A Real-Time Java Component Model

Ales Plsek, Philippe Merle, Lionel Seinturier

A Real-Time Java Component Model*

Aleš Plšek, Philippe Merle, Lionel Seinturier
INRIA Lille - Nord Europe, Project-team ADAM
USTL-LIFL CNRS UMR 8022
Haute Borne, 40, avenue Halley
59650 Villeneuve d'Ascq, France

Abstract

The Real-Time Specification for Java (RTSJ) [10] is becoming a popular choice in the world of real-time programming. However, the complexities introduced by RTSJ bring the needs for an extensive framework comprising all the aspects of RTSJ development. As the first contribution of this paper, we present a real-time component model directly fitting the needs of RTSJ. Our motivation is to clearly separate real-time and business concerns of applications. We further argue that the RTSJ concerns need to be considered at early stages of architecture design in order to mitigate the complexities of the implementation phase. Therefore, as our second contribution, we propose a design process introducing gradually RTSJ concepts into the architecture. We are thus able to alleviate the development of real-time systems and to tailor them for different real-time conditions. Finally, we demonstrate the feasibility of our solution on an example scenario.

1 Introduction

When looking into the future of distributed, real-time and embedded systems, we can see large-scale, heterogeneous, dynamically highly adaptive systems fulfilling tasks of different complexities. This will bring variously stringent QoS demands presented at the same time for different but interconnected parts of systems. In the context of this paper, meeting these challenges represent a task of developing applications composed from hard-, soft- and non-real-time units.

A recently popular solution in this domain represents the Real-Time Specification for Java (RTSJ) [10]. However, using RTSJ at the implementation level is an error-prone process where developers have to obey non-intuitive rules and restrictions. Despite numerous efforts for designing RTSJ compliant patterns [1, 3, 12], a systematical development approach considering RTSJ limitations at higher abstraction levels is desperately needed. Here, a clear separation of real-time concerns from the rest of the system would allow developers to obtain applications that are more modular and where RTSJ-related properties are isolated in clearly identified software entities.

One of the answers to these issues are component-oriented frameworks for RTSJ, such as [11, 9, 7]. The basic motivation lays in abstracting complexities of the RTSJ development from the developers. Nevertheless, the state-of-the-art solutions still need to fully address adaptability issues of real-time systems, separation of real-time and business concerns, and suffer from the absence of a design process that would introduce real-time concerns at the architectural level.

The goal of this paper is to propose a real-time component model designed to directly fit the needs of RTSJ. Although our primary motivation is the separation of real-time and business concerns, in this paper we argue that RTSJ concerns need to be considered at early stages of the architecture design in order to mitigate the complexities of the implementation phase. The challenge is therefore to express RTSJ concerns at the architectural level and thus alleviate the design of RTSJ-based real-time systems.

Additionally, we propose a design process tailored to our real-time component model that gradually introduces the RTSJ concerns into the architecture. Finally, we address the adaptability issues by proposing means to compose different assemblies of real-time components thus adapting a system for different real-time conditions.

To reflect these goals, the structure of the paper is as follows. Section 2 provides an overview of RTSJ and introduces our example scenario. In Section 3, we discuss related work and identify non-addressed challenges. We propose a new real-time component model in Section 4. Section 5 incorporates concepts of the model into the real-time systems design process and demonstrates its feasibility on

[1]This work has been partially supported by the ANR/RNTL project Flex-eWare.

the example scenario. Section 6 further elaborates selected implementation aspects of our proposal. We evaluate our contributions in Section 7. Section 8 concludes and discusses our future work.

2 Background

2.1 Real-Time Java Specification

The Real-Time Java Specification [10] (RTSJ) is a fully fledged specification for development of predictable real-time Java applications. Between many constructs which mainly pose special requirements on underrunning JVM, two new programming concepts were introduced - real-time threads (`RealTimeThread`, `NoHeapRealTimeThread`) and special types of memory areas (`ScopedMemory`, `ImmortalMemory`).

`RealTimeThread` and `NoHeapRealTimeThread` (NHRT) are new types of threads that have precise scheduling semantics. Moreover, NHRT can not be preempted by the garbage collector, this is however compensated by a restriction forbidding to access the heap memory. RTSJ further distinguishes three memory regions: `ScopedMemory`, `ImmortalMemory`, and `HeapMemory`, where 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. Another important limitation is the *single parent rule* [2] defining that a memory region can have only one parenting scope.

