Applying the CBSE Paradigm in the Real Time Specification for Java

Jean-Paul Etienne Julien Cordry Samia Bouzefrane

Laboratoire CEDRIC
Conservatoire National des Arts et Métiers
252, rue Saint-Martin
Paris, France
{jean-paul.etienne, julien.cordry, samia.bouzefrane}@cnam.fr

ABSTRACT

During the past few years, Component Based Software Engineering (CBSE) has emerged as an excellent candidate to achieve greater software understanding, reuse and reliability. Through our research work, we have designed a component model that provides abstractions and means to facilitate design, analysis and validation of real-time systems. In this paper, we investigate the applicability of our model in the Real Time Specification for Java. We also show how our framework provides a programming model that facilitates real-time development using RTSJ by abstracting away memory management and traversal complexities.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects, control structures, frameworks, patterns*

General Terms

Design, Experimentation, Languages

Keywords

Component, Programming model, scoped memory management, RTSJ Design Patterns

1. INTRODUCTION

With the growing complexity of real-time systems, new methodologies are needed to facilitate design, implementation and maintenance of such systems, while providing means to capitalize software development. Such methodologies should rely on bulletproof concepts such as **increased level of abstraction**, to reduce software complexity and **software reuse** to boost software development and ease certification. Component Based Software Engineering (CBSE) [9] provides answers to these needs by offering effective structuring mechanisms for achieving greater software understanding, reuse and reliability. Through our research work [6] [7], we have developed a component model dedicated to the development of real-time systems, which takes into consideration commonly used abstractions of such systems, provides means to cater for temporal properties of the component architecture, and provides techniques to analyze and validate a system of real-time components. In this paper, we investigate the feasibility of our component model in Real-time Java.

Though Real Time Specification for Java (RTSJ) [3] leverages Java technological benefits for the development of real-time and embedded systems, its new memory model, as well as unusual memory management features, makes real-time application development a rather painstaking activity. As such, through our framework, we also seek to provide a programming paradigm that will enable the developer to design real-time applications using RTSJ, without worrying about how to deal with the application memory architecture and management.

In this paper, we start by giving, in section 2, a brief introduction to RTSJ, as well as the implications involved when developing real-time applications using its constructs. Thereafter, we present the requirements that need to be fulfilled when applying the CBSE paradigm to RTSJ. Our real-time component model is described in section 3. In section 4, we describe the general memory structure of the component framework and explain the benefits and implications of using such a memory configuration. In sections 5 and 6 we present the RTSJ Component framework, whereby we explain how the various component types are structured. Moreover, we describe the constructs needed to allow client components to be loosely coupled and also explain how the composite entity plays an important role in the framework to handle sub-components management and allow interaction of components found within different scopes. In section 7 we discuss the performance issues regarding the use of the framework. We present related work in section 8 and finally conclude and present future works in section 9.

2. RTSJ

Java's great palette of ingredients(robustness, ease of use, cross-platform capabilities and security features) has made it become a language of choice for general-purpose programming. Nevertheless, it has serious limitations that hinder its applicability for real-time and embedded programming. Indeed, Java lacks predictability primarily due to its automatic

JTRES'06 , October 11-13, 2006, Paris, France.

memory management feature, achieved through automatic garbage collection, and because it does provide weak mechanisms for handling scheduling and priorities of threads. To circumvent these limitations, the Real-Time Specification for Java (RTSJ) [3] has been proposed. To enable development of predictable real-time applications, RTSJ introduces, amongst other constructs, new types of threads (`RealtimeThread`, `NoHeapRealtimeThread`) having precise scheduling semantics and that may run at higher priority (`NoHeapRealtimeThread` only) than the garbage collector, and special types of memory (`ScopedMemory`, `ImmortalMemory`) that are outside the scope of action of the garbage collector to ensure predictable memory access.

However, the introduction of new memory regions complicates the development of hard real-time applications. Since these memory types are not managed by the garbage collector, memory management is put back upon the shoulder of the developer. Moreover, RTSJ does provide unusual memory management features [3] [16]: An object allocated in a scoped memory is available while there is a `Runnable` active within the scope, otherwise scoped memory is freed up, thus deallocating the object. Besides, objects allocated in Immortal Memory remains for the whole lifetime of the application. Furthermore, memory reference, assignment and single-parent rules [16] involve subtle mechanisms to enable interaction between objects within distinct scopes. All these restrictions force the developer to be careful about how to manage an application memory footprint without causing memory leakage and without violating RTSJ's memory reference rules.

