Design Patterns for Supporting RTSJ Component Models

Mohammed Alrahmawy and Andy Wellings
Department of Computer Science, University of York, York, UK
{mrahmawy, andy}@cs.york.ac.uk

ABSTRACT

The Real-time Specification of Java (RTSJ) has new memory management and scheduling models. These models require modification to existing software models and patterns or even the invention of new ones in order to be able to provide the patterns necessary to build reusable software components. In this paper we present a new memory model pattern associated with a set of integrated patterns and with it build simple and configurable software components.

1. INTRODUCTION

A wide range of applications from embedded applications (e.g. cell phones) to distributed applications (e.g. telecommunications) requires real-time support. The complexity of these software systems complicates their development and testing as current languages and tools used for building such systems have low levels of abstraction. One of the most successful strategies used to simplify and speedup software development is to use a component based system (CBS) approach, in which the system is designed from a set of components that are developed individually and then integrated. However, using a CBS strategy for developing real-time systems presents its own challenges, due to the performance overhead it entails.

Java has proved to be a successful platform for building complex non-real-time systems either central or distributed. One of the main reasons of this success is due to its support for building applications using reusable components (e.g. Java Beans, Enterprise Java Beans). The RTSJ is an extension to Java that aims to solve the unpredictability problems of Java and to support the real-time concepts and requirements directly in the language itself. However, not only due to the real-time constraints; but also due to its new memory model and scheduling model, building reusable software components in RTSJ for real-time systems using the current CBS strategies is a complicated task, and it is not easy to enforce the use of the RTSJ rules into them especially when components integrate together.

In this paper we propose a new simple memory model that directly enforces the use of RTSJ memory access rules in a simple and efficient way. We also present some new patterns that integrate with this model to provide efficient and easy to use components. In the following section we will present the research work related to ours, then we will discuss our proposed model and patterns, and finally we provide our conclusion.

JTRES '09, September 23-25, 2009, Madrid, Spain

2. RELATED WORK

Component based programming has proved to be an efficient software engineering solution for systems development and has been used in many Java based systems. Hence, there is a trend in the research in the RTSJ community toward developing models for components based on the RTSJ. As software design patterns are basic elements in building such software components, the research for creating RTSJ based components is closely related to that of developing design patterns.

One of the main areas, that impact the development of component models, is the introduction of memory areas into the RTSJ. The constraints and memory access rules required by scoped memory inhibit developers from the direct use of general software design patterns. Hence, there has been a requirement to enhance existing software patterns or even to present new ones that can be integrated with scoped memory areas. For example, in [1] software patterns that predictably execute loops and methods in scoped memory areas were presented moreover, the *wedge thread* pattern was proposed to keep a certain scoped memory area with shared objects alive even without any schedulable object being active inside it. Furthermore, a *handoff* pattern was presented as a mechanism to enable communication among objects running in different scoped memory areas that have a common outer scoped memory area.

In [2], a survey of software patterns for RTSJ was presented, where an object factory pattern for allocating objects in a specific memory area was proposed. Also, in the same survey, the *memory pools* and *memory blocks* patterns were defined as design patterns for reusing objects especially those allocated in the immortal memory area.

In [3], the authors presented a memory-scoped version of the *leader-follower* software pattern. In this pattern, a leader-follower selector thread is proposed to be running in a single scoped memory area and can select the leader thread from a pool of threads allocated in the same memory area. Authors in [3, 4], proposed the *memory tunnels* pattern as an extension to the RTSJ specification; in this pattern, data transfer among objects in different scoped memory areas is done by deep copying objects in a temporary memory tunnel proposed by the authors.

In [5], the authors presented their experience in using software patterns for developing RTZEN, a real time CORBA Object Request Broker using the RTSJ. The *immortal exception* pattern is one of the patterns they used in their implementation. This pattern provides an exception handling mechanism capable of handling exceptions thrown from objects allocated in scoped memory areas by using reusable exception objects created in a pool in immortal memory.

Other research work has concentrated on developing RTSJ based components using some of the patterns mentioned above. For example, authors in [6], proposed a component

framework for RTSJ in which they classified components according to the existence of schedulable objects running within them into *passive* and *active* components and they proposed a framework for interaction among these components. Then the same authors, in [7], enhanced their proposed framework to provide an XML-based component definition language, and extended their model to enable composite components definition. Their composite component model was based on the *message passing* design patterns across memory scopes, e.g. the *handoff* pattern mentioned above, shared objects and serialization. In addition, they assumed the communication among parent components and their child components is to be done through a scoped memory manager defined for each component.

Researchers in [8, 9] provided another RTSJ component model that was based on the Fractal component model [10, 11]. In their model, they extended the classification of components in RTSJ to define *passive*, *active*, *composite*, and *binding* components (cross threads and cross scopes components). Furthermore, they adopted the separation of concerns concept in designing their model, where they divided the design flow of the component model into a business design flow, and a real-time design flow. Moreover, they divided the real time design flow to include a thread management view and a memory management view.

In [12], another component model for RTSJ was proposed that also adopted the fractal component model, but their aim was to make the components contract aware, based on the assumptions first presented in [13].

3. REQUIREMENTS FOR SUPPORTING COMPONENT MODELS

Beside the usual requirements of designing components in general (i.e. configurability, integrity with other components, etc), designing a model for components based on RTSJ must also take into consideration both its memory and threading models. This involves the patterns and techniques used to construct the internal elements of the components and how they can be integrated together to satisfy the general constraints defined in the RTSJ. Hence, from our viewpoint, the structure of any component model based on the RTSJ should satisfy a set of requirements that include:

- it must be constrained to conform to the RTSJ memory access rules,
- it must provide an efficient use of memory resources with minimum overhead,
- it has to avoid the redundant use of resources, and
- it has to provide a coherent model that is easy to build and to use by the developer.

In the following sections we will present our proposed new memory model, ***the Forked Memory Model***, and present some new patterns to support it. We then consider the patterns that can be used to help in developing component models using this new memory model.