2.2 Motivation Example

To better illustrate all the complexities of the RTSJ development we introduce an example scenario that will be revisited several times through the course of this paper. The task is to design an automation system controlling an output statistics from a production line in a factory and report all anomalies. The example represents a classical scenario, inspired by [5], where both real-time and non-real-time concerns coexist in the same system.

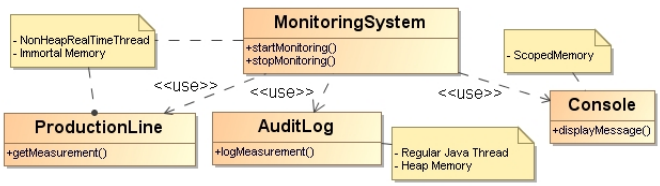

Figure 1. Motivation Example

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. Since the production line operates in 10ms intervals, the system must be designed to operate under hard real-time conditions.

A class diagram of the system is depicted in Fig. 1. As we can see, real-time and non-realtime concerns are mixed together. Identification of those parts of the system that run under different real-time constrains is difficult, hence the design of communication between them is clumsy and error-prone. As a consequence, the developer has to face these issues at the implementation level which brings many accidental complexities.

Solving these issues during the implementation is therefore an error-prone process. To avoid this, a clear separation of real-time and memory management from the business concerns is required. Moreover, the RTSJ concerns need to be considered at the design phase since they influence the architecture of the system. Therefore a proper semantics for manipulating RTSJ concerns at the architectural level has to be additionally proposed.

2.3 Component Frameworks

Component frameworks simplify development of software systems. A proper component model represents cornerstone for each component framework, the models are usually hierarchical which allows component reuse at a coarser grain. Basic building units are components communicating through exactly defined points - interfaces. Bindings between interfaces represent communication, different concepts (such as synchronous/asynchronous or event-based) can be used for their implementation. Components are sometimes divided into *passive* and *active*. Whereas passive components generally represent services, active components contain their own thread of control. It is therefore a component model and its extensiveness that substantially influences capabilities of a component framework.

3 Related Work

Recently significant increase of interest in RT Java is reflected by an intensive research in the area. However, focus is laid on implementation layer issues, e.g. RTSJ compliant patterns [1, 3, 12], rather than on RTSJ frameworks where only a few projects are involved.

Compadres[11], one of the most recent projects, proposes a component framework for distributed real-time embedded systems. A hierarchical component model where each component is allocated either in a scoped or immortal memory is proposed. However, the model supports only

event-oriented interactions between components. Components can be allocated only in scoped or immortal memories, therefore communication with regular non-real-time parts of applications can not be expressed. 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. Also the design process of real-time applications is addressed here. However, a solution introducing systematically the real-time concerns into the business code is not proposed, thus the complexities of designing real-time systems are not mitigated.

Work introduced in [9] also defines a hierarchical component model for Real-Time Java. Here, components can be either active or passive. Active components with their own thread of control represent real-time threads. However, the real-time memory management concerns can not be expressed independently of the business architecture, systems are thus designed already with real-time concerns which not only lay additional burdens on designers but also hinders later adaptability.

The project Golden Gate [7] introduces real-time components that encapsulate the business code to support the RTSJ memory management. However, the work is focused only on the memory management aspects of RTSJ, the usage of real-time threads together with their limitations is not addressed.

As we can see, there is much to be done on the field of component based frameworks for RTSJ. The current solutions are still rigid, supporting no adaptation, and mainly do not allow to express both real-time and non-real-time systems at the architectural level. To meet all the challenges an adequate component model allowing to fully describe real-time concerns independently of the business needs to be proposed. Furthermore, a design process that will consequently incorporate real-time concerns into the business architecture has to be designed.

4 Our Real-Time Component Model

When designing a component model for real-time Java, two seemingly opposite tasks have to be met. A sufficient level of abstraction from RTSJ complexities has to be provided. On the contrary, we simultaneously pursue a motivation to adequatly express real-time concerns at the design level. As already said, RTSJ concepts need to be considered at the early stages of the architecture design in order to achieve effective development process that mitigates all the complexities.

