A Component Framework for Real-time Java[*]

Juan A. Colmenares[†], Shruti Gorappa, Mark Panahi, and Raymond Klefstad
Department of Electrical Engineering and Computer Science
University of California, Irvine, CA 92697, USA
{jcolmena, sgorappa, mpanahi, klefstad}@uci.edu

Abstract

The Real-time Specification for Java (RTSJ) offers predictable memory management and scheduling required for real-time applications. However, the usage of its memory model is difficult for standard Java developers, who are accustomed to automatic memory management. Components offer the opportunity to hide the complexities of the RTSJ's memory model, in addition to benefits such as reusability and modularity. To this end, we propose a component framework that brings together the ease of programming in Java and the predictability of RTSJ.

1 Introduction

Java's ease of use and platform independence have made it the programming language of choice in the desktop and enterprise domains. However, Java's inherent sources of unpredictability (e.g., on-demand memory (de)allocation) have prevented it from being used in real-time (RT) systems. The Real-Time Specification for Java (RTSJ) [1, 14] has been defined to address this concern. RTSJ provides threads with more carefully defined scheduling attributes, and a new memory model that makes memory management more predictable. Particularly, in addition to heap memory, RTSJ introduces two new memory regions (immortal and scoped memory) that are not managed by the garbage collector, but the restrictions (assignment and single-parent rules) imposed on these memory regions make programming using RTSJ much more complicated and tedious than with regular Java. Programmers are required to design the system's memory structure based on the characteristics of the application and then explicitly create memory scopes for object allocation. Moreover, they are responsible for ensuring that there are no illegal inter-scope object references. According to our experience [8, 7], this additional complexity is the main reason for RTSJ's steep learning curve, and the reluctance of its adoption by RT application developers.

Component-based real-time (CBRT) systems [9, 5, 12, 13, 11, 10] have received increasing attention because component reuse and modularity can potentially facilitate the development, deployment, and maintenance of RT software, and provide greater reliability at lower cost. Another reason for this interest is the success of enterprise component technologies (e.g., EJB[1]). Components also offer the opportunity to hide the complexities introduced by the RTSJ's memory model. Therefore, this paper presents a framework that eliminates the need for programmers to manually design the scoped memory architecture and explicitly manage memory allocation. They need only to provide the application's Java business code in the form of components, similar to those in EJB, while the supporting RTSJ-compatible code is automatically generated following RTSJ design patterns [4, 2, 6, 8, 7]. To the best of our knowledge, neither the research community nor industry has witnessed the convergence of CBRT technologies and RTSJ; our work would be the first effort in this direction.

2 The RTSJ Component Framework

2.1 Requirements

Besides allowing for modular programming and reusability, the RTSJ component framework must: 1) hide the complexities of RTSJ and allow programmers to develop RT applications as if using standard Java as much as possible, 2) provide a programming model easy to understand, 3) facilitate the development of a majority of RT applications, and 4) be based only on standard RTSJ features and not depend on language extensions or on any special characteristics of particular VMs. Moreover, in terms of performance it must: 1) not introduce unpredictability, 2) not add significantly more overhead than that associated to the equivalent application developed by other means (e.g., hand-coding from scratch), and 3) not have memory leaks itself, and assist in avoiding memory leaks in applications.

2.2 RTSJ Components

Inspired by [3], the proposed RTSJ component model consists of fine-grained object-oriented software artifacts (i.e., the components) that represent fundamental concepts of RT models[2], and facilitate the logical architectural design of RT applications. We distinguish two basic categories of components: active and passive. **Active components** represent RT tasks. Each active component has an associated `Schedulable` object (e.g., a `RealtimeThread`) that controls the execution of predefined operations of the component. According to the

[*]This material is based upon work supported by the National Science Foundation under Grant No. 0410218. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

[†]Also with the Applied Computing Institute, Faculty of Engineering, University of Zulia.