Applying CBSE to RTSJ

Apart from investigating the use of RTSJ constructs to structurally implement our component model, we seek to provide a framework that facilitates real-time development using RTSJ. As such, the framework should provide means to ease component development, composition, adaption and reuse, while abstracting away the memory management and traversal intricacies involved generally when developing real-time application with RTSJ. Furthermore, though flexible, the framework should not introduce considerable execution overhead as compared to an RTSJ application developed without CBSE in mind and should not have any memory leaks that would eventually lead to application runtime and allocation errors.

3. REAL-TIME COMPONENT MODEL

Through our research work, we have designed a component model [6] [7] dedicated to the development of real-time systems, which takes into consideration commonly used abstractions of such systems (active, passive and communication entities), provides means to cater for temporal properties of the component architecture and provides techniques to analyze and validate a configuration of real-time components. Furthermore, through our model, we position ourselves at a higher conceptual level, a la ADL ("Architecture Definition Language") [11], and also advocate the use of contractual specifications, which provide us a formal basis for verifying compatibility between components.

Components form the first class entities of our architecture. Component development can be done using any programming paradigm, imperative, object-oriented, etc. However real-time applications are designed exclusively as assemblies of pre-existing components. Thus to facilitate component reuse within the same configuration, a distinction is made between type and instance of components.

3.1 Base entity: The Component

A component is a software entity that interacts with its environment only via well-defined interfaces, making it ready for composition and reuse. To increase potential of reuse, a component will not only describe the functionalities it offers, through so-called offered interfaces, but also those that it requires, through required interfaces, in order to fulfill its role.

Our model comprises four types of components, active component, passive component, connector and composite.

3.1.1 Active Component

The active component is a software entity having its own thread of control. It generally represents real-time tasks. At the temporal level, it includes properties regarding its periodicity, deadline and priority. These temporal specifications enable us to characterize two types of active components, those having a periodic execution and those having an event-based execution (I/O interrupt) or whose execution is initiated by other active components (aperiodic execution). From a functional point of view, the active component does only make use of services provided by other types of components. Thus at this level, it is composed only of required interfaces. In addition to functional interfaces, the active component also comprises a control interface used to direct its execution, for example, to initiate its execution if it is dependent upon external events or other active components. In general, an active component interacts with its counterparts only via communication connectors or passive components.

3.1.2 Passive Component

A passive component, in opposition to its active counterpart, does not have its own thread of control. It generally represents libraries, shared entities or high-level APIs used to access Hardware. When being called upon, its execution is carried out in the context of the active component demanding its services.

3.1.3 Connector

The concept of connector represents the set of remote communication mechanisms (example RPC) used to enable interaction between components from different sites. It can be perceived as a special type of component.

3.1.4 Composite

In our architecture, hierarchical design is also present through the concept of composite. The composite entity allows us to regroup basic as well as composite components to form a reusable entity of a much coarser grain. Moreover, it does not offer or demand more functionalities than those provided or required by its sub-components. As a result, it is equipped only with delegated interfaces, that is, interfaces that are directly connected to the interfaces of the components it comprises. In general, the notion of composite

makes it possible to follow an **extension by composition** design philosophy, where by the functionalities of the component are extended by first plugging its existing offered interfaces to a new component that provides the additional functionalities and secondly by regrouping the components into a composite to facilitate reuse of the enhanced configuration.

3.2 Specification

In our architecture, a component is equipped with a set of well-defined specifications in order to get precise description of what functionalities it offers and requires, thus facilitating its reuse. These contractual specifications, provide information about the structural, behavioral and temporal properties of the component. During the assembly phase, components undergo contract negotiations to ensure conformance of their operational requirements. In general, a component is equipped with four types of contracts, syntactic, assertion, interaction and temporal.

3.2.1 Syntactic contract

The syntactic contract provides a description of the signatures of the component services. At binding time, it ensures structural compliance of components being glued together.

3.2.2 Assertion contract

The assertion contract defines the functional properties of operations in terms of pre-post conditions and invariants. It ensures behavioral compliance of components being assembled.

$$long\ get() \left\{ \begin{array}{l} pre : \neg empty() \\ post :\leftarrow first() \wedge size()' = size() - 1 \end{array} \right\}$$

3.2.3 Interaction contract

The interaction contract describes the component dynamics, by providing a specification of the behavior and interaction of the component, in terms of transitions on its visible operations (offered and required). It is specified using timed-automata [1], since this formalism enables us to cater for both the dynamic behaviors and temporal specifications of the system. In addition, this contract is also exploited to carry out verification of safety and liveness properties of the system.

