Model-based Development for RTSJ Platforms

Miguel A. de Miguel
ETSI Telecomunicación
Universidad Politécnica de Madrid
+34 914533574
miguel.demiguel@upm.es

Emilio Salazar
ETSI Telecomunicación
Universidad Politécnica de Madrid
+34 914533574
esalazar@dit.upm.es

ABSTRACT

The integration of real-time concepts into modeling tools and Java virtual machines and APIs are two problems addressed along last decade. Examples of standards addressing these problems are RTSJ (Real-Time Specification for Java) and MARTE (Modeling and Analysis of Real-time Embedded Systems). These standards, in general, have common fundaments (time predictability of software systems based on scheduling analysis methods and object-oriented languages).

Model driven developments methods are based on the application of generators and transformations, on source models, to generate code and artifacts of specific run-time platforms. Common fundaments of RTSJ and MARTE make possible their integration in a common model-driven software development framework. But this integration requires the developments of generators and transformations, and the customizations of UML extensions for the specific run-time platform. Integration of UML profiles into code generations requires specific customizations of generators. This paper studies these problems and proposes solutions for the application of model driven development techniques to develop of RTSJ software systems.

Categories and Subject Descriptors

The D.2.0 [**Software Engineering**]: General – *Standards*. D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Object-oriented design methods, Computer-aided software engineering*. J.7 [**Computer Applications**]: Computers in Other Systems – *Real-Time.*

General Terms

Performance, Design, Experimentation, Standardization, Verification.

Keywords

Model driven development of real-time Java applications, analysis of real-time Java models, RTSJ in UML.

1. INTRODUCTION

Real-Time Java standards such as RTSJ [11] define real-time Java platforms for the execution of high integrity applications. UML profile standards such as MARTE [15] and SPT [16] define UML extensions for the development of real-time software systems. Time predictability is the main topic addressed in both technologies (real-time Java and real-time UML), but from different viewpoints (real-time Java execution platforms and implementations, and high level real-time software development).

JTRES '12, October 24 - 26 2012, Copenhagen, Denmark
Copyright 2012 ACM 978-1-4503-1688-0/12/10…$15.00

Model driven development methods propose general solutions for the development of software systems in specific domains and for specific platforms. Generators and transformations are applied to generate platform specific software. General modeling languages (e.g. UML) must be extended for the description of domain/platform/technical concepts; generators and transformations reuse these extensions for the generation of platform specific software. In the case of real-time applications the extensions represent basic real-time concepts such as time values (e.g. execution times, response times, and release times), resources (e.g. hardware computation and communication resources, schedulers, process and threads) and real-time schedulable elements interdependencies (e.g. precedence in the execution of real-time actions). Model driven development tools for real-time systems reuse these extensions to generate real-time specific code and artifacts (e.g. reuse of real-time libraries, generation of real-time execution configurations).

RTSJ extends Java runtime environment to make Java applications time predictable. These extensions include Java libraries for handling real-time features such as: real-time scheduling, time-predictable memory management, time-predictable synchronization, asynchronous events handling and physical memory access. Java run-time environments (and Java Virtual Machines in particular) must be adapted to support some RTSJ specific concepts (memory management in particular).

RTSJ Java extensions can be reused in model driven generators to implement real-time extensions of general modeling languages (e.g. MARTE extension of UML). Some Ada and real-time Linux platforms are examples of integrated platforms in some UML-based code generators and modeling tools (e.g. Rational Software Architect RealTime Edition, and Rational Rhapsody Developer). But RTSJ has not been integrated in commercial UML modeling tools yet. Some exploratory implementations have been developed and this paper provides results and issues based on a specific implementation (http://www.erma-assets.org/).

MARTE and SPT are two examples of UML profiles for real-time software development. Both standards address common problems: i) representation of time expressions and concepts, ii) specification of resources, and iii) UML extensions for scheduling and performance analysis. MARTE includes some additional sub-profiles for handling non-functional requirements and component oriented modeling. Both profiles are designed for the application of analysis methods in early design phases. MARTE does not include platform specific extensions and it is not designed with any kind of real-time platform in mind. But some MARTE sub-profiles (GQAM (Generic Quantitative Analysis Modeling) and SAM (Schedulability Analysis Modeling)) were designed to be reused in the application of scheduling analysis methods for UML models. Early software development phases can apply these analyses to decide about alternative solutions and to limit the

resource consumption in time-critical software elements. RTSJ integrates scheduling analysis concepts too.

Around fifteen years ago started to appear multiple research results about the integration of object-oriented techniques into real-time software developments approaches. Volume 33 Issue 6 of IEEE Computer Magazine includes results about these topics, and two research areas in particular: modeling real-time systems with UML and implementation of real-time systems in Java [3,22]. Real-time system in Java and in UML produced standards [11,20,15,14], frameworks and tools that support all the standards. The integration of both approaches in a common environment requires a good knowledge of different technologies and tools. Solutions presented in this paper have been integrated in JamaicaVM[1], Websphere real-time[2], Rational Software Architect[3] and Eclipse Papyrus[4].