[1]Enterprise JavaBeans, `http://java.sun.com`

[2]These concepts are RT tasks and passive resources.

RTSJ's types of releases, active components can be: 1) **periodic**, which are released in regular time intervals; 2) **aperiodic**, which are released at random (upon the occurrence of external stimuli); and 3) **sporadic**, which are released irregularly but with a minimum time between releases. On the other hand, **passive components** contain user-defined methods that are invoked by other components. Unlike active components, a passive component does not have a fixed `Schedulable` object associated with it. Active components invoke passive components directly or through other passive components (i.e., invocation chain). Access control mechanisms allow passive components to safely handle concurrent method invocations. Thus, an application consists of at least one active component and zero or more passive components, assembled together along with a descriptor file (Section 2.4).

2.3 Scoped Memory Structure

The first major step in the design of our component framework is the definition of the memory structure. We propose a shallow memory structure (Figure 1) in which, by default, each component has its own scoped memory area (with linear-time allocation) whose parent is immortal memory, unless one specifies that some components must share a memory area.

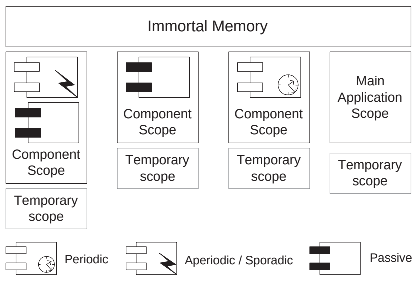


Figure 1. Framework memory structure.

Besides simplicity, the main advantage of this memory organization is that it enables independent control of the lifetime of components. This feature has demonstrated its importance in RT applications since developers are required to: 1) create and destroy components during run-time in order to implement transitions among the system's operation modes (e.g., from cruising mode to landing mode in aircraft), and 2) minimize the application's memory footprint by instantiating only the components that are actually needed (e.g., an assembly robot only needs the components associated with its current task). Furthermore, previous research [8, 7] shows that scoped memory structures can improve predictability without severely affecting performance. On the other hand, a disadvantage is that invocations on components (allocated in sibling scopes) require complex memory traversals. RTSJ design patterns [2, 6, 4, 7] can, however, overcome this drawback, and thus, developers never deal with these details. Also, due to RTSJ's assignment and single parent rules [1, 14], this memory layout imposes some restrictions on arguments and return values of methods, and private variables of components. For example, components do not allow public variables, after initialization components cannot assign (directly or indirectly) new objects to their private variables, and arguments of methods of passive components are read-only. Moreover, all component implementation classes must have a default constructor, and developers must not create and suspend threads, explicitly traverse the memory structure, and invoke `newInstance` and `newArray` on `MemoryArea` objects. Although they may seem somewhat restrictive at first glance, the restrictions do not actually limit the development of most RT applications.

2.4 Component Software Structure

The framework includes the `ActiveComponent` and `PassiveComponent` interfaces that extend the `Component` interface (Figure 2). Invocations on `init`, `terminate`, and `execute` are only permitted for the framework, not for the application code. To develop an active component, we need to implement the `ActiveComponent` interface, and to develop a passive component, we need an interface with user-defined methods that extends the `PassiveComponent` interface[3], and a class that implements that interface.

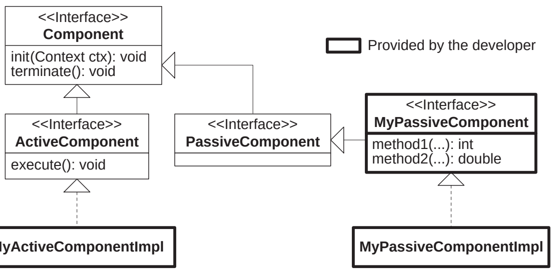


Figure 2. Component interfaces.