Our goal is therefore to design a component model directly fitting the concepts introduced by RTSJ. While keeping an appropriate level of abstraction, it is needed to define a proper representation of RTSJ concepts in the model. Based on this, we define a hierarchical component model with sharing depicted in Fig. 2. The abstract entity *Component* defines that each component has sub components - expressing hierarchy, and super components - expressing component sharing. From Component, we derive *Active/Passive* components representing basic composition units of a system. Furthermore, entities representing real-time concerns are defined: *ThreadDomain* and *MemoryScope*. These extensions allow the developers to express both RTSJ and non-realtime concerns, at the architectural level. They are further described in Section 4.2 and 4.3.

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 this in Sections 4.4 and 4.5.

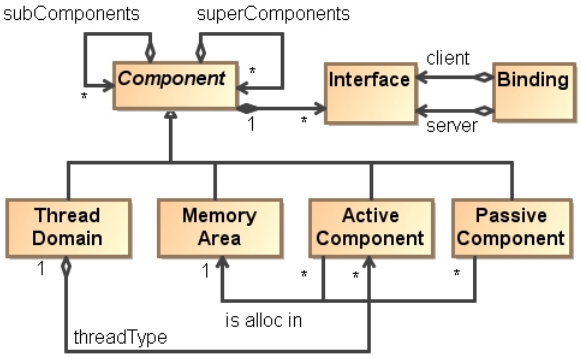

Figure 2. Real-Time Component Metamodel

4.1 Active and Passive Components

Active and *Passive* components, basic building units of our model, represent business concerns in the system. Passive components are standard component-oriented units providing and requiring services. Active components contain their own threads of execution, every active component is deployed in an instance of the *ThreadDomain* which determines a type of the execution thread. Moreover, a relation between an Active/Passive component and a *MemoryArea* defines the memory region in which the component is allocated.

4.2 Thread Domains

ThreadDomain represents `RealTimeThread`, `NoHeapRealTimeThread`, and `RegularThread` in a system. Each *ThreadDomain* groups threads with the same properties (type, priority, etc.) and encapsulates all the active components that contain a thread of control with such properties. 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 in the system.

4.3 Memory Areas

MemoryArea represents the memory areas distinguished by RTSJ: `ImmortalMemory`, `ScopedMemory`, and `HeapMemory`. MemoryArea component instance thus encapsulates all subcomponents that are allocated in the same memory area. This allows to detect a communication between different memory areas (also known as *interscope 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 a system.

Additionally, nested memory scopes can be easily modeled as subcomponents. This allows us to straightforwardly identify a parent of each memory scope and thus comfortably respect the *single-parent rule*.

4.4 Composing Real-Time Components

The restrictions introduced by RTSJ impose several rules on the composition process. Fortunately, the component model includes RTSJ concerns and therefore we are able to validate a conformance to RTSJ during the composition process. Additionally, to express both business and real-time concerns simultaneously in the architecture, the model allows sharing of components. Therefore the set of super components of a given component directly defines its business and also its real-time role in the system. Eg. deploying an active component into an instance of the *ThreadDomain* entity defines all the properties for the thread of control of this active component. At the same time, the active component is also part of the business composition.

4.5 Binding Real-Time 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.

4.5.1 Cross-Thread Communication

When modeling a RTSJ compliant binding between active components, *Queue Communication* and *Scope Sharing* concepts can be used. Whereas using specially designed queues is simple, it introduces asynchronous communication. On the contrary, Scope Sharing poses no limitations, but can be applied only in the RTSJ compliant cases.

Since adherence to RTSJ rules can be verified, the designer is able to decide which types of bindings can be used. This unloads unnecessary burden from the implementation phase where only the implementation of chosen binding concepts has to be done.

4.5.2 Cross-Scope Communication

Our model additionally allows to clearly express cross-scope communication as a binding between a scope memory component and its environment. Here, many patterns introduced in [1, 3, 12] can be used depending on a designers choice and a specific situation.

5 Designing Component-Oriented Real-Time Applications

The elevation of RTSJ concepts to the architectural level may hinder our task for an appropriate level of abstraction, since we are combining business with real-time concerns. To avoid increased complexity of the design phase we therefore propose to modify the process of designing a component-based system architecture.

