

As the second step we will focus on architectural refinement of the remaining two components - `Console` and `AuditLog`. Since they communicate extensively with the `MonitoringSystem` component, we also need to design this communication.

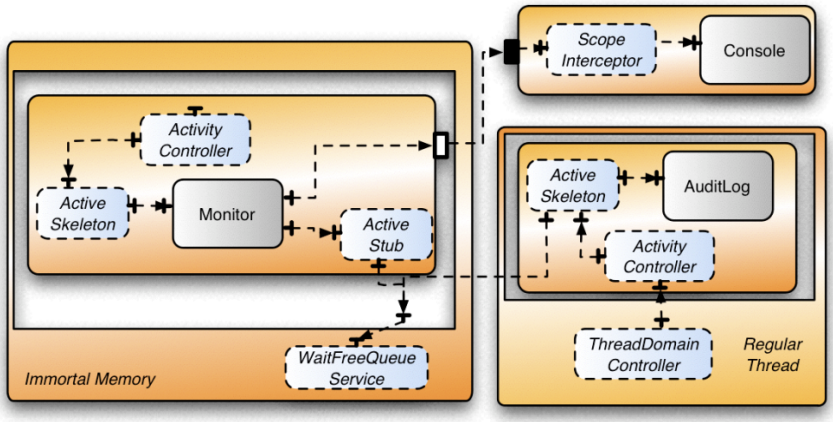


Figure 5.26: `AuditLog` and `Console` Architecture Refinement

Considering the `Console` component, its refinement is simple since it is a passive component residing in a scoped memory, according to defined rules, we will refine its structure by deploying a `ScopeInterceptor`, described in Section 5.2.5, into its container.

The `AuditLog` component is an active component executed by a regular thread residing on a heap, however, it communicates with the `MonitoringSystem`. Therefore, applying the rules from Figure 5.14 we derive that the asynchronous communication will use *WriteFreeQueue* with the deep-copy pattern. Therefore, the resulting architecture can be seen in Fig. 5.26.

5.5 Summary

In this chapter we propose our approach to implementation of domain components and we further employ it in the HULOTTE framework - a toolset for automatic instantiation of runtime execution platforms (**G2.2**).

First, we extend the SOLEIL component metamodel with the *platform-level concepts* that provide a set of entities used to refine architectures of domain components. Consequently, we identified architectural patterns that facilitate refinement of these components. Second, we apply the architectural patterns and the platform-level concepts in order to design and implement RTSJ-specific domain components proposed by the SOLEIL framework.

Furthermore, we develop HULOTTE— a generic framework that uses generative programming methods to instantiate RTSJ applications together with their runtime platform. The whole approach is highly transparent since it is fully based on component-oriented principles, which allows developers to easily extend the framework with additional domain components.

Finally, we have applied the concepts introduced by this chapter on our motivation scenario.

In the following chapter, we use the SOLEIL and HULOTTE frameworks in three different case studies, showing the benefits of our approach and evaluating it from different perspectives, addressing the goal **G3**.

Part III

Validation

Chapter 6

Case Studies

Contents

IN THIS CHAPTER we apply the SOLEIL and HULOTTE frameworks in several case studies spanning different domains and challenges. The case studies[5] will serve both as examples and validation of the proposed concepts.

The first case study, *SweetFactory*, is based on the motivation scenario that we have introduced in Section 2.2.3. The goal of the case study is to validate our approach from the quantitative point of view. First, we summarize the implementation of the SweetFactory example as it was conducted through the course of this dissertation, and consequently, we evaluate performance issues of our framework from various perspectives. Finally, we validate our framework based on the RTSJ metrics [BV07] for code generation. Therefore, this case study confronts our approach with goals **G3.1** and **G3.2**.

The second case study, *Real-time Collision Detector*, presents a large real-time Java application that is well known as a validation case study of many RTSJ projects. The goal is to demonstrate application of the SOLEIL framework in a real-life scenario and thus confront our approach with goals **G3.3** and **G3.4**.

The third and final case study - *Distributed and Ambient Programming in* HULOTTE , evaluates the potential of the SOLEIL and HULOTTE frameworks to be extended towards different domains.

[5]The source codes of the case studies are available at ttp://adam.lille.inria.fr/soleil/

Therefore, we describe challenges of distributed Real-time Java programming and Ambient Programming, consequently we propose a new domain components and show their implementation. Finally, we demonstrate contributions of our approach on examples.

At the end of this chapter, we discuss specific shortcomings of our solution and compare our framework with the frameworks presented in Section 3.2.

Contributions

The major contribution of this chapter is the evaluation of the framework from several different perspectives:

- **Performance.** As the first contribution, we evaluate performance of our solution. We measure overhead of our framework comparing to manually implemented object-oriented application and consequently, we measure the impact of optimization heuristics that we have introduced in the framework.
- **Generic-Programming Evaluation.** Since our framework extensively employs code generation techniques, we evaluate that the generated code and manually implemented functional components form programs that can be easily maintained and are transparent for application developers. The evaluation is conducted based on the metrics published in [BV07].
- **Real-time Collision Detector.** We evaluate our approach on a large-scale application that is widely accepted in the Real-time Java community. In the case study we show how our approach mitigates RTSJ development complexities by bringing benefits even in large applications.
- **Extendability of the Framework.** Finally, we evaluate the ability of the HULOTTE framework to extend towards different application domains. We present challenges of two domains - distributed real-time programming and ambient programming, and show how the HULOTTE framework can be extended to meet their requirements.

Structure of the Chapter

This chapter is divided into three parts, each dedicated to a case study. Each of the case studies starts by giving its overview and highlights the key challenges. Afterwards, we apply our framework and show its benefits. Finally the case study closes with an evaluation. Therefore, Section 6.1 presents the SweetFactory case study that is focused on illustration of general framework contributions, consequently evaluates both performance and memory overhead introduced by the SOLEIL framework and finally evaluates the framework from the RTSJ-code generation perspective.

Section 6.2 describes the Real-time Collision Detector case study. First, we introduce the case study and consequently show the state-of-the-art approaches used to its implementation. Afterwards, we present solution in the SOLEIL framework and finally compare the benefits of concerned approaches. The case study is therefore focused on implementation of a large system, evaluating benefits of our approach from the software engineer perspective.

Section 6.3 is dedicated to the final case study - Distributed and Ambient Programming in HULOTTE. First, it presents Distributed- and Ambient- Programming and their challenges. Consequently, we define according domain components and show how they can be implemented in the HULOTTE framework. Finally, we evaluate this approach on simple examples to demonstrate the ability of HULOTTE to extends towards different domains.

After the case studies, we discuss the limitations or the approach identified in the case studies in Section 6.4. In Section 6.5 we compare the SOLEIL framework with the state-of-the-art RTSJ frameworks. The chapter is summarized in Section 6.6.

6.1 Sweet Factory

The Sweet Factory application is based on the motivation scenario presented in Section 2.2.3. The goal of this case study is to illustrate development of RT Java applications in the SOLEIL framework, measure the performance of applications implemented in our framework, and finally validate the applications developed in our framework from an code generation perspective (using metrics from [BV07]).

6.1.1 Description

The SweetFactory case study [GHMS07] consists of a production line that periodically generates measurements, and of a monitoring system that evaluates them. The goal is to notify a worker console whenever abnormal values of measurements appear. The last part of the system is an auditing log where all the measurements are stored for auditing purposes. The production line operates in 10ms intervals, the system must be designed to operate under hard real-time conditions. However, the console and logging part does not superimpose any additional real-time constraints. Therefore, system is thus composed of parts that must meet hard real-time deadlines (the production line and monitoring system) and parts that are non real-time (console). Such a case study represents the best candidate for applying real-time Java technology, since it embraces real-time and non-real-time demands in one application.

Despite relative simplicity of the case study, it brings many challenges and contains the most significant challenges in real-time Java programming: hard real-time threads operating with different priorities (production line and monitoring system), a passive service suitable for scoped memory application (logging), non-real-time part (console), and both synchronous and asynchronous ways of communication. This case study was also used as a running example in this dissertation. We have described designing of the application in Section 4.4. Consequently, implementation of the example was discussed in Section 5.4.

6.1.2 Performance Evaluation

The goal of this benchmark is to show that our framework does not introduce any non-determinism and to measure the performance and memory overhead of the framework. As one of the means of evaluation, we compare differently optimized applications developed in our framework with a manually written object-oriented application.

The performance of the component-based software systems is a well known issue [LP08]. The platform infrastructure presented in Chapter 5 is based on the container composition paradigm. This introduces several indirections in the functional execution flow initially described at the design time by the developer. These indirections allow developers to superimpose domain-specific functionalities around functional components, however, they also introduce additional overhead caused by the traversal of the incoming invocation through the container structure. Our primary goal when evaluating the framework is thus to measure its overhead in terms of execution time and memory footprint. Moreover, since the RTSJ specifications have been defined to reduce the unpredictability introduced by the dynamic memory management within the Java Virtual Machine, we also use the jitter as a measure of the system's predictability.

In order to employ a standardized and well known form of performance evaluation, the benchmark is inspired by the evaluation case study presented in the Compadres project [HGCK07].

Benchmark Scenario

We measure the execution time of a complete iteration starting from the `ProductionLine` component. Its execution behavior consists of a production of a state message that is sent to the `MonitoringSystem` component using an asynchronous communication. The latter is a *sporadic active component* that is triggered by an arrival notification of the message from its incoming server interface. The scenario of this transaction finally ends after invocation of a synchronous method

provided by the passive `Console` component and an asynchronous message transmission to the active `AuditLog` component.

Evaluation Platform

The testing environment consists of a Pentium 4 mono-processor (512KB Cache) at 2.66 GHz with 1GB of SDRAM, with the Sun 2.1 Real-Time Java Virtual Machine (a J2SE 5.0 platform compliant with RTSJ), and running the Linux 2.6.24 kernel patched by RT-PREEMPT. The latter converts the kernel into a fully preemptive one with high resolution clock support, which brings hard realtime capabilities, see [MG07].

Measurement Collection

The measurements are based on *steady state observations* - in order to eliminate the transitory effects of cold starts we collect measurements after the system has started and renders a steady execution. For each test, we perform 10 000 observations from which we compute performance results.

Our first goal is to show that the framework does not introduce any non-determinism into the developed systems, we therefore evaluate a "worst-case" execution time and an average jitter. Afterwards, we evaluate the overhead of the framework by performance comparison between an application developed in the framework (impacting the generated code) and an implementation developed manually through object-oriented approach. Therefore, in the results presented below, we compare four different implementations of the evaluation scenario. First, denoted as `OO`, is the manually developed object-oriented application. Then, denoted as `SOLEIL`, `MERGE_ALL`, and `ULTRA_MERGE` are applications developed in our framework constructed with different levels of optimization heuristics. We refer the reader to Section 5.3.5 for detail description of the optimization levels.

Results Discussion

The results of the benchmarks are presented in Fig. 6.1. It presents the execution time distribution of the 10,000 observations processed. Table. 6.1 sums up these results and gives the *median* of observed times - the range of observations between the minimum and the maximum execution time observed, as well as the corresponding *jitter*. Fig. 6.2 presents the memory footprints observed at runtime.

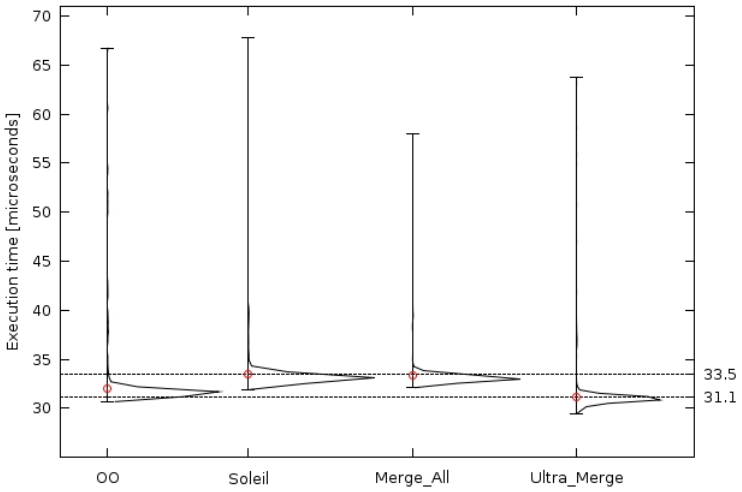


Figure 6.1: Benchmark Results: Execution Time Distribution

In Fig. 6.1 we show performance results of different variants of SweetFactory application - OO, SOLEIL, MERGE_ALL and ULTRA_MERGE respectively. For each implementation we show a performance graph showing distribution of the results on a time scale, where each point of the graph represents the number of measured iterations that finished in a given time. Consequently, the peak of each graph represents the most frequent time length of the iteration.

	Median (μs)	Jitter (μs)
OO	31,9	0,457
SOLEIL	33,5	0,453
MERGE_ALL	33,3	0,387
ULTRA_MERGE	31,1	0,384

Table 6.1: Execution Time Median and Jitter

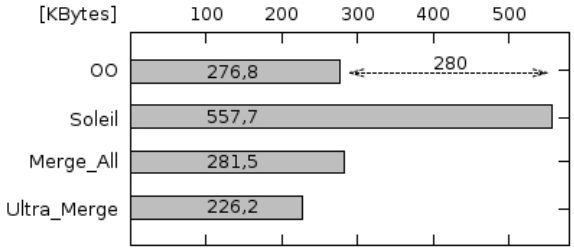


Table 6.2: Memory Footprint

Non-Determinism As the first result, we can see that our approach does not introduce any non-determinism in comparison to the object-oriented one, as the execution time curves of OO and SOLEIL are similar. Moreover, the jitter is very subtle for all tests. This is caused by the execution platform which ensures that real-time threads are not preempted by GC, and provides a low latency support for full-preemption mechanisms within the kernel.

Performance Time The median execution time for the SOLEIL test is 4.7% higher than for the OO one. This corresponds to the overhead induced by our approach based on component-oriented containers. However, the performance of the ULTRA_MERGE is comparable to the manually implemented OO - it is even slightly better since ULTRA_MERGE implementation is more compact since it removes indirections between objects.

Memory Footprint Considering the memory footprint, SOLEIL consumes 50% (280KB) more memory than OO. The price paid for generated containers providing RTSJ interception mechanism. MERGE_ALL, a test introducing the first level of optimizations, gives a more precise idea of the injected code which provides these non-functional capabilities at runtime: 4.7KB. The memory overhead purely corresponds to the algorithms and data structures used by our component framework. Finally, the ULTRA_MERGE is the most lightweight - even in comparison to OO, here, the whole applicative code and RTSJ interceptors are instantiated within a unique Java object.

6.1.3 RTSJ Code Generation Perspective

The work introduced in [BV07] investigate fitness criteria of RTSJ in model-driven engineering process that includes automated code generation. The authors identify a basic set of requirements on code generation process. We further confront our approach and the generation process that we integrate with this set of code generation requirements.

Therefore, we consult the requirements, give their descriptions and than we discuss how they are met in the HULOTTE framework. The key generation requirements can be divided into two groups.

- **The Code Generator Validation.** In order to be able to place trust on generated code, it is necessary to validate the generation process. Achieving a code generator easy to validate follows from meeting three more concrete requirements.
  - **The Generated Code Should be Compact** Usually, the compactness of the generated code is a good indication of the complexity of the code generator. One of the approaches how to achieve compactness of the generated code is to create a shared library of patterns upon which the generator can lean. Considering HULOTTE and its

generation process, we implement a library of platform-level components (containers, controller, interceptors, etc.) that are frequently used during the generation, in Section 5.2.

- **Direct Model-to-Code Traceability** Since our approach is model-based, model-to-code traceability is a central factor. Moreover, the most stringent development standards (such as DO-178 [IH92]) require full traceability across the different abstraction levels, the model-to-code traceability becomes a necessary asset to trust the code generation.

  However, when looking at our approach, both the core-level to platform-level model transformation and the platform-level model to code transformation are clearly defined and a clear and direct model-to-model and model-to-code semantics mappings can be found. This is guaranteed by the fact that CBSE principles are used as basic concepts at both model and code level, where resulting application implementation is implemented in component-based programming. Therefore the component-based concept represents the same functionality in the model and also in the resulting code.
- **Code Generator Development Tools Should Allow Easy Model Navigation** Finally, the generator development tools massively influence the development complexity, it is therefore important to provide an extensive tool support.

  In our approach, we employ several tools at different stages of the development process to mitigate its complexities. First, during the design of the application we employ the Alloy Analyzer [KM08] and Alloy for Eclipse plugin [BB08] to validate model instances. Second, during the implementation of both functional and platform-level concepts, different tools based on the FRACTAL project can be used. Notably F4E [Dav08] - an Eclipse plug-in for FRACTAL, and FRACTAL-ADL [LOQS07].