These interfaces relate to the **component life cycle**, whose phases are: initialization, execution, and termination. In the **initialization phase** the framework invokes `init`. In this method we can obtain (through the `Context` object) constant values specified in the descriptor file and references to passive components, allocate new objects, and assign them all to private variables of the component. Once a component is initialized, it enters the **execution phase** in which the framework invokes `execute` on active components at regular time intervals or upon the occurrence of a happening (depending on the specific component type), whereas passive components receive invocations from other components. During this phase component's methods (except `init` and `terminate`) execute in a temporary scope that is a child of a component's scope (Figure 1).[4] This allows us to create objects (e.g., intermediate results) without worrying about the allocation context and wasting memory space of the component's scope. Nevertheless, it imposes restrictions on the use of the component's variables because no variable within the component's scope can reference memory in the temporary (child) scope [1, 14]. Thus, `execute` can only modify the component's primitive data (and internal primitive data type variables of mutable objects), but object references must be considered read-only and no method invocation on them may create new objects. References to passive components represent an exception to this restriction because they refer to objects allocated in immortal

[3]Note that `PassiveComponent` adds no methods; it was included to offer a coherent set of types, consistent with the terminology introduced in this work.

[4]It is based on the design patterns Scoped Memory Entry per RT Interval [2], Scoped Run Loop and Encapsulated Method [6].

memory (i.e., immortal façades [7]). Note that all objects created inside `execute` are discarded before each invocation. Finally, the framework invokes `terminate` in the **termination phase**. Here methods on passive components may be invoked. After executing `terminate`, the component's thread ends, the `ScopedMemory` object is returned to its pool, scope objects are reclaimed, and supporting objects (e.g., façades and `Runnables`) are reset and returned to their pools.

Passive components support two types of methods: 1) **state-dependent methods** (SDMs), that access or modify the internal state of the component, and 2) **state-independent methods** (SIMs), that do not. This distinction is important because the framework treats SIMs and SDMs differently. Whereas SIMs may execute in any temporary scope and use any instance of the component implementation class (e.g., obtained from a pool or created "on the fly"), SDMs must execute in a specific temporary scope (that is a child of the passive component's scope) and use the instance of the component implementation class that keeps the component's state. Specific restrictions apply to passive components; they are: 1) the arguments of methods of passive components are read-only, and 2) the class of the return value of a SDM must implement the Java `Cloneable` interface, and the `clone` method must return a deep copy. Furthermore, the framework synchronizes concurrent invocations of SDMs by using RTSJ's access control mechanisms. By default, a passive component's method is a SIM, unless one specifies otherwise in the descriptor file.

To implement an application we must provide an XML descriptor file along with the component implementation classes. The file specifies the name of the application, and the name, type, and implementation class for every component. Global and component-specific constants are also specified in the file as well as parameters of active components (e.g., priority, start time and period of periodic components, and inter-arrival time and happenings of sporadic components), SDMs of passive components, and components that share scoped memory areas. An XML schema is used to validate the file.

Once the components and the descriptor file are ready, a tool, developed in regular Java, uses them as input to produce the RTSJ application. The tool first verifies whether the implementation of the components and the descriptor file are consistent, and whether the framework's conventions and restrictions are followed. If the verification succeeds, the tool generates the code responsible for the initialization, execution and interaction of the components, and then the resulting RTSJ application is compiled. This approach allows for minimizing runtime errors and, consequently, increasing the reliability of RTSJ applications. Additionally, it facilitates the adaptation of applications to changes in RTSJ and leaves room for many improvements (e.g., new types of components, and support of RT distributed systems). Note that the development process described above bears some resemblance to EJB development, which is a successful and well-known server-side component technology.

2.5 Framework Support Classes

The framework provides some classes that support the execution of active and passive components. `ACFacade`, `ACCSIRunnable` and `ACPortal` are common to all types of active components, whereas `PeriodicRunnable` is only used by periodic components, and `ACAsyncEventHandler` is used by aperiodic and sporadic components (Figure 3).