4. A MEMORY MODEL FOR CBS

In this section we provide the details of our proposed new memory model, the *Forked Memory model,* as a model for building components in RTSJ that directly enforce the use of its memory access rules. We presented an initial view of this model in [14] to be used for building memory architectures of the components in RTSJ. In this paper we extend our view in order to build the inner structure of this model based on a set of assumptions that satisfies the requirements mentioned in the last section. We will provide the architecture of this model and we will show how it can simplify and enforce the RTSJ memory access rules. Then, we show some patterns that we have proposed to be integrated together in order to support this new memory model.

4.1 The Basic Memory Model Infrastructure

As mentioned in our initial view of this model in [14], in order to develop our proposed *Forked Memory Model*, the following two assumptions have been considered:

- The component can consist of one or more tasks co-operating together to perform a single integrated service offered by this component.
- Each task can be executing a nested set of inner subtasks.

From an RTSJ's viewpoint, we propose that these assumptions can be mapped into the Forked Memory model (shown in Figure 1), where this proposed model consists of:

- **A set of single memory area stacks (SMA_x),** where each memory area stack represents the memory assigned for each inner task *x,* and its nested inner tasks.
- **A single parent memory area (SMA_{parent}),** this memory area acts as a primordial parent memory area (***root MA***) for all memory area stacks (***leaves MAs***) of the tasks within the component.

The configuration, initialization and creation of a memory hierarchy within the component according to this pattern are explained in the following:

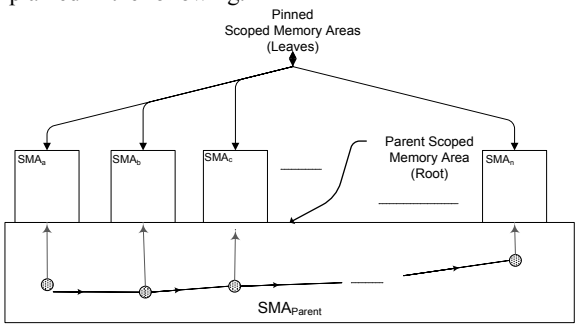

Figure 1: Basic Component's Forked Memory Model

4.1.1 Memory Configuration

The configuration of the memory areas in this model is either:

1. **Configuration of the parent memory area (*root*).**

The parent memory area must be externally configurable by the developer during the *component configuration stage* at design time (in statically created systems) or during runtime (in dynamically created systems, e.g. by dynamic contract negotiation through an external interface).

The memory configuration interface should enable the developer to configure the memory source from which the parent memory would be allocated, according to the required lifetime and requirements of the program using this component; e.g. for non-real-time applications, it can be created from heap memory. Otherwise, for real-time systems, it can be allocated from immortal memory as long as this component is going to be alive for the system lifetime, or from a linear (or variable) time scoped memory area if the component has a shorter lifetime than the program using it. The selection of a certain memory area can be reflected later on using some patterns internally to construct the component's

memory model, e.g. the use of portals of the parent memory areas, as explained later, will be restricted to parent memory area that is configured to be of a scoped memory area type. Hence, we have added a restriction on all the patterns associated with the forked memory model presented in this paper to by assuming that the parent memory area is of a scoped memory area type, where this restriction does not limit these patterns to be extended to work with other memory area types with some modifications.

2. **Configuration of the child memory areas (*leaves*).**
All the tasks' (*leaves*) memory areas are by default configured by the component creator within the parent memory area to be of a single subtype of RTSJ scoped memory areas. However, the developer can externally configure each one of them individually according to the operation specified for each task attached to them. However, he is not allowed to configure them to be created neither from the heap memory area nor from the immortal memory areas; this is a basic constraint, as we assume that these memory areas are only for tasks that have lifetimes less than the program in which they are running in and they require predictable memory management.

4.1.2 Memory Creation

In this section we show how the memory model is constructed and how its parts are integrated together by using our proposed *Multi Named-Object Pattern* in order to enforce the use of RTSJ memory access rules.

At the initialization stage of the component, its memory areas are created as configured at design time. Firstly, the root memory area is created from the component's source memory area. Then, the scoped memory area of each inner task is created. The creation of objects resembling these scoped memory areas (***leaves***) in our proposed *Forked Memory model* is done by an initialization thread running within the *root* memory area (e.g. by the *forkThread* as will be explained later). Moreover, we assume that the references for these objects are allocated within the *root* memory area and all these created references are kept inside a single predefined collection (*memoryForkQueue*) within this root memory area.

Adopting this mechanism enforces the RTSJ memory access rule in this model, as any real-time thread that wants to access an object within any of the *leaf* scoped memory areas has first to enter the parent memory area to be able to get a reference to the required scoped memory area from the references collection, i.e. *memoryForkQueue,* stored in the parent memory area.

As it is created in the root memory area, the *memoryForkQueue* collection is required to be accessible as a shared object from any thread running within the component. Hence, according to the RTSJ memory model, it has to be saved as a memory portal object for the memory area in which it is created, i.e. the root memory area. However, RTSJ allows the definition of only one single object to be a portal for any scoped memory area and using it. Hence, in order to extend the use of the portal for multiple objects, i.e. objects other than the memoryForkQueue, in our proposed model, we assumed the new simple pattern "*Multi-Named Objects Portal*" pattern (MNPORTAL), in which shared objects are given names in order to be accessed by these names through a single portal of an RTSJ scoped memory. This pattern (shown in Figure 2) assumes that the portal of the scoped memory is assigned a single object as required by the RTSJ. This shared object, the MNPORTAL in our model, is a shared linked list object, where each element in this linked list is an object holding both a reference to a shared object created in the memory scope, and a string name assigned to this reference. As seen in Figure 2, we assumed a simple shared-object naming scheme in which the name of the object to be saved is the same as its object references preceded by the prefix "ref". Hence, according to this proposed pattern, entering a *leaf*-scoped memory is done using the following three predefined step:

- Retrieve the object saved as portal of the root memory by calling the method `getPortal()`, defined in the `javax.realtime.ScopedMemoryArea` class, and then cast it as `MNPORTAL` object.
- Retrieve the shared object representing the *memForkQueue* collection from the `MNPortal` object using the proposed method `MNPortal.getObject ("refMemForkQueue")` and cast the returned object as `memForkQueue` object.
- Retrieve the reference object of the required scoped memory using its predefined name from the `memForkQueue` object using the method `memFork Queue.getScopedMemoryRef("refRequiredScope dMemory")`. Where the string `"refRequired ScopedMemory"` is the name of the *Required ScopedMemory* shared object.