2. RELATED WORK

UML 2.4 [17] is a general modeling language that supports language extension based on profiles. MARTE [15] is a UML profile standard for the description of real-time software architectures. SPT [16] standard addressed the same topics but it was designed for UML 1.4.

CNES and Obeo are working in the development of a UML to RTSJ generator in the context of TOPCASED (TOPCASED toolkit includes a set of model transformation tools). There is not result yet. This project does not address problems of scheduling analysis presented in this paper.

There are many articles about Java code generation from specific UML subsets (e.g. state machines, collaborations and iterations). In this paper we address the problem from the reusable viewpoint. Ming [14] proposes a general framework for the construction of reusable UML code generators, and Stevens [23] proposes solutions for mapping UML L0 into Java. Both approaches have been reused in our studies.

Many solutions have been proposed to generate scheduling analysis models from UML annotated models [5,21]. Generation of analysis models from MARTE is a special case, and in this paper we address a specific problem: to make consistent analysis models and code generations. Two different approaches are to integrate analysis modeling language concepts into UML language [15], and to integrate timing properties into UML modeling elements like design or architecture specifications languages [1,4]. Both approaches are conceptually different. Second approach makes simpler the problem of consistency of code generation and analysis generation; standard MARTE uses the first approach, and it is integrated with UML 2.2 (there is not standard solution of first approach integrated whit UML).

3. TWO ALTERNATIVE MODELLING SOLUTIONS TO MODEL RTSJ SOFTWARE APPLICATIONS

Two alternative solutions for integration of RTSJ concepts into UML designs are:

1. To represent directly RTSJ Java classes into UML models and to define the relationships (e.g. generalizations, types, and associations) from application modeling elements to RTSJ libraries. UML modeling tools use two alternative solutions to integrate Java classes into UML models and diagrams:

   a. A UML model library integrates the RTSJ library and application models import this model library and reuse its model elements (e.g. extension of *RealtimeThread* class, definition a *ScopedMemory* instance). Figure 1 is an example of application of this library to design the *Cruise Control* example included in [24] (Section 16). This diagram includes two real-time threads with their miss handlers (the thread for the throttle and the thread that control the car speed).

   b. Another solution is to annotate UML modeling elements with some kind of extension (stereotypes or model annotations) that include references (e.g. URIs with some specific type of scheme) to the Java element. In this solution the references will target Java elements in RTSJ libraries.

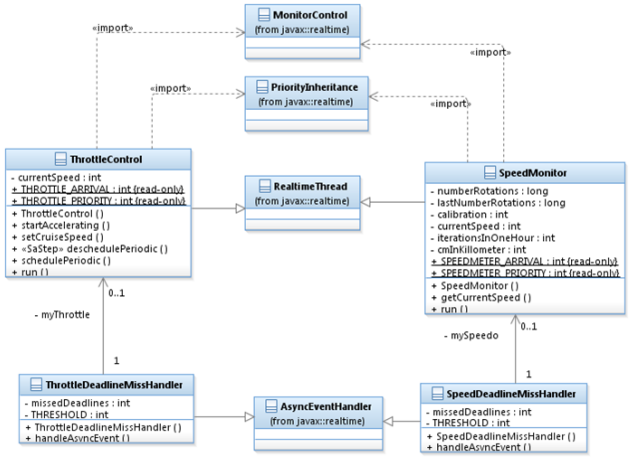

Figure 1. Reuse of RTSJ UML model library

2. To define the mapping from MARTE to RTSJ and formalize the representation of a UML/MARTE subset for the generation of RTSJ java code. For example, MARTE extensions such as *GRM::SchedulableResource* can be used for the representation of time critical threads, which would be mapped to *javax::realtime::RealtimeThread* instances. Figure 2 includes a diagram of another version of *Cruise Control System* UML model annotated with MARTE extensions (Figure 1 includes a class diagram, and Figure 2 includes an Object diagram with a set of instance specifications that represent the execution scenario for the analysis. They are included in two different models that use different modeling artifacts but they represent the same problem).

First solution is simpler and its learning curve would be optimal for RTSJ experts. The second solution provides advantages when we use scheduling analysis methods in early development phases. UML extensions in MARTE sub-profiles GRM (Generic Resource Modeling), GQAM and SAM can be used for the generation of scheduling analysis models; they include some

[1] http://www.aicas.com/jamaica.html

[2] http://www-01.ibm.com/software/webservers/realtime/

[3] http://www-1.ibm.com/software/awdtools/swarchitect/index.html

[4] http://www.eclipse.org/modeling/mdt/papyrus/

runtime properties annotated in their extensions (e.g. worst case executions times, release time of schedulable resources, and scheduling algorithms). Scheduling analysis results are only acceptable when final implementations fulfill the MARTE annotations assumptions. In the first modeling approach, scheduling analysis annotations and the reuse of RTSJ elements must be consistent, but they are two independent concepts represented in the same model; the modeler is responsible for their consistency (the design based on RTSJ library and the scheduling analysis model can be inconsistent, and the modeling tool will not detect these problems). When the transformation from MARTE to analysis models and the transformation from MARTE to RTSJ are designed to be consistent, the consistency problem is resolved in the transformations. Figure 3 represents the general models and transformations for the generation of consistent analysis models and real-time Java code.

