A Component Framework for Java-based Real-Time Embedded Systems*

Aleš Plšek, Frédéric Loiret, Philippe Merle, Lionel Seinturier

INRIA Lille - Nord Europe, ADAM Project-team
USTL-LIFL CNRS UMR 8022
Haute Borne, 40, avenue Halley
59650 Villeneuve d'Ascq, France
{ales.plsek | frederic.loiret | philippe.merle | lionel.seinturier}@inria.fr

Abstract. The Real-Time Specification for Java (RTSJ) [13] is becoming a popular choice in the world of real-time and embedded programming. However, RTSJ introduces many non-intuitive rules and restrictions which prevent its wide adoption. Moreover, current state-of-the-art frameworks usually fail to alleviate the development process into higher layers of the software development life-cycle. In this paper we extend our philosophy that RTSJ concepts need to be considered at early stages of software development, postulated in our prior work [2], in a framework that provides continuum between the design and implementation process. A component model designed specially for RTSJ serves here as a cornerstone. As the first contribution of this work, we propose a development process where RTSJ concepts are manipulated independently of functional aspects. Second, we mitigate complexities of RTSJ-development by automatically generating execution infrastructure where real-time concerns are transparently managed. We thus allow developers to create systems for variously constrained real-time and embedded environments. Performed benchmarks show that the overhead of the framework is minimal in comparison to manually written object-oriented applications, while providing more extensive functionality. Finally, the framework is designed with the stress on dynamic adaptability of target systems, a property we envisage as a fundamental in an upcoming era of massively developed real-time systems.

Key words: Real-time Java, RTSJ, component framework, middleware

1 Introduction

1.1 Current Trends and Challenges

The future of distributed, real-time and embedded systems brings demand for large-scale, heterogeneous, dynamically highly adaptive systems with variously

[1] This work has been partially funded by the ANR/RNTL Flex-eWare project and by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

stringent QoS demands. Therefore, one of the challenges is the development of complex systems composed from hard-, soft-, and non-real-time units. The Java programming language and its Real-Time Java Specification [13] (RTSJ) represent an attractive choice because of their potential to meet this challenge. Moreover, they bring a higher-level view into the real-time and embedded world, which is desperately needed when avoiding accidental complexities and steep-learning curves. However, using RTSJ at the implementation level is an error-prone process where developers have to obey non-intuitive rules and restrictions (single parent rule [3], cross-scope communication [17], etc.).

One of the answers to these issues are component-oriented frameworks for RTSJ, such as [10,12,14], 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 functional concerns, and suffer from the absence of a high-level process that would introduce real-time concerns during the design phase.

1.2 Goals of the Paper

A complete process for designing of real-time and embedded applications comprise many complexities, specially timing and schedulability analysis, which has to be included in a design procedure. The scope of our proposal is focused on non-distributed applications and 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 real-time and embedded systems. Our motivation is to consider real-time 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 real-time system development and offload the burden from users by providing a middleware layer for management of RTSJ concerns. We therefore summarize the main contributions that are addressed to achieve the goals:

- *Development Process* To propose a methodology to develop RTSJ-based systems that mitigates possible complexities and allows full-scale introduction of code generation technics.
- *Transparently Implemented Systems* To provide transparent implementation of systems with comprehensive separation of concerns and extensive support of non-functional properties.
- *Performance* To achieve minimal overhead of the framework, its performance and memory overhead should be subtle enough to address real-time and embedded platforms. Different code-optimization levels should be introduced to address variously constrained environments.

1.3 Structure of the Paper

To reflect the goals, the paper is structured as follows. Section 2 provides an overview of RTSJ, introduces our example scenario, and presents the component-oriented principles we integrate in our solution. Section 3 proposes a new framework for developing real-time and embedded systems. In Section 4 we present selected design and implementation aspects of our framework. Section 5 evaluates our approach; we show benchmark results measuring the overhead of the framework and discuss further contributions of our work. We present related work in Section 6. Section 7 concludes and draws future directions of our research.