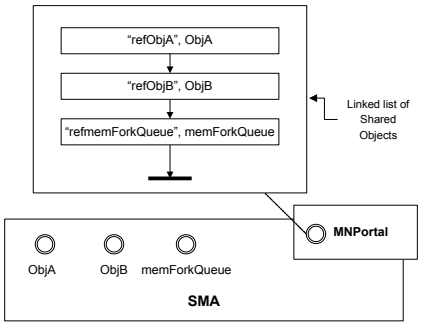

Figure 2: The Multi Named-Object Portal Pattern

4.2 Scoped Memory Lifetime Management

In the proposed structure for the component presented above, the memory is composed of a set of nested memory stacks, these memory stacks can be shared among the schedulable objects running within this component or using it. Hence, their lifetimes must be well defined and manageable.

In the RTSJ, a certain scoped memory area is assumed to be valid as long as there is at least one schedulable object running inside it either explicitly using `ScopedMemory-Area.enter()` or `MemoryArea.executeInArea()` family of methods, or implicitly as its default memory area. This model was presented as a memory management model that predictably and dynamically manages memory de-allocation to avoid the unpredictability due to Java's garbage collector. However, this imposes a restriction on using shared objects created within these scoped memory areas. Although shared objects are handled in RTSJ's memory model using a portal object defined for each scoped memory area instance, the model assumes that the shared object (or any other object in the memory area) will be valid only as long as a schedulable object is running in it. This has led to the use of the *wedgeThread* pattern presented in [1].

A wedge thread is a real-time thread with a high priority that does nothing more than that it enters the scoped memory area and waits inside it as long as there is a shared object(s) that is needed to be accessed from this scoped memory area by a schedulable object which has not yet entered it. The use of this pattern has an overhead due to the amount of resources needed for the wedge thread. This overhead becomes greater when there multiple shared objects exist and each one of them is created in a different scoped memory area, in this case a wedge thread will be required for each scoped memory scope area to keep it alive [12]. In future releases of RTSJ, a *pinned* memory model is to be added to the specification. Here, we try to propose an enhanced solution that can be applicable for applications running on JVM built based on the current RTSJ specification. Our solution is dependent on two new integrated software patterns, the *forkThread* pattern, and the *dualFork Pattern*. The structure of these patterns and how they work are explained in the next sections.

4.2.1 The ForkThread Pattern

In the *forkThread* pattern, instead of creating a single wedge thread for each scoped memory area, we assume that we use a single thread for *all* scoped memory areas that have a common parent. A simple illustrating diagram for this pattern is shown in Figure 3. In the diagram, all the scoped memory areas share the same parent memory area. Hence, a real-time thread can enter them all at the same time by making a sequence of a pair of `MemoryArea.enter`, and `MemoryArea.executeIn Area`() calls. Then finally, it waits either in the last scoped memory area or in the parent memory area.

Due to the RTSJ memory access constraints and the nesting required, implementation of this pattern in RTSJ is not a trivial task. Here, we provide our own implementation based on our proposed component memory model presented above. In our proposed implementation we assume that all the scoped memory areas to be kept alive are all those saved in the `memoryForkQueue`; and sharing the same common parent (the component common memory). Also, we assume that the shared objects are accessible through the multi named-object portals defined for each scoped memory area as mentioned before.

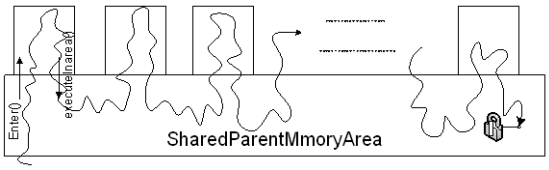

Figure 3: ForkThread Pattern

Figure 4 shows a detailed sequence diagram of the pattern implementation based on our component model. The sequence of operations in this pattern is summarized as follows:

1- The real-time thread, `forkThread` object, is created and starts in the common memory of the component.

2- The `forkThread`, retrieves the `MNPortal` object of the common memory.

3- A reference of the `forkedMemoryQueue` is retrieved from the `MNPortal` object, Then,

4- the first scoped memory area is retrieved from the queue.

5- The `forkThread` starts a recursive propagation process among all the scoped memory areas saved in the queue. The recursion process is a repeated sequence of two operations:

a. Entering scoped memory area using a reference retrieved from the `memoryforkQueue`.

b. Executing back into the common memory area.

The implementation of these two operations is done using an enhanced version of the *encapsulated runnable* pattern presented in [1], where each operation is implemented as a runnable object and then executed by the `forkThread` when it enters (or calls `executeInArea`() of) the appropriate scoped memory area. These two operations are processed by running three encapsulated methods. Hence, as shown in the sequence diagram, they require three nested runnable objects as follows:

- `enterBranchRunnable`: represent the method of entering the scoped memory area of a certain *leaf* memory area
- `executeInComMemRunnable`: for executing back in the common memory area.
- `enterNextBranchRunnable`: to call recursively the two operations mentioned above for the next scoped memory area (leaf); if there is any more existing ones.

6- The termination of the propagation operation. This operation acts as a terminator of the recursion process defined in the previous step, i.e. instead of entering the next scoped memory area, the `forkThread` stops propagation at this memory area waiting for external notification. This operation is again implemented using the *encapsulated runnable* pattern, where a single runnable, `TailRunnable`, holds the necessary code for causing the `forkThread` to wait at the current (last) scoped memory area.

The recursive nature of the `forkThread` pattern requires careful consideration of the creation of objects to avoid waste of memory resources. `Runnable` objects are the main objects used as they encapsulate the methods that constitute the recursion process mentioned above. These objects can be created using the RTSJ based class factory pattern, presented in [2]. However, for optimizing the memory usage, we enhanced this pattern to support object reuse. In our proposed model, these objects are created in the common memory area as singletons, i.e., the `RunnableFactory` creates these objects when they are requested for the first time only to execute their associated methods. The runnable factory inserts named references to each one of these objects in the common memory's multi named-object. Then, each time one of them is requested, the runnable factory checks the common memory's multi-object portal to retrieve a reference to it and forward it to the requestor `forkThread`. To be reusable, these objects are created as a parameterized object, i.e. the objects that are used from within the `run()` methods of these objects and which varies from one call to another are defined as member variables that can be set at runtime before running the methods. These values can be set either from a set of predefined named shared variable in the common memory or through an accessory method, e.g. `setParameters (Object[] parameters)`. These parameters have to be available in the common scope stack of both the assigning and assigned object (in this case, the `RunnableObject`). The choice between the two methods is a developer choice according to the runnable object access requirements, and the type and location of the parameters, e.g. assigned object parameters already in the component memory are loaded from a portal, whereas primitive parameters are passed using access method(s).