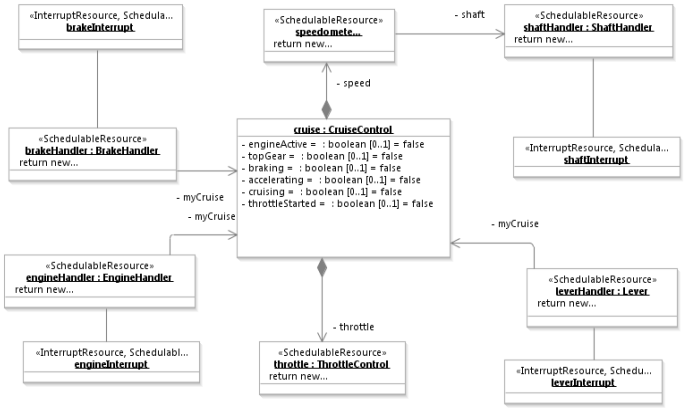

Figure 2. MARTE annotations for *Cruise Control System*

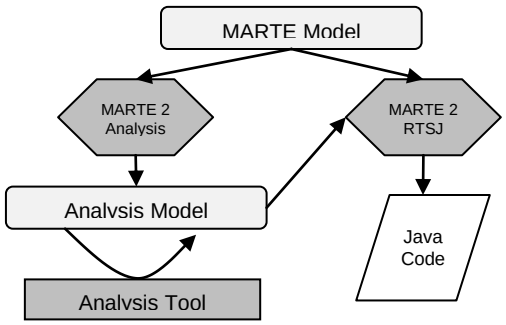

Figure 3. General models and transformations

4. A REUSABLE JAVA CODE GENERATOR

UML standard [17] specifies the abstract syntax of modeling language; this specification has a layer-based structure. Each layer provides new modeling capabilities for representation of concepts not supported in previous layers. Each new layer includes the specification of new modeling elements, and the extension (the extensions are new relations and properties for the modeling element) of elements introduced in previous layers. UML 2.X Superstructure layers are:

L0. This level reuses the *UML Infrastructure* specification and it contains a single language unit for modeling class-based structures and instance specifications of classes. This specification layer can represent most of modeling elements included in class and object diagrams (this representation would not include some properties such as template parameters). The modeling elements can represent most of class structures encountered in Java language. But this level does not support the specification of behaviors (there is not modeling elements equivalent to Java language elements such as control sentences).

L1. This level adds new language elements and extends the capabilities provided by Level 0. Specifically, it supports use cases, interactions, structures, actions, and activities.

L2. This level adds language elements for specification of deployments, components, state machine modeling, and UML profiles.

L3. This level represents the complete UML. It adds new language elements for modeling information flows, templates, and model packaging.

L0 level specification does not include any specification about active objects or concurrent behaviors. The results of mappings from L0 levels into any programming language should include only passive objects without concurrent properties.

Most of UML modeling tools provide Java code generators, but they do not handle modeling elements included in all layers. Most of generators support L0 and, frequently, some kind of behavior specification (e.g. state machines, activities or iteration). But, often, one kind of behavior excludes the other behaviors.

UML profiles include additional specifications and there are many code generators that reuse some specific profiles for code generation purposes (e.g. EJB (Enterprise Java Beans) profiles for generation of Java EJB code and artifacts).

In general, generators define a mapping from source language into target language. If we would use Java JRE+RTSJ as target platform, the generators requires very complex mappings for some UML modeling elements, because there is not software elements that support these UML elements in the target language. For example, UML component modeling elements (e.g. *Connector*, *Component* and *Port*) are designed for the specification of logical and physical components such as EJB, WSDL (Web Service Definition Language) and OSGi Bundles. We could extend the target platform (JRE+RTSJ) with some frameworks (e.g. OSGi or EJB) to support some source elements in the target, but RTSJ platforms does not well integrate these component frameworks, because of specific RTSJ memory limitations (e.g. the integration of non-heap real-time threads into OSGi bundles or EJB beans will have memory inconsistencies, because most of EJB and bundle implementations are designed for their execution into heap memories) [19,6,2]. These kinds of generation problems are out of the scope of this paper.

In general, Java code generators are designed with template based languages such as MOFM2Text [18,10] and JET (Java Emitter Templates) [7]. These generators define some Java text templates, and some parts in the Java text are refilled with expressions that reuse source model elements values. The template defines the mapping from a source pattern to an equivalent text code. This approach makes complex to reuse the generators for different kinds of generators (e.g. generate code for different kinds of behaviors, or customized generator for specific profiles), and it

must integrate in the templates very different kinds of concepts (e.g. code text for the generation of class structure, code for the integration of behavior into the Java class, and code for the generation of concurrent behaviors defined into profiles).