In our modeling, interaction contract descriptions follow a specification convention for representing method call executions. Each method call is described using two synchronized actions, one for indicating the beginning of its execution and one for indicating its end. For example, given a method *get*, we will have as synchronized actions *get* and *endget*. Furthermore, each beginning action is defined as urgent in order to ensure that the system will not delay if a component is ready to make a method call. The reason for adopting such convention is to enable specification of execution times and temporal requirements of offered and required services respectively. In general, the specification of a service call in the interaction contract of a component is dependent upon whether the service is required or offered by the component. If a component is offering a service, the service execution is specified as follows:

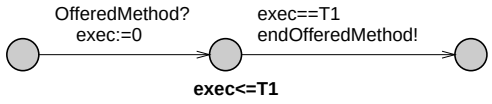

Figure 1: Component offering a service

The variable *T1* represents the execution time of the method. Upon reception of the method call, via the synchronized action *OfferedMethod?*, the local clock *exec* is reset to zero and the automaton moves to the location having invariant $exec \leq T1$. It remains there until clock *exec* reaches *T1*. Once $exec = T1$ the signal *endOfferedMethod!* is trigged to inform the caller that the execution of the method is over. The invariant $exec \leq T1$ is added to the location to force the transition once $exec = T1$.

For the component requesting the service, the service execution is specified as follows:

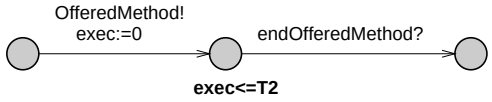

Figure 2: Component requesting a service

Once ready, the automaton of the requester will make the method call via the signal *OfferedMethod!* and resets its local clock *exec* to zero before moving to the location having invariant $exec \leq T2$ where it remains until it receives notification of the method completion. In this example, the variable *T2* represents execution time requested by the component for the method. The invariant $exec \leq T2$ is used as a monitor to verify if the temporal constraint of the required method is met.

3.2.4 Temporal contract

The temporal contract takes into account the temporal properties relative to the execution or interaction of components. As such, it is generally used in conjunction with the assertion or interaction contract to describe temporal properties based on the behavioral specifications of the component. It is also associated with active components to provide information about their periods, deadlines and priorities. Finally it is also used at the composite or architectural level to describe global temporal constraints that need to be satisfied by components or sub-components comprising the composite/architecture. An example of an invariant property, specified in a real-time temporal logic [13], is shown below. It describes the fact that once a Producer component has inserted data in the Buffer, the Consumer component should retrieve it within 10 time units.

$$G((Producer1.put \vee Producer2.put) \rightarrow F_{<10}(Consumer.get))$$

3.3 Compatibility test

In [6] [7], we also shown how compatibility is established based on each type of contract used. In general, at the syntactic and behavioral levels, compatibility is established for

each pair of components having their offered/required interfaces in a binding relationship by verifying that each operation of the required interface has a corresponding offered operation that fulfills its structural and behavioral requirements. At the syntactic level, this verification consists in determining if the types of the input/output parameters of the required and offered operations are in a sub-type relationship. At the behavioral level, compatibility of the offered and required operations is ascertained if the pre-condition of the offered operation satisfies that of the required operation (contra-variance properties) and if the post-condition of the latter satisfies that of the offered operation (covariance properties).

At the interaction level, compatibility is established by verifying that there are no conflicts in the sequence of calling methods between components. This is determined by verifying that the synchronized product of the graphs of their interaction protocols does not deadlock.

Finally at the temporal level, compatibility of the components is verified by ascertaining that the execution times of offered operations are meeting the temporal requirements of the required operations, while guaranteeing that deadline of active components are met during execution. Since we are using timed-automata to model behavioral and temporal properties of components, any incompatibility is detected if the synchronized product of the automata of components deadlocks. In such formalism, a deadlock will arise if there are either absences or miss-sequencings of methods in the interactions or temporal incompatibilities between required and offered operations or deadline misses of active components.

3.4 Schedulability analysis

In our architecture, we perform schedulability analysis of the configuration of components using timed-automata. To reduce state space exploration, the analysis is done on a functionally abstracted version of the system, which considers only active components, together with their temporal properties, as well as system-specific actions (signal, acquire, release, wait) and functional constraints likely to influence their execution. The timed-automata formalism was chosen firstly because its expressive power allows us to consider execution models of much greater complexity and secondly, because it enables us to achieve more precise estimations [4], compared to classical analysis. In timed-automata, the behavior and the interaction of tasks are modeled explicitly. The schedulability analysis is transformed into a reachability problem by performing an exhaustive search of all the possible behaviors of the task set. We perform this modeling using the UPPAAL [15] tool.

In general, the specification and verification techniques presented above help us ensure that real-time application design is sound from both a behavioral and temporal point of view.