We decouple the design phase into several steps where each step focuses on different concepts of RTSJ. The abstractions introduced in our model allow designers to gradually integrate real-time concerns into the architecture. We define three basic views on the real-time component-based system architecture: *Business View*, *Thread Management View*, and *Memory Management View*. Whereas the business view considers only functional aspects of the system, the two others stress on different aspects of real-time programming and allow developers to design these aspects independently on the business architecture. Finally, we propose a new methodology of the real-time architecture design that helps designers to integrate seamlessly real-time concerns into the business architecture.

5.1 Business View

Business view deals only with composition of active and passive components. This helps developers to focus exclusively on designing functional aspects of the system. To illustrate the idea, we use our example scenario presented in Section 2.2 and construct the business architecture of the system, see Fig. 3.

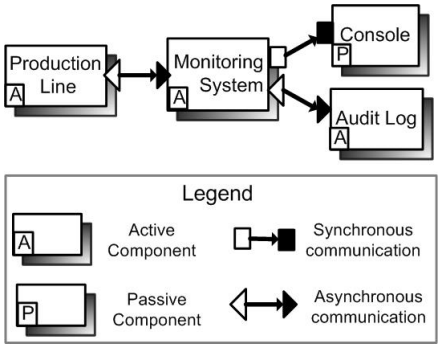

Figure 3. Business View

5.2 Thread Management View

The tread 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.

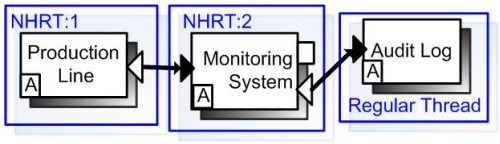

Figure 4. Thread Management View

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. 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, the *ProductionLine* and *MonitoringSystem* will be deployed in the NHRT domain, since they have to meet 10ms deadlines. Both of them are in different instances of NHRT because they run under different thread priorities. Unlike the other components, the *AuditLog* component is not timing-critical, thus we can design it for use under regular Java. The whole composition is depicted in Fig. 4.

5.3 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 allocated.

Additionally, the bindings crossing different memory regions can be easily identified. This facilitates to appropriately deploy a glue code managing the cross-scope communication.

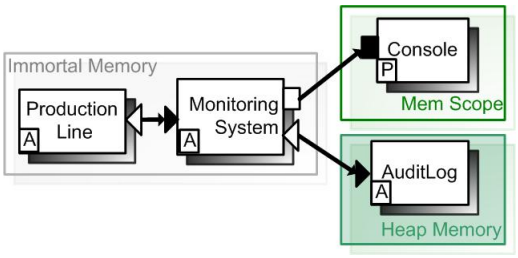

Figure 5. Memory Management View

Similarly to the thread management view, different assemblies of components into memory regions tailored to fit various real-time conditions can be delivered.

Considering our motivation example, 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. 5.

5.4 Design Methodology

To progressively incorporate all the views into the design process, we propose a new architecture design flow, depicted in Fig. 6. First, we analyze system specification which is divided into *Business* and *Real-Time Specification*. From the *Business Specification*, describing functional tasks of the system, we design interfaces, components, and consequently the *Business Architecture*, in Fig. 6 denoted as step 1. So far, we follow a well-known concept of component-architecture design and the business view can be employed.

The *Real-Time Specification* describes non-functional properties of the system related to the real-time, here we are able to determine parts of the system that will be real-time aware (using the *Thread Management View*). Consequently, we deploy different parts of a business system into various real-time and non-real-time components to obtain the *Real-Time Business Architecture*, step 2. This can be easily achieved since our component model allows us to abstract different real-time units through `ThreadDomain` components.

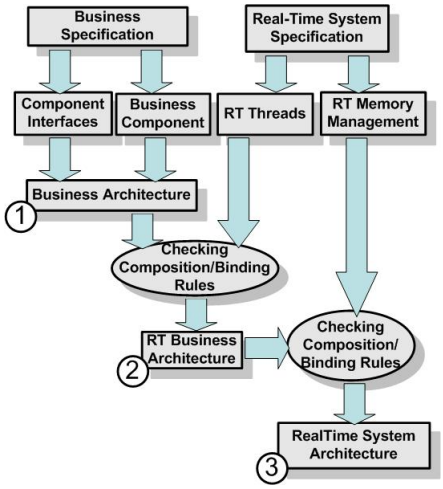

Figure 6. RealTime Component Architecture Design Flow

Then we extend the *Real-Time Business Architecture* by an appropriate memory management (using the *Memory Management View*), thus achieving a complete and RTSJ compliant architecture of a real-time system, step 3.