This section introduces a different approach for the Java code generation. This approach is based on model-to-model transformations. Source models are UML+profiles models, and target model is a model that represents Java code in abstract format (an abstract syntax tree (AST) for the Java code). OMG Meta-Object Facilities supports the specification of abstract syntax of modeling languages, but they can be used for the representation of AST for programing languages too. The model-to-model transformation generates Java code in abstract format and, at the end of the generation, the AST will be transformed into text (the Java concrete syntax equivalent to the Java AST). Specific generators for each UML layers and profiles fill a common Java AST with their own Java structures. Upper layers knows about lower layers AST generators (they know the mappings supported and the results generated, but they do not know details of mapping implementations), but lower layers are not designed taking into account upper layers.

Most of Model-to-Model transformation languages (e.g. QVT and ATL) [9] are based on rules that transform a set of source modeling element into a set of target elements. These rules are applied in the transformation, and the rules can get references to modeling element previously generated in other rules and can complete the rule base on rule results that will be provided in the future. Rules defined in different kinds of transformations can reference the Java structures generated for lower levels, and based on these references, they increase Java sub-trees (e.g. new methods and fields are included in previously generated classes).

4.1 Structure of Java Code Generator

Our implementations of Java code generators are based on following modeling languages and artifacts:

1. *UML meta-model*. This is the UML specification language.
2. *Java AST*. This is the meta-model for the representation of Java application in abstract format. Eclipse Modisco project [8] uses this approach to implement Java reverse engineer. But the main difference is that Modisco AST was designed to support reverse engineer and it is based on a single model that does not include references to elements in library models. Java Development Tools (JDT) in Eclipse supports another kind of Java AST designed for the construction of editors, debuggers and compilers. JDT AST format is not integrated with Eclipse modeling tools (Eclipse Modeling Framework tools and Eclipse model transformation languages).
3. *Java libraries in UML and in Java AST*. *javax.realtime* is the fundamental library for the support of RTSJ. But some other basic Java packages (e.g. *java.util*, *java.io* and *java.lang*) are needed in any RTSJ Java application. Some RTSJ platforms have adapted implementations of some classes in these packages. The adapted methods are executable in the context of no-heap threads. UML model libraries include the two kinds of Java libraries, one library represents *rt.jar* (Java JDK) in UML. The second model library represents *javax.realtime* and all Java classes that can be executed without memory inconsistency problems.

To make analysis models and runtime execution consistent, when the Java generator supports MARTE profile, UML model should not include explicit references to the RTSJ UML library. The Java code generated that includes references to RTSJ will be based on MARTE annotations and these mapping are consistent with scheduling analysis results.

Java generator is decomposed in layers and lower layers can be executed without the execution of upper layers. Figure 4 represents the general structure with two layers: the Java generator for L0 and the Java generator for MARTE. Some additional generators could be integrated for the generation of Java code for UML behaviors just in between previous layers.

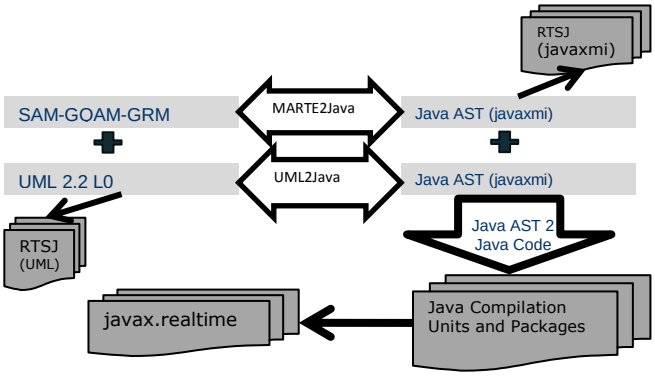

Figure 3. Multi-layered UML 2 Java generators

This generator includes two transformations and one generation:

1. *UML 2.2 to Java AST*. This is a QVTo transformation that handles UML L0 modeling elements (*Class* and *InstanceSpecification* in particular). This generator handles UML opaque expressions in Java for the representation of *Operation* implementations and values expressions in general. Java code included in *OpaqueExpression* is parsed and the parse results are included as part of Java AST generated. The code associated to *InstanceSpecifications* (their classifiers must be UML L0 *Classiffiers*) is a set of Java classes that implement the instantiation of classes and the initialization based on UML Slots and attribute values specifications.
2. *MARTE 2 Java AST*. This QVTo transformation extends the Java code generated for L0 modeling elements, and introduces new Java expressions and properties, for handling MARTE extensions attached to UML elements. Generator *UML 2.2 to Java AST* knows nothing about this generator.
3. *Java AST 2 Java Code*. This generator transforms Java AST in the equivalent concrete syntax. We have implemented this generator in MOF2Text and a second version implements this in OCL. Every Java AST element has associated an OCL operation that transforms the element into text. This OCL operation represents the mapping from Java AST to Java code for each AST element. Our implementation of Java AST includes around 120 non-abstract meta-classes that implement the OCL mapping from Java AST to Java text and they are OCL expressions around 6 lines long.

Different kinds of templates are used for the generation of Java AST elements. When they are complex templates that require complex Java AST sub-trees and the construction of these sub-trees in QVTo requires large implementations. Complex templates

are parsed with Eclipse JDT and then JDT AST is transformed into Java AST; the new values are introduced into this generated sub-tree. This approach combines the advantages of template based generation languages and model-to-model based code generation.