2 Background

2.1 Real-Time Java Specification

The Real-Time Java Specification [13] (RTSJ) 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 (`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 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. Another important limitation is the *single parent rule* [3] 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 goal 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 [8], where both real-time and non-real-time concerns coexist in the same system.

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.

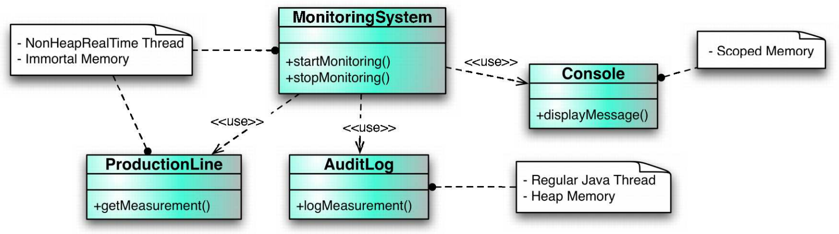


Fig. 1. Motivation Example

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.

To avoid this, a clear separation of real-time and memory management from the functional 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 during all the steps of system development 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, its extensiveness substantially influences the capabilities of a component framework.

We have investigated several component models [7,9,11] to identify features suitable for our framework. Based on this we extract a fundamental characteristic of a state-of-the-art component model: A lightweight 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 *functional* or *control*. Whereas *functional interfaces* provide external access points to components, *control interfaces* are in charge of non-functional properties of the component (e.g. life-cycle or binding management). Components are sometimes divided into *passive* and *active*. Whereas passive components generally represent services, active components contain their own thread of control. Additionally, a feature so far provided only by the Fractal component model [11] is *sharing of components* which defines that a component could have several super-components.

Component models usually provide *container* (also referred as *membrane* or *membrane paradigm* in [18]) - a controlling environment encapsulating each component and supporting various non-functional properties specific to a given component. This brings better separation of functional and non-functional properties, which can be hidden in membranes, thus simplifying utilization of components by end users.

3 Component Framework for RTSJ-based Applications

In our previous work [2], we claim that an effective development process of RTSJ-compliant systems needs to consider RTSJ concerns at early stages of the system design. Following this philosophy, our framework proposes a new methodology that facilitate design and implementation of RTSJ-based systems. We thus clarify manipulation of non-functional properties during all phases of the system life cycle.

The cornerstone of our framework represents a component model, proposed by our prior research, which allows us to fully separate functional and non-functional concerns through all the steps of system development. We recapitulate the basic model characteristics in Section 3.1. Then, a design methodology incorporated into our framework is introduced in Section 3.2. As an outcome of this process we obtain a real-time system architecture that can be used for implementation of the system. Here, 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 3.3.

Our hierarchical component model with sharing [2] is depicted in Fig. 2. The abstract entity *Component* defines that each component has sub components expressing hierarchy, and super components, expressing component sharing. We derive *Active* and *Passive* components, basic building units of our model, representing business concerns in the system. Each active component contains its own thread of execution.

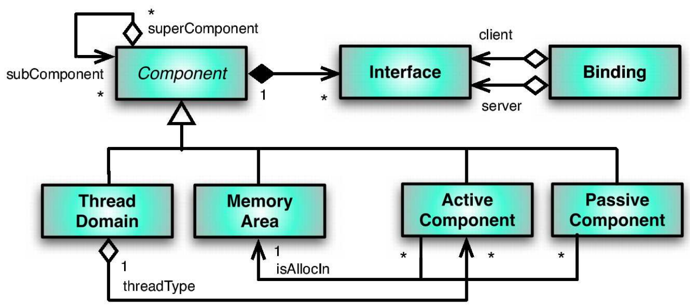


Fig. 2. A Real-Time Component Metamodel

3.1 A Real-Time Java Component Metamodel

ThreadDomain represents `RealTimeThread`, `NoHeapRealTimeThread`, and `RegularThread` in a system. Each *ThreadDomain* component encapsulates all the active components containing threads of control with the same properties (such as thread-type or priority). *MemoryArea* representing `ImmortalMemory`, `ScopedMemory`, and `HeapMemory` encapsulates all subcomponents that are allocated in the same memory area. Therefore, we are able to explicitly model RT-concepts at the architectural level by using *ThreadDomain* and *MemoryArea* components. This brings us the advantage of creating the most fitting architecture according to real-time requirements of the system.