The compliance with RTSJ is enforced by checking composition and binding rules during the design process. This provides an immediate feedback and the designer can appropriately modify an architecture whenever is in a conflict with RTSJ.

5.5 Example Scenario

To fully demonstrate the design process proposed in this section, we revisit our example scenario. Therefore, after using the business view to obtain business architecture we can gradually integrate real-time concerns. We design the Real-time Business Architecture first. Then we deploy active components into appropriate `ThreadDomain` components, thus determining which parts of the application will be real-time oriented. Here, the thread management view can be used as already illustrated in Fig. 4.

After deploying all components into corresponding `ThreadDomain` components, the composition and binding rules verification is conducted. As a result, the bindings between components will be identified as RTSJ violation since they express communication between threads of different types. Consequently, possible solutions will be proposed, e.g. implementing the bindings through `WaitFreeQueues`.

In the next step, the memory management of the system has to be designed. The memory management view can be used, as presented in Fig. 5.

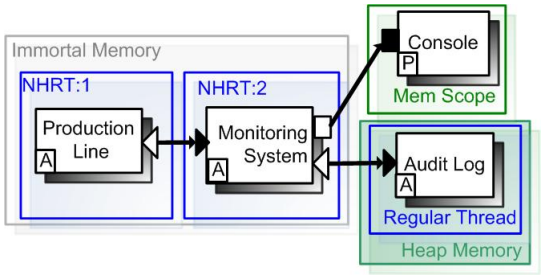

Figure 7. RT System Architecture

To finally create a complete RT System Architecture, the business view, the thread and memory management views have to be merged together. The final RT System Architecture can be seen in Fig. 7. Again, the verification process identifies the bindings that violate RTSJ and an appropriate intercepting code can be injected.

6 Implementation Issues

This section further elaborates selected implementation aspects of the component model. We demonstrate feasibility of our solution on selected examples implementing different parts of the model. As the cornerstone of the implementation we have chosen the Fractal component model that brings extendable light-weight approach together with wide range of features.

6.1 The Fractal Component Model

The Fractal component model [8][1] is a light-weight hierarchical component model that stresses on modularity and extensibility. It allows the definition, configuration, dynamic reconfiguration, and clear separation of functional and non-functional concerns. The central role is played by interfaces, which can be either *business* or *control*. Whereas *business interfaces* provide external access point to components, *control interfaces* are in charge of non-functional properties of the component (e.g. life-cycle management or binding management).

Although we focus on the Fractal component model, the concepts proposed by this work can be applied to many different component models, e.g. [4, 6].

6.2 Cross-Scope Communication

Implementing a communication crossing different memory scopes is a challenging process since developers have to obey the *single parent rule*. However, the complexities are mitigated with the introduction of our component

[1] Available at http://fractal.objectweb.org/

model where all the memory areas are exactly defined. Thus the code handling cross-scope communication can be automatically deployed. Moreover, the component-oriented approach allows a separation of real-time and business concerns in the system.

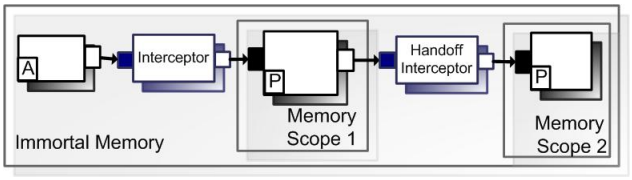

Figure 8. Memory Scope Component

We demonstrate these ideas in Fig. 8 that shows communication between several different memory scopes. While the business code remains unmodified, the real-time concerns are represented by the interceptors deployed in the bindings that cross different memory scopes.

```
class Interceptor implements ServiceInterface
{
  ScopeMemory scope;
  InterceptorRunnable intRunnable;
  public Result service() {
    intRunnable.setData();
    scope.enter(intRunnable);
    return intRunnable.getResult();
  }
  ....
}
```

Figure 9. MemoryArea Interceptor Implementation

A code snippet from Fig. 9 shows implementation of the Interceptor component that manages entering and leaving of the memory scope and uses a simple deep-copy pattern for returning results from the scope.