4.2 Integration of MARTE Code Generator and Java Code Generator

MARTE 2 Java AST generates the code based on MARTE profiles GRM (and its extension DRM (Detailed Resource Modeling)), GQAM and SAM. This generation is designed to be consistent with the transformation to scheduling analysis models. The kinds of RTSJ classes handled in the generators are:

- *RealtimeThread*. RTSJ *RealtimeThreads* are generated for UML elements annotated with *SchedulableResource* and some of its specializations, and for *GaWorkloadEvent*. *RealtimeThreads* are configured with some parameters. The RTSJ *SchedulingParameters* will depend on the priority configuration of *SchedulableResource*. RTSJ *Release* parameter will depend on *SchedulableResource* and *GaWorkloadEvent* attributes; values supported are: i) *periodic scheduling parameters*, ii) *sporadic scheduling parameters* and *iii) aperiodic scheduling parameters*. Aperiodic scheduling parameters are represented with *Bursty* for *GaWorkloadEvent* and *Deferred* for *SchedulableResource*.
- *AsyncEvent*. The RTSJ class *AsyncEvent* is executed when interrupts and some other asynchronous events occur; in general, *AsyncEvent* invoke *AsyncEventHandlers* for handling the asynchronous event. *AsyncEvents* are represented with *InterruptResource* combined with *SchedulableResource* extension. We combine both extensions because of some inconsistencies of MARTE profiles. There are some design errors in MARTE for the representation of *InterruptResources* because they cannot be handled correctly in the scheduling analysis.
- *AsyncEventHandler*. RTSJ *AsyncEventHandlers* are generated for different purposes. They are generated for handling errors in *RealtimeThreads*, and they are generated as handlers of interrupt events. MARTE *SchedulableResource* that handle responses to events handled in *InterruptResource* are represented as *BoundedAsyncEventHandler*.
- *PriorityInheritance* and *CeilingProtocol* monitor control. This code is generated for *SaSharedResource* with priority inheritance and ceiling protocols, and the code generated sets the monitor control for the object instances. The code generated ensures that the object is blocked before setting the monitor.

When we combine L0 generator and MARTE generator, UML modeling elements that can have attached MARTE extensions are:

- *UML Class* and *UML Classifier*. The code generated for these elements ensures that any instance will have associated *RealtimeThread*, *AsyncEvent*, *AsyncEventHandler* or monitor control for UML elements with MARTE annotations. When the classifier is instantiated, the threads are created or the monitor is set.
- *UML Enumeration*. Enumerations will have associated monitors (they can have associated threads too, but that is unusual).
- *UML Interface*. When an interface is annotated with MARTE stereotypes, any class that implements the annotated interface will have associated the equivalent thread or will support the monitor control (this will have the same effect as if we would annotate each class that implements the interface with the interface annotations).
- *UML Property*. For each property annotated, a new property will be created to support the thread, or the instance that references the property will be set to the specific monitor.
- *UML InstanceSpecification*. When this specific instance (but not the other instance of the classifier) is created, the thread that supports the annotation will be created, or the monitor is set.

Templates supported in implemented generators include: periodic server, sporadic server, external events handler, aperiodic server, and protected objet. Next sub-section introduces the periodic server template. Implementations of aperiodic and sporadic servers and external event handlers have a similar complexity; protected object template is simpler (10 lines long).

4.2.1 Periodic Server template

The generation of RTSJ *RealtimeThread* from MARTE is complex because there are multiple combinations to take into account: different kinds of UML elements can be annotated, different MARTE extensions can characterize the thread and RTSJ supports different kinds of *RealtimeThread*. This section resumes all combinations, but examples are oriented to UML *InstanceSpecification* annotations based on MARTE *SchedulableResource*, for generation of *RealtimeThread* (generation of periodic *NoHeapRealtimeThread* has some differences).

MARTE includes three kinds of periodic event handlers: i) sources of external work load (*GaWorkLoadEvents), ii*) periodic schedulable resources (*SchedulableResource*), and iii) specific real-time specifications (*RtSpecification*) for real-time units (*RtUnit*). RTSJ can handle periodic events with threads and handlers. If a *GaWorkloadEvent* were handled with MARTE *InterruptResource*, the handler would be a *BoundedAsyncEvent-Handler*. In other case the handler is a *RealtimeThread* (or a *NoHeapRealtimeThread*). In this section we introduce the templates that generate code for *RealtimeThreads* and their error handlers.