Composing and Binding RT-Components The restrictions introduced by RTSJ impose several rules on the composition process. Since the component model includes RTSJ concerns, we are able to validate a conformance to RTSJ during the composition process. Additionally, our model allows sharing of components. Therefore, a set of super components of a given component directly defines its business and also its real-time role in the system. To give an example of such rules, the `ThreadDomain` and `MemoryArea` components are exclusively composite components, since they do not implement a functional behaviour. They specify non-functional properties which are commonly shared by their sub-components. Therefore, while `MemoryArea` components can be arbitrarily nested [1], it does not apply for `ThreadDomain`. Indeed, an *active component* should always be nested in a unique `ThreadDomain`. An another example of RTSJ constraints between thread and memory model concerns the `NoHeapRealTimeThread` 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 `NHRT ThreadDomain` which should not encapsulate a `Heap MemoryArea`, regardless of the hierarchical level specified by functional components.

Similarly, also the RTSJ conformance of bindings between components is evaluated at the design time. This allows developers to mitigate complexities of their implementation by choosing one of several communication patterns [1,5,17] already at the design time.

All these constraints are verified during the design process, which is presented in the following section.

3.2 Designing Real-time Applications

This section further explains how we integrate the component model into the process of designing real-time applications. The *Design Views* and the *Design Methodology* are proposed with motivation to fully exploit the advantages of the component model at the design time.

[1] RTSJ specification defines a hierarchical memory model for scoped memories, as introduces in Section 2.1.

Design Views We define three basic views that allow designers to gradually integrate real-time concerns into the 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 - realtime threads and memory areas management. These views therefore allow designers to architect real-time concerns independently of the business functionality. Additionally, 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.

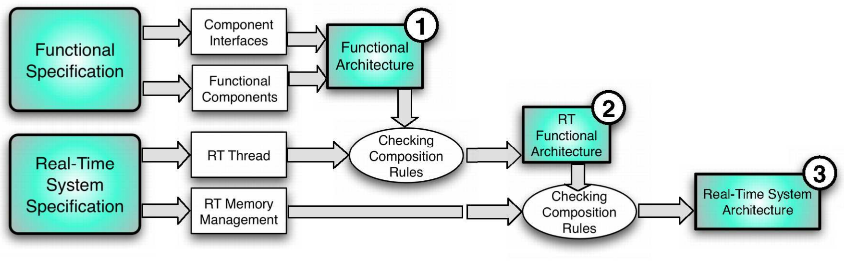
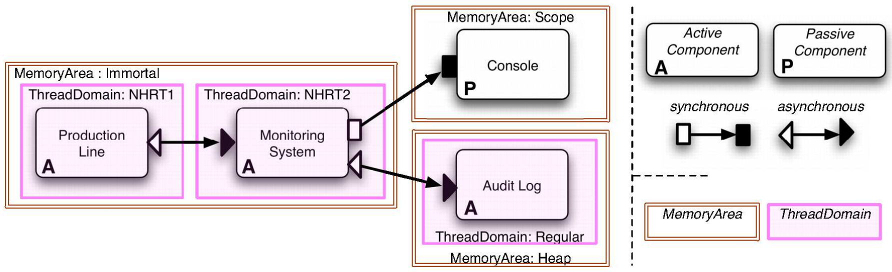

Functional Specification
Component Interfaces
Functional Components
Functional Architecture
1
Real-Time System Specification
RT Thread
RT Memory Management
Checking Composition Rules
RT Functional Architecture
2
Checking Composition Rules
Real-Time System Architecture
3

Fig. 3. RealTime Component Architecture Design Flow

Design Methodology The methodology we propose progressively incorporates all the views into the design process. The new architecture design flow, depicted in Fig. 3, represents a procedure gradually introducing real-time concerns into the architecture. In three steps, we consequently employ the *Business*, *Real-timeThread* and *Memory Management views* to finally obtain RTSJ compliant architecture. The compliance with RTSJ is enforced during the design process. This 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 the implementation phase.