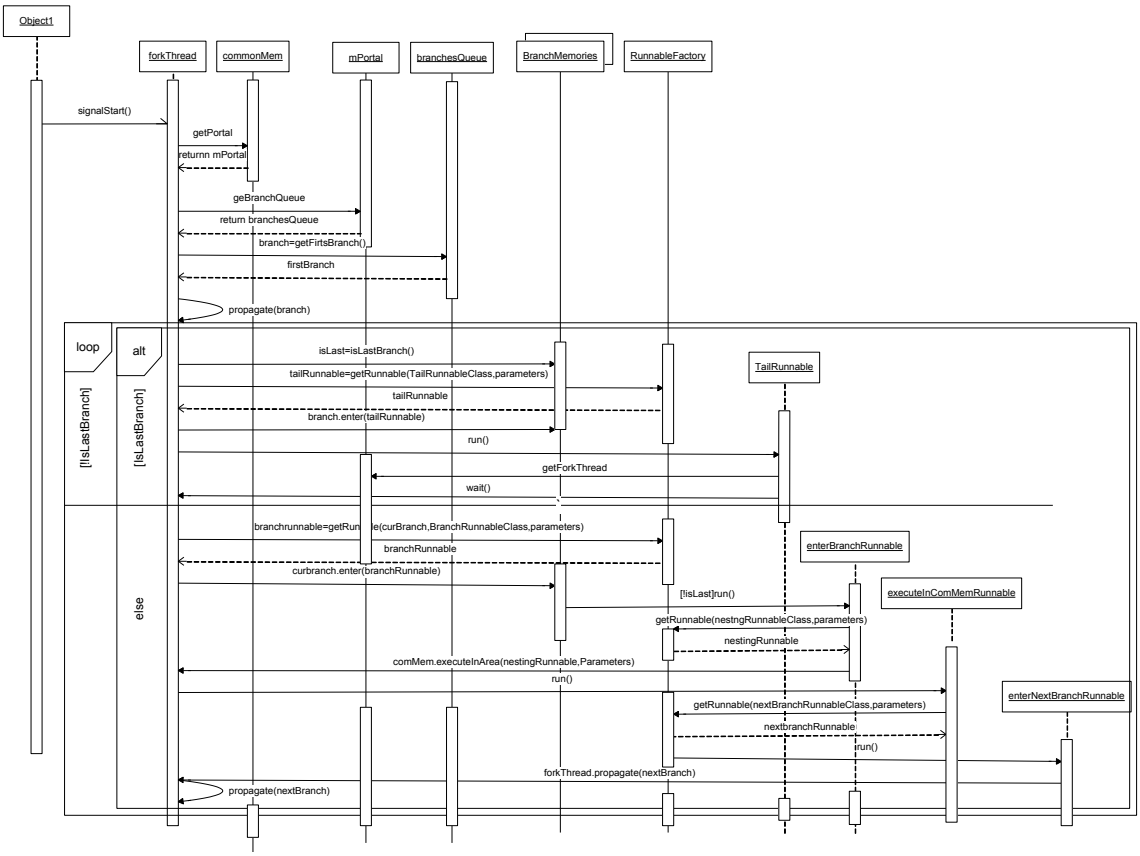

Figure 4: ForkThread Sequence Diagram

To enforce this model, we assume that all `Runnable` objects in our component model, extend the interface `IRunnableWithParameter`, which is defined in Figure 5. Hence, for each `Runnable` object `Robj` in our proposed model, a class `RobjClass` will exist that is defined according to the template shown in Figure 6.

```
interface  IRunnableWithParameters extends Runnable {
    public setParameters(Object[] parameters); }
```

Figure 5: Interface IRunnableWithParameters

```
public class RobjClass implements IRunnableWithParameters
{ //Define here fields representing the method parameters
        RobjClass()
        {
        //No-arg constructor needed for object creation,
        //and load parameters from shared memory
        }
        setParameters(object[] parameters)
        {//Assign values to the fields representing the
        //method parameters
        }
         run()
        {//Encapsulated method's body
        }
}
```

Figure 6: Templates for Classes of Reusable Runnable Objects

Limitations of the forkThread pattern

The *forkThread* pattern enables the pinning of a set of RTSJ scoped memory areas however, it is not flexible enough to handle dynamic creation and deallocation because only two variations of the *forkThread* pattern can be built according to the scoped memories list associated with it:

1. A *forkThread* with a *fixed scoped memory list*. Here neither individual addition nor individual removal of scoped memory areas to/from the `forkedMemoryQueue` is allowed. Hence, applying this model is very restrictive to components that have internal memory stacks with a requirement to be waiting for a certain event (e.g. waiting for other schedulable objects to enter them), and then unpinned when there is no longer need for keeping all of them alive at the same time. Once unpinned, objects in these scoped memory areas can be deallocated if there is no other schedulable objects running in them. Hence, this model is not suitable for systems with dynamic scoped memory creation, or deletion, which is one of the main purposes of RTSJ scoped memory areas. One simple use of this form of the *forkThread* pattern is in building components with pre-initialized memory structure, i.e. the inner memory areas of the component are created and kept alive using a fork thread running within them. When all schedulable objects enter the memory areas created within this component, the fork-thread, can be un-forked to save some of the system resources.

2. A *forkThread* with *append-only/remove-last scoped memory list.* This model is a simple extension of the previous

one, by allowing the *forkThread* to propagate only within a new memory area(s) appended to the end of its associated scoped memory list, or de-propagate from the last memory area(s) removed from its associated memory list. This model does not add much to the previous model, as it only relaxes the restrictions on a subset of the associated scoped memory areas (i.e. the memory areas appended or removed to/from the tail) not all of them.