4. FRAMEWORK MEMORY STRUCTURE

In our framework, we adopt a memory structure that is patterned after our component model hierarchy, whereby each component is allocated within a distinct scoped memory. This allocation policy allows us to control the lifetime of each component individually while providing an efficient mean to manage the memory footprint of an application. A scoped memory remains occupied while the component allocated within it is active. Once the component is terminated, the memory area becomes available again for reallocation. Thus, this memory structuring allows us to change the configuration of an application at run-time while ensuring memory availability. Such concern is particularly important for highly available real-time systems, whereby undergoing an offline phase to perform software adaptation may lead to fatal failures. However, adopting such a memory structure raises an important issue, which is how to perform component interface bindings without violating RTSJ's memory reference, assignment and single-parent rules [14] [16]. This problem is addressed by incorporating within composite entities glue codes, based on RTSJ's design patterns [2] [14] [12], to ensure seamless component accessibility. As we will see in section 6.3, the configuration of the composite entities will vary depending on the client or server nature of its sub-components.

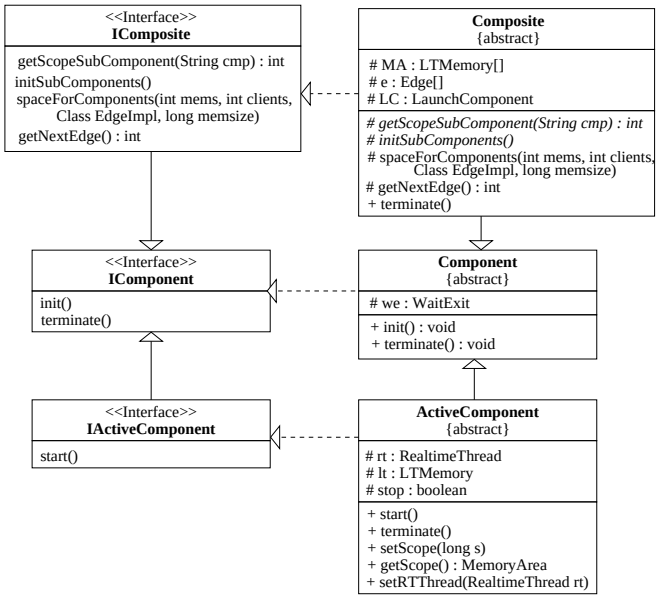

Figure 3: RTSJ Component framework

5. RTSJ COMPONENT STRUCTURE

As shown in figure 3, the RTSJ component framework is composed basically of three abstract classes, corresponding to the active, passive and composite structures respectively. The `Component` abstract class implements the `IComponent` interface, which provides methods for initializing and terminating the component. It incorporates a **Wedge Thread** [12] whose function is to keep alive the `ScopedMemory` within which the component object has been instantiated and initialized (via the `init` method). As explained in [12], the **Wedge Thread** is just a sleeping thread and as such will not perform any processing during the lifetime of the component until being called, via the terminate method, to free the `ScopedMemory`. The `ActiveComponent` abstract class extends the `Component` class and implements the `IActiveComponent` interface, which provides a method to start its `RealtimeThread`. It also includes a `ScopedMemory` used as a temporary execution space for the `RealtimeThread`. Moreover,

the `ActiveComponent` abstract class overrides the terminate method of the `Component` abstract class to include the necessary codes for stopping the `RealtimeThread` and provides additional methods for setting the `RealtimeThread` to be used, as well as the memory area space within which the `RealtimeThread` will execute.

In general, implementing a passive or active component consists in implementing a class that extends the corresponding abstract class (as shown in programs 1 and 2). The writing of the component's inner logic does not require any concern about RTSJ's memory management. The necessary codes to handle these are generated at the composite level. Furthermore, as the structure of the implementing classes are generated automatically from the component model abstract specifications, the developer just has to fill the body of the skeleton methods of the classes. For instance, when implementing an active component, the developer has to write only the inner logic of the `RealtimeThread` (in the `run` method) of the component. The component and its real-time thread structure, as well as its scheduling and temporal parameters are generated automatically following the component's abstract specification. However, as explained in [2], special care needs to be taken concerning the use of standard Java libraries. The developer has to ensure that these libraries do not cause memory leakage within the allocation context of the component.

```
public interface IBuffer {
    public void put(int i);
    public int get();
    public boolean full();
    public boolean empty();
}

public class BufferOne extends Component implements IBuffer {
    private int[] list;
    private int head;
    private int tail;
    private int numofelements;

    public BufferOne()
    {
        list = new int[1];
        head=0;
        tail=0;
        numofelements=0;
    }

    public void put(int i)
    {
        list[tail]=i;
        tail=(tail+1)%list.length;
        numofelements++;
    }
    ...
}
```

Program 1: A passive component implementation class

Specifying offered and required services

In our framework, a component offers a particular service by implementing an `interface`, which provides the signatures of the methods characterizing the latter (as shown in program 1). When being used thereafter, the component is referenced and accessed only via the interfaces it implements. This allows us to gain access to components services without explicitly accessing their implementations. As a result, implementation change or update is facilitated as the overall application architecture need not be reviewed each time a component is substituted or updated. On the other hand, services required by components are made explicit through the **Dependency Injection** pattern [8]. This mechanism allows a component class to be loosely coupled by inverting how it obtains external component references. This is achieved through the use of private interfaces. The concrete component references are provided externally. In particular, we use a form of dependency injection called **Interface Injection**. As shown in program 3, any component depending on one or more external components needs to implement the `IBindController` interface to provide the methods required to enable binding of its private interfaces with concrete component references.