- **Separation of Concerns.** The separation of concerns in our approach is implicitly met since already at the model level we define clear entities that represent domain-specific concerns - `Domain Components`. Looking at the separation of concerns more in details, it can be applied at multiple levels, in particular:

  - **The Separation of Generated from Manually-Written Code** Distinguishing generated code from manually-written code is important for two main reasons. First, the generated code is expected to be correct, meaning that it should correctly map the semantics expressed at the model level. Second, the generated code needs not to be verified, on the ground that the originating model has been checked extensively.

    We achieve the separation of generated from manually-written code through the following features of the HULOTTE framework. First, the clear separation is already achieved at the model level, where functional concerns are represented by the concepts of the core-level of the model and the domain-specific concerns are represented by the concepts defined in the platform-level of our model, as defined in Section 5.1. Consequently, when generating the implementation of platform-level concepts, we deploy generated code into clearly defined component entities, as described in Section 5.3.
  - **The Separation between Functional and Non-functional Semantics** The separation functional and non-functional semantics brings several advantages. To name one, it foster reuse, since it allows one to reuse functional and non-functional components independently of each other. Furthermore, the separation of concerns increases simplicity of the application implementation since the functional design and implementation is not tangled with the non-functional concerns.

    This requirement, similarly as the previous one, is implicitly met since we represent both functional and non-functional concerns as clearly identified software components – either functional or domain components (defined in Section 4.1).

Based on the discussion above, we conclude that SOLEIL and HULOTTE frameworks are meeting the most important criteria from the code generation perspective. Mainly, the concept of

domain components proves to be highly beneficial since it allowes us to achieve separation of concerns along the whole development lifecycle.

6.1.4 Evaluation

The following aspects of the above evaluation are noteworthy:

Performance Perspective

The bottom line is that our approach does not introduce any non-determinism. Moreover, the overhead of the framework is minimal when considering *MERGE_ALL*, but with the same functionality as our non-optimized code. Finally, we demonstrate a fitness for embedded platforms by achieving a memory footprint reduction (*ULTRA_MERGE*) that provides better results than the OO-approach.

Code-Generation Perspective

We have evaluated the HULOTTE framework from the code generation perspective by confronting it with code generation requirements from [BV07]. Evaluation showed that the HULOTTE framework is meeting this criteria and that particularly the separation of concerns is achieved by using the domain component concept proposed in the SOLEIL framework.

6.2 Real-time Collision Detector

The purpose of the second case study is to validate our approach on a real-life application. The key motivation is to employ the domain component concept and the HULOTTE framework in order to achieve a better separation of concerns in RTSJ systems and to mitigate complexities of the RTSJ-based development process.

Furthermore, this case study was conducted to evaluate the SOLEIL framework and its benefits from a software engineering point of view. The software system used in this experiment is modeling a real-time collision detector (RCD), described in [ZNV04]. The collision detector algorithm consists 241 classes (around 30 KLOC) and was originally written with RTSJ, and since then it was used in several research projects as evaluation case study [PFHV04, ACG+07, ABG+08].

Therefore, first, we describe the RCD case study in Section 6.2.1. Consequently, in order to illustrate properly challenges of RCD implementation, we discuss prior implementations of this case study, published in [ZNV04, PFHV04], in Section 6.2.2. We present implementation of RCD in the SOLEIL framework in Section 6.2.3. Finally, we discuss relative benefits achieved in Section 6.2.4.

6.2.1 Description

The RCD case study represents a real-time safety-critical application developed to monitor air traffic. RCD must proceed positions of each aircraft and compute trajectories of aircraft in order to detect possible collisions. By the nature of the task, the application has to meet hard real-time constraints, since the positions of aircraft are periodically updated in very short intervals. Furthermore, the application must proceed all the data received in every interval. Dropping arriving information in order to increase throughput can not be tolerated in any case, since a missed deadline could potentially lead to an aircraft collision resulting in material and also people casualties.

The detection algorithm [ZNV04, PFHV04] implemented by RCD is a single threaded hard real-time task which periodically receives a stream of aircraft positions and must determine if any of these aircraft are on a collision course. In RTSJ, the task is implemented as 10Hz NoHeapRealtimeThread. In every loop it creates a list of motions representing movements of aircraft

from their previous states and computes a list of collision courses. A list of recent aircraft positions, called `StateTable`, is being updated and kept between each loop of the algorithm. Since the computation logic requires allocation of temporary memory to store intermediate results, a scoped memory region is used.

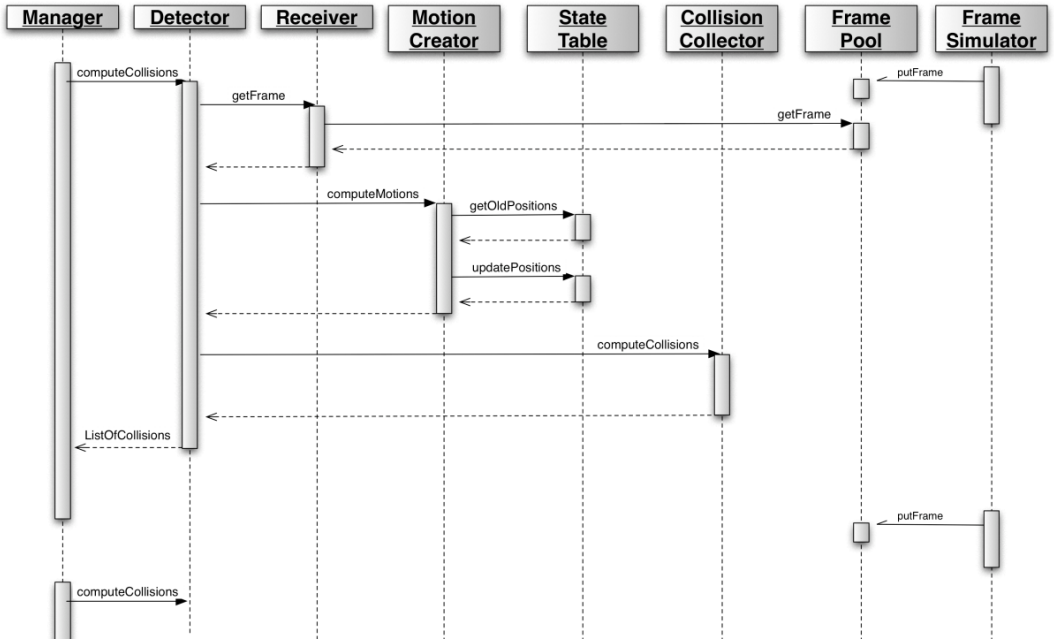


Figure 6.2: RCD, Sequence Diagram

In order to better illustrate the algorithm, we present its sequence diagram in Fig. 6.2. The computation starts with the `Frame Simulator` that is responsible for simulating a real air traffic by creating a list of aircraft positions. `Frame Simulator` generates the aircraft positions in 10Hz intervals, the data are stored in an instance of the `Frame` object. Since `Frame Simulator` is a hard real-time task, its interaction with the system is isolated, therefore, it only stores every frame in an `FramePool` from where they can be further processed by the collision detector algorithm.

The collision detector algorithm is then launched by the `Manager` that is activated in every iteration to compute collision courses for given positions of aircraft. Therefore, for every iteration it instructs the `Detector` to start computations. `Detector` first contacts `Receiver` to receive a stream of aircraft positions stored in a `Frame`. `Receiver` retrieves the first available `Frame` from the `FramePool` and returns it to the `Detector`.

Consequently, motions of aircraft needs to be computed. These motions are computed from the current positions of aircraft received from the `Receiver` and the last known positions of aircraft stored in the `StateTable`. This is performed by the `MotionCreator` that contacts the `StateTable` and computes a list of motions. Finally, the list of motions is delegated to the `CollisionDetector` that computes possible collision courses.

Since the system is operating under hard real-time deadlines and must operate in very strict time conditions, a regular Java can not be used. Instead, non-heap real-time thread (NHRT) must be used, restricting the developers to use immortal and scoped memories. Thanks to the characteristic of the algorithm, each of its iteration can be performed in a scoped memory - applying for `Detector`, `Receiver`, `MotionCreator` and `CollisionDetector`. In order to keep positions of aircraft between each iteration, the `StateTable` must be placed outside of the scoped memory allocation context to prevent its collection after each iteration.

However, implementation of the `StateTable` is a good example of the intricacy of RTSJ programming. An extract of computed data from a child scope must be stored in a parenting

scope while the computation in the child scope continues. Developers must constantly switch between different allocation contexts and perform deep copying of selected data, making the programming process highly error-prone.

In the following section we will therefore demonstrate how prior research is dealing with challenges of RCD development. Although the sequence diagram explains clearly the RCD algorithm, its implementation in presented approaches is usually hampered with the concepts of RTSJ. The code must be tangled at many places and additional trade-offs are introduced, modifying the algorithm and the system architecture itself. We will therefore show these proprietary solutions in order to finally present solution in the SOLEIL framework and thus highlighting its contributions.

6.2.2 Current Approaches and Their Limitations

In order to properly illustrate the challenges of RCD implementation, we will discuss prior implementations of this case study published in [ZNV04, ACG+07].

Original RCD Implementation

In the original RCD implementation [ZNV04, PFHV04] the functional code was monolithic and highly tangled with the RTSJ-related code, making the implementation hard to understand, debug, or potentially update.

Furthermore, the non-intuitive form of memory management prevents developers to exactly determine in which scope area a given method is being executed. A drawback that is particularly tricky when allocating data that needs to be preserved across the lifespan of memory scopes. Specifically, this situation occurs when updating the data in the `StateTable`. We show a code-snippet in Fig. 6.3, which is implemented using a multiscope pattern [PFHV04] - an instance of class allocated in one scope but with some of its methods executing in a child scope.

Looking at the `StateTable`, the method `createMotions()` is executed in the child scope and computes motions of aircraft. At lines 14 to 15, the new and old position of the aircraft is computed. Afterwards, if the `old_pos` is `null` then we have detected a new aircraft and we need to add it into the list of aircraft - lines 18–23. Otherwise – lines 24-28, a position of already detected aircraft is updated.

However, to ensure that these changes will be permanent, we need to switch from a child scope to a parent scope – lines 24-28, and then using the classes `Putter` or `Putter2` we distinguish whether a new aircraft will be added - line 37, or a position of an old one will be updated – line 44.

Bottom Line To summarize, the RTSJ and functional code are highly tangled. Moreover, the multi-scoped object pattern does not distinguish clearly when a scoped-memory switch should be performed, this is however putting a lot of burden on developers who must reason about the current allocation context and if necessary, switch it. As already shown in [Nil04], manual scoped memory control and switching performed by developers is a highly error prone practice. Finally, when looking at the RCD sequence diagram, the responsibilities of `MotionCreator` and `StateTable` were merged into the `StateTable` class, which does not influence functionality, but, however, mixes different functional concerns.

STARS Project

The issues of the original implementation were partially mitigated in the STARS project [ACG+07] using AOP. The project proposes a novel approach for programming real-time systems in order to shield developers from many accidental complexities that have proven to be problematic in practice. One of the goals is to mitigate complexities of memory switches, by making them implicit. The approach uses a program's package hierarchy to represent the structure of its memory use, making clear where objects are allocated and thus where they are accessible. This means that

```
1  public class StateTable {
2    HashMap prev = new HashMap();
3    Putter putter = new Putter();
4
5    public List createMotions (RawFrame f) {
6      final List result = new LinkedList();
7      for (...) {
8        x = f.decodeX(...);
9        y = f.decodeY(...);
10       z = f.decodeZ(...);
11       cs = f.getAircraftCallSign(...);
12       Aircraft craft = new Aircraft(cs);
13
14       Vector3d new_pos = new Vector3d(x, y, z);
15       Vector3d old_pos = prev.get(craft);
16
17       Vector3d old = (Vector3d) prev.get(craft);
18       if (old_pos == null) {
19         putter.c = craft;
20         putter.v = new_pos;
21         MemoryArea current = MemoryArea.getMemoryArea(this);
22         current.executeInArea(putter);
23       }
24       else {
25         putter2.c = craft;
26         putter2.v = new_pos;
27         MemoryArea current = MemoryArea.getMemoryArea(this);
28         current.executeInArea(putter2);
29       }
30     }
31     return result;
32   }
33   public class Putter implements Runnable {
34     Aircraft c;
35     Vector3d v;
36     public void run() {
37       prev.putNewAirCraft(c, new Vector3d(v));
38     }
39   }
40   public class Putter2 implements Runnable {
41     Aircraft c;
42     Vector3d v;
43     public void run() {
44       prev.updateOldAircraft(c, new Vector3d(v));
45     }
46   }
47 }
```

Figure 6.3: `StateTable` Original Implementation

developers define an exact memory scope for every program's package thus defining an allocation context for each class contained in the specific package. Real-time Aspects, defined by the approach, then weave in allocation policies and implementation-dependent code — separating real-time concerns further from the functional implementation.

To demonstrate this approach, we again show a code snippet of the `MotionCreator` and `StateTable` communication in Fig. 6.4. In order to define allocation contexts, the classes implementing the algorithm are scattered across different packages. The package `imm.runner` gathers classes with the allocation context of the parent scope, therefore, the `StateTable` class, keeping the aircraft positions between each iteration, is defined here. Furthermore, a class `Vector3d` is defined here to store an aircraft position.

The classes executing in the child scope are defined in the package `imm.runner.detector`. Therefore, the class `StateTable2` responsible for computation of motions (thus corresponding to `MotionCreator` class in the sequence diagram) is defined here. Furthermore, another class `Vector3d` is defined to store an aircraft position in this allocation context. As we can see, `StateTable` and `Vector3d` are defined in two variants, each in a different package.

The algorithm starts in the method `createMotions` in `StateTable2` – line 19. We compute a new aircraft position - line 28, and store in the childscope `Vector3d`. Afterwards, we retrieve

```
1  package imm.runner;
2
3  public class Vector3d { ... }
4
5  public class StateTable {
6    HashMap prev = new HashMap();
7    public void put(...) {
8      ...
9    }
10 }
11
12 package imm.runner.detector;
13
14 class Vector3d { ... }
15
16 class StateTable_2 {
17   StateTable table;
18
19   public List createMotions (RawFrame f) {
20     final List result = new LinkedList();
21     for (...) {
22       x = f.decodeX(...);
23       y = f.decodeY(...);
24       z = f.decodeZ(...);
25       cs = f.getAircraftCallSign(...);
26       Aircraft craft = new Aircraft(cs);
27
28       new_pos = new Vector3d(x, y, z);
29       final imm.runner.Vector3d old_pos = (imm.runner.Vector3d) table.previous_state.get(craft);
30
31       if (old_pos == null) {
32         table.put(craft.getCallsign()), new_pos.x, new_pos.y, new_pos.z);
33       } else {
34         final Vector3d save_old_position = new Vector3d(old_pos.x, old_pos.y, old_pos.z);
35         old_pos.set(new_pos.x, new_pos.y, new_pos.z);
36         ...
37       }
38       ...
39     }
40     return result;
41   }
42 }
```

Figure 6.4: StateTable, STARS Project Implementation

an old position and store it in the parent scope `imm.runner.Vector3d` - line 29. In order to add a new aircraft to the `StateTable`, we call `table.put()` - line 32. Note, that the `StateTable2` stores a reference to the `StateTable` class from the parent scope - line 17. However, updating an old aircraft position is tricky. Instead of calling an `updatePosition` method, we allocate a new `Vector3d` in the parent scope holding the new position of the aircraft - line 34, and update the `old_position` class explicitly - line 35. The Real-time Aspect Weaver then assures that appropriate context switching code will be weaved in the functional code during the compilation time.

Bottom Line When looking at this solution, we can see that although resulting code is more compact and context switching is performed implicitly on the basis of packages, the implementation is not very intuitive since even now developers are not sure where are the allocated objects stored - e.g. looking at the `Vector3d` class at the line 34. Moreover, the approach forces developers to define the same classes redundantly in different packages, making potential modification of the code extremely difficult.

Additionally, the RTSJ code is weaved into the functional during the development process and thus, the final functional implementation is still polluted by the RTSJ-related concepts. To allow the weaver to inject the RTSJ code the application has to be designed and deployed into different packages. However, these packages does not represent the functional logic of the application but obey the RTSJ restrictions. E.g. the `StateTable` logic divided into two packages.

On the other hand, an interesting coherence can be found between STARS and SOLEIL. STARS

defines a memory area per package, while SOLEIL can achieve a more fine-grained specification – a memory area per component (using domain components). Therefore, we comprehend the STARS approach for explicit memory area definition as an effort in a correct direction, which is however pursued more fundamentally in the SOLEIL achieving better results, as will be demonstrated in the next section.

6.2.3 RCD Implementation in the SOLEIL Framework

Finally, we have developed RCD in the SOLEIL framework. Therefore, we will focus on design and implementation of this case study.