Motivation Example Revisited To fully demonstrate the design process proposed in this section, we revisit our example scenario. By using the business view, we construct the functional architecture. Then we deploy active components into appropriate `ThreadDomain` components, determining which parts of the application will be real-time oriented - the thread management view can be used here. After deploying all components into corresponding `ThreadDomain` components, the adherence to RTSJ is verified. As a result, the compositions violating RTSJ are identified and possible solutions proposed, for example using RTSJ-compliant patterns [1,5,17]. In the next step, the memory management of the system has to be designed - the memory management view can be used.



```
  <!-- Functional Components -->

<ActiveComponent name="ProductionLine"
    type="periodic" periodicity="10ms">
  <interface name="iMonitor"
     role="client"
     signature="IMonitor" />
  <content class="ProductionLineImpl"/>
</ActiveComponent>
<ActiveComponent name="MonitoringSystem"
    type="sporadic">
  <interface name="iMonitor"
    role="server"
    signature="IMonitor" />
  ...
</ActiveComponent>

<PassiveComponent name="Console">
  ...
</PassiveComponent>
<ActiveComponent name="Audit"
  type="sporadic" />
  ...
</ActiveComponent>

<!-- Bindings -->
<Binding>
  <client cname="ProductionLine"
          iname="iMonitor" />
  <server cname="MonitoringSystem"
          iname="iMonitor" />
  <BindDesc protocol="asynchronous"
          bufferSize="10" />
</Binding>
```

```
  <!-- 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>
```

Fig. 4. Motivation Example: Real-time System Architecture

To finally create a complete RT System Architecture, the business view, the thread and memory management views are merged together. The final RT System Architecture can be seen in Fig. 4. The lower part of the figure presents the XML serialization of the resulting architecture. The structure of this language is consistent with the metamodel sketched out in Fig. 2. It provides the whole information needed to implement the execution infrastructure described in Section 3.3, 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 and its associated *message buffer size*,
– 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`.

3.3 Implementing Real-time Applications

The design analysis described in the previous section yields in the *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. Indeed, we adopt a generative-programming approach where the non-functional code (e.g. the RTSJ-specific code) is generated.

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.

We therefore introduce a new implementation process incorporating code generation technics, depicted in Fig. 5.

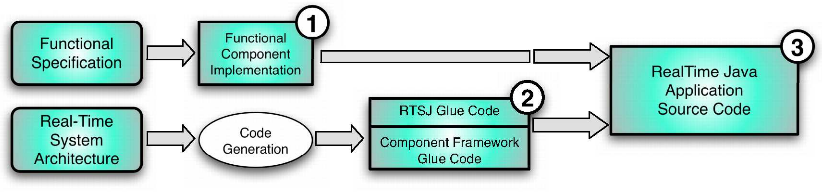

Functional Specification
Functional Component Implementation
1
Real-Time System Architecture
Code Generation
RTSJ Glue Code
Component Framework Glue Code
2
RealTime Java Application Source Code
3

Fig. 5. Execution Infrastructure Generation Flow

Implementing Functional Concerns of Applications As the first step of the implementation flow, see Fig. 5 step 1, functional logic of the system is being developed. The development process thus follows our approach where developers implement only component content classes.

Infrastructure Generation Process As the second step of our development process, we generate an execution infrastructure of the system, in Fig. 5 step 2. We exploit an already designed RT System Architecture in order to generate a glue code managing non-functional properties of the system. The generation process implements several tasks, they are listed below, their implementation is described in further details in Section 4.