The generation includes two kinds of generations: code generation of fields that support the references to the thread and the handlers (in the template the identifiers of these fields are *schedulableResourceOverrunHandler*, *schedulableResourceMiss-Handler* and *schedulableResource*). The Java element that supports the UML base element (e.g. class, interface, attribute, operation, enumeration, component, instance specifications) will support these three fields. When the MARTE annotations are at instance level, there will be specific fields for the specific instance, and they are initialized in the construction of the specific instance. When the MARTE annotations are at classifier level, the instantiation will be done in the construction. The template is simplified and it is configured for heap real-time threads in particular (no-heap real-time threads requires additional annotations for the specification of their memory space). The thread and its handlers are supported by Java anonymous classes embedded in the Java class that supports the base element:

```
1 <instance>.schedulableResourceOverrunHandler=new BoundAsyncEventHandler (
2       new PriorityParameters(<schedulable_resource priority>) ,
3                               null,null,null,null,false,
4                               new Runnable ( ) {
5                                       public void run() {
6                                               <instance>.overrunHandler( );
7                                       }
8                               });
9 <instance>.schedulableResourceMissHandler=new BoundAsyncEventHandler (
10      new PriorityParameters(<schedulable_resource priority>),
11                              null,null,null,null,false,
12                              new Runnable ( ) {
13                                      public void run() {
14                                              <instance>.missHandler();
15                                      }
16                              });
17 <instance> schedulableResource=new RealtimeThread (
18      new PriorityParameters(<schedulable_resource priority>),
19      new PeriodicParameters (null, new RelativeTime(
20                                      <schedulable_resource period>,0),
21                                  new RelativeTime(
22                                      <schedulable_resource cost>,0),
23                                  new RelativeTime(
24                                      <schedulable_resource deadline>, 0),
25                                  <instance>.schedulableResourceOverrunHandler,
26                                  <instance>.schedulableResourceMissHandler),
27                                  null,null,null,
28                                  new Runnable() {
29                                      public void run() {
30                                              boolean noProblems=true;
31                                              while (noProblems) {
32                                                      <instance>.run() ;
33                                                      noProblems=
34                                                              RealtimeThread.waitForNextPeriod();
35                                                      if (!noProblems)
36                                                              noProblems=<instance>.deadlineHandler();
37                                              }
38                                      }
39                                  })
```

The template has two parameters: ***instance*** **(**represents the identifier of base element) and ***schedulable_resource*** (represents the MARTE annotation for the period), and they provide the information needed in the template: priority, period, deadline, and error handlers. Lines 1 to 16 include the error handler (miss and overrun); this kind of specifications is not supported in MARTE and we assume that the logical handlers are operations with specific names (*missHandler* and *overrunHandler*). Line 36 includes another error detection that depends on the results of *waitForNextPeriod* in *RealtimeThread*. When the classifier of the base UML element does not have associated error handler, empty handlers are generated.

The periodic server executes periodically the logical code (line 32), and after the end of execution waits for next period.

This generator implementation does not handle all MARTE annotation values (MARTE is platform independent and provides general solutions for most of real-time platforms). MARTE includes real-time patters not supported in RTSJ. For example, *SchedulableResource* can have associated a table for the description of table driven scheduling and *MutualExclusionResource* can be annotated as *StackBased*. When generator cannot generate RTSJ code, the generator produces warnings and non-fatal errors.

5. CONSISTENCY OF ANALYSIS MODELS AND CODE GENERATORS

Some MARTE profiles (e.g. GRM, GQAM and SAM) are design for the automatic generation of scheduling analysis models. These models are generated and analyzed during early modeling phases and design decisions could depend on the analysis results.

MARTE analysis annotations are practically a parallel model represented with UML extensions (analysis model only depends on UML specification of sequence behaviors). These extensions include references between them, and all together define an analysis model (an UML model will include as many analysis models as *SaAnalysisContext* stereotype applications).

Figure 4 depicts some relations across analysis stereotypes to resume the general structure of these models. The root is the analysis context (typically a package or a model; alternative solutions can include several analysis contexts). It identifies the set of work load behaviors and the platform resources. The work load behaviors have associated the work load event, and the sequence of steps and scenarios (a scenario can include a set of steps and sub-scenarios). Work load events have associated their effect (scenarios or step). The specification of the platform resources includes the executable resources, the mutual exclusion resources and the schedulable resources. The schedulable resources identify their host and scheduler, and the steps reference their schedulable resources.

The analysis model is based on four basic concepts:

- *Specification of load events in the system*. These events specify the sources of load in the system and they raise the execution of behavior steps. *GaWorkloadEvents* represent these kinds of events. Each event has associated time distributions of their occurrences (periodic, sporadic and aperiodic) and deadlines for the execution of their responses. Some arrival patterns of *GaWorkloadEvents* are not handled in analysis tools (e.g. *IrregularPatterns*) and they are not handled in the analysis and transformations produce non-fatal errors.

- *Event responses*. A sequence of steps define behavior associate to external events occurrences. Examples of MARTE extensions that represent these concepts are *SaStep*, *GaStep*, *GaScenario* and their specializations. The steps include references to the resources needed for the step execution and their execution times. The steps define their execution precedence; this precedence can be derived of UML behavior specifications.
- *Resources*. The execution of each steps require a specific set of resources for their execution. They must be executed in the context of specific schedulable resources (e.g. threads, process or tasks); steps may need some additional resources. There are different kinds of resources:

  a. *Schedulable Resource*. It is a resource that uses the capacity of processing resource for the execution of steps. Schedulers decide about the execution of schedulable resources, which has associated scheduling parameters such as priorities.

  b. *Computing Resource*. They represent processing resources needed for the execution of schedulable resources. They have associates schedulers.

  c. *Communication Resource*. They represent communication media needed for data transmissions.

  d. *Synchronization Resource* and *Mutual Exclusion Resource*. Some data resources need mutual exclusion or some kind of synchronizations. These resources have associated some protocols used during their scheduling (e.g. priority and ceiling).