Design of RCD

The architecture of RCD developed in our framework is depicted in Fig. 6.5. The original object-oriented RCD application was very easily re-engineered into a component-oriented one, moreover, the components implement only the functional logic of the application. Our approach allows us to design the application intuitively in correspondence with the logic of the algorithm (described in Fig. 6.2). `Manager` is periodically starting the computations, which are proceeded in every iteration by the `Detector`. `Detector` first asks `FrameReceiver` for new aircraft positions, creates motions in `MotionCreator`, and collisions are computed by the `CollisionCo--llector` component. Moreover, the `MotionCreator` updates the current aircraft positions which are stored between each iteration in the `StateTable` component. The `FrameReceiver` revives frames from the `FrameSimulator` that simulates aircraft movements. The designed architecture is very clear and easily understandable, thus the developers are able to reason about the system without being limited by the RTSJ-related issues. Consequently, domain components are applied to determine the allocation and execution context of each component.

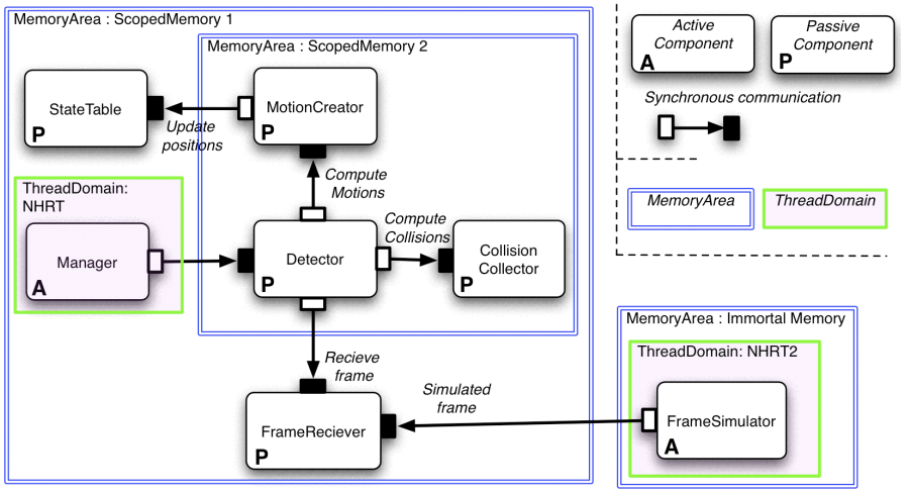


Figure 6.5: RCD Architecture

Implementation of RCD

After designing RCD architecture, we refine it with the platform-level concepts, fully evolved view on the architecture is presented in Fig. 6.6. Since the design performed by the user specified different domain components, we can perform reasoning about the compositional and binding rules that we have specified in the SOLEIL and HULOTTE frameworks. As a result, we can see

that several interfaces and controllers were deployed into the refined architecture in order to represent RTSJ concerns. Notably, a `HandOff Interceptor` was deployed in the binding between `MotionCreator` and `StateTable`, and `ScopeInterceptor` deployed between `Manager` and `Detector`.

Since we have already defined the allocation contexts (represented by the memory area domain components) at design time, together with other RTSJ-specific concerns, we can focus only on implementation of functional components at implementation time. In Fig. 6.8 we are showing the SOLEIL implementation of the `StateTable` logic. Thanks to the separation of concerns, we can implement the logic in the same way as described in the RCD sequence diagram in Fig. 6.2. Looking at the `MotionCreator`, we can see that it is connected with the `StateTable` component through the `IStateTable` interface - line 3, that is defined in Fig. 6.7. Therefore, the communication with the `StateTable` component is performed through this interface. We can further observed that no RTSJ-related code is present.

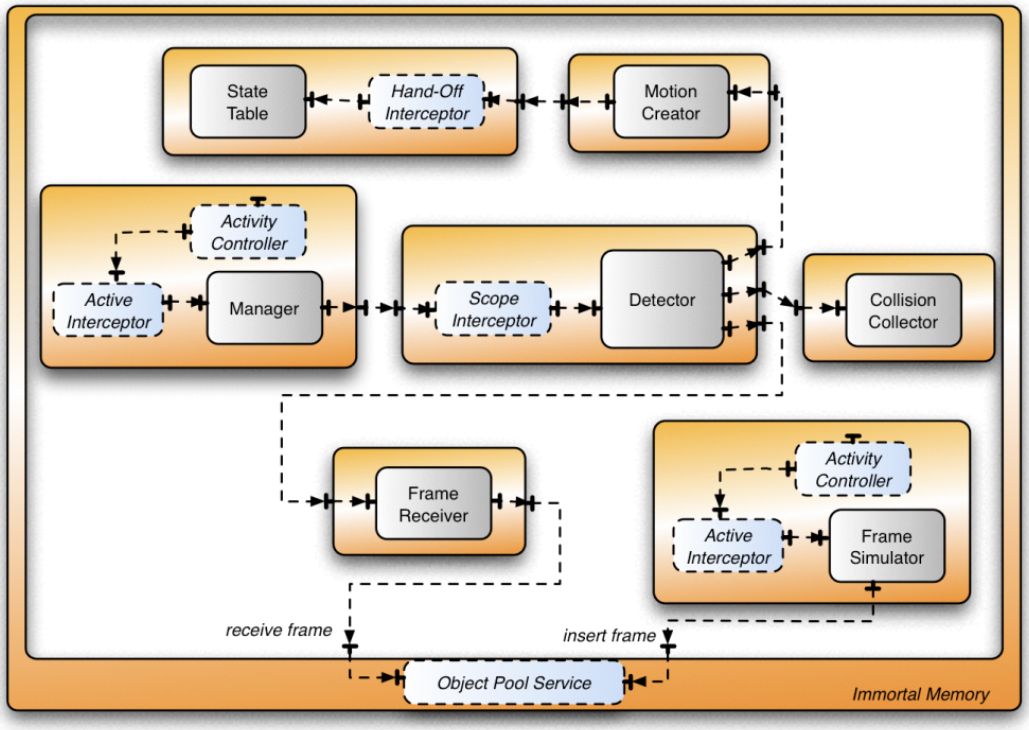


Figure 6.6: RCD Refined Architecture

```
1 interface IStateTable {
2    Vector3d getAircraftPosition(CallSign cs);
3    CallSign createCallSign(CallSign cs);
4    void putNewPosition(CallSign cs, Vector3d v);
5    void updatePosition(CallSign cs, Vector3d v);
6 }
```

Figure 6.7: `IStateTable` Interface

Again, the flow of the algorithm is starting in the `createMotions` method in the `MotionCreator` – line 5, we first proceed the new and old positions of an aircraft – lines 14-15. Afterwards, we determine whether the `old_pos` is `null` and either insert a position of a new aircraft – line 14,

or update a position of an already known aircraft – line 22.

```
1  public class MotionCreator implements
2              IMotionCreator {
3    IStateTable iStateTable;
4
5    public List createMotions(RawFrame f) {
6      ...
7      for (...) {
8        x = f.decodeX(...);
9        y = f.decodeY(...);
10       z = f.decodeZ(...);
11       cs = f.getAircraftCallSign(...);
12       Aircraft craft = new Aircraft(cs);
13
14       Vector3d new_pos = new Vector3d(...);
15       Vector3d old_pos = iStateTable.getAircraftPosition(craft);
16       if (old_pos == null) {
17         CallSign cs2 = iStateTable.createCallSign(cs);
18         iStateTable.putNewPosition(cs2,new_pos);
19         ...
20       }
21       else {
22         iStateTable.updatePosition(cs,new_pos);
23         ...
24       }
25     }
26     ...
27 }
```

Figure 6.8: `MotionCreator` Implementation

Bottom Line Finally, the implementation of the `StateTable` is very intuitive, as we can see in Fig. 6.9, only functional code is present. Moreover, the code is clear and reflecting the needs of the application without any constrains imposed by the real-time properties.

```
1  public class StateTable implements IStateTable  {
2    HashMap prev = new HashMap();
3    public Aircraft getAircraftPosition(...) {
4      ...
5      return aircraft;
6    }
7    CallSign  createCallSign(CallSign cs) {
8      return new CallSign(cs);
9    }
10   ...
11 }
```

Figure 6.9: `StateTable` Implementation

The RTSJ related concerns are expressed by the domain components, depicted in Fig. 6.5, here the memory components play a key role since the communication between different scopes is needed. This is therefore beneficial for the functional implementation since the functional and RTSJ-related concerns are separated, as shown above.

Furthermore, the code arbitrating the cross-scope communication between `MotionCreator` and `StateTable` components can be automatically generated and deployed in the interceptors hidden in the containers. We demonstrate such a code example in Fig. 6.10 where a `HandOff-Interceptor` arbitrating the communication between the `MotionCreator` and `StateTable` is shown. Here, a simple cross-scope pattern, defined in Section 4.2.4, is used.

6.2.4 Evaluation

The following aspects of the RCD implementation in the SOLEIL framework are noteworthy.

```
1 public class HandOffInterceptor
2     implements IStateTable {
3  IStateTable iStateTable;
4  ScopedMemory scope;
5   public void putNewPosition(...) {
6     ...
7     scope.executeInArea(putter);
8   }
9 }
```

```
1 public class Putter
2    implements  Runnable {
3  public void run() {
4     ...
5  }
6 }
```

Figure 6.10: `SoleilInterceptor` Implementation

The domain components simplified expression of RTSJ specific properties, since these properties are present in the architecture as first-class entities. A full separation of functional and real-time concerns is achieved, therefore, the functional code is more readable — reflecting the functional needs of the application without any constrains imposed by the real-time properties. As the second benefit of our approach we consider application of the HULOTTE tool-chain for automatic generation of the runtime platform implementing RTSJ-related code, which is highly error-prone when implementing by hand.

Comparing to other projects implementing RCD, we achieve better separation of concerns, resulting implementation is not tangled by RTSJ-specific code and we are able to follow the original logic of the RCD algorithm without proprietary trade-offs enforced by the RTSJ limitations.

6.3 Distributed and Ambient Programming in SOLEIL and HULOTTE

The goal of our final case study is to evaluate the SOLEIL and HULOTTE frameworks in a more general view without focusing solely on RTSJ. The challenge is to evaluate the concept of domain components and its support in SOLEIL and HULOTTE frameworks towards various domain-specific challenges. We have conducted two case studies: the Distributed Real-time Programming with SOLEIL case study [MPL+08] and the Ambient-programming with Hulotte case study [PMS07], in this section we revisit them in the context of this dissertation.

6.3.1 Distributed Real-Time Programming with SOLEIL

The aspect of distribution in RTSJ systems still represents a challenge and brings many open issues. The state-of-the-art of distributed and real-time Java lies at its very beginning. A few initial studies introducing specifications, profiles or frameworks [AJ06, WCJW02, TAdM07] have been conducted, however, there is still a need of a comprehensive solution proposing a full-fledged approach that would mitigate complexities of real-time programming in distributed systems.

We further envisage that supporting development of distributed real-time systems is a highly desired feature, therefore as one of the key contributions of this case study we focus on extensions of our framework towards distribution support. Following the philosophy of the SOLEIL framework, our goals are twofold.

- Employ SOLEIL framework in order to propose appropriate abstractions in the form of domain components that clarify specification of model artifacts and properties that will cover distributed real-time requirements. Thus create an abstract layer which will hide low-level distribution concerns from component designers.

- Use the HULOTTE framework to implement semantics of such components and manage transparent deployment and execution of distribution support inside the platform layer.

Requirements and Challenges

Integration of distribution into the SOLEIL component framework is a challenge involving an analysis of requirements dedicated to real-time systems as well as requirements coming from RTSJ. All these requirements affect not only a way of specifying model artifacts (components, bindings and their properties) but also its runtime structure and the process of its initialization. We therefore determine a scope of requirements which have to be reflected by a distributed system within a real-time environment at all stages of the application lifecycle.

Real-time Properties. Since real-time programming introduces specific requirements on distributed systems (e.g. priorities of running tasks, computational deadlines), they play a substantial role during the development. These properties influence remote connections and superimpose new constraints over them. Some real-time properties have to be propagated between remote parts of applications (e.g. priority of a client thread) and others have to be reflected during creation of the connection (e.g. end-to-end time).

RTSJ Requirements. Moreover, employing RTSJ in development of distributed Real-Time Java-based systems is also affected by the particularities of its specification. However, the specification silences about distribution aspects and therefore these complexities need to be resolved by the developers.

Integration level. Furthermore, the integration of distribution into a component framework yields a decision at which level of abstraction the distribution will be incorporated into the framework and how a component-application developer will manipulate with real-time properties. Whether to hide the manipulation from the developer or not. These questions were discussed in the scope of RMI integration into RTSJ presented in [WCJW02]. It distinguishes three basic levels of RMI integration (denoted as `L0`, `L1` and `L2`) from different views.[6] `L0` is the minimal level of the integration with no support for real-time properties from underlying technology. The level `L1` requires a transparent manipulation with scheduling parameters or timing constraints and finally `L2` declares semantics for the distributed thread concept [AJ06] which represents a fully transparent real-time programming model.

We partially adopt this idea of integration levels in our approach. The primary objective is an integration of distribution into the RTSJ-based component system at a level corresponding to `L1`. We however generalize the idea of the level `L1`, originally tightly coupled with RMI, to address the full span of possible communication middlewares (RMI, CORBA) in distributed environments. We therefore address the following contributions to meet this generic goal:

(i) **Scheduling Parameters.** To handle transparently scheduling parameters which are associated to component threads. The task involves a transportation of parameters from a client to a server where it is required, configuration of an underlying middleware (e.g. in case of CORBA, creating priority lanes), pre-reservation of connections for selected priorities, etc.;

(ii) **Determinism.** To ensure that the generated runtime infrastructure does not affect determinism and timely delivery assured by a used underlying middleware;

(iii) **RTSJ Rules and Restrictions.** To handle memory and thread differences between components and a used middleware. This also covers handling of a memory allocation of call parameters (e.g. CORBA parameter holders) and auxiliary artifacts (e.g. adaptors, call serializers),

(iv) **Communication Styles** To provide different communication styles [Bur06, MMP00] which are common in the real-time and embedded systems world (synchronous and asynchronous method call, asynchronous messaging).

`DistributedNode` Domain Component

The purpose of this case study is therefore to introduce a notion of distribution simply by defining a new domain component – `DistributedNode` (DN). Each DN represents a distributed node of

[6] *Programming model* (identification of remote objects), *development tools* and *implementation model* (real-time properties transport mechanism). In our case, the first two models are realized by the SOLEIL framework, therefore the following text is interested in the third one.

the application, a functional component in DN will be thus deployed on the corresponding node together with its runtime support extended towards the specifics of distributed communication.

We illustrate an example of DN in Fig. 6.11. As we can see, `Client` and `Server` are both deployed in different DNs. This means that each of them will be deployed in a distributed environment on a dedicated node. Moreover, every communication crossing different DNs must be implemented as a distributed communication based on the properties of each DN. In Fig. 6.12 we show formal definition of the `DistributedNode` component.

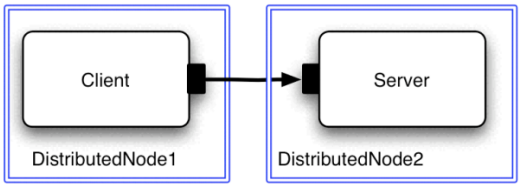


Figure 6.11: `DistributedNode` Component Example

```
1 sig DistributedNode extends DomainComponent()
2
3 fact EveryComponentHasOnlyOneDistributedNode{
4   all c:Component |
5     one d: DistributedNode |
6       c in d.^subComponents
7 }
8
9 sig AmbientNode extends DistributedNode{}
```

Figure 6.12: `DistributedNode` and `AmbientNode` Formal Definitions

Moreover, the HULOTTE framework is responsible for implementing semantics of DN, meaning that the framework generates each `DN` component as a self-standing application allowing deployment of the components into the corresponding nodes. Furthermore, each distributed binding must be implemented as a distributed communication. We present this in the following section.

`DistributedNode` Implementation

The basic idea of our approach is inspired by a solution in which components communicate through architecture-level software connectors that are implemented using a middleware [MDT03]. This approach preserves the properties of the architecture-level connectors while leveraging the beneficial capabilities of the underlying middleware. This approach is integrated into the HULOTTE framework and represented by the `ChainComposite` pattern.

The role of the framework developer is therefore to apply the `ChainComposite` pattern on each distributed binding, corresponding stubs and skeletons will be refined as subcomponents of the `ChainComposite`. The HULOTTE framework provides the platform-level generator that generates a system's infrastructure on the basis of a given architecture. Thus we automatically obtain container implementations, consequently mitigating complexities of the system development. Additionally, the process of designing and implementing containers addresses the real-time challenges identified in the previous section.

From the high-level point of view, we adopt a general approach to a generation of component connectors presented in [Bur06], we however focus on more lightweight and especially RTSJ tightly coupled solution.