By and large, a component offering one or more services will implement one or more corresponding interfaces and a component requiring one or more services will encompass one or more corresponding private interfaces, which will be glued at binding time with concrete implementations.

```
public class Producer extends ActiveComponent implements
IBindController {
    private IBuffer b;
    private IMutex m;
    ...
    public Producer()
    {
        SchedulingParameters sp = new PriorityParameters(...);
        PeriodicParameters pp = new PeriodicParameters(...);
        SizeEstimator s1 = new SizeEstimator();

        s1.reserve(MyHeap.class, 1);
        setScope(s1.getEstimate());
        setRTThread(new MyHeap(sp,pp,getScope()));
    }

    public void bindInterface(String itfName, Object o)
    {...}

    public void unbindInterface(String itfName)
    {...}

    private class MyHeap extends NoHeapRealtimeThread
    {
        private boolean con;
        ...
        public MyHeap(SchedulingParameters sp,
        PeriodicParameters pp, MemoryArea ma)
        {
            super(sp,pp,ma);
            ...
        }

        public void run()
        {
            int i=0;
            while(con && !stop)
            {
                m.acquire();

                if(!b.full())
                {
                  b.put(i);
                  ...
                }
                else
                {...}
            ...
            con=waitForNextPeriod();
         }
    }
}
```

Program 2: An active component implementation class

```
public class MyDependentComponent extends Component implements
IBindController {
    private IOneInterface a;
    private IAnotherInterface b;

    public void bindInterface(String itfName, Object o)
    {
         if (itfName.compareTo("IOneInterface")==0) then
            a = (IOneInterface)o;
         else if (itfName.compareTo("IAnotherInterface")==0) then
            b = (IAnotherInterface)o;
    }

    public void unbindInterface(String itfName)
    {
        if (itfName.compareTo("IOneInterface")==0) then
            a = null;
        else if (itfName.compareTo("IAnotherInterface")==0) then
            b = null;
    }
    ...
}
```

Program 3: Dependency injection

6. RTSJ COMPOSITE STRUCTURE

The composite entity is much more of interest as regards to RTSJ constructs. In our model, the composite is a mere structuring entity which allows component reuse at a coarser grain. In the RTSJ framework, it also handles scoped memories of sub-components, initialization, starting (active components), termination and interface bindings of sub-components, as well as concurrent access. As shown in figure 3, the `Composite` abstract class implements the `IComposite` interface. It maintains a list of scoped memories, each of which corresponding to the memory area of a distinct sub-component, and a pool of `Edge` runnables, whose working mechanisms are inspired by the **Handoff** [12] and **Cross-Scope Invocation** [14] patterns, to allow components in distinct scopes to communicate without violating memory access rules. The reason for using a pool of such runnables is to enable concurrent access to sub-components while providing means to prevent scope memory leakage. In fact, the use of a pool prevents new `Edge` runnable instances from being created within the allocation context of caller components upon service calls. Otherwise, this would lead to memory leakage as each newly allocated runnable would remain within the memory area of the caller component until its termination. Moreover, referencing an `Edge` runnable(obtained from the pool) from the allocation context of a component is permissible since an object in a child scope can have a reference to an object found in a parent scope. In general, the size of the `Edge` pool will depend upon the number of potential concurrent client components accessing the composite and its sub-components. During its initialization phase, the implementing composite has to make a call to the `spaceForComponents` method (as shown in program 4) to specify:

- the number of scoped memories needed to initialize sub-components, as well as their size,
- the implementing `Edge` runnable class that contains the necessary mechanisms to handle interaction with the sub-components of the composite,
- and the number of `Edge` runnable instances needed to handle a potential number of concurrent accesses.

```
public class MyComposite extends Composite
   implements IBuffer, IMutex, ...
{
   ...
    public void init()
    {
        super.init();
        spaceForComponents(5, 4, MyEdge.class, 8000);
        initSubComponents();

        LC.setCName("Producer1");
        LC.setExeOpt(LaunchComponent.LC_START);
        MA[getScopeSubComponent("Producer1")].enter(LC);
        ...
    }

    public void initSubComponents()
    {
        //INITIALIZING COMPONENTS
        LC.setExeOpt(LaunchComponent.LC_INIT);
        LC.setCName("Producer1");
        MA[getScopeSubComponent("Producer1")].enter(LC);

        LC.setCName("SynchronizedBuffer");
        MA[getScopeSubComponent("IBuffer")].enter(LC);

        //BINDING INTERFACES
        LC.setExeOpt(LaunchComponent.LC_BIND);
        LC.setCName("IBuffer");
        LC.setObjectToBind(this);
        MA[getScopeSubComponent("Producer1")].enter(LC);
        ...
    }

    protected int getScopeSubComponent(String cmp)
    {
        int tmp=-1;
        if (cmp.compareTo("IBuffer")==0)
        {tmp=0;}
        else if (cmp.compareTo("Producer1")==0)
        {tmp=1;}
        ...
        return tmp;
    }

    public void put(int i) //IBuffer interface method
    {
        int tmp = getNextEdge();
        e[tmp].setItfOpt("IBuffer", "put");
        ((MyEdge)e[tmp]).setParamInt(i);
        MemoryArea.getMemoryArea(this).executeInArea(e[tmp]);
        e[tmp].unlock();
    }
    ...
}
```

Program 4: A `Composite` implementation class

The `Composite` abstract class also overrides the terminate method of the `Component` abstract class to include the necessary codes for terminating sub-components.

6.1 Handling cross scoping

As shown in program 4, to allow cross-scope invocations, the composite should implement the interfaces needed by caller components to enable them perform seamless interface bindings. The body of the implementing methods at the composite level, should include an `executeInArea` method call (using an instance of the `Edge` runnable obtained from the pool) to switch execution to the allocation context of the composite when the caller components will request services. Upon start up, the `Edge` runnable in turn uses its private member runnable to enter the scope of the callee component to perform the required operation (see program 5).

```
private class MyEdge extends Edge
{
  private boolean ParamBool;
  private int ParamInt;
  private InChildScope IC;
  ...
  public void run()
  {
    //ENTER SCOPE OF SUB-COMPONENT
    MA[getScopeSubComponent(getItfOpt())].enter(IC);
  }
  public int getParamInt()
  {return ParamInt;}

  public void setParamInt(int i)
  {ParamInt = i;}

  private class InChildScope implements Runnable
  {
    public void run()
    {
      //EXECUTE IBUFFER INTERFACE GET METHOD
      if (getExeOpt().compareTo("get")==0)
      {
        ParamInt=((IBuffer)((ScopedMemory)RealtimeThread.
                            getCurrentMemoryArea()).
                            getPortal()).
                            get();
      }
      //EXECUTE IMUTEX INTERFACE ACQUIRE METHOD
      else if (getExeOpt().compareTo("acquire")==0)
      {
        try
        {
          ((IMutex)((ScopedMemory)RealtimeThread.
                      getCurrentMemoryArea()).
                      getPortal()).
                      acquire();
        }
        catch(InterruptedException ie)
        {...}
        }
        ...
     }
...
}
```

Program 5: An `Edge` implementation class

Handling input and return parameters

As shown in program 5, to allow relaying of data among components found within different scopes, the `Edge` runnable should contain private member variables of types corresponding to those of the input and return parameters of the methods defined within the offered and required interfaces of components. This is required to enable arguments accessibility from the callee scope and return parameters accessibility from the caller scope without violating memory assignment and access rules. However, to ensure exception-free data transit, some restrictions are imposed on the use of the offered interfaces methods of components. The arguments of the methods should be read-only and their returning parameters having non-primitive Java types should implement the Java `Cloneable` interface in order to obtain a deep copy of the returning object (via the `clone` method), instead of a reference.

6.2 Sub-components management

Sub-components management is handled by the composite member class called the `LaunchComponent`. The `LaunchComponent` is a `RealtimeThread` of greater priority than the sub-components comprising the composite. It is responsible for initialising, starting, terminating and performing interface

```
public class LaunchComponent extends RealtimeThread
{
  ...
  public void run()
  {
    if(exeopt==LC_INIT)
    {
      try
      {
        ComponentFactory CF = ComponentFactory.getInstance();
       ((ScopedMemory)RealtimeThread.getCurrentMemoryArea()).
            setPortal(RealtimeThread.getCurrentMemoryArea().
            newInstance(CF.getClass(str)));
        ((IComponent)((ScopedMemory)RealtimeThread.
                      getCurrentMemoryArea()).
                      getPortal()).
                      init();
      }
        catch(ClassNotFoundException cnfe)
        {
          ...
        }
    }
    else if(exeopt==LC_START)
    {
      ((IActiveComponent)((ScopedMemory)RealtimeThread.
             getCurrentMemoryArea()).
             getPortal()).
             start();
    }
    ...
  }
  ...
}
```

Program 6: The `LaunchComponent` class

bindings of sub-components. Following the selected execution option and the targeting sub-component, the `LaunchComponent`, upon start up via the enter method of the corresponding scoped memory, will perform the required operations within the memory area of the sub-component. An excerpt of its innerworkings is shown in program 6. When being called upon with the `LC_INIT` option, the `LaunchComponent` will:

- create an instance of a component implementation class obtained from the `ComponentFactory` singleton,
- set the newly created component as the portal object of the scoped memory, so as to be able later to obtain a reference to the component,
- call the `init` method of the component, such that the scoped memory area remains active after the `LaunchComponent` has left it.