4.2.2 DualForkThread Patterns

In order to remove the limitations of the *forkThread* pattern, we propose the *dualForkThread* pattern. The *dualForkThread* pattern is formed simply of two individual fork-threads that are running concurrently and cooperatively to achieve the requirement of keeping a set of scoped memory areas alive as long as needed, where this set of scoped memory areas is a dynamic set, i.e. inserting new scoped memory areas or removing some existing ones is allowed during the lifetime of the *dualForkThread*. To explain how the *dualForkThread* pattern works, the diagram shown in Figure 7 shows the different states of the two fork-threads (T1, T2). This is explained in the following:

(a) Initially, there is no scoped memory areas assigned to the two threads so, they wait in the parent memory area.

(b) Once a scoped memory area set has been created, the first thread T1 propagates among all scoped memory areas defined in it, as explained before for the *forkThread* pattern, and finally waits at the last scoped memory area, while the other thread T2 is still waiting in the parent memory area.

(c) Then, at any time, a new scoped memory area can be inserted anywhere into the scoped memory area set hence, the two threads are required to take actions to update their state.

(d) So, the second thread starts to propagate within all the scoped memory areas within the memory areas set including the new one, and finally stops and waits at the last one.

(e) Once the second thread arrives at the last scoped memory area, it notifies the first one to de-propagate, i.e. to exit from all scoped memory areas it has entered before and then it stops and waits at the parent memory area.

(f) Hence, in this manner, the *dualForkThread* pattern enables dynamically inserting scoped memory area(s) to its associated list, which was not possible with the *forkThread* pattern.

(g) Now assume that one of the current scoped memory areas within the memory set is not needed any more and it is required to release it. So, the two threads swap their actions done in steps (c, d, e), i.e. thread T1 propagates within the scoped memory area set, which does not include the removed scoped memory area, then it stops and wait at the last scoped memory area however, before stopping, it notifies T2, to de-propagate back to exit all the scoped memory areas it has entered before,

(h) Hence, as the removed scoped memory area has no thread running in it, it can be freed (if it has no more schedulable objects running within it), and the threads waits for new requests for updating its status or for terminating.

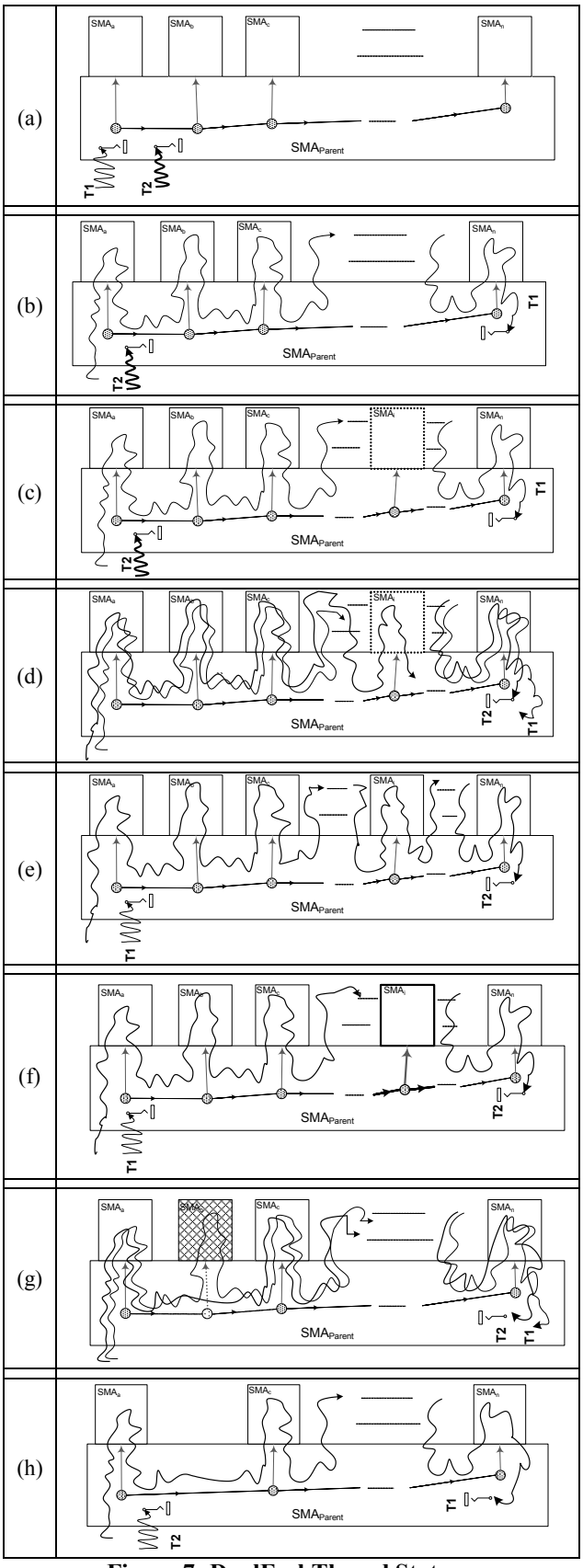

Figure 7: DualForkThread States

In our design, we assume that the *dualForkThread* pattern with its associated concurrency control mechanism of the operations explained above is encapsulated into a single control component, where the main function of this component is initializing and managing two fork thread object (T1,T2) as shown in the activity diagram given in **Figure 8** where T1 and T2 have the same sequence of activities along their lifetime within the component, and each one of them is either in one of four states:

a- Waiting for a request to propagate.

In this state, the fork-thread is waiting for an update command on a `forkLock` object. The `forkLock` object is a simple object saved in the shared parent memory area accessible by both fork-threads. Once an update command is received, a signal is sent to the `forkLock` to notify both fork-threads to start to propagate through the scoped memory areas, if both of the fork-threads are waiting in this state, i.e. no scoped memory areas were associated with the dual-fork threads. Then, only one of them is woken, while the other is enforced to wait for the next update operation to be woken, the choice of the thread to be woken is simply done by testing a simple primitive value *turn* defined in the dual-fork object which alternates its value between 1 and 2 on each update call. However, before the woken thread proceeds, it has to be confirmed that the other thread is not active, i.e. it is not currently propagating or de-propagating, to ensure exclusive operation of each of them. Hence, it has to check if the other thread is in states (b or d). If the other thread is neither in state (b nor d) then, this thread moves directly to state (b). Otherwise, the thread stops again on another shared object `updateLock` waiting for a notification signal from the other thread to inform it that it is safe to continue to move to state (b).

b- Propagating through the scoped memory area set.