  GRM includes stereotypes for the description of these kinds of resources; these stereotypes are extended in SAM, DRM and other sub-profiles.
- *Schedulers of resources*. The schedulers define scheduling algorithms for computing and communication resources. *Scheduler* and *SecondaryScheduler* are GRM stereotypes for the description of scheduling characteristics in the execution platforms. The schedulers include references to their scheduled resources.

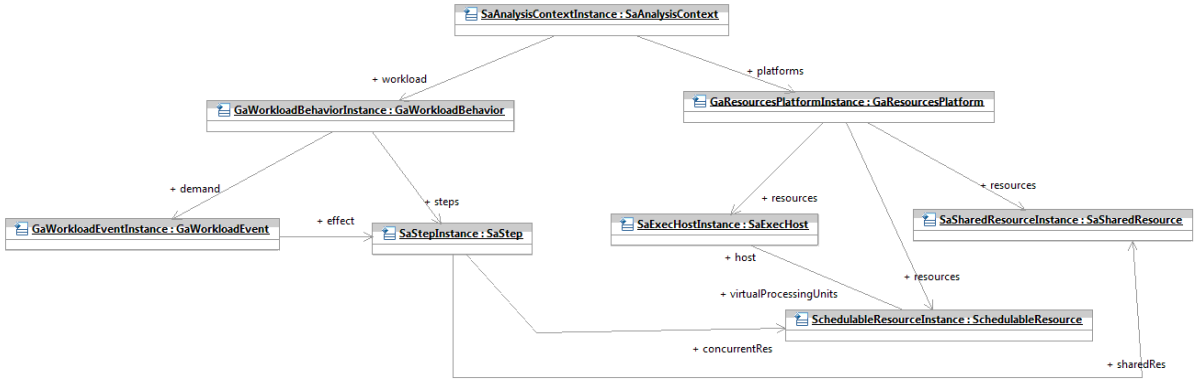

Figure 4. General structure of analysis models

The scheduling analysis tools [13] implement scheduling analysis theories to make the analysis of model; they provide results such as worst case response time for events and steps, occupation of resources and optimal protocol and scheduling parameters of resources.

The code generated in RTSJ generators must be consistent with scheduling analysis result. Java code generated must have the same time behavior. RTSJ supports some analysis methods, but they are not as detailed as MARTE analysis specification. RTSJ *Scheduler* class includes some methods such as *addToFeasibility(Schedulable)* and *isFeasible()* to determine the schedulability of schedulable resources; but the specification does not include information about monitors shared and *MonitorControls*. This is important because blocking times is not handled in the feasibility analysis. Another kind of information not handled is the execution times of *AsynEvents* and their hardware priority (when they are used for handling hardware interrupts).

To make MARTE analysis results and RTSJ consistent, in the code generated must be there a one to one correspondence from *RealtimeThread*, *AsyncEvent*/*AsyncEventHandlers* to scheduling analysis events (these RTJS elements will determine the load in the system). Code generated and analysis models must support the same synchronizations and mutual exclusions. The sequence of steps for handling *GaWorkloadEvents* must be equivalent to the methods, their implementations and their synchronizations (the methods that support the execution of RTSJ real-time threads and event handlers). *Schedulers* in analysis models must have the same configurations as RTSJ *Scheduler* (*PriorityScheduler* and its priority ranges). If RTSJ would support some other schedulers handled in MARTE GRM, they should be consistent too.

Some additional RTSJ implementation dependent properties should be integrated in the scheduling analysis generated models. They are improvements in the transformations to get precise analysis results (our implementations do not support these improvements). This would provide a customization of analysis methods for the specific RTSJ implementations. Tree examples of implementation specific dependencies are:

- JVM (Java Virtual Machine) implementations that support RTSJ integrate specific garbage collectors. RTSJ standard does not provide specific guidelines or restrictions for the garbage collector implementations. But these garbage collectors affect to heap *RealtimeThreads* response times. The effect of garbage collector should be integrated in the

analysis (some garbage collectors analysis effect could too pessimistic; and execution times have similar problems). An optimal solution depends on the specific garbage collector properties and on the analysis results expected (pessimistic analysis vs optimal resources allocation).

- Implementations of *javax.realtime* classes use some specific system resources such as timers and mutexes. These resources could have associated blocking times that should be included as part of scheduling analysis.
- Different RTSJ environments use different approaches for the interpretation of Java byte code: interpretation of byte code, just in time translation of java class files, ahead of time precompiled code, and translation of Java class files into C and compilation in target binaries. Each solution has important effects in the evaluation of execution time costs.