When being executed with the other options, the `LaunchComponent` gets access to the component within the scope by performing a `getPortal` and then performs corresponding operations. To generalize the execution of the `LaunchComponent` upon the sub-components, we cast the returning objects of the `getPortal` methods with interfaces corresponding to the type of services that we want to access. This allows us to operate on components without accessing explicitly their implementations. For example, as shown in program 6, to start an active component, we just need to have access to the `start` method of the component. To gain access to the method, we just have to cast the returning object with the

IActiveComponent interface, which provides the method's declaration.

In general, the initialization and interface bindings of sub-components should be performed within the initSubComponents implementing method of the concrete composite. The getScopeSubComponent implementing method should provide the necessary codes to obtain the array index of the ScopedMemory that corresponds to the component identifier given as argument (see program 4).

6.3 Composite configuration

The code configuration of the composite will vary, depending on the client or server nature of its sub-components. For instance, if the composite comprises only client sub-components, it will perform a mere delegation of services without considering any memory traversals, since offered services are found at the parent level of the composite. To achieve this, the composite should be set up both as server and client of the interfaces required by its sub-components. In the former case, this configuration would allow the sub-components to perform interface bindings and in the latter case, it would allow the composite to access services at its parent level. Since the composite is also a client component, it should implement the IBindController (as presented in section 5) to enable binding of its required interfaces to the offered ones at the parent level. In general, the mechanisms offered through the Edge runnable are needed if the composite comprises at least one server sub-component. This would allow client components to make service calls to the server component either from sibling scoped memory areas or from scoped memory areas on the same sibling level or at a higher level than that of the composite encompassing the server component. For the last cases, the invocation of the executeInArea method within the body of the implementing methods of the composite (as shown in program 4 - put method) would not cause an InaccessibleAreaException. The reason is that the Edge runnable, initiated from the parent scope of the composite to perform the service call, would already be within the allocation context of the composite.

In our RTSJ framework, the composite entity can also be viewed as connector, in which case it achieves two purposes: first, it is used as a **ScopedMemory Connector** to enable components interaction and secondly, it is used as an **Interface Adaptor** to enable interaction of client and server components that make use of different interface definitions, but which are in a structural subtyping relationship. However, in the latter case, this is made possible only if the offered interface of the server component is a subtype of the required interface of the client component. In order to be configured as an **Interface Adaptor**, the composite should be set up as a client of the offered interface of the server component and as a server of the required interface of the client component. Then the composite should furnish the necessary codes to enable smooth relaying of data between the components. As compared to our model, the RTSJ Component framework cannot allow multiple connectors at each hierarchy level due to the RTSJ's memory single-parent rule. For this reason, an application built with the framework, will always contain a **top-level** composite (defined in a scoped memory) that acts as a **ScopedMemory Connector** to enable binding and interaction of first-level component entities. Thus, the ImmortalMemory region will contain only a LaunchComponent used to initialize the **top-level** composite and the ComponentFactory singleton containing a list of component implementation classes that will be used at each hierarchy level to create new instances of component entities.

In general, the choice of managing sub-components at the composite level, as regards to scoped memory management and traversal, allows the framework to scale up easily hierarchically. Runtime component adaptation or extension does not require the revision of the overall framework setup, as it would be the case if the necessary mechanisms are allocated within the ImmortalMemory. Memory management and traversal concern is local to each level of the hierarchy. When performing cross-scope invocation at a particular level, a composite does not have to cater for whether or not more memory traversals are needed within its sub-components or within its parent component. These will be handled at either level. As such, if we need to replace a component by a composite entity, we need not have to cater for additional scoped memories required to create the sub-components of the composite, as well as Edge runnable mechanisms. All these will be created automatically upon instantiation of the composite.