Architecture Refinement Process of Distributed Connections. At design time we perceive `ChainComposite` Interceptors as representations of bindings between functional components. A binding has attached non-functional properties such as benchmarking, enforcement of a dedicated connection or prescribed utilization of a given middleware. Furthermore, the binding connects components which also have associated properties (e.g. call deadlines for interface operations) or they receive derived properties from non-functional components in which they are placed (e.g. memory allocation context, thread priorities). All these properties are reflected in the `ChainComposite` Interceptor architecture representing the binding.

The chosen architecture also brings advantages in dealing with issues triggered by using RTSJ such as memory scopes crossing or copying between memory areas.

Generation Process. The connector generation process includes:

(i) **Chain Structure Selection.** Which involves selecting interceptors and their order in according to binding properties (specified and derived) and also to RTSJ requirements, e.g. selecting memory allocation areas and adapting memory or thread differences;

(ii) **Interceptor Code Generation.** The task involves generation of interceptors and of a selected middleware specific code (e.g. initialization of middleware, setting connection parameters).

Furthermore, different optimizations in the chain or in its selected parts are possible, similarly as proposed in Section 5.3.5.

Runtime. The preservation of the container architecture at the runtime level permits modification of interceptor attributes. Either simple attribute modifications affecting only one interceptor are possible (e.g. modification of middleware threads priority) or even more advanced adaptations of the whole `ChainComposite` Interceptor structures can be performed (e.g. update of interceptors in a chain, change of the interceptors order). Moreover, the framework generates each DN component as a self-standing application allowing deployment of the components into the corresponding nodes.

Example Scenario

The proposed concept was applied in an implementation of the motivation scenario presented in Section 2.2.3. Concretely, we model a real-time communication between two active components - *ProductLine* and *MonitoringSystem* allocated in a non-heap memory. Both components have associated properties defining components' thread priorities. The binding between these components is modeled as a remote binding with two associated non-functional properties — the first one enforces utilization of a distribution enabling technology and the second one identifies asynchronous method call. In our case we use *RTZen* middleware [RZP+05] as the distributed enabling technology, however, different middleware technologies can be used, depending on user's choice. Furthermore, the HULOTTE framework could be extended by a process for automatic instantiation or the middleware layer, e.g. introduced in [ZPH08], which would implement only the services required by the application, achieving performance gain and footprint reduction.

These simple properties involve several tasks which have to be covered by the generated interceptors and its *chain* architecture: (i) implementation of core distribution with help of RTZen. This also involves generation of low-level CORBA interfaces, helpers, value holders in according to a specified IDL; (ii) configuration of underlying middleware - adjustment of CORBA policies to reflect components' thread properties; (iii) asynchronous method calls in case the underlying middleware does not support them; and (iv) adaptation between memory areas; (v) adaptation between functional and internally generated interfaces.

These requirements are reflected by a selected chain of interceptors at the client as well as at the server side (see Fig. 6.13). The core of the distribution implementation is generated in interceptors called *RTZenStub* and *RTZenSkeleton* which mediate the communication with help of the RTZen middleware.

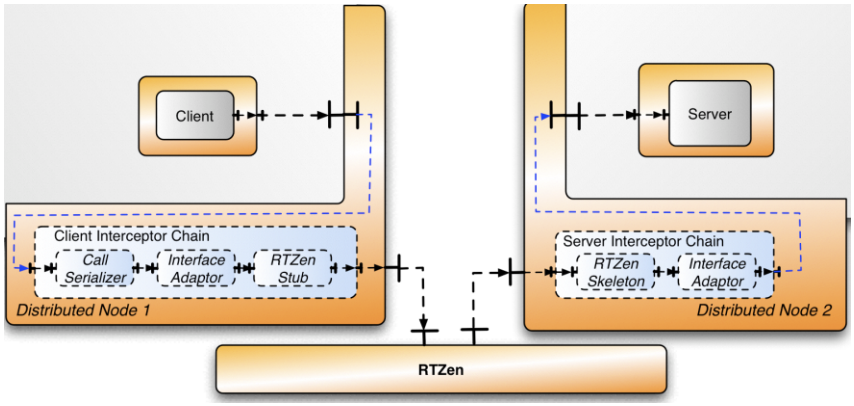


Figure 6.13: `DistributedNode` Refinement

At the server side, *RTZenSkeleton* registers itself as a remote object in RTZen and serves like a proxy which delegates calls to a following interceptor which adapts an internally generated interface to the server component's functional interface. The *RTZenSkeleton* interceptor also configures a priority with which remote calls will be handled. The delegation needs a simple interface adaptation (interceptor *Interface adaptor*) to bridge differences between the server functional interface and interface required by RTZen.

At the client side, *RTZenStub* obtains, via calling the encapsulated RTZen middleware, a reference to the remote object and delegates all incoming calls to it. However this reference implements the internally generated interface, therefore it has to be adapted to the functional interface by another interceptor called *Adaptor*. Finally, the *Serializer* interceptor arranges asynchronous semantics for method calls - each call on its provided interface is stored in a local queue and then served by a thread associated with the queue.

6.3.2 Ambient Programming with HULOTTE

Ambient-Oriented Programming (AmOP) [JCS+05] as a new trend in software development comprises a suite of challenges which are yet to be addressed fully. So far, only a few solutions facing the obstacles of ambient programming have been developed. In this case study we focus on AmbientTalk [JCS+05] since in our opinion it represents one of the most sophisticated solutions. Although AmbientTalk conceptually proposes a way to implement applications for the ambient environment, this is achieved by defining a new programming language. Consequently, AmbientTalk potentially introduces a steep learning curve for the developers. From this point of view, it is reasonable to search for an approach which uses well-known techniques and is powerful enough to face the obstacles of ambient programming. We believe that these requirements can be met by the using the technology provided by the HULOTTE framework.

This case study is based on our prior research [PMS07] and its goal is to provide a support for ambient-oriented programming in component-based systems. The key idea of the solution is to use the concept of domain components to describe ambient concerns in the system and by employing the HULOTTE framework we strive to manage ambient software components transparently for users, mitigating the complexities of the ambient world.

Besides, considering the challenges of ambient programming in the context of real-time Java is reasonable, since the ambient-oriented software usually encompasses some real-time constraints and RT Java represents a promising choice for this technology, as discussed in [AMPN+06].

Challenges and Requirements

Ambient Intelligence [Gro03] represents a new trend of computing where technology is gracefully integrated into the everyday life of its users. This new field in distributed computing comprises wireless devices which spontaneously communicate with each other.

The specific character of a highly dynamical mobile environment however imposes special constraints (facing the connection volatility, the ambient nature of resources, etc.). These challenges form a new group of programming techniques – Ambient-Oriented Programming. Although the main stress here is laid on facing the so-called Hardware Phenomenon [JCS+05], we believe that software engineering aspects supporting more effective development of ambient oriented applications should be more emphasized.

AmbientTalk is a programming language that explicitly incorporates potential constraints of the distributed mobile environment in the very heart of their basic computational steps, thus addressing directly the obstacles of application development for mobile devices. To deal with the ambient environment characteristics, AmbientTalk implements several features. For this discussion we focus on two keystone concepts: Ambient Reference and Non-blocking Futures.

The ***Ambient Reference*** concept represents a powerful solution to referencing objects in ambient environment. Ambient reference operates in two states - unbound and bound. When an ambient reference is *unbound*, it acts as a discovery channel looking for remote service objects in the environment to bind to. Once such a suitable object is found, the ambient reference becomes *bound*. Once bound, an ambient reference is a true remote object reference to the remote service. When the service object to which an ambient reference is bound moves out of communication range, the ambient reference can become unbound again. Then it acts as a peer discovery mechanism again and tries to rebind to the same or another matching service.

Since the concept represents an asynchronous way of communication, it is necessary to face the challenge of returning the result of a client's request. To provide this, the ***Non-blocking Futures*** concept is introduced. It allows to associate a block of code which will be triggered on the client once its request is resolved – the returning value from the server is thus processed. The main motivation for employing this feature is to manage the returning value processing without the introduction of callback methods.

Other solutions to the ambient environment challenges exist, we refer interested reader to [GK03].

`AmbientNode` Domain Component

In order to define ambient concerns in the system, we employ the concept of domain components. We define a new domain component - `AmbientNode`, in Fig. 6.14, in order to represent those software components that are extended towards specifics of the ambient world.

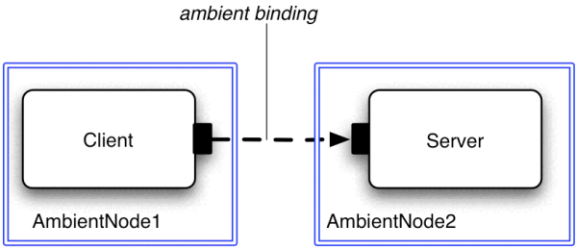


Figure 6.14: `AmbientNode` Component Example

The `AmbientNode` component, as defined in Fig. 6.12, is an extension of the `DistributedNode` component, and represents a node that exists in an ambient environment. Furthermore, each binding connecting different `AmbientNode` components is an ambient binding - a component-based variant of the ambient reference.

In order to implement the concept of non-blocking futures, we associate to every ambient binding a call-back binding that is transparently managed by the container of the `AmbientNode`, see the next section for further details.

`AmbientNode` Implementation

The adaptability of the container, supported by the HULOTTE framework, is the key feature we want to employ during the implementation of our solution. As already said, each component container can be extended individually thus perfectly fitting the specific needs of particular component. Applied to our experiment, we extend membranes of components implementing the communication between both actors with the ambient functionality. Thus the functionality is deployed only on specific components, they are extended with following units:

- **Ambient Controller** The ambient controller is a new managing unit introduced into the component membrane architecture. The task of the controller lays in managing the ambient functionality of the component. Particularly, the key responsibilities of this unit are the control of the ambient references and the deployment of ambient interceptors.
- **Ambient Interceptor** The interceptors deployed on every component interface allow to trace the component communication and to adjust the communication towards the specific needs of the ambient environment. E.g. either forward the messages to the recipient or buffer them when the recipient is unavailable.
- **Discovery Service** The solution is supported by the discovery service which manages the list of available services in the communication domain and notifies actors whenever a required resource appears or disappears from the domain. The motivation for introducing the centralized unit into the experiment is to allow to focuss only on implementation of ambient references and concepts.

Example Scenario

To demonstrate the potential abilities of our proposed solution, we have conducted an experiment that implements a middleware layer supporting the Ambient Reference and Non-blocking Future concepts - the fundamental features of AmbientTalk.

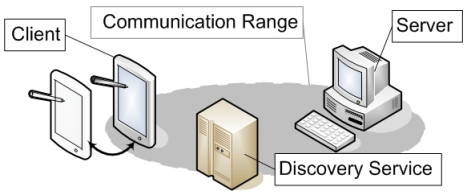


Figure 6.15: Ambient Scenario

The experimental implementation involves two actors : a *server* that provides a given service and a *client* that is searching for the service and that spontaneously enters and leaves the communication range of the server. The task is to use Ambient Reference and Non-blocking Futures concepts and thus hide the ambient character of the environment. To focus only on the implementation of these two concepts, we have extended this system by a third actor - a *discovery service*, which manages the service provisions and requirements in the environment. The discovery service operates at the middleware layer, communicating only with ambient-aware parts of actors. The whole scenario is depicted in Fig. 6.15.

Through the container adaptation we are able to achieve the ambient functionality, obtaining an *ambient component*. The functional code is not affected thus putting no extra burden on

the developer. Moreover, ambient-awareness extensions are transparent and can co-exist with the remaining unmodified components – achieving that potentially every component systems developed in HULOTTE can be extended.

When applying the approach to our experiment, the containers of components participating in the ambient communication are extended by the ambient controllers. An ambient binding, a component-oriented variant of the Ambient Reference concept, is instantiated once the *discovery service* announces that a client's desired service becomes available. Then, the `AmbientController` creates the binding between ambient components and deploys the `AmbientInterceptor` on the interface of the client component. Once the ambient binding is instantiated, the `Ambient-Controller` keeps this reference updated and notifies the interceptor every time the discovery service announces that an ambient resource is unavailable. The role of the interceptor is to either transmit messages to the server interface or to buffer them when the ambient service is currently unavailable.

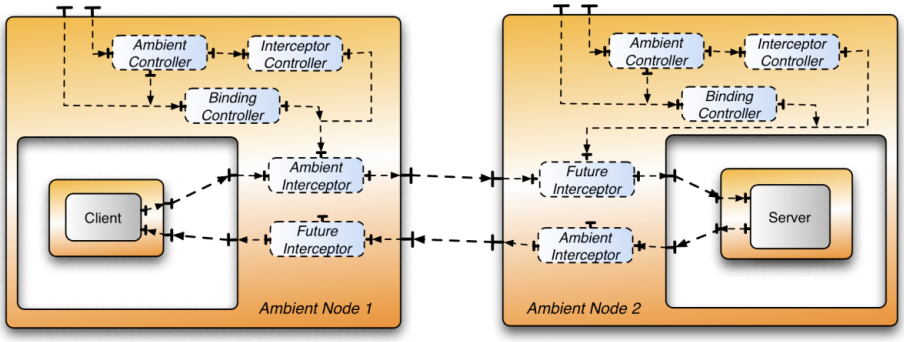


Figure 6.16: Ambient Communication Scenario

To implement the Future concept, the callback technique is used even though the original implementation of the concept in AmbientTalk avoids a callback. Every communication of the component with its environment has to be provided through an interface, it is therefore necessary to define a method for resolving a returning value and to expose this method in an interface definition. However, the callback binding is created automatically with the creation of an ambient reference. Both bindings are managed by the `AmbientController` and thus no special burden is laid on the shoulders of the developer.

The architecture of the ambient component is depicted in Fig. 6.16, where we can see container extensions - `AmbientController` and `AmbientInterceptor`, the ambient binding, and the Future callback binding which is created simultaneously and is managed in cooperation of ambient controllers on both client and server components.

6.3.3 Evaluation

The following aspects of the above implementation are noteworthy:

Distributed Real-Time Programming with SOLEIL

Distribution support in RTSJ-based component systems is a highly demanded property which is however neglected in the state-of-the-art solutions. Our case study proposed an approach to introduction of distribution support into the SOLEIL framework. Therefore, we have proposed and implemented a new domain component - `DistributedNode`. The application of the `DistributedNode` component was illustrated in a scenario introducing distribution using the RTZen middleware [RZP+05]. We have shown that our approach allows developers to hide the distribution support in container implementations which are automatically generated by our

framework tool – HULOTTE. When comparing our approach with the component connector technology [MDT03], we notice that the `ChainComposite` pattern allows framework developers to express any form of a component connector, while still using the general concept defined by our approach. Moreover, automatic generation of the runtime platform and separation of generated files according to distribution nodes mitigates complexities of distributed programming.

Ambient Programming with HULOTTE

In this second case study focused on evaluation of extendability of our approach we have been addressing challenges of ambient programming. Similarly as in the previous case, we have proposed a new domain component - `AmbientNode`, and we have demonstrated its implementation in the HULOTTE framework. The conducted experiment showed that HULOTTE provides sufficient extendability to develop ambient-oriented components. Furthermore, it indicates that the proposed solution potentially represents an equivalent alternative to AmbientTalk.

6.4 Limitations of our Approach

In this section we discuss specific technical shortcomings of our solution that were identified when conducting presented case studies.

Domain Component Dependencies and Policies

In this dissertation we focus on definition of domain components and their integration in the HULOTTE framework. However, an open research issue still remains specification of policies and constraints that regulate application of domain components at the functional level. Since some domain-specific services are non-orthogonal - competing or dependent on each other, their application must be exactly delimited in a form of policies that will manage non-trivial combinations of domain-specific services. A taxonomy of domain components should be proposed to clarify relations between specific domain components.

Furthermore, also at the runtime platform level, a consistent and symmetric approach to construction of containers needs to be specified in a form of policies that will manage non-trivial combinations of domain-specific services.

Development Process

Considering an implementation of each component, the designed architecture considerably simplifies this task. Functional and real-time concerns are strictly separated and a guidance for possible implementations of those interfaces that cross different concerns is proposed. However, the application developers have to fully follow the development methodology and the SOLEIL profile we propose, be aware of the restrictions imposed by RTSJ, and must not deliberately tangle the functional code with RTSJ concerns.

Separation of Concerns

Although we strive to achieve separation of concerns, we do not fully separate all the RTSJ concerns from the functional ones. This is particularly the case when representing Immortal memory as a component which provides the `ObjectPool` service. The functional logic of a component must be aware of this service and in special cases is forced to directly communicate with it through an interface. Furthermore, as already discussed in [BCC+03], this introduces a burden on developers since a strict discipline within the component is required to recycle object references in an organized manner.