In this state the thread recursively propagates to enter all the current members of its associated scoped memory set. Once, the thread enters the last scoped memory area, it becomes safe for the other thread to continue propagation if it is waiting for this thread to finish propagation. So, this thread sends a signal (`updateFork.notifyAll()`) to release the other thread.

c- Stopping in the last scoped memory area.

In this state, the thread stops waiting for the other fork-thread to receive a command to update in order to replace this thread. Once the other fork-thread receives the update command and propagates and reaches the last scoped memory area of the scoped set, it sends a notification signal (`updateLock.notifyAll()`) to this thread to release it and enable it to move to state (d).

d- De-propagate back to the beginning.

In this state the thread exits of all the scoped memory areas it is currently entering them, i.e. it returns back to its initial state (a). Once it returns to the initial state, it sends a notification signal (`updatLock().notifyAll()`) to enable any waiting fork-thread to start to propagate as explained before.

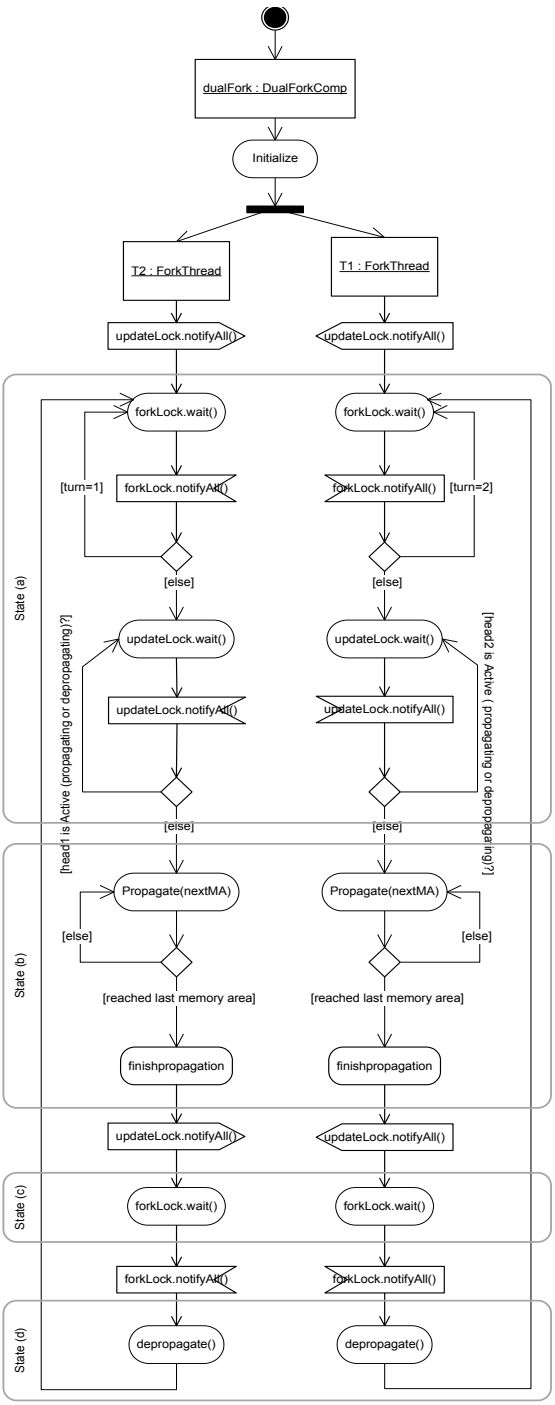

Figure 8: Activity Diagram of the DualForkThread Pattern

5. PATTERNS TO SUPPORT INTERNAL TASKS

As assumed in our model above, any component will consist of one or more internal tasks that collaborate to perform the component's functionality. Hence, the memory model of each task constituting the component must satisfy the following:

- It is subject to the RTSJ memory rules.
- It has well-defined mechanisms for communication and integration with other tasks within the same component and, if required, with other tasks in other components.
- It must provide flexibility to the user in configuring the task function(s).

In order to satisfy these requirements we present two different patterns for the tasks, and we show how they can cooperate with other tasks in the same component and how they can be exposed if (necessary) to work with other components.

5.1 The Executable Runnable Stack Pattern

In the *Executable Runnable Stack* pattern, as shown in Figure 9, the internal tasks can be modeled as a set **DOPS**, which consists of *n mostly dependent operations* Op_i that runs in sequence, i.e.

$$DOPS=\{Op_0, Op_1, Op_2, ..., Op_n\} \text{ where,}$$
$$Op_0 => Op_1 => Op_2 => ..... => Op_n$$

This pattern of execution is very common in some real-time systems; one example of its possible use is in systems manipulating images, e.g. pattern recognition applications, where the processing of the image goes through several steps, e.g. image capture, noise filtration and enhancements, features extraction, recognition, and finally the required action depending on the recognized object. In this example, the execution of each stage can be run in different ways according to different algorithms, filters, mechanisms etc. From an RTSJ memory management perspective, the data used by these operations can be modeled as a stack of scoped memory areas. A real-time thread can run the sequence of these operations to handle the image data at each stage in different memory areas either:

- **Top-Down**: if the data used and created need not to exist during the next operation, the real-time thread can start the first operation by entering all the scoped memories in the stack until the top scoped memory area, where it executes the first operation. Then, it exits this memory area and returns to the one just beneath it in the stack to execute the next operation, and so on, until it finishes the execution of all operations in the stack. In this approach, the objects allocated during any stage is allocated at the scoped memory area assigned for it, and it is deallocated once the operation finishes to enhance memory usage. To enable an operation (Op_x) to pass an object to another operation following it (Op_y where y>x). The source operation Op_x must allocate this object in the scoped memory area of the other operation Op_y, to satisfy the RTSJ memory access rules.
- Or, **Bottom-Up**: this approach can be useful when objects and data created during an operation are not to be de-allocated during the execution of any of the following operations (e.g. to enable re-execution of the same operation).