- **RTSJ-related Glue Code**
  - *Realtime Threads and MemoryArea management* Real-time Thread and Memory Areas management is the primary task of the generated code. Automatical initialization and management of these aspects in conformance to RTSJ thus 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 the bootstrapping process.
- **Framework Glue Code**
  - *Active Component Management* For active components, the framework manages their lifecycle - generating code that activates their functionality.
  - *Communication Concepts* Automatical support for synchronous/asynchronous communication mechanisms is important aspect offloading many burdens from developers.
  - *Additional Functionality* Additionally, many other non-functional properties can be injected by the framework, e.g. a support for introspection and reconfiguration of the system.

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. 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 Framework Implementation Issues

The key design decision characterizing the framework is to employ component-oriented approach also during the implementation process of developed systems. Our motivation is therefore to preserve components at the implementation layer. Apart from well-know advantages of this concept, e.g. reusability of the code, this approach brings better transparency and separation of concerns. Specially separation of concerns is important here, since we need to implement functional and real-time concerns of the system but deploy them in separate entities.

Following these goals, we introduce *non-functional components* that are present at runtime. These components represent `ThreadDomain` and `MemoryArea` components, architected at the design time, and manage RTSJ-concerns of the system. This contributes to a full separation of functional and non-functional code. Moreover, this approach is further expanded by the membrane paradigm, introduced in Section 2.3, defining that each component is encapsulated by a membrane layer that manages its non-functional properties. RTSJ management is thus deployed at two places - in non-functional components, providing a coarse-grain approach, and in the membrane of each functional component, providing a fine-grain approach to management of RTSJ concerns for the specific functional logic.

Therefore, in Section 4.1 we first present the membranes and how they are employed in our solution to support RTSJ concerns of components. We also introduce non-functional components here. Consequently in Section 4.2, we explain detail implementation of membranes. Finally, Section 4.3 describes the infrastructure generation process generating membranes of components and introduces various levels of optimization heuristics which reduce overhead of the framework.

4.1 Component Framework Implementation

Component-Oriented Membrane Membrane paradigm, originally introduced in [18], defines that each component is wrapped by a controlling environment called *membrane*. Its task is to support various non-functional properties of the component. The control membrane 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 membrane. Since membranes can be parameterized, the framework allows developers to deploy for each component its unequally designed membrane that directly fits its needs.

RTSJ-oriented Membrane We employ the concept of membranes to develop our own set of controllers and interceptors which are specially designed to manage RTSJ-concerns in the system. We provide the following extensions of the control layer:

- ***Active Interceptors*** encapsulate *active components*. They implement a run-to-completion execution model[2] for each incoming invocation from their server interfaces and are configured by the properties defined by the enclosing *ThreadDomain* component.

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

– ***Memory Interceptors*** implement cross-scope communication and are deployed on each binding between different *MemoryAreas.* Their implementation depends on the design procedure choosing one of many RTSJ memory patterns [1,5,17].

Non-Functional Components An additional construct for manipulation of RTSJ-concerns at the implementation layer represent *non-functional components.* These components correspond to `ThreadDomain` and `MemoryArea` components, created at the design time, and provide thus a coarse-grain approach to management of RTSJ-concerns in the system. More preciously, membranes of non-functional components contain real-time controllers and interceptors, which superimpose non-functional concerns over their subcomponents. Thus we manage RTSJ concepts of groups of functional components with identical RTSJ properties.

4.2 Membrane Architecture Analysis

The control components incorporated in membrane can be divided into two groups. First, the controllers which are specific to the non-functional needs of the component - e.g. asynchronous communication controller, 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. Access to membrane functionality is provided through *control-interfaces*, which are hidden at the functional level to avoid confusion with functional implementation.