On the other hand, the authors of SOFA 2.0 [BHP06] claim that the code implementing the functional component should be aware of the control part; moreover, they state that the functional code should have access to the control part of the component. Although this could be seen

as a cross cutting concern that hopefully should be avoided according to the Fractal community, in the case of `ObjectPools` provided by the `Immortal Memory` it is necessary.

6.5 Related Work Comparison

In this section we confront the SOLEIL framework with the component frameworks presented in Section 3.2.3. In Table 6.3 we therefore repeat the characteristics of these frameworks and we compare them with SOLEIL.

	Compadres	Etienne et al.	SOFA HI	Golden Gate	SOLEIL
CBSE Criterions					
Component Model					
Communication Model					
Development Methodology					
Adaptation					
RTSJ Criterions					
Thread Model Support					
Memory Model Support					
Formalization and Validation					

Legend

Low Support — *High Support*

Table 6.3: Recapitulation and Comparison of Component Frameworks for RTSJ

The SOLEIL framework, summarized in the last column, represents a matured component model that is comparable to the state-of-the-art general-purpose component frameworks - e.g. SOFA or FRACTAL. Similarly, the communication model supports a wide range of communication types, others may be easily added. Furthermore, the framework proposes a development methodology, with both meets the requirements of general purpose methodologies but also provides solutions specific to RTSJ. However, only adaptation of non-real-time parts of applications is supported.

From the RTSJ perspective, the framework provides a support for RTSJ thread model. The concept of active and passive components is provided together with a higher level approach using the domain components (the `ThreadDomain` component). The framework further employs the domain components when addressing the memory model complexities, the solution is fostered by a wide set of supported patterns and idioms at the implementation level. Finally, since the component model is formalized in the Alloy language and we specify the SOLEIL profile, an approach to validation of component model instances and their implementations can be proposed.

6.6 Summary

In this chapter we have evaluated the SOLEIL and HULOTTE frameworks from several different perspectives. These evaluations were divided into three independent case studies, each focusing on different aspects of our proposal.

SweetFactory The main purpose of this case study was to illustrate basic challenges of RTSJ programming and also to illustrate the main ideas of our proposal along the course of the dissertation. Furthermore, we have also used the implementation of the SweetFactory to evaluate performance characteristics of our framework. Finally, as a part of the evaluation case study we have evaluated the code generation process implemented in the HULOTTE framework.

As the result of this case study, we have effectively demonstrated the basic ideas proposed by the SOLEIL framework. Also, performance benchmarks showed that the framework is not introducing any overhead comparing to manually implemented variant of the SweetFactory. Finally, from the code generation perspective we have discussed that the HULOTTE framework is meeting the set of requirements posed on code generators for RTSJ systems, notably we have achieved a full separation of concerns.

Real-time Collision Detector The purpose of the RCD case study was to evaluate the SOLEIL framework from a software engineering point of view, showing the benefits the software engineers will exploit when using the SOLEIL framework. As one of the means of evaluation, we have compared prior implementations of RCD with the application implemented in SOLEIL, showing better scalability of the solution, separation of concerns in practice, and mitigation of RTSJ programming complexities.

Distributed and Ambient Programming The purpose of this final case study was to evaluate extendability of the framework. We have chosen two domain specific challenges - distributed real-time computing and ambient computing. For both we have proposed new domain components, implemented their semantics, and demonstrated the benefits of the HULOTTE framework on simple illustration scenarios. The case studies showed how easily and intuitively the frameworks can be extended, evaluating their potential in more general scenarios.

In the next chapter, we will present perspectives and conclude.

Part IV

Conclusion and Perspectives

Chapter 7

Conclusion and Perspectives

Contents

In THIS concluding chapter, we revisit our research goals as stated in the introduction with hindsight and highlight the contributions of this dissertation once more. We discuss the rough edges to our proposal and outline those aspects of the proposal which may lead to interesting avenues of future research. We also point out other research which has been – to some extent – influenced by the work described in this dissertation.

7.1 Summary of the Dissertation

The vision behind this dissertation is to ultimately close the gap between real-time programming and today's software technology. Therefore, this dissertation investigates scalable software engineering techniques for RTSJ [BGB+00] programming. The fundamental philosophy that we adopt is to introduce high-level abstractions of RTSJ concepts in order to leverage development of real-time Java systems. The work in this dissertation lies within the intersection of three technological domains: real-time systems development, component-based software engineering (CBSE), and Real-time Specification for Java (RTSJ). Our basic goal is to reconcile traditional component-based software engineering abstractions in the context of RTSJ. Following this goal, the appropriate abstractions of RTSJ concerns should mitigate complexities related to the RTSJ programming model.

As the cornerstone of our approach, we introduce the concept of *domain component* that allows developers to express RTSJ concerns - e.g. real-time threads or memory areas, as special software components in the system. By abstracting RTSJ we thus permit to manipulate with its concerns as first-class entities along the whole development lifecycle. This approach ultimately provides a full separation of functional and RTSJ-specific concerns. The resulting applications are more transparent and can be easily tailored to various real-time conditions without modifying the functional implementation.

Furthermore, we embrace the concept of domain components in SOLEIL– a full fledged component framework for development of RTSJ-based applications. The framework proposes a development methodology providing a continuum between the design and implementation process where RTSJ concerns are transparently managed. The framework further mitigates the complexities of real-time programming by automatically generating the RTSJ-specific code, thus we achieve a development process that highly resembles the standard Java one. Additionally, we formalize the concept of domain components and the rules and restrictions introduced by RTSJ using the Alloy [Jac06] language into the SOLEIL profile. Based on the profile, we propose an approach to validate that developed architectures and implementations are in conformance to RTSJ.

Finally, we extend the SOLEIL framework with the HULOTTE tool set that provides a component-based approach to automatic instantiation of runtime execution platforms for applications developed in SOLEIL. The toolset can be easily extended with new domain components and their semantics in order to address various domain specific challenges, without the restriction only to RTSJ. Additionally, various optimization levels are provided to balance between functionality and performance of instantiated applications.

To validate the approach, we have conducted three case studies, each challenging our proposal from different perspectives. In the *SweetFactory* case study we have evaluated performance characteristics of our approach, showing that we introduce only a slight overhead in comparison to manually written object-oriented applications while providing more functionality. The *Real-time Collision Detector* case study evaluated the benefits of the approach for software engineers, we have showed that in comparison to different RTSJ programming models we bring better separation of concerns and significantly facilitate development of RTSJ programs. The Distributed and Ambient programming case study demonstrated ability of the *domain component* concept to be extended towards various domain-specific challenges. Furthermore, we have compared the SOLEIL framework with the state-of-the-art RTSJ frameworks and we have justify why SOLEIL surpasses them.

7.2 Contributions of the Dissertation

We further summarize the main contributions of our work and put them into the context of the goals stated in Section 3.5:

- **Domain Components** We have proposed this new kind of components that represents domain-specific concerns in systems. In this way we present these concerns as first-class entities and developers can manipulate with them along the whole development lifecycle. Domain components are used in our proposal to represent RTSJ features and thus provide an appropriate level of abstractions – meeting the **G1.1**. We thus achieve better separation of functional and non-functional concerns (represented as domain components) – **G1.2**.

- **RTSJ Component Model** Based on the concept of domain components we propose a component model embracing RTSJ concepts - **G1**. `ThreadDomain` and `MemoryArea` components are designed to alleviate the complexities of thread and memory models of RTSJ. Furthermore, we formalize the component model in Alloy in order to provide a background for validation methods – **G1.3**.

- **SOLEIL Framework** The SOLEIL framework further extends our contributions by introducing a development methodology - **G2.1**. The methodology provides a guiding principles to facilitate development of RTSJ systems. Furthermore, the SOLEIL profile defines rules that must be obeyed by developers in order to develop RTSJ compliant applications. Furthermore, based on the profile and on the formalization of the component framework we propose a validation process validating correctness of developed applications - **G1.3**.

- **HULOTTE Framework** The HULOTTE framework provides a unified approach to instantiation of runtime platforms for applications developed with the SOLEIL framework. The framework respects the separation of concerns by keeping the software components also at runtime. Furthermore, runtime platforms are instantiated through the methods of generative programming – **G2.2**. The functional code is fully separated from an automatically generated RTSJ-related one, both at development time and also at runtime. The HULOTTE framework also introduces optimization heuristics in order to mitigate the performance overhead caused by the componentization of applications – **G3.2**.

- **Evaluation** Our performance evaluations show that we deliver predictable applications – **G3.1** and the overhead of the framework is considerably reduced by the optimizations heuristics we implement – **G3.2**. Additionally, the presented case studies showed that development of RTSJ applications is substantially alleviated by separating the functional and RTSJ code, which is furthermore automatically generated – **G3.3** and **G3.4**.

7.3 Limitations of the Approach

We already highlighted specific technical shortcomings of our proposal in Section 6.4. Important as they may be, we will not repeat them here, but rather focus on the limitations of the concept of the SOLEIL and HULOTTE frameworks as a whole.

Activity View

As discussed in the previous section, the SOLEIL framework provides an effective approach to development of RTSJ-compliant applications. The complexities of the development process are mitigated and RTSJ-related code automatically generated, however, the framework does not guarantee that such applications will finally meet the requirements of their real-time model. RTSJ itself guarantees solely predictable execution of Java code, the SOLEIL framework is proposed to maximize the electivity of this process.

Analyzing real-time applications is fully in competence of real-time model construction tools (e.g. TIMES [AFM+02] and SYMPTA/S [HHJ+05], MAST [HGGM01], or MARTE [MTdS07]). SOLEIL however does not provide any support for such tools. Although, this was not stated as a goal for this dissertation, it can be considered as a limitation of the approach. We further address it in the perspectives of this dissertation, Section 7.5.

Runtime Adaptation

Our proposal is developed with a motivation to address the issues of adaptation of real-time systems. To achieve this, we can extensively benefit from the FRACTAL component framework that supports reliable ways of runtime reconfiguration [LLC07]. Our framework follows this approach by allowing developers to design containers supporting such way of reliable adaptation.

However, we meet the ultimate goal of adaptable real-time components only partially. As discussed in [WRM+05a, CL02], when addressing adaptability of real-time component systems, the runtime changes in the system must not jeopardize the end-to-end timing constraints. Furthermore, the reconfiguration operations must be deterministic [FSSC08]. Our approach does not support adaptation up to this extend.

Additionally, RTSJ characteristics further hamper adaptation of applications, since for example performing adaptation in the allocation context of an immortal memory could eventually lead to its exhaustion. To resolve this issue, the SOLEIL framework should be extended with formalisms defining precisely when the adaptation is possible with respect to the RTSJ restrictions. Possibly, a new domain component expressing which software component can be adapted should be proposed. Finally, to provide a support of adaptation of components residing in immortal and scoped memories, e.g. a wedge thread pattern [PFHV04] could be used to control life span of components.

7.4 Impact of the Dissertation

The impact of the dissertation can be put in two different perspectives: collaborations with other universities/research labs and new research topics inspired or influenced by the dissertation.

7.4.1 Collaborations

During the course of the dissertation, we have collaborated within several research projects.

Distributed Programming in Real-time Java

Within an INRIA internship funding, a visiting PhD student Michal Malohlava from DSRG[7] has been supervised. The goal of the internship was to provide a distributed programming support for Real-time Java. The work resulted in two publications [MPL+08, LMP+09], and is still continuing as a collaboration in the HULOTTE project. We further present the outcomes of this work as a case study in Section 6.3.1.

Ambient Programming in FRACTAL

In early stages of this dissertation a case-study oriented on challenges of ambient and ubiquitous programming has been conducted. The work was done in collaboration with Dr. Tom Van Cutsem from PROG[8]. As the result of the work, we have published a workshop paper [PMS07], furthermore, the outcomes were also presented as a case study in Section 6.3.2. Interested reader can find more information about the concept proposed by this work - the *ambient bindings*, in [Van08].

Carmen Project – Model Checking of Software Components

Furthermore, we have carried out the Carmen project [Pls06] - a software component model checker. In this project we have proposed a new approach to model checking of isolated software components [PA08]. This work was conducted in collaboration with Dr. Jiří Adámek and Dr. Pavel Parizek from DSRG, currently we prepare a journal publication on this topic.

7.4.2 Research Projects Influenced by the Dissertation

In this section, we highlight recent projects influenced by the work put forward in this dissertation.

Annotation Framework for Domain-Specific Component Applications

Inspired by the concept of domain components and by the HULOTTE framework, this research proposes an extension to the HULOTTE component framework that allows the definition and checking of domain-specific concerns. In this approach, from the components' architecture to

[7] Distributed Systems Research Group, Charles University, Prague, Czech Republic, http://dsrg.mff.cuni.cz/
[8] The Programming Technology Lab, VUB, Brussels, Belgium http://prog.vub.ac.be/doku.php

their implementation, concerns are defined and checked in an homogeneous manner. An interesting reminiscence of the concept of domains component is present here, instead of expressing the domain-specific requirements as components, they are represented by domain-specific annotations supported by an extensive annotation framework [ND08].

The work is conducted in collaboration between Dr. Frédéric Loiret from INRIA ADAM team and Dr. Carlos Nogura from SSEL[9] lab and have been published in [NL09]. Furthermore, we have applied the preliminary outcomes of this research in the SOLEIL framework to implement validation of software applications developed in our framework, see Section 4.3.4.

Real-time Java in Space

An ongoing research [KPV+09], supported by grants from ESA, is being carried out on the topics of using Real-time Java in onboard software for space missions. The research is conducted by the R&D lab at SciSys[10], DSRG, and the real-time Java research group from Purdue University[11]. During the course of this dissertation we have collaborated with or provided external consultations to some of these research groups.

7.5 Perspectives

In this section, we discuss how our research could be extended or studied in a different context without the emphasis on addressing limitations, as was the case in the Section 7.3.

7.5.1 Short Term Perspectives

Domain Components Taxonomy

In this dissertation we have introduced only a small set of domain components, moreover, these components were specifically targeted on RTSJ. We can however use this concept in a more sense and propose various types of domain components.

The growing family of domain components brings a demand for a taxonomy which will describe these components and will clarify their relations. As we have already discussed in Section 6.4, dependencies between different domain components exist and must be considered when applying these components. Taxonomy of domain components can however be easily constructed, since we can extend our component model with new domain components and clarify precisely their constraints in the Alloy language. As we have shown for the `DistributedNode` and `AmbientNode` domain components in Section 6.3.

Asynchronous Transfer of Control

Asynchronous transfer of control (ATC) is a mechanism that lets one thread throw an exception into another thread. The ATC was invented because some real-time programmers find this type of function crucial, other find it repulsive [Dib08]. ATC is considered as one of the most controversial element of RTSJ. Since RTSJ does not force programmers to learn to use ATC in order to use its other features, we have not covered this feature in our proposal.

ATC is usually used to terminate a thread or abort one iteration of a loop, however, since we manage the threads through the `ActivityController` presented in the container, we could implement the ATC mechanisms in this controller. However, to investigate potential impact of this feature to other parts of the system, we consider this problem as a future work.

[9] The System and Software Engineering lab, VUB, Brussels, Belgium http://ssel.vub.ac.be/ssel/
[10] http://www.scisys.co.uk/
[11] http://www.cs.purdue.edu/homes/jv/

7.5.2 Mid Term Perspectives

Runtime Adaptation of RTSJ-based Systems

A new collaboration emerging from our publication [PLMS08] is focused on runtime adaptation of RTSJ-based system. The project is based on the SOLEIL framework [PLMS08] and inspired by [FSSC08, FC08]. The goal is to provide a support for bounded adaptation of software components at runtime.

While the SOLEIL framework provides potentially a strong support for runtime adaptation of software components, inherited already from the FRACTAL component model, there is no support for an adaptation with regards to the real-time requirements of the system. This limitation was discussed in Section 7.3 and currently is being addressed by the research conducted in collaboration with Serena Fritsch[12].

7.5.3 Long Term Perspectives

Tinap

The Tinap project [LSSD09] has shown that component-oriented design facilitates extraction of activity diagrams that allow developers to reason about activities (composed of tasks and mapped to chains of component operations), synchronization, execution times, deadlines, etc. Based on the activity diagram, external validation tools can be used to ensure that developed applications are dead-lock free or they could provide performance and scheduling analysis.

Therefore, by extending the SOLEIL framework with Tinap, we could bridge the gap between effective development of RTSJ-compliant applications, provided by SOLEIL, and development of real-time applications that are schedulable and in conformance to their real-time model.

Model Checking of RTSJ-Components

Work is currently in progress to exploit the idea of formal verification of real-time Java software components. Here, the goal is to develop approach to model checking of applications implemented in Real-time Java. The research is inspired by the initial study published [G. 05] and the goal is to exploit the results of the Carmen project - our previous research [PA08]. We plan to extend the SOLEIL framework with Behavior protocols [F. 02] using their new version [KPS08] to specify the contracts between components and we consequently model check these properties by the Java PathFinder model checker extended towards RTSJ. Work is conducted jointly in collaboration with Dr. Pavel Parizek from DSRJ and Dr. Tomáš Kalibera from Purdue University.