Building components to support these kinds of applications should be flexible enough for configuration by the developer at design time, or even by the user at run time. Our proposed pattern structure assumes the following as a base for building such components:

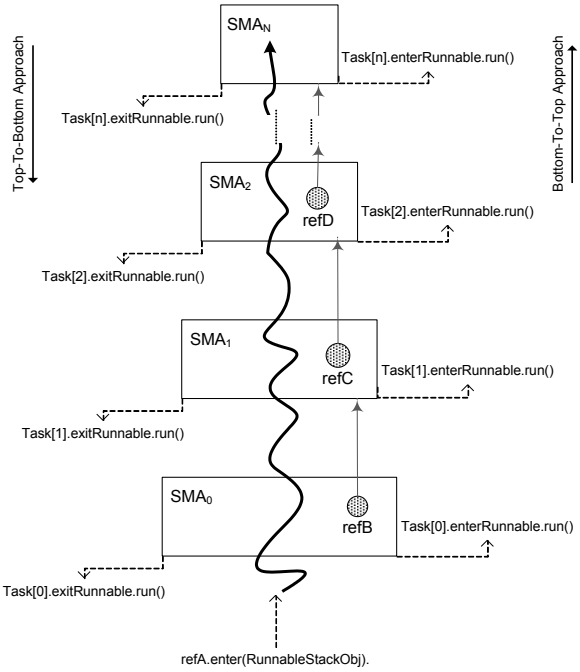

Figure 9. The Executable Runnable Stack Pattern

- Only one RTSJ schedulable object (e.g. real-time thread) is executing all the operations defined in the set DOPS.
- Each operation Op_i can be executed in a single configurable scoped memory area.
- Each operation Op_i is dependent on a previous operation Op_j where j<i.
- Shared Objects created during the execution of an operation Op_j can be accessed by one of its following operations Op_i through the portal object of scoped memory SM_j where, SM_j is the scoped memory in which Op_j has been executed and these shared objects are a result of its execution.

According to these assumptions the *Executable Runnable Stack* pattern can be constructed and created by the following elements (see Figure 10):

1- An Executive Actor: This element defines a configurable RTSJ schedulable object, which represents the executor element running along the memory areas constituting the stack to run the operations defined in the DOPS. This schedulable object can be a `RealtimeThread`, a `NoHeapRealtimeThread`, or even an `AsynchronousEventHandler`. This element can be assigned statically at design or run time, or it can be assigned dynamically at run time, e.g. from a schedulable object pool.

2- Multi Named-Object Portal. This element is created dynamically for each scoped memory area constituting the stack. This element uses the Multi Named Object pattern to define a portal for the scoped memory. As described before, this pattern enforces the single parent memory access rules defined by RTSJ. Where the reference of each memory area SM_i associated with the operation Op_i, is saved within its direct parent memory area SM_J (i.e. j=i-1).

3- Runnable Stack: This stack is composed of *n* configurable objects defining operations executed by the executive actor, where n is the number of operations defined in the DOPS set defined above, i.e. it has an object corresponding to each

operation Op_i. This object defines the following for each operation:

a. Scoped Memory Type, Name and Size. Associates a scoped memory with a certain size and name to be the allocation context during running the operation Op_i.

b. Entry Runnable Class: This is useful in case of using the Bottom-Up approach as it defines the encapsulated method that contains the operation's logic to be executed each time the executive actor enters the scoped memory assigned for this operation.

c. Exit Runnable Class: this can be useful to run Top-Down approach, where it defines an encapsulated method that holds the logic to be executed each time the executive actor finishes execution inside the associated scoped memory and it is about to leave it.

The Runnable Stack Object itself is a runnable object that encapsulates the method that executes the logic that is responsible of building and propagating the executive actor along the nested scoped memories defined by the runnable stack. So, entering the Runnable Stack Object starts the executive actor.

5.2 The Executable Runnable Fork Pattern

The Executable runnable stack pattern is suitable more to model dependent tasks, while the *Executable Runnable Fork* pattern, as described here and shown in Figure 10, is suitable for modeling a set of mostly *parallel independent* operations ***PIOPS*** that consists of m different operations Op_x, i.e.

$$PIOPS = \{Op_1, Op_2, \ldots, Op_m\}, \text{ where}$$
$$Op_1 || Op_2 || \ldots || Op_m.$$

One example of a component that can be built using this pattern is in multiple-producer single-consumer structures, where multiple products are produced independently of each other but have to be handled by a single consumer. The waiting queue of a call-request-handling server is one such example of this pattern, where an object is created to encapsulate each call request received by the server thread, this object is created in a separate scoped memory area, the memory pattern created then looks much like a fork. On the other hand, the severer thread uses a certain policy, (e.g. FCFS, HPF, etc.) to select the appropriate call-request object to be handled in sequence. For this pattern, we have assumed that to execute all input operations defined in the PIOPS set, either more than one RTSJ schedulable object (e.g. real-time thread) can be responsible for creating the shared objects created by the producers (request-call objects) to the component, or a single schedulable object is creating them from multiple input sources. Hence,

- Each operation Op_i can be executed in a single configurable scoped memory area.
- Each operation within the task Op_i is mostly independent of the other operations Op_j where $j \neq i$.
- Shared objects created during the execution of an operation Op_j can be accessed by one of the other parallel operations Op_i either by:
  - Using the *handOff* Pattern [1]: when the shared objects are created in the scoped memory SM_j of this operation.
  - Or, by using the *sharedMemory* pattern: where the shared object is created inside the component shared memory.

According to these assumptions the *multiple producer-single – consumer* version of *the Executable Runnable Fork pattern* can be constructed and created from the following elements (see Figure 10):

1- The Producers: The producers are the set of RTSJ schedulable objects that are responsible of producing the shared objects used within this pattern. Each one of these schedulable objects runs in a separate memory stack originating from the component common memory. Hence, these producers should be configured by the following:

a. Total number of producers

b. Scheduling, release, memory parameters of each of these schedulable objects

c. The classes encapsulating the methods implementing the runnable logic assigned to each of these objects.