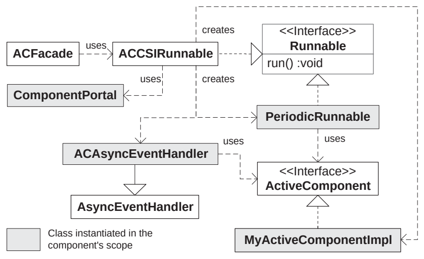


Figure 3. Active component class diagram.

`ACFacade`, which implements the Immortal Façade pattern [7], allows the framework to create, initialize, execute, and terminate active components. It also serves as a pool manager of objects of its type. By implementing the Cross-scope Invocation pattern [7], `ACFacade` is able to execute the `ACCSIRunnable` within the component's scope. `ACCSIRunnable` is responsible for instantiating all the objects required by the component to operate within its scope, and invoking `init`, `execute`, and `terminate` on the instance of the component implementation class. The portal of every component's scope is a `ComponentPortal` instance. It maintains references to objects that keep the scope alive (i.e., the `PeriodicRunnable` for periodic components, the `AsyncEvent` for aperiodic and sporadic components, and `WedgeRunnable` object for passive components). It allows the framework to obtain the reference of these objects and terminate the components.

Figure 4 illustrates the classes associated with an example passive component called *Control Law*. In this example, the developer provides the `ControLaw` interface and the implementation class (`ControLawImpl`). Then the rest of the classes are generated from the component interface and the information specified in the descriptor file. Except the `ControlLawReturnRunnable` class, which is responsible for copying the return object of a SDM in the scope where the invocation originated, the generated classes are similar to those that support the execution of active components.

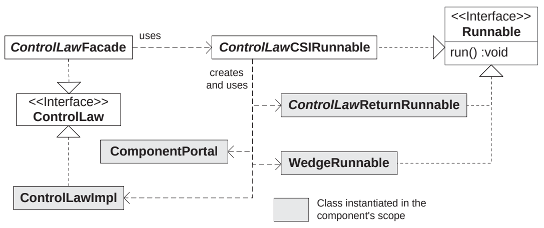
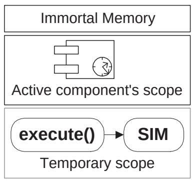


Figure 4. Passive components class diagram.

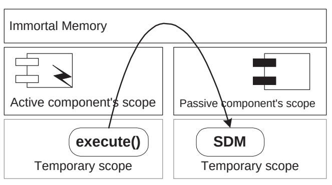
2.6 Interaction between Active and Passive Components

Most of the time passive components are invoked (directly or indirectly) by active components[5]; hence, in this case invocation chains begin with the `execute` or `terminate` method of active components.

[5]Recall that the `terminate` method of a passive component may invoke other passive components. In this case, the interactions between passive components are similar to those described in this section.



(a) AC → SIM



(b) AC → SDM

Figure 5. Basic component interactions.

Active Component Invoking a SIM (AC→SIM). The invocation of a SIM takes place in the `execute` and `terminate` methods of an active component (Figure 5(a)). In `execute`, the thread is executing in a temporary scope (a child of the active component's scope), and since no state needs to be accessed, the façade of the passive component can create (or obtain from a pool) an instance of its implementation class and invoke the requested method. The case of `terminate` is similar but here, the thread executes in the active component's scope before the component terminates. Note that SIMs can return new objects as in regular Java, but these objects can only be used inside `execute` and `terminate`.

Active Component invoking a SDM (AC→SDM). The invocation of a SDM also starts in the `execute` and `terminate` methods of an active component (Figure 5(b)). Since the SDM accesses the passive component's internal state, the thread jumps (using the Cross-scope Invocation pattern) from the current scope to a temporary scope that is a child of the passive component's scope. Once in the temporary scope, the instance of the passive component's implementation class is obtained from the portal and the requested method is invoked on it. Like SIMs, SDMs can return new objects, but in this case the framework allocates copies of them in the scope where the invocation originated. After the completion of the method, the thread exits the passive component's temporary scope and returns to the initial scope.