7.6 Publications

The outcomes of this dissertation appear in the following publications.

7.6.1 International Conferences

1 Frédéric Loiret, Michal Malohlava, Aleš Plšek, Philippe Merle, Lionel Seinturier. *Constructing Domain-Specific Component Frameworks through Architecture Refinement*. In Proceedings of Euromicro SEAA 2009, August 2009, Patras, Greece. *(To appear.)* [LMP+09]

2 Aleš Plšek, Frédéric Loiret, Philippe Merle, Lionel Seinturier. *A Component Framework for Java-based Real-time Embedded Systems*. In Proceedings of ACM/IFIP/USENIX 9th International Middleware Conference, volume 5346/2008, pages 124–143, Leuven, Belgium, December 2008. IEEE Computer Society; *Rank A conference, Acceptance rate: 18%* [PLMS08]

[12] Distributed Systems Group, Trinity College, Dublin, Ireland, http://www.dsg.cs.tcd.ie/

3 Aleš Plšek, Jirí Adámek. *Carmen: Software Component Model Checker*. In Proceedings of Fourth International Conference on the Quality of Software-Architectures (QoSA'08), October 2008, Karlsruhe, Germany; *Rank A conference, Acceptance rate: 33%* [PA08]

4 Aleš Plšek, Philippe Merle, Lionel Seinturier. *A Real-Time Java Component Model*. In Proceedings of the 11th International Symposium on Object/Component/Service-oriented Real-Time Distributed Computing (ISORC'08), pages 281–288, Orlando, Florida, USA, May 2008. IEEE Computer Society; *Acceptance rate: 33%* [PMS08]

7.6.2 International Workshops

5 Michal Malohlava, Aleš Plšek, Frédéric Loiret, Philippe Merle and Lionel Seinturier. *Introducing Distribution into a RTSJ-based Component Framework*. In RNTS 2008: 2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC'08), October 2008, Rennes France. [MPL+08]

6 Aleš Plšek, Philippe Merle, Lionel Seinturier. *Ambient-Oriented Programming in* FRACTAL . In ECOOP 2007: 3rd Workshop on Object Technology for Ambient Intelligence, July 2007, Berlin, Germany. [PMS07]

7.6.3 Poster Sessions

7 Aleš Plšek, Frédéric Loiret, Michal Malohlava, Philippe Merle, Lionel Seinturier. *Soleil: A Component Framework for Java-based Real-time Embedded Systems.* OW2 Annual Conference 2009: FRACTAL Poster Session, Paris, France, April 2009.

8 Aleš Plšek, Frédéric Loiret, Michal Malohlava, Philippe Merle, Lionel Seinturier. *Soleil: A Component Framework for Java-based Real-time Embedded Systems.* GDR-GPL 2009, Toulouse, France, January 2009.

7.6.4 Presentations

Besides the publications above, the results of the thesis have been presented in the following invited talks.

9 Aleš Plšek *Real-time Programming in Java*. Invited talk at Czech Java User Group, November 2008, Prague, Czech Republic.

10 Aleš Plšek, Frédéric Loiret, Michal Malohlava, Philippe Merle, Lionel Seinturier. *Soleil: A Component Framework for Java-based Real-time Embedded Systems*. Distributed Systems Research Group, Charles University, November 2008, Prague, Czech Republic.

11 Aleš Plšek, Frederic Loiret, Michal Malohlava, Philippe Merle, Lionel Seinturier. *Applying Component- based Software Engineering in On-board Software*. Invited talk at SciSys, October 2008, Bristol, UK.

12 Aleš Plšek, Philippe Merle, Lionel Seinturier. *A Real-Time Java Component Model*. Invited talk at Cinvestav, I.P.N., May 2008, Mexico City, Mexico.

13 Aleš Plšek, Philippe Merle, Lionel Seinturier. *Evolvable Middleware Container Architectures for Distributed Embedded Systems*. RUNES Summer School, July 2007, London, UK.

14 Aleš Plšek, Philippe Merle, Lionel Seinturier. *Evolvable Middleware Container Architectures for Distributed Embedded Systems*. Junior Seminar, CALA Days, February 19-20, 2007, Vrije Universiteit Brussel (VUB), Belgium.

Bibliography

[A. 05] A. Corsaro, C. Santoro. The Analysis and Evaluation of Design Patterns for Distributed Real-Time Java Software. *16th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005. 2, 17, 19, 21, 41, 54

[ABB+07a] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time java virtual machine. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 249–258, New York, NY, USA, 2007. ACM. 1, 20

[ABB+07b] Joshua Auerbach, David F. Bacon, Florian B¨omers, , and Perry Cheng. Real-time Music Synthesis in Java Using the Metronome Garbage Collector. *In Proceedings of the International Computer Music Conference*, August 2007. 1, 21

[ABC+07] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007. 1, 20, 21

[ABG+08] Joshua Auerbach, David F. Bacon, Rachid Guerraoui, Jesper Honig Spring, and Jan Vitek. Flexible Task Graphs: a unified restricted thread programming model for Java. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 1–11, New York, NY, USA, 2008. ACM. 19, 22, 97

[ACG+07] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped Types and Aspects for Real-time Java Memory Management. *Real-Time Syst.*, 37(1):1–44, 2007. 19, 21, 97, 99

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - a tool for modelling and implementation of embedded systems. In Springer-Verlag, editor, *8th International Conference, TACAS 2002, part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002*, pages 460–464, Grenoble, France, April 2002. Lecture Notes in Computer Science, Vol.2280. 11, 121

[AJ06] Jonathan S. Anderson and E. Douglas Jensen. Distributed Real-Time Specification for Java: a Status Report (digest). In *Proceedings of the 4^{th} International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '06)*, pages 3–9, New York, NY, USA, 2006. ACM. 35, 105, 106

[Ale09] Aleš Plšek. Soleil and Hulotte Project Web Page, 2009. http://adam.lille.inria.fr/soleil/ as of June 2009. 56, 80, 141

[All09] OSGi Alliance. OSGi Service Platform Release 4. as of June 2009. http://www.osgi.org/. 30

[AMPN+06] Alonso Alejandro, Bianconi Maria-Paola, Francois Nicolas, Cortese Giovanni, and Yu Erik. Flexible Java Real-Time Profile for Business-Critical Systems. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 135–143, New York, NY, USA, 2006. ACM. 109

[Ang02] Angelo Corsaro and Doug Schmidt. The design and performance of the jRate Real-Time Java implementation. 2002. 20

[APPZ04] R. Aigner, C. Pohl, M. Pohlack, and S. Zschaler. ailor-made containers: Modeling non-functional middleware service. In *In Workshop on Models for Non-functional Aspects of Component-Based Software (NfC'04) at UML conference*, 2004. 25

[BB08] Antoine Bourre and Francois Blarel. Alloy4Eclipse Plug-In. 2008. http://code.google.com/p/alloy4eclipse/. 38, 96

[BCC+03] Greg Bollella, Tim Canham, Vanessa Carson, Virgil Champlin, Daniel Dvorak, Brian Giovannoni, Mark Indictor, Kenny Meyer, Alex Murray, and Kirk Reinholtz. Programming with non-heap memory in the real time specification for java. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 361–369, New York, NY, USA, 2003. ACM. 2, 17, 19, 21, 33, 76, 113

[BCF+06] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, Austin Armbruster, Edward Pla, and David Holmes. A Real-time Java Virtual Machine for Avionics - An Experience Report. *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006. 20

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software: Practice and Experience*, 36:1257 – 1284, 2006. 23, 24, 27, 35, 47, 80

[BCS09] Gordon S. Blair, Thierry Coupaye, and Jean-Bernard Stefani. Component-based Architecture: the Fractal Initiative. *Annales des Télécommunications*, 64(1-2):1–4, 2009. 27

[BEA06] BEA. Weblogic Real-Time. www.bea.com. 2006. 1, 17, 21

[BGB+00] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000. 1, 9, 13, 119

[BHM09] Tomas Bures, Petr Hnetynka, and Michal Malohlava. Using a Product Line for Creating Component Systems. In *Proceedings of ACM SAC 2009, Honolulu, Hawaii, U.S.A., Mar 2009*, March 2009. 27, 83, 85

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *SERA '06: Proc. of the 4th International Conference on Software Engineering Research, Management and Applications*, pages 40–48, USA, 2006. IEEE Computer Society. 24, 28, 113

[BKT+06] Krishnakumar Balasubramanian, Arvind S. Krishna, Emre Turkay, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha S. Gokhale, and Douglas C. Schmidt. Applying model-driven development to distributed real-time and embedded avionics systems. *IJES*, 2(3/4):142–155, 2006. 12

[BN03] E. G. Benowitz and A. F. Niessner. A Patterns Catalog for RTSJ Software Designs. *Lecture notes in Computer Science*, 2003. 2, 17, 19, 21, 34, 41, 54, 78

[Boe05] Boeing and Insitu. ScanEagle, 2005. http://www.insitu.com/scaneagle as of June 2009. 21

[Bou95] R Bourgonjon. The evolution of embedded software in consumer products. *International Conference on Engineering of Complex Computer Systems*, 1995. 1

[BP04] Tomas Bures and Frantisek Plasil. *Communication Style Driven Connector Configurations*, pages 102–116. 2004. 26

[BR02] Greg Bollella and Krik Reinholtz. A Real-Time RMI Framework for the RTSJ. In *In Proceedings of the Fifth International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC02)*, 2002. 15

[BSE+04] F. Budinsky, D. Steinberg, R. Ellersick, E. Merks, S.A. Brodsky, and T.J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2004. 63, 81

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-wise Refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society. 68, 80

[Bur06] Tomas Bures. *Generating Connectors for Homogeneous and Heterogeneous Deployment*. PhD thesis, Department of Sofware Engineering, Mathematical and Physical Faculty, Charles University, Prague, 2006. 26, 106, 107

[But05] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Springer, 2nd Edition, 2005. 11

[BV07] Matteo Bordin and Tullio Vardanega. Real-time Java from an Automated Code Generation Perspective. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 63–72, New York, NY, USA, 2007. ACM. 21, 91, 92, 93, 95, 97

[BVGVEA05] P. Basanta-Val, M. Garcia-Valls, and I. Estevez-Ayres. Towards the Integration of Scoped Memory in Distributed Real-Time Java. In *Proceedings of the 8^{th} IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, pages 382–389, 2005. 35

[BVGVEADK06] Pablo Basanta-Val, Marisol García-Valls, Iria Estevez-Ayres, and Carlos Delgado-Kloos. Extended portal: violating the assignment rule and enforcing the single parent rule. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 30–37, New York, NY, USA, 2006. ACM. 21

[BW01] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 13

[BW03] A. Borg and A. Wellings. A Real-Time RMI Framework for the RTSJ. In *Proceedings of the 15^{th} Euromicro Conference on Real-Time Systems.*, pages 238–246, July 2003. 35

[CBG+08] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A Generic Component Model for Building Systems Software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008. 24, 27

[CC03] Angelo Corsaro and Ron K. Cytron. Efficient Memory Reference Checks for Real-Time Java. In *In Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003. 20

[CCL06] Ivica Crnkovic, Michel R. V. Chaudron, and Stig Larsson. Component-Based Development Process and Component Lifecycle. In *ICSEA*, page 44, 2006. 23, 25, 59

[CCPS03] V. Cechticky, P. Chevalley, A. Pasetti, and W. Schaufelberger. A Generative Approach to Framework Instantiation. *Proceedings of GPCE*, pages 267–286, September 2003. 27, 30

[CDM+05] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu, and Kefei Wang. An empirical study of programming language trends. *IEEE Softw.*, 22(3):72–78, 2005. 1, 13

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 26, 60, 83

[CL02] I. Crnkovic and S. Larsson. *Building Reliable Component-based Systems*. Addison-Wesley Professional, Boston, 2002. 28, 82, 84, 121

[Cle02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd ed.* Addison-Wesley Professional, Boston, 2002. 2, 24, 83

[CS04] A. Corsaro and C. Santoro. Design Patterns for RTSJ Application Development. *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, 2004. 34, 78

[Dau07] Jean-Marie Dautelle. Fully Time Deterministic Java. In *AIAA Space 2007*, 2007. http://www.osgi.org/. 62

[Dav08] Yann Davin. F4E: FRACTAL plug-in for Eclipse. 2008. http://fractal.ow2.org/f4e/index.html as of June 2009. 37, 96

[Daw08] Michael H. Dawson. Challenges in Implementing the Real-Time Specification for Java (RTSJ) in a Commercial Real-Time Java Virtual Machine. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:241–247, 2008. 1, 17

[DBC+04] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. In *ISORC*, pages 15–22, 2004. 33, 41

[DEM02] Frédéric Duclos, Jacky Estublier, and Philippe Morat. Describing and using non functional aspects in component based applications. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 65–75, New York, NY, USA, 2002. ACM. 25, 83

[DHT01] Eric M. Dashofy, André Van der Hoek, and Richard N. Taylor. A highly-extensible, xml-based architecture description language. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 103, Washington, DC, USA, 2001. IEEE Computer Society. 83

[Dib08] Dr. Peter C Dibble. *Real-Time Java Platform Programming: Second Edition*. BookSurge Publishing, 2008. 9, 13, 123

[dNBR06] Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar. Model-Based Development of Embedded Systems: The SysWeaver Approach. In *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 231–242, Washington, DC, USA, 2006. IEEE Computer Society. 28

[DR04] Daniel L. Dvorak and William K. Reinholtz. Hard Real-time: C++ versus RTSJ. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 268–274, New York, NY, USA, 2004. ACM. 31, 33

[E. 00] E. Clarke, O.Grumberg, and D.Peled. Model Checking. *MIT Press*, Jan 2000. 27

[ECB06] J.P. Etienne, J. Cordry, and S. Bouzefrane. Applying the CBSE Paradigm in the Real-Time Specification for Java. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 218–226, USA, 2006. ACM. 32, 41

[EM08] George Edwards and Nenad Medvidovic. A methodology and framework for creating domain-specific development infrastructures. In *ASE*, pages 168–177. IEEE, 2008. 68

[EPFD01] Denis Conan Erik, Erik Putrycz, Nicolas Farcet, and Miguel Demiguel. Integration of non-functional properties in containers. In *In Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP*, 2001. 25

[F. 02] F. Plasil, S. Visnovsky. Behavior Protocols for Software components. *IEEE Transactions on Software Engineering*, 28, no. 11, Nov 2002. 124

[FC08] Serena Fritsch and Siobhán Clarke. Timeadapt: timely execution of dynamic software reconfigurations. In *MDS '08: Proceedings of the 5th Middleware doctoral symposium*, pages 13–18, New York, NY, USA, 2008. ACM. 124

[FDDM08] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the Grid with DeployWare. In *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID'08)*, pages 177–184, Lyon, France, may 2008. IEEE. 37

[FLV06] Peter H. Feiler, Bruce A. Lewis, and Steve Vestal. The SAE Architecture Analysis and Design Language (AADL) a standard for engineering performance critical systems. pages 1206–1211, 2006. 28

[FS07] Mike Fulton and Mark Stoodley. Compilation techniques for real-time java programs. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 221–231, Washington, DC, USA, 2007. IEEE Computer Society. 1, 20

[FSSC08] Serena Fritsch, Aline Senart, Douglas C. Schmidt, and Siobhán Clarke. Time-bounded Adaptation for Automotive System Software. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 571–580, New York, NY, USA, 2008. ACM. 24, 83, 121, 124

[G. 05] G. Lindstrom, P. Mehlitz and W. Visser. Model Checking Real Time Java Using JavaPathfinder. *Proceedings of the Third International Symposium on Automated Technology for Verification and Analysis (ATVA)*, October 2005. 124

[GDFSB08] Ning Gui, Vincenzo De Flori, Hong Sun, and Chris Blondia. A framework for adaptive real-time applications: the declarative real-time osgi component model. In *ARM '08: Proceedings of the 7th workshop on Reflective and adaptive middleware*, pages 35–40, New York, NY, USA, 2008. ACM. 30

[Geo99] Phipps Geoffrey. Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.*, 29(4):345–358, 1999. 1, 13

[GER08] Eli Gjørven, Frank Eliassen, and Romain Rouvoy. Experiences from Developing a Component Technology Agnostic Adaptation Framework. In *CBSE*, pages 230–245, 2008. 24, 83

[GHMS07] C. Gough, A. Hall, H. Masters, and A. Stevens. Real-Time Java: Writing and Deploying Real-Time Java Applications, 2007. http://www.ibm.com/developerworks/java/library/j-rtj5/. 17, 93

[GK03] A. Gaddah and T. Kunz. A Survey of Middleware Paradigms for Mobile Computing. *Carleton University Systems and Computing Engineering Technical Report SCE-03-13*, 2003. 110