A more sophisticated solution of cross-scope communication handling is demonstrated by the HandoffInterceptor from Fig. 8. We employ the HandOff pattern [12] to store data in a scope while still operating in the original one. Fig. 10 shows a code snippet of the HandOffInterceptor implementation.

6.3 Scope Sharing

Our solution further allows designer to construct dynamical views focusing on different aspects of the architecture. By an introspection of the architecture we are able to identify components shared between different active components. This brings not only advantage of automatical deployment of synchronization procedures but we can also model sharing of different memory scopes.

```
class HandoffInterceptor implements
    ServiceInterface {
  ScopeMemory parentScope;
  Bridge bridge;
  public void service(Data data) {
    bridge.setData(input)
    parentScope.executeInArea(bridge);
  }
}
```

Figure 10. HandOff Interceptor Implementation

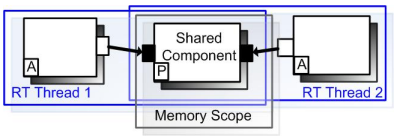

Figure 11. Memory Scope Sharing

Such sharing is illustrated in Fig. 11. Here, we are able to verify at design-time whether this sharing is compliant with RTSJ. If it is, the deployment of different types of interceptors depending on desired cross-scope communication patterns can be performed. If this sharing violates RTSJ, the designer can choose a different communication concept.

7 Evaluation

We further summarize the main contributions of our work.

- **Component Model** The proposed component model allows designers to explicitly express an architecture combining real-time and business concerns.
- **Design Views** The component model further allows a separation of real-time concerns and to design them independently of the rest of the system. Business, Thread and Memory Management Views were proposed to alleviate this process.
- **Design Methodology** Our methodology facilitates the real-time system design by providing an approach to merge composition views. By combining different Thread and Memory Management compositions we

can smoothly tailor a system for variously hard real-time conditions without necessity to modify the business architecture. The verification process moreover ensures that compositions violating RTSJ will be refused.

- **Implementation Phase** Considering an implementation of each component, the designed architecture considerably simplifies this task. Business and real-time concerns are strictly separated and a guidance for possible implementations of those interfaces that cross different concerns is proposed.

8 Conclusion & Future Work

Although RTSJ allows programmers to develop real-time systems with the Java language, it brings many complexities and restrictions that prevent from a straightforward application. In this paper, we define a component model tailored directly to the specifics of RTSJ. Our contributions include separation of real-time and business concerns, and the ability to express these concerns at the architectural level. We are therefore able to model real-time concerns and, as our additional contribution, we design a process that gradually integrates these concerns into the architecture.

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

As for our future work, we consider the proposed component model as a fundamental cornerstone of a component framework that will focus on developing dynamically adaptable real-time systems for distributed and embedded environments. Additionally, similarly as we achieved for the Fractal component model [13], our future work comprises development of optimization heuristics to reduce overhead of the framework. An extensive analysis has to be conducted to show that the framework does not introduce execution overhead comparing to RTSJ systems developed in object-oriented way.

References

[1] 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] A. Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley and Sons, 2004.

[3] E. G. Benowitz and A. F. Niessner. A Patterns Catalog for RTSJ Software Designs. *Lecture notes in Computer Science*, 2003.

[4] T. Bures, P. Hnetynka, and F. 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, 2006.

[5] C. Gough, A. Hall, H. Masters, A. Stevens. Real-Time Java: Writing and Deploying RT-Java Applications, 2007. http://www.ibm.com/developerworks/java/library/j-rtj5/.

[6] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. *Lecture Notes in Computer Science*, 2218:160, 2001.

[7] 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.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software: Practice and Experience*, 36:1257 – 1284, 2006.

[9] J. 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, 2006.

[10] G. Bollela, J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull . *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[11] 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.

[12] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-Time Java Scoped Memory: Design Patterns and Semantics. In *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 101–110, 2004.

[13] L. Seinturier, N. Pessemier, L. Duchien, and T. 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*, page 139—153. Springer, 2006.