Motivation Example Revisited To illustrate the membrane architecture, we revisit our motivation example from Section 2.2, Fig. 6 shows a membrane of the `MonitoringSystem` component. In the picture we can see an active `MonitoringSystem` component encapsulated by its membrane, this composition is then deployed in a non-functional component `NHRT2`, an instance of a `ThreadDomain` entity representing a NoHeapRealtimeThread. Inside the MonitoringSystem membrane, various controllers and interceptors are present. *ActiveInterceptor* implements execution model of an active component; *Asynchronous Skeleton* implements asynchronous communication. Both of them represent non-functional interceptors specific to the `MonitoringSystem` component. Contrarily, *Lifecycle* and *Binding Controllers* are present to implement introspection and reconfiguration of the system, and represent optional part of the membrane that is independent of functional architecture of the component. Finally, the NRHT2 component contains a ThreadDomain controller that implements logic for management of NoHeapRealtimeThread subcomponents.

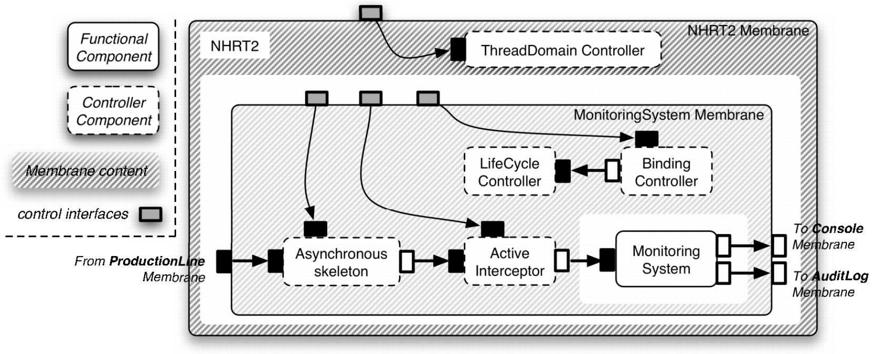


Fig. 6. Membrane Architecture, Illustration Example

Runtime Adaptability Already the basic set of controllers - Binding Controller, Content Controller, and Lifecycle controller, supports introspection and dynamic adaptation of the system. Whereas regular-Java components in our framework can be flawlessly reconfigured, however, adaptation of real-time code brings additional challenges and complexities. Since every manipulation of RTSJ concepts is bounded by their specification rules, the reconfiguration process has to adhere to these restrictions as well. Investigation and research of these controlling mechanisms is however out of the scope of this paper. We therefore settle for basic support of the adaptability issue and plan to fully tackle this challenging topic in our future work.

4.3 Soleil - Execution Infrastructure Generator

For the infrastructure generation process we employ Soleil, an extension of *Juliac* - a Fractal [11][3] toolchain backend which generates Java source code corresponding to the real-time architecture specified by the designer, including membrane source code, a framework glue code and a bootstrapping code. Moreover, the 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 execution infrastructure. The RTSJ interceptors and the reconfigurability management code are therefore implemented as non-functional components, within the *membranes.* The structure of the latter is also reified at runtime, as well as the `ThreadDomain` and `MemoryArea` composite components. This generation mode provides the complete introspection and reconfiguration capabilities of the component framework at functional and at membrane level.

[3] Available at http://fractal.ow2.org/

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 membrane architecture are replaced by direct method calls. As component membrane structures are not preserved at the runtime, the `MERGE-ALL` mode do not provides reconfiguration capabilities at membrane level. However, these capabilities are provided at the functional level.
   The source-to-source optimizations performed by the generation process are based on Spoon [16], 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 embeddes the functional implementations merged to the code which takes into account the component activations, the asynchronous communications, and the RTSJ dedicated code.

5 Evaluation

To show the quality of our framework, we evaluate it from several different perspectives. First, we conduct benchmark tests to measure performance of the framework. Then we evaluate the development process introduced by our solution from the code generation perspective. Finally, we summarize the contributions of the framework to the field of real-time and embedded systems development.

5.1 Overhead of the Framework

The main goal of this benchmark is to show that our framework does not introduce any non-determinism and to measure the performance/memory-consumption overhead of the framework. As one of the means of evaluation, we compare differently optimized applications developed in our framework against a manually written object-oriented application.
Benchmark Scenario The benchmark is performed on the motivation case-study presented in Fig. 4. 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 monoprocessor (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 preemptible one with high resolution clock support, which brings hard realtime capabilities[4].