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997. 81

[Gre05] Gregory Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, and Fr´ed´eric Parain. Mackinac: Making Hotspot real-time. 2005. 20

[Gro03] IST Advisory Group. Ambient Intelligence: From Vision to Reality. 2003. 110

[HACT04] Hans Hansson, Mikael Akerholm, Ivica Crnkovic, and Martin Torngren. SaveCCM - A Component Model for Safety-Critical Real-Time Systems. In *EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference*, pages 627–635, Washington, DC, USA, 2004. IEEE Computer Society. 24, 83

[HGCK07] J. Hu, S. Gorappa, J. A. Colmenares, and R. Klefstad. Compadres: A Lightweight Component Middleware Framework for Composing Distributed, Real-Time, Embedded Systems with Real-Time Java. *In Proc. ACM/IFIP/USENIX 8th Int'l Middleware Conference (Middleware 2007)*, Vol. 4834:41–59, 2007. 31, 41, 93

[HGGM01] M. González Harbour, J.J. Gutiérrez García, J.C. Palencia Gutiérrez, and J.M. Drake Moyano. Mast: Modeling and analysis suite for real time applications. *Real-Time Systems, Euromicro Conference on*, 0:0125, 2001. 11, 29, 121

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. In *Computers and Digital Techniques, IEE Proceedings*, volume 152, Issue 2, pages 148–166, Mar 2005. 11, 121

[HT08] M. Teresa Higuera-Toledano. Making stronger and Flexible the Single Parent Rule in the Real-Time Specification of Java. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 19–28, New York, NY, USA, 2008. ACM. 16, 21

[IBM07] IBM. DDG 1000 Next Generation Navy Destroyers, http://www-03.ibm.com/press/us/en/pressrelease/21033.wss. 2007. 1, 17, 21

[IH92] Jesús A. Izaguirre and Scott S. Hampton. Software considerations in airborne systems and equipment certification, do-178-b/ed-12-b, requirements and technical concepts for aviation/european organisation for civil aviation equipment. *Journal of Computational Physics*, 200:581–604, 1992. 1, 96

[J. 06] J. A. Colmenares, S. Gorappa, M. Panahi, and R. Klefstad. A Component Framework for Real-time Java. *12th IEEE Real-time and Embedded Technology and Applications Symposium, Work-In-Progress session*, 2006. 31

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. 38, 46, 120

[JASB07] Nicolas Juillerat, Stefan Müller Arisona, and Simon Schubiger-Banz. Real-time, low latency audio processing in Java. *In Proceedings of the International Computer Music Conference*, August 2007. 1, 21

[JCS+05] J.Dedecker, T. Van Cutsem, S.Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-Oriented Programming. *In "OOPSLA '05: Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications"*, 2005. 109, 110

[Jen07] E. Douglas Jensen's. Overview of Fundamental Real-Time Concepts and Terms. *http://www.real-time.org/realtimeoverview.htm*, 19 Feb 2007. 1, 10

[JHA+05] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Dmitriy Kopylenko. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK, 2005. 83

[JKSS04] Greenfield Jack, Short Keith, Cook Steve, and Kent Stuart. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004. 26

[JS00] Daniel Jackson and Kevin Sullivan. COM revisited: Tool-Assisted Modelling of an Architectural Framework. *SIGSOFT Softw. Eng. Notes*, 25(6):149–158, 2000. 38

[KM08] Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In *MoDELS 2008*, LNCS 5301, pages 690–704, 2008. 38, 59, 63, 96

[KPS08] Jan Kofron, Tomas Poch, and Ondrej Sery. TBP: Code-Oriented Component Behavior Specification. In *32nd Annual IEEE Software Engineering Workshop*, 2008. 124

[KPV+09] Tomas Kalibera, Filip Pizlo, Jan Vitek, Marek Prochazka, M. Zulianello, and Martin Decky. Real-Time Java in Space: Potential Benefits and Open Challenges. In *Proceedings of Data Systems in Aerospace (DASIA 2009)*, Jan 2009. 29, 123

[KRP+93] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-time Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. 11

[Lam05] David Lammers. Programming Tools: IBM sets real-time tempo for Java code with Metronome, http://www.eetimes.com/showArticle.jhtml?articleID=170701426&printable=true&printable=truev. EETimes.com, 2005. 2

[LLC07] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable Dynamic Reconfigurations in the Fractal Component Model. In *In Proc. 6th Workshop on Adaptive and Reflective Middleware (ARM2007) at Middleware*. ACM Digital Library, November 2007. 25, 28, 121

[LMP+09] Frederic Loiret, Michal Malohlava, Ales Plsek, Philippe Merle, and Lionel Seinturier. Constructing Domain-Specific Component Frameworks through Architecture Refinement. In *Euromicro SEAA 2009*, 2009. 4, 80, 122, 124

[LOQS07] Matthieu Leclercq, Ali Erdem Ozcan, Vivien Quema, and Jean-Bernard Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 209–219, Washington, DC, USA, 2007. IEEE Computer Society. 37, 56, 81, 96

[LP08] Olivier Lobry and Juraj Polakovic. Controlling the Performance Overhead of Component-Based Systems. In *Software Composition*, pages 149–156, 2008. 26, 82, 93

[LSSD09] Frédéric Loiret, Lionel Seinturier, David Servat, and Laurence Duchien. Tinap: A Component Design Framework for Multitasking Applications. In *Submitted for ISORC'09*, 2009. 124

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Trans. Softw. Eng.*, 33(10):709–724, 2007. 27

[McM04] Robert McMillan. GM CTO Sees More Code on Future Cars.", 2004. `http://www.infoworld.com/` as of June 2009. 1

[MDP] P. Mantegazza, E. L. Dozio, and S. Papacharalambous. Rtai: Real time application interface. *Linux J.*, page 10. 30

[MDT03] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. The Role of Middleware in Architecture-Based Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367–393, 2003. 26, 72, 107, 113

[MG07] Ingo Molnar and Thomas Gleixner. The RT-PREEMPT patch set for Linux, available at `http://www.kernel.org/pub/linux/kernel/projects/rt/`. 2007. 1, 20, 21, 94

[Mic] SUN Microsystems. Enterprise JavaBeans. http://java.sun.com/ejb. 25, 83

[MMP00] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 178–187, New York, NY, USA, 2000. ACM. 106

[Mor06] Gabriel A. Moreno. Creating custom containers with generative techniques. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 29–38, New York, NY, USA, 2006. ACM. 25, 83, 84

[MPL+08] Michal Malohlava, Aleš Plšek, Frédéric Loiret, Philippe Merle, and Lionel Seinturier. Introducing Distribution into a RTSJ-based Component Framework. In *In RNTS 2008: 2nd Junior Researcher Workshop on Real-Time Computing (JRWRTC'08)*, Rennes, France, October 2008. 4, 83, 105, 122, 125

[MS08] Philippe Merle and Jean-Bernard Stefani. A formal specification of the Fractal component model in Alloy. In *INRIA technical report, inria-00338987, version 1*, November 2008. 38

[MTdS07] Faugere Madeleine, Bourbeau Thimothee, Simone Robert de, and Gerard Sebastien. Marte: Also an uml profile for modeling aadl applications. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 359–364, Washington, DC, USA, 2007. IEEE Computer Society. 121

[MWP+08] Prochazka M., R. Ward, Tuma P., Hnetynka P., and Adamek J. A Component-Oriented Framework for Spacecraft On-Board Software. In *DASIA 2008, DAta Systems In Aerospace, European Space Agency Report Nr. SP-665, ISBN 978-92-9221-229-2*, Palma de Mallorca, Spain, May 2008. 29

[NB03] Albert F. Niessner and Edward G. Benowitz. RTSJ Memory Areas and Their Affects on the Performance of a Flight-like Attitude Control System. In *In Proceedings of the International workshop on Java technologies for real-time and embedded systems (JTRES)*, pages 508–519, 2003. 21

[ND08] Carlos Noguera and Laurence Duchien. Annotation framework validation using domain models. In Ina Schieferdecker and Alan Hartman, editors, *ECMDA-FA*, volume 5095 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008. 123

[Nil04] Kelvin Nilsen. Quantitative Analysis of Developer Productivity in C vs. Real-Time Java. *Defense Advanced Research Projects Agency Workshop on Real-Time Java*, 2004. 1, 13, 99

[Nil06] Kelvin Nilsen. A type system to assure scope safety within safety-critical java modules. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 97–106, New York, NY, USA, 2006. ACM. 19, 21

[Nil07] Kelvin Nilsen. Applying COTS Java Benefits to Mission-Critical Real-Time Software. *Crosstalk The Journal of Defense Software Engineering*, pages 19–24, June 2007. 21, 31

[NL09] Carlos Noguera and Frederic Loiret. Checking Architectural and Implementations Constraints for Domain-Specific Component Frameworks using Models. In *In Proceedings of Euromicro SEAA 2009*, August 2009. 63, 80, 123, 147

[NLLT08] Pessemier Nicolas, Seinturier Lionel, Duchien Laurence, and Coupaye Thierry. A Component-based and Aspect-Oriented Model for Software Evolution. *Int. Journal of Computer Applications in Technology*, 31(1/2):94–105, 2008. 37, 50

[OMG] OMG. Real-time CORBA. `http://www.omg.org/technology/documents/formal/real-time_CORBA.htm`. 35

[OW209a] OW2. Fractal Specification. In *`http://http://fractal.ow2.org/`*, 2009. 27

[OW209b] OW2. Fractal Specification. In *`http://http://fractal.ow2.org/specification/`*, 2009. 37, 64

[PA08] Aleš Plšek and Jiří Adámek. Carmen: Software component model checker. In *QoSA '08: Proceedings of the 4th International Conference on Quality of Software-Architectures*, pages 71–85, Berlin, Heidelberg, 2008. Springer-Verlag. 27, 122, 124, 125

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972. 2

[Paw06] Renaud Pawlak. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distributed Systems Online*, 7(11), 2006. 63, 83

[PFHV04] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-Time Java Scoped Memory: Design Patterns and Semantics. In *Proceedings of the 7^{th} IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 101–110, 2004. 2, 16, 17, 19, 21, 32, 34, 41, 54, 78, 79, 97, 99, 122

[PFP08] M. Prochazka, S. Fowell, and L. Planche. DisCo Space-Oriented Middleware: Architecture of a Distributed Runtime Environment for Complex Spacecraft On-Board Applications. In *4th European Congress on Embedded Real-Time Software (ERTS 2008)*, Toulouse, France, 2008. 29

[PLMS08] Aleš Plšek, Frédéric Loiret, Philippe Merle, and Lionel Seinturier. A Component Framework for Java-based Real-time Embedded Systems. In *Proceedings of ACM/IFIP/USENIX 9^{th} International Middleware Conference*, volume 5346/2008, pages 124–143, Leuven, Belgium, December 2008. IEEE Computer Society. 4, 24, 83, 124

[Pls06] A. Plsek. Extending Java PathFinder with Behavior Protocols, August 2006. Master Thesis, available at `http://www.lifl.fr/~plsek/Projects/Carmen/Download/Documents/masterThesis.pdf`. 122

[PMPL08] Lopez Martinez Patricia, Drake Jose M., Pacheco Pablo, and Medina Julio L. Ada-ccm: Component-based technology for distributed real-time systems. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 334–350, Berlin, Heidelberg, 2008. Springer-Verlag. 28, 29

[PMS07] Aleš Plšek, Philippe Merle, and Lionel Seinturier. Ambient-Oriented Programming in Fractal. In *Proceedings of the 3^{rd} Workshop on Object Technology for Ambient Intelligence at ECOOP*, July 2007. 4, 36, 105, 109, 122, 125

[PMS08] Aleš Plšek, Philippe Merle, and Lionel Seinturier. A Real-Time Java Component Model. In *Proceedings of the 11^{th} International Symposium on Object/Component/Service-oriented Real-Time Distributed Computing (ISORC'08)*, pages 281–288, Orlando, Florida, USA, May 2008. IEEE Computer Society. 4, 125

[PV03] Krzysztof Palacz and Jan Vitek. Java Subtype Tests in Real-Time. In *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 378–404, July 2003. 20

[PV06] Filip Pizlo and Jan Vitek. An empirical evaluation of memory management alternatives for Real-time Java. In *In Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS)*, pages 248–254, December 2006. 17, 19

[PV08] Filip Pizlo and Jan Vitek. Memory Management for Real-Time Java: State of the Art. In *2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 248–254, Orlando, Florida, 2008. 17

[RFP08] Marco Roccetti, Stefano Ferretti, and Claudio E. Palazzi. The brave new world of multiplayer online games: Synchronization issues with smart solutions. In

ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, pages 587–592, Washington, DC, USA, 2008. IEEE Computer Society. 1

[RG02] Mark Richters and Martin Gogolla. OCL: Syntax, Semantics, and Tools. pages 447–450. 2002. 56, 63

[RHN+07] Sven Gestegård Robertz, Roger Henriksson, Klas Nilsson, Anders Blomdell, and Ivan Tarasov. Using real-time java for industrial robot control. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 104–110, New York, NY, USA, 2007. ACM. 1, 21

[RM09] Romain Rouvoy and Philippe Merle. Leveraging component-based software engineering with fraclet. *Annales des Télécommunications*, 64(1-2):65–79, 2009. 37, 147

[RPV+06] O. Rohlik, A. Pasetti, T. Vardanega, V. Cechticky, and M. Egli. A UML2 Profile for Reusable and Verifiable Software Components for Real-Time Applications. In *International Conference on Software Reuse*, Torino, Italy, June 2006. 28, 30

[RZP+05] Krishna Raman, Yue Zhang, Mark Panahi, Juan A. Colmenares, Raymond Klefstad, and Trevor Harmon. RTZen: Highly Predictable, Real-Time Java Middleware for Distributed and Embedded Systems. In *Middleware 2005*, pages 225–248, December 2005. 35, 108, 112

[Sch99] Douglas C. Schmidt. Why Software Reuse has Failed and How to Make It Work for You. *C++ Report magazine*, 1999. 25

[Sie99] F. Siebert. Real-time garbage collection in multi-threaded systems on a single processor. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 277, Washington, DC, USA, 1999. IEEE Computer Society. 1, 17

[Sie04] Fridtjof Siebert. The impact of realtime garbage collection on realtime Java programming. In *In Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 33–40, 2004. 17

[SPDC06] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153, Vasteras, Sweden, Jun 2006. Springer. 22, 25, 36, 72

[SR98] John A. Stankovic and K. Ramamritham, editors. *Tutorial on Hard Real-Time Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998. 10

[SR08] SUN and Returs. Thomson Reuters Market Data Platform, 2008. http://www.sun.com/solutions/landing/industry/financial_services.xml as of June 2009. 1, 21

[SUN08] SUN. Sun Java Real-time System, 2008. http://java.sun.com/javase/technologies/realtime/ as of June 2009. 1, 17

[SVB+08] Séverine Sentilles, Aneta Vulgarakis, Tomás Bures, Jan Carlson, and Ivica Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In *CBSE*, pages 310–317, 2008. 24, 28, 83

[TAdM07] Daniel Tejera, Alejandro Alonso, and Miguel A. de Miguel. RMI-HRT: Remote Method Invocation - Hard Real Time. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, pages 113–120, New York, NY, USA, 2007. ACM. 35, 105

[TDB+07] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and Pascal Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1 (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. 29

[Van08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008. 122

[Vit08] Jan Vitek. Programming models for concurrency and real-time. In *TiC'08: Second International School on Trends in Concurrency*, 2008. 12

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000. 24, 28, 83

[WCJW02] A. Wellings, R. Clark, D. Jensen, and D. Wells. A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation. In *Proceedings of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '02).*, pages 13–22, 2002. 35, 105, 106

[Wel04] Andrew Wellings. Concurrent and Real-Time Programming in Java. 2004. 13, 16

[WHJ04] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Preemption-based avoidance of priority inversion for java. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing*, pages 529–538, Washington, DC, USA, 2004. IEEE Computer Society. 15, 54

[WP03] Andy Wellings and Peter Puschner. Evaluating the expressive power of the Real-Time Specification for Java. *Real-Time Systems*, 24(3):319—-359, 2003. 15

[WRM+05a] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-Time Component-based Systems. *in Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, March 2005. 28, 121

[WRM+05b] Shengquan Wang, Sangig Rho, Zhibin Mai, Riccardo Bettati, and Wei Zhao. Real-time component-based systems. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 428–437, Washington, DC, USA, 2005. IEEE Computer Society. 28

[WSBR01] Jr. William S. Beebee and Martin Rinard. An implementation of scoped memory for Real-Time Java. In *In Embedded Software Implementation Tools for Fully Programmable Application Specific Systems (EMSOFT)*, pages 289–305, 2001. 20

[ZNV04] Tian Zhao, James Noble, and Jan Vitek. Scoped Types for Real-Time Java. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society. 20, 97, 99