Active Component invoking a Chain of Methods. There are four basic cases of a SIM and a SDM invoking each other in a chain. They are: 1) AC→SIM→SIM, in which both SIMs execute in the same scope; 2) AC→SIM→SDM, which is equal to AC→SDM; 3) AC→SDM→SIM, in which the SDM and the SIM execute in the same scope (i.e., a child of the scope of the SDM's passive component); 4) AC→SDM→SDM, in which an interaction similar to AC→SDM is performed for the second SDM (i.e., every SDM executes in its own temporary scope that is a child of the scope of the corresponding passive component). These interactions can be similarly extrapolated for longer chains.

3 Final Remarks

This paper presented a component framework that facilitates the development of real-time Java applications by hiding RTSJ's complexities, especially those related to its memory model. Its design is based on requirements derived from previous experiences with developing RTSJ middleware [8, 7]. Although our framework imposes some restrictions on the implementation of components, it still allows Java programmers to develop many real-time applications. Like the Ravenscar-Java profile [14], we assume that schedulability of applications is determined off-line; thus, parameters and mechanisms for handling overruns and missed deadlines are ignored. Also, thread priorities are static and the creation of new components at runtime is not allowed.

Currently, experimental tests are being conducted to demonstrate that the framework fulfills the performance requirements listed in Section 2.1. Preliminary results show that the framework's software architecture does not introduce unpredictability[6], but they are the subject of another publication. Also, a prototype version of the code generation tool is being developed, and the descriptor file is written manually, but a tool for constructing the descriptor file will be provided in the future. In the future we plan to support one-way and two-way asynchronous method invocations for passive components, networked applications, and applications with multiple operational modes.

References

[1] Real-Time Specification for Java (RTSJ) 1.0.1(b). `http://www.rtsj.org`.

[2] E. G. Benowitz and A. F. Niessner. A patterns catalog for RTSJ software designs. In *LNCS 2889*, 2003.

[3] A. Burns and A. Wellings. HRT-HOOD: A design method for hard real-time systems. *Real-Time Systems*, 6(1), 1994.

[4] A. Corsaro and C. Santoro. Design patterns for RTSJ application development. In *LNCS 3292*, 2004.

[5] K. H. Kim. APIs for real-time distributed object programming. *IEEE Computer*, 33(6), June 2000.

[6] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: Design patterns and semantics. In *Proc. of the 7th IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing*, May 2004.

[7] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, and R. Klefstad. Patterns and tools for achieving predictability and performance with Real-Time Java. In *Proc. of the 11th IEEE Int'l Conference on Real-Time and Embedded Computing Systems and Applications*, August 2005.

[8] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, R. Klefstad, and T. Harmon. RTZen: highly predictable, Real-Time Java middleware for distributed and embedded systems. In *Proc. of the 6th Int'l Middleware Conference*, December 2005.

[9] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12), December 1997.

[10] A. Tešanović, D. Nyström, J. Hansson, and C. Norström. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1), February 2004.

[11] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3), 2000.

[12] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. Rodrigues. QoS-enabled middleware (Chapter 6). In *Middleware for Communications*. John Wiley & Sons, 2004.

[13] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-time component-based systems. In *Proc. 11th IEEE Real Time and Embedded Technology and Applications Symposium*, March 2005.

[14] A. Wellings. *Concurrent and real-time programming in Java*. John Wiley & Sons, 2004.

[6]We performed 15000 runs on RTSJ-RI v1.0.1(b)/Timesys Linux of the most complex interaction between active and passive components, i.e., an active component invokes an SDM that returns an object. We observed that jitter is less than 20% of the minimum execution time ($max = 297.32ms$, $min = 247.90ms$, and $jitter = 49.42ms$), which is acceptable for soft real-time systems.