Benchmarking Method 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 bellow, 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 4.3 for detail description of the optimization levels.

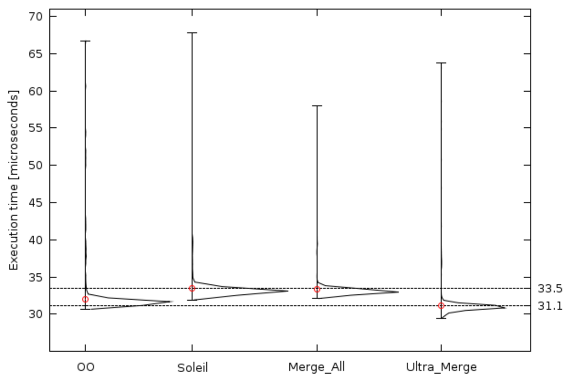
Results Discussion The results of the benchmarks are presented in Fig. 7, where the graph (a) presents the execution time distribution of the 10,000 observations processed. Fig. 7(b) sums up these results and gives their corresponding jitters. Fig. 7(c) presents the memory footprints observed at runtime.

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 membranes. 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.

Memory Footprint Considering the memory footprint, SOLEIL consumes 280KB more memory than OO. The price paid for generated membranes providing RTSJ interception mechanisms, introspection and reconfigurability. MERGE_ALL, a test introducing the first level of optimizations, gives a more precise idea of the

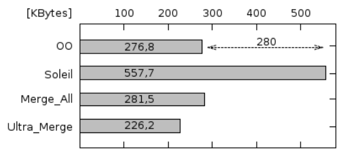
[4] The Linux RT-PREEMPT patch is available at `www.kernel.org/pub/linux/kernel/projects/rt/`



(a) Execution Time Distribution

	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

(b) Execution Time Median and Jitter



(c) Memory Footprint

Fig. 7. Benchmark Results

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.

Bottom Line 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.

5.2 RTSJ Code Generation Perspective

We further confront our generation process against the set of code generation requirements identified in [6]. The authors highlight importance of separation of concerns, stress on compactness of generated code, and demand clear distinction between generated and manually written code. All these requirements are met by our generation process since both generated and manually written code are deployed in clearly identified components. Moreover, an additional requirement demands a clear separation between functional and non-functional semantics. This is however supported directly in our component metamodel (`ThreadDomain` and `MemoryArea` components) and thus we inherently meet this requirement.

5.3 Summary of our Contribution

We further summarize the main contributions of our work, they can be divided into two categories:

– **RTSJ-based Systems Development**

  - **Component Model** The proposed component model allows designers to explicitly express an architecture combining real-time and business concerns.
  - **Designing Real-time Applications** The component model further allows a separation of real-time concerns and to design them independently of the rest of the system. 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 functional architecture. The verification process moreover ensures that compositions violating RTSJ will be refused.
  - **Implementing Real-time Applications** 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.
- **Framework Implementation**
  - **Separation of Concerns** The separation of concerns is consistently respected through all the steps of development lifecycle. Membrane extensions and non-functional components are preserved also at the implementation layer to manage real-time concerns of the system.
  - **Code Generation** The code generation approach we integrate in our framework respects the set of requirements [6] that are key for the fitness of generated code from the RTSJ perspective.
  - **Performance** Our evaluations show that we deliver predictable applications and the overhead of the framework is considerably reduced by the optimizations heuristics we implement (MERGE_ALL optimization level). Moreover, we achieve an effective footprint reduction suitable for embedded systems (ULTRA_MERGE optimization level). Despite the wide functionality we provide through out the applications development and execution life-cycle, performance results are comparable with the object-oriented approach.
  - **Dynamic Adaptation of Real-time Systems** Although the dynamic adaptation of Java-based real-time systems is a novel and complex topic, we tackle this challenge by introducing a basic support for runtime adaptation of systems developed in our framework. We consider this feature as a potent starting point for our future research.