[ZPH08] Bechir Zalila, Laurent Pautet, and Jérôme Hugues. Towards automatic middleware generation. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 221–228, Washington, DC, USA, 2008. IEEE Computer Society. 26, 27, 108

Appendixes

Appendix A

Alloy Formalization of the RTSJ metamodel

The full Alloy specification of SOLEIL component model can be downloaded from [Ale09]. In total we have defined around 400 signatures, facts, and predicates in 4.5KLoC in order to fully specify the component model together with its composition and RTSJ-related rules.

```
abstract sig Component{
  externalInterfaces : set Interface,
  superComponents    : set Composite
}

sig Interface {
  owner     :    one Component,
  boundTo   :   lone Interface,
  type      :    one Type,
  binding   :    one Binding
}{
  boundTo != this
}

sig Attribute{}

sig Binding {
  client            : one Interface,
  server            : one Interface,
  communicationType : one CommunicationType,
  attributes        : set Attribute
}

sig CommunicationType, Type {}

one sig Asynchronous, Synchronous
        extends CommunicationType {}

one sig Client, Server extends Type {}

sig Composite in Component {
  internalInterfaces  : set Interface,
  subComponents       : set Component,
}

sig Primitive in Component {
  content : lone PrimitiveObject
}

fact ComponentIsCompositeOrPrimitive {
  all c:Component |
        (c in Composite or c in Primitive)
    and (
       c in Composite implies
         c not in Primitive )
    and (
      c in Primitive implies
        c not in Composite )
}
```

```
sig FunctionalComponent extends Component{}

abstract sig DomainComponent
                  extends Composite {} {
  no internalInterface
  no externalInterface
}

fact NoDomainAsSubcomponentOfFunctional {
   no d:DomainComponent |
     some c:FunctionalComponent |
        d in c.subComponents
}

sig Active extends FunctionalComponent{
  periodicity : int
}

sig Passive extends FunctionalComponent {}

sig Periodic, Sporadic extends Active {}


fact periodicANDsporadic {
  all a:Active |
    a in Periodic or a in Sporadic
  if a in Periodic
    #a.getServerInterfaces == 0
  if a in Sporadic
    all i:Interface |
      if i in a.getServerInterfaces {
        i.interfaceType = ASYNCHR
      }
}

pred getServerInterfaces[a: Component]
                          : set Interface {
  all i: Interface | {
    i in a.externalInterfaces
    and
    i.type == Server
  }
}
```

Figure A.1: Generic Component Model Formalization in Alloy

```
sig ThreadDomain extends DomainComponent {
  priority       : int
  memoryArea     : one MemoryArea
}

fact NoThreadDomainInDomainComp {
    all t:ThreadDomain |
      no c: DomainComponent |
        c in t.superComponents
}

fact EveryActiveIsInThreadDomain {
   all a:Active |
     one tDom : ThreadDomain |
       a in tDom.subComponents
}

sig RegularThread, RTThread, NHRT
   extends ThreadDomain {}

sig MemoryArea
       extends DomainComponent {
  size: int
}

one sig HeapMemory, ImmortalMemory
          extends MemoryArea {}

sig ScopedMemory extends MemoryArea {}
```

```
ct EveryComponentHasMemoryArea {
  all fc: FunctionalComponent |
    one ma: MemoryArea |
      fc in ma.^subComponents
}

fact ScopedMemoryParent {
  // all scopes are subcomponents
  //     of the ImmortalMemory
  all c:ScopedMemory |
      one t:ImmortalMemory |
        c in t.^subComponents
  // only Immortal and Scoped memory
  //     can be a parent
  all c:ScopedMemory |
     no f: (Component -
              ImmortalMemory - ScopedMemory) |
        c in f.subComponents
  // scope is non-empty
  all c:ScopedMemory |
     some k:(Component - DomainComponent)|
             k in c.subComponents
}

fact SingleParentRule {
  all sc : ScopedMemory |
    some parent : ScopedMemory |
      sc in parent.subComponents or
      sc in ImortalMemory.subComponents
}
```

Figure A.2: ThreadDomain and Memory Area

```
fact ThreadHasMemory{
  all th:ThreadDomain | all a:Active |
    if a in th.subComponents {
      th.memoryArea = getMemoryArea[a]
    }
}

pred Component.getMemoryArea[ a : Component]
            : one MemoryArea {
   one m : MemoryArea |
     a in m.^subComponents
}

fact NHRTnotInHeap {
  no c:FunctionalComponent |
    some r:NHRT |
      some h:HeapMemory |
        c in r.^subComponents
          and c in h.^subComponents
}
```

```
fact cross-thread-communication {
   all a1,a2:Active |
     all t1,t2:ThreadDomain {
       a1 in t1.subComponents
       a2 in t2:subComponents
     }
     implies {
       assertOnlyAsynchrComm{a1,a2}
     }
}

pred assertOnlyAsynchrComm
          [a1,a2:ActiveComponent] {
  all i1,i2: Interface {
    i1 in a1.externalInterfaces
    i2 in a2.externalInterfaces
    i2 in i1.boundTo

    a1 in t1.subComponents
    a2 in t2.subComponents
  }
  implies {
    i1.binding.communicationType
                = Asynchronous
    i2.binding.communicationType
                = Asynchronous
  }
}
```

Figure A.3: Composition and Binding Rules for RTSJ Domain Components

```
fact cross-scope-pattern {
  all c1,c2 : Component |
    all sc1,sc2: Component {
      c1 in sc1.subComponents
        and c2 in sc2.subComponents
            sc2 in sc1.subComponents
      isClientTo[c1,c2]
    }
    implies
      applyCrossScopePattern{c1,c2}
}

fact multi-scope-pattern {
   all c1,c2 : Component |
    all sc1,sc2: Component {
      c1 in sc1.subComponents
        and c2 in sc2.subComponents
      sc2 in sc1.subComponents
      isClientTo[c2,c1]
    }
    implies
      applyMultiScopedPattern{c1,c2}
}

fact hand-off-pattern {
  all c1,c2 : Component |
    all sc1,sc2: Component {
      c1 in sc1.subComponents
        and c2 in sc2.subComponents
      areSiblings[sc1,sc2]
      areSiblings[c1,c2]
    }
    implies
      applyHandOffPattern{c1,c2}
}
```

```
pred isClientTo[c1,c2] {
  // returns True if c2 provides a server
  // interface to c1
}

pred areSiblings[sc1,sc2] {
  // returns true if exist a sc3
  // that is a parent to sc1 and also to sc2
}

one sig CROSS_SCOPE_PATTERN,
   MULTI_SCOPE_PATTERN,
     HANDOFF_PATTERN in Attributes

fact applyCrossScopePattern {
  one b: Binding | {
    b.c in a.externalInterfaces
    b.s in b.externalInterfaces
    b.type = Asynchronous
    CROSS_SCOPE_PATTERN in b.atributes
  }
}

pred ApplyMultiScopePattern[c1,c2] {
  ...
  MULTI_SCOPE_PATTERN in b.atributes
}

pred ApplyHandOffPattern[c1,c2] {
  ...
  HANDOFF_PATTERN in b.atributes
}
```

Figure A.4: Cross-scope Communication Patterns

```
sig Container extends Composite {
  content          :  one Component
  chains           :  set InterceptorChain
  controllers      :  set Controller
  controlInterface :  set ControlInterface
} {
  controlInterfaces in ExternalInterface
  content, chains,
      controllers in this.subComponents
}

sig Component {
  container : one Container
  ...
}

sig Controller extends Composite {
  controlInterfaces :  set ControlInterface
  content           :  lone PrimitiveObject
  owner             :  one Container
}

sig Non-FunctionalComponent extends Composite{}
```

```
sig InterceptorChain extends Composite {
  owner          :  one Component
  inInterface    :  one FunctionalInterface
  outInterface   :  one FunctionalInterface
  trapInterface  :  one ControlInterface
} {
  subComponents  in Interceptor
    inInterface, outInterface, trapInterface
     in this.externalInterfaces
}

sig ControlInterface, FunctionalInterface
      extends Interface {}

fact Interfaces {
  Interface = FunctionalInterface
              + ControlInterface
}

sig Interceptor extends Component {
  owner :  one InterceptorChain
}
```

Figure A.5: Platform Level Concepts Specified in Alloy

```
fact WaitFreeQueues_ApplicationRules {
  all a,b : Active {
    if isAsynchronousBinding(a,b)  {
      if getThreadType(a) == getThreadType(b){
        if getPriority(a) > getPriority(b)
          setWriteFreeQueue(a,b)
        if getPriority(a) < getPriority(b)
          setReadFreeQueue(a,b)
        if getPriority(a) == getPriority(b)
          // can not happen here
      }
      if isBiggerType(a,b)
        setWriteFreeQueue(a,b)
      if isBiggerType(b,a)
        setReadFreeQueue(a,b)
    }
  }
}

// from a to b
pred isAsynchronousBinding
       (a,b: ActiveComponent) {
  one b: Binding | {
    b.client in a.externalInterfaces
    b.server in b.externalInterfaces
    b.type = Asynchronous
  }
}
```

```
fact DefiningPoolPatternApplication {
  all a,b : Active {
    if isAsynchronousBinding(a,b)  {
      if a,b in ImmortalMemory.^subComponents
        useObjectPoolPattern[a,b]
      if getMemoryArea(a) != getMemoryArea(b)
        useDeepCopy(a,b)
    }
  }
}

pred useDeepCopy[a,b : ActiveComponent] {
  one b: Binding | {
    b.client in a.externalInterfaces
    b.server in b.externalInterfaces
    b.type = Asynchronous
    DEEP_COPY in b.atributes
  }
}


pred useObjectPoolPattern
       [a,b : ActiveComponent] {
  // ...same se useDeepCopy
  OBJECT_POOL in b.atributes
}

pred setReadFreeQueue
       [a,b : ActiveComponent] {
  // ...same se useDeepCopy
  WAITFREE_READ_QUEUE in b.atributes
}

pred setWriteFreeQueue
       [a,b : ActiveComponent] {
  // ...same se useDeepCopy
 WAITFREE_WRITE_QUEUE in b.atributes
}

one sig DEEP_COPY, OBJECT_POOL,
  WAITFREE_WRITE_QUEUE,
    WAITFREE_READ_QUEUE in Atributes {}
```

Figure A.6: WaitFreeQueue and ObjectPool Formalization

```
pred Component.component [n : String] {
 c : this.subComponents | c.name = n
}

pred Component.interface [n: Name, r : Type] {
  one i : Interface | {
    i in c.externalInterfaces
    and i.name = n and i.role = r
  }
}

pred Component.primitive {
  this in Primitive
}
```

```
pred binding[clien : Interface
       serv : Interface] {
   one b : Binding |
     b.client = clien
     b.server = serv
     b.communicationType = Synchronous
   }
}

pred bindingAsynchronous[clien : Interface
       serv : Interface] {
   one b : Binding |
     b.client = clien
     b.server = serv
     b.communicationType = Asynchronous
  }
}
```

Figure A.7: SOLEIL ADL defined in Alloy

Appendix B

OCL Constraints for SOLEIL Profile

The Table B.1 provides several SOLEIL profile rules specified in OCL for illustration. The whole specification and validation process is described in [NL09].

Looking at the table, for the definition of the constraints that domain-specific components (DSC) impose on implementation, we employ the link between the HULOTTE and Java models that is provided by the Fraclet-metamodel [RM09]. All of the constraints defined in table B.1 use this link through a helper class called `SpoonUTIL`. The first constraint checks that methods in distributed bindings define parameters with primitive types. The `SpoonUTIL` helper class is used to get the Java interfaces (and methods) that implement given distributed components. The second constraint is implemented in a similar manner, and represents the constraint that states that *asynchronous* bindings must define methods with no return type. Constraints three and five check that certain statements are not present in the body of methods implementing the DSCs: synchronized statements in *protected* components, and instantiation of threads in *active* components. Finally, constraint number four checks that classes that implement *periodic* components extend Java's Runnable interface.

1	**Context**:DistributedNode **inv**:self.annotatesSet->**forAll**(c | c.bindings->**forAll**(eb | SpoonUTIL.getFracletInterface(eb.source).target.Methods.Parameters **union** SpoonUTIL.getFracletInterface(eb.dest).target.Methods.Parameters->flatten()->**forAll**(p | p.Type.isPrimitive())))
2	**Context**:Asynchronous **inv**:self.annotatesSet->**forAll**(b | SpoonUTIL.getFractalInterface(b.source).target.Methods.Type.SimpleName = 'void' **and** SpoonUTIL.getFractalInterface(b.dest.target.Methods.Type.SimpleName = 'void')
3	**Context**:Protected **inv**:self.annotatesSet->**forAll**(c | SpoonUTIL.getFracletComponent(c).target.Methods->**forAll**(m | m.body.statements->flatten()->**select**(s | s.oclIsKindOf(CtSynchronized))->isEmpty()))
4	**Context**:Periodic **inv**:self.annotatesSet->**forAll**(c| SpoonUTIL.getFracletComponent(c).target.SuperInterfaces->**select**(s| s.QualifiedName = 'java.lang.Runnable')->notEmpty())
5	**Context**:Active **inv**:self.annotatesSet->**forAll**(c | SpoonUTIL.getFracletComponent(c).target.Methods->**forAll**(m | m.body.statements->flatten()->**select**(s | s.oclIsKindOf(CtNewClass) **implies** s.asOclType(CtNewClass).Type.QualifiedName = 'java.lang.Thread')->isEmpty()))

Table B.1: OCL expressions for implementation constraints

Appendix C

SweetFactory Architecture in FRACTAL-ADL

In Fig. C.2 we show architecture of the SweetFactory scenario expressed in the FRACTAL-ADL notation. The architecture corresponds to the same architecture expressed in the Alloy ADL in Fig. 4.20.

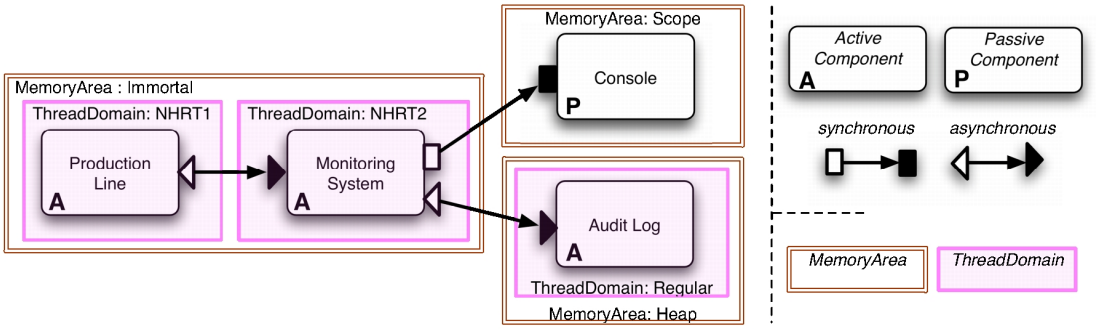
```
1   <!-- Functional Components -->

3<ActiveComponent name="ProductionLine"
4     type="periodic" periodicity="10ms">
5   <interface name="iMonitor"
6       role="client"
7       signature="IMonitor" />
8   <content class="ProductionLineImpl"/>
9</ActiveComponent> <ActiveComponent name="MonitoringSystem"
10    type="sporadic">
11  <interface name="iMonitor"
12    role="server"
13    signature="IMonitor" />
14  ...
15</ActiveComponent>

17<PassiveComponent name="Console">
18  ...
19</PassiveComponent> <ActiveComponent name="Audit"
20  type="sporadic" />
21  ...
22</ActiveComponent>

24<!-- Bindings --> <Binding>
25  <client cname="ProductionLine"
26          iname="iMonitor" />
27  <server cname="MonitoringSystem"
28          iname="iMonitor" />
29  <BindDesc protocol="asynchronous"
30          bufferSize="10" />
31</Binding>
```

Figure C.1: SweetFactory Architecture in FRACTAL-ADL, Part 1




```
<!-- Non-Functional Components -->

<MemoryArea name="Imm1">
  <ThreadDomain name="NHRT1">
    <ActiveComp
      name="ProductionLine"/>
    <DomainDesc type="NHRT"
        priority="30" />
  </ThreadDomain>
  <ThreadDomain name="NHRT2">
    <ActiveComp
      name="MonitoringSystem"/>
    <DomainDesc type="NHRT"
        priority="25" />
  </ThreadDomain>

  <AreaDesc type="immortal"
    size="600KB" />
</MemoryArea>

<MemoryArea name="S1">
    <PassiveComp name="Console"/>
  <AreaDesc type="scope"
    name="cscope" size="28KB" />
</MemoryArea>

<MemoryArea name="H1">
  <ThreadDomain name="reg1">
    <ActiveComp
      name="Audit"/>
    <DomainDesc type="Regular" />
  </ThreadDomain>
  <AreaDesc type="heap" />
</MemoryArea>
```

Figure C.2: SweetFactory Architecture in FRACTAL-ADL, Part 2

Index