2- The Consumer: The class that implements the runnable logic of the consumer is an RTSJ schedulable object that can access the shared objects created by the producer(s) in order to process them. As a schedulable object, the scheduling, release, memory parameters associated with it should be reconfigurable according to the logic assumed for this component.

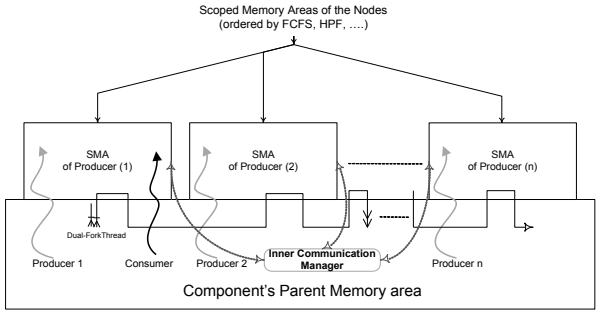

Figure 10: Executable Runnable Fork Pattern

3- Inner Communication Manager. Two types of communication managers can coexist in this model. A shared memory manager and *handoff* memory manager. These models are needed to enable communication among the internal schedulable objects running in different memory stacks within this model. The choice of one of them for communication is related mainly to the lifetime of the object that needed to be shared as explained below (also see Figure 11).

a. Shared Memory Manager. This model of communication among the producers is appropriate when the shared object is needed for the lifetime of the component. To enforce the single parent rule, the common memory of the component acts as a shared memory among all its nested memory stacks of the internal schedulable objects. Hence, objects saved in this memory area are accessible by all schedulable objects (producers or consumers). The shared memory manager is a multi-name portal object that acts as the portal of this shared memory area. So any producer schedulable object can create an object and save a named reference of it in the shared memory manager. Later when one of these saved shared objects is required, the requestor retrieves this named reference from the portal, as described before, in order to access it.

b. Inter-scopes Memory Manager. Although the shared memory model is a straightforward mechanism of communication, it is not necessarily the optimal solution of communication among RTSJ schedulable objects, as it

assumes that the shared object live the lifetime of the scoped memory containing it. However, sometimes the created shared objects are just temporary objects created for short periods. So, a model that enables inter-scope communication can be the best solutions in such cases. The inter-scoped memory manager can be built using the *handOff* pattern presented in [1].

4- Internal Lifetime Controller. This can be an instance of either a fork-thread or a dual-fork thread described earlier in this paper to manage the lifetime of the scoped memory area stacks of the inner schedulable objects.

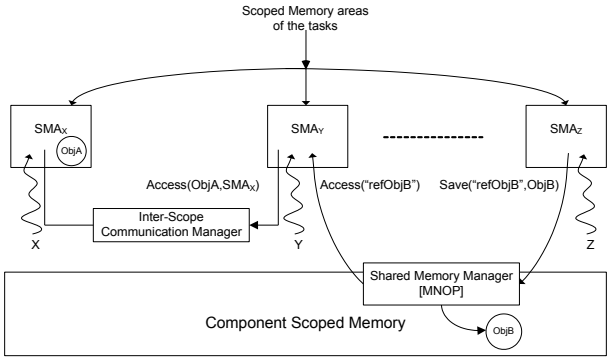

Figure 11: Models for Internal Communication

6. CONCLUSION

In order to satisfy the predictability and timing requirements of real-time systems, the RTSJ presented new memory and scheduling models that presented a new set of constraints and limitations to the Java programming models. Consequently, current software design pattern cannot be directly applied in building RTSJ based applications.

We have presented in this paper a set of software patterns that can be integrated to build reusable software components based on the RTSJ specification. These patterns include patterns for defining an easy RTSJ compatible component memory model that enforces the constraints of the RTSJ memory model, and provide easy to configure and use elements. Moreover, we associated with this a life manager subcomponent to manage the lifetime of the elements that constitute the proposed model. The patterns used to build the memory model have been designed considering reusing objects to avoid the waste of systems resources, which are very limited for many real-time systems.

Moreover, we presented execution patterns for running real-time tasks on the proposed component memory model. In these execution patterns, we investigated how to integrate the proposed memory model elements with schedulable object elements to support building re-configurable real-time components based on the RTSJ.

In the future we intend to extend this model by integrating it with a set of communication patterns in order to build a distributed component model. These communication patterns are to be configurable to support either synchronous or asynchronous communication. Moreover, we aim to extend our model to be integrated with our approach for real-time mobility, proposed in [14], in order to build distributed components that have the capability to migrate in real-time among the nodes of a distributed real-time system.

7. REFERENCES

[1] Pizlo F, Fox J, Holmes D, Vitek J (2004) "*Real-time Java scoped memory: design patterns and semantics*". In: Proceedings of the IEEE international symposium on object-oriented real-time distributed computing (ISORC'04), Vienna, Austria, May 2004

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

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

[4] 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.

[5] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, and R. Klefstad. "*Patterns and Tools for Achieving Predictability and Performance with Real-time Java*". 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), pages 247-253, 2005.

[6] J. A. Colmenares, S. Gorappa, M. Panahi, and R. Klefstad. A Component Framework for Real-time Java. 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), 2006.

[7] 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.

[8] A. Plšek. P. Merle, L. Seinturier. "*A Real-Time Java Component Model"*. In Proceedings of the 11thInternational Symposium on Object/Component/Service-oriented Real-Time Distributed Computing (ISORC'08), pages 281{288, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[9] A. Plšek, F. Loiret, P. Merle, and L. Seinturier. A Component Framework for Java-based Real-time Embedded Systems. In Proceedings of 9th International Middleware Conference, Leuven, Belgium, December 2008.

[10] Thierry Coupaye, Jean-Bernard Stefani: Fractal Component-Based Software Engineering. ECOOP Workshops 2006: 117-129.

[11] Bruneton E., Coupaye T., Leclercq M., Quéma V. and Stefani J-B. "*The Fractal component model and its support in Java"*, Software - Practice and Experience, Volume 36, p1257-1284, 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] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, D. Watkins. "*Making Components Contract Aware"*, in: *IEEE Computer*, July 1999, vol. 32, no 7.

[14] M. AlRahmawy, A. Wellings. "*A model for real- time mobility based on the RTSJ*". In Proceedings of JTRES'2007, pp.155-164