6 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 [1,5,17], rather than on RTSJ frameworks where only a few projects are involved. Apart from these few frameworks, other projects are recently emerging with features similar to our work.

Compadres [14], 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 designed. However, the model supports only event-oriented interactions between components. On the contrary to our 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. 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. Compadres also proposes a design process of real-time applications. 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.

Work introduced in [12] also defines a hierarchical component model for Real-Time Java. Here, components can be either active or passive. Similarly to our work, 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 functional architecture, systems are thus developed already with real-time concerns which not only lay additional burdens on designers but also hinders later adaptability.

The project Golden Gate [10] introduces real-time components that encapsulate the functional 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.

The work published in [4] presents a new programming model for RTSJ based on aspect-oriented approach. Similarly to our approach, the real-time concerns are completely separated from applications base code. Although, as we have shown in [18], aspect- and component-oriented approaches are complementary, but the component-oriented approach offers more higher-level perspective of system development and brings a more transparent way of managing non-functional properties with only slightly bigger overhead.

The DiSCo project [15] addresses future space missions where key challenges are hard real-time constraints for applications running in embedded environment, partitioning between applications having different levels of criticality, and distributed computing. Therefore, similarly to our goals, the project addresses applications containing units that face variously hard real-time constraints. Here, an interesting convergence of both solutions can be revealed. The DiSCo Space-Oriented Middleware introduces a component model where each component provides a wide set of *component controllers* - a feature extensively supported by our solution.

The work introduced in [6] 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. From this point of view, we can consider our generation tool as an implementation fully compatible to the ideas proposed in this work. We further confront our approach with these requirements in Section 5.2.

7 Conclusion and Future Work

This paper presents 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 execution infrastructure.

Therefore, we employ a component model comprising the RTSJ-related aspects that allows us to clearly define real-time concepts as software entities and to manipulate them through all the steps of the system development. Consequently, we define a methodology that gradually introduces real-time concerns into the system architecture, thus mitigating complexities of this process. Finally, we alleviate the implementation phase by providing a process generating automatically a middleware layer that manages real-time and non-functional properties of the system.

Our evaluation study shows that we deliver predictable systems and the overhead of the framework is considerably reduced by the optimization heuristics we implement. Moreover, we achieve an effective footprint reduction making the output systems suitable for the embedded domain.

As for the future work, our primary goal is to extend our framework to support design and infrastructure generation for additional non-functional properties, e.g. distribution support. Furthermore, we design our framework with stress on adaptability of real-time and embedded systems, thus the framework provides a basic support for dynamic adaptability of all system components. However, to comprehensively address this issue, adaptation of real-time components needs to be managed, we therefore plan to fully tackle this challenging topic in our future work.

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. Plďž″ek, P. Merle, L. 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.
3. A. Wellings. *Concurrent and Real-Time Programming in Java.* John Wiley and Sons, 2004.

4. C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped Types and Aspects for Real-time Java Memory Management. *Real-Time Syst.*, 37(1):1–44, 2007.
5. E. G. Benowitz and A. F. Niessner. A Patterns Catalog for RTSJ Software Designs. *Lecture Notes in Computer Science*, 2889:497–507, 2003.
6. M. Bordin and T. 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.
7. 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, USA, 2006. IEEE Computer Society.
8. 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/`.
9. 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.
10. 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.
11. 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.
12. 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, USA, 2006. ACM.
13. G. Bollela, J. Gosling, B. Brosgol, P. Dibble, S. Furr, M. Turnbull . *The Real-Time Specification for Java.* Addison-Wesley, 2000.
14. 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.
15. M. Prochazka, S. Fowell, 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.
16. R. Pawlak, C. Noguera, and N. Petitprez. Spoon: Program Analysis and Transformation in Java. Technical report rr-5901, INRIA, 2006.
17. 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.
18. 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*, pages 139–153, Vasteras, Sweden, June 2006. Springer.