7. FLEXIBILITY VS PERFORMANCE

Though the choice of assigning a distinct scoped memory to each component of the system brings a lot of flexibility regarding component management, it may however involves a high execution cost, especially if the configuration of components spans over several hierarchy levels. Such consequence is due to the fact that each service call performed by a client component will require one or more memory traversals, each of which will be done via the execution of two runnables. To increase performance, components could be allocated within the same scope, wherever possible. For instance, if all the sub-components of a composite are of passive type, they can be allocated within the scope of the composite itself. Thus sub-component access and management will be simplified. However, in this case, sub-component adaptation would cause memory leakage within the scope. Each time a sub-component class variable is assigned a new instance, the scoped memory size will be reduced as the old instance will remain within the scope (even though not accessible) for the lifetime of the composite. To circumvent this problem, the only solution is to design the composite as a monolithic block in which sub-components interfaces are statically bound and adaptation is not allowed. As a result, when developing a real-time application in RTSJ using our approach, the developer has to find the exact balance between flexibility and performance, depending on the criticity of the application and the hierarchy depth of its configuration.

8. RELATED WORK

To our knowledge, research concerning the applicability of CBSE onto RTSJ has been carried out only by [5]. Though they also advocate a memory structure similar to ours, their component framework is however quite dissimilar. First of all, their component model is not hierarchical. Besides, the necessary mechanisms to handle component management and memory traversal are defined in the ImmortalMemory region, whereas in our case, these mechanisms are included

at each composite level. As presented in section 6.3, a direct consequence of their design choice, is that runtime component adaptation is difficultly achievable. Moreover, in [5], accesses to state-dependent methods and state-independent methods of passive components are handled differently. Indeed, in their framework, an active component accesses the services offered by a stateless passive component within its execution scope by obtaining an instance of the passive component implementation class (either obtained from a pool stored in `ImmortalMemory` or created "on the fly"). In our framework, since each passive component is assigned a scoped memory, services offered by stateful and stateless passive components are accessed the same way. Furthermore, in our framework, components deal only with interfaces and as such do not need to cater about how to access services offered by server components (like when accessing a passive component in [5]). Component access is handled automatically at the composite level.

9. CONCLUSION

The growing complexity of real-time systems has entailed development of new methodologies to reduce software complexity and provide support to facilitate software reuse. During the past few years, CBSE has emerged as a new revolutionary paradigm to achieve greater software understanding, reuse and reliability. Through our research work, we have designed a component model, which provides the necessary abstractions and means to design, analyze and validate real-time systems. In this paper, we investigated the use of RTSJ to structurally implement our component model. We also shown how our framework provides a programming model to facilitate real-time development using RTSJ. Indeed, through our framework, the developer does no longer have to cater for memory management and traversal intricacies generally involved when developing real-time application with RTSJ, while taking advantage of the benefits offered through the CBSE paradigm. The necessary codes to handle memory management issues are generated automatically at the composite level.

Future works

We have implemented our framework using jRate [10] and currently, the RTSJ codes are written manually. In the near future, we plan to develop a code generation tool that will, from an abstract component architecture description, generate automatically the core structure of the component architecture in RTSJ, with the necessary codes at the composite level to handle management of sub-components memory areas, concurrent accesses and interactions. We also plan to conduct experimental tests on the framework to analyze its performance and also see to it that it does not introduce unpredictable jitter. Furthermore, even though our framework is set up in a way to ease runtime component adaptation, by now, the necessary mechanisms to handle this has not yet been implemented. Finally, we also plan to include temporal contract monitoring mechanisms in the framework to watchdog if the temporal requirements specified and validated at the abstract level are still met at the implementation level.

10. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] E. G. Benowitz and A. F. Niessner. A Patterns Catalog for RTSJ Software Designs. In *OTM Workshops*, pages 497–507, 2003.

[3] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, 1st edition, January 2000.

[4] S. P. Bradley, W. D. Henderson, and D. Kendall. Using timed automata for response time analysis of distributed real-time systems. In A. H. Frigeri, W. A. Halang, and S. H. Son, editors, *24th IFAC/IFIP Workshop on Real-Time Programming (WRTP 99)*, pages 143–148, May 1999.

[5] 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.

[6] J. Etienne and S. Bouzefrane. Applying the CBSE Paradigm on Real-Time Systems. *6th IEEE International Workshop on Factory Communication Systems (WFCS'06)*, 2006.

[7] J. Etienne and S. Bouzefrane. Vers une approche par composants pour la modélisation d'applications temps réel. *6ème Conférence Francophone de Modélisation et Simulation (MOSIM'06)*, pages 1–10, 2006.

[8] M. Fowler. Module assembly. *IEEE Software*, 21(2):65–67, 2004.

[9] G. T. Heineman and W. T. Councill. *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 1st edition, June 2001.

[10] jRate. http://jrate.sourceforge.net/.

[11] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

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

[13] R. Alur and T.A. Henzinger. Logics and Models of Real-Time: A Survey. In *Real Time: Theory in Practice*, volume 600, pages 74–106. Springer-Verlag, 1991.

[14] 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.

[15] UPPAAL. http://www.uppaal.com/.

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