6. MARTE LIMITATIONS FOR RTSJ CODE GENERATION

MARTE is a very extensive language, but some RTSJ concepts are not well supported in MARTE extensions and some additional extensions are need for RTSJ code generation. The two main concepts are:

- Memory management in RTSJ is a fundamental topic. Most of real-time languages (e.g. Ada languages, and C/C++ on Real-Time Operating Systems), in general, do not use memory management structures used in RTSJ (immortal and scoped memories) or Java (garbage collector). *RealtimeThreads* and *NoHeapRealtimeThreads* in particular must provide parameters about their memory areas. MARTE does not provide support for the specification of memory areas used for object allocation (in RTSJ: *HeapMemory*, *ScopedMemory*, *LTMemory*, *VTMemory* and *ImmortalMemory*). Some MARTE extensions can represent memory spaces and MARTE can represent schedulable resources sharing common memory spaces (e.g. *ResourceUsage* can specify used and allocated memories, and *StorageResource* and *HW_Memory* represent memory resources). But we cannot specify RTSJ specific memory spaces, and their specific parameters. And MARTE does not support the specification of time properties of time predictable garbage collectors.
- RTSJ supports allocation of processing resources for groups of schedulable resources (*ProcessingGroupParameters*). This is not well supported in MARTE. MARTE scheduling parameters are associated to individual schedulable resources, and some other extensions such as *ResourceUsage* cannot reference a common processing resource allocation.

Some additional extensions (e.g. extended MARTE or specific RTSJ profile) are needed for generation of that code. We have experimented with an independent profile to support MARTE conformance of models. MARTE has some additional problems that RTSJ generator and analysis models transformations must take into account. Some examples are:

- *MARTE is redundant*. Same time values can be repeated several times. For example a periodic thread for handling a periodic external event represented in MARTE will include a *GaWokloadEvent* with the time distribution *pattern: ArrivialPattern*; a *SchedulableResource* will represent the thread and its property *schedParameter:SchedParameters* will include the period distribution (*SchedParameter* in GRM and *ArrivalPattern* in GQAM are different values and data types). In addition, *RtSpecification* in HLAM would repeat this value too. Some other type values such as *NFP_Duration* (used for the description of time periods, delays and many other time values) can have associated three values (*value*, *best* and *worst*), these three values will have different interpretation in different context (e.g. *worst* has no meaning for periods, but it could be use full for jitters). *GaScenario* has different approaches (non-exclusive) for the description of sequences of steps that represent the steps behavior (*root*, *parent step* and *steps*).
- *Semantic of MARTE is imprecise*. MARTE includes some natural language constraints for some stereotypes, but there is not constraints for the specification of values precedence (e.g. we could create a *RealtimeThread* with the deadline specified in the *GaWorkloadEvent* and the period and execution times specified in *SchedulableResource*, or we can use the period specified in *GaWorkloadEvent*).
- *MARTE has inconsistencies between the different profiles*. In particular for the representation of interrupt behaviors. GRM includes the class *ConcurencyResource*, it is an abstract stereotype and there is not stereotype in MARTE that extends *ConcurrencyResource*. It was designed to represent any kind of resource that executes in concurrency (e.g. schedulable resource and interrupt resource). DRM includes *InterruptResource* for the description of interrupt routines, but it does not extend *ConcurrencyResource*. GQAM reuses GRM (and indirectly should reuse DRM), *GaStep* reference *SchedulableReurces* to identify the execution context of steps, but it cannot reference (directly or indirectly) *InterruptResource* to specify the step as part of interrupt routine. In MARTE we cannot specify scheduling analysis that include interrupts with hardware priorities (most of scheduling analysis tools support this).

RTSJ analysis transformations and RTSJ generators must include restriction rules and consistency rules to reduce MARTE imprecisions and to make consistent redundant values. These rules restrict the application of MARTE but they provide more precise models.

7. CONCLUSIONS

MARTE and RTSJ have common fundaments and they are good candidates for the construction of model-based development environments. But MARTE requires improvements: to provide a precise interpretation of MARTE extensions based on RTSJ concepts, to introduce constraints and precedence rules to make MARTE models consistent for code generation, and to include new extensions for representation of RTSJ specific topics (memory spaces and processing groups).

Scheduling analysis is a fundamental tool to decide about alternative software architecture solutions. Precision of analysis models for MARTE models and RTSJ implementations requires specific improvements, because there are some concepts not well integrated yet in the analysis (e.g. the effect of garbage collector, the effect of memory management, and evaluation of execution time costs). In this paper we propose to use the scheduling analysis to improve the design decisions, but the application

analysis to guarantee time predictability of RTSJ applications requires improvements.

When we use analysis methods to decide about software designs, these analysis results must be consistent with implementations. If we would choose one solution because it is schedulable, and we discard some other because it is not schedulable, the two implementations that derive of designs should have the same time properties. If this is not true the design decision would be wrong. To make consistent analysis models and detailed software designs is complex, because the modifications in one model affect to the other, and to maintain consistency of both models is complex. When the consistency is integrated in the transformations the consistency problems are solved.

UML code generators for specific real-time platforms and for specific UML extensions such as MARTE require innovative solutions. The application of traditional methods requires very complex generators that must handle many different concepts for the same target code elements. We propose incremental solutions based on generation layers and intermediate Java AST models. In this approach upper layers need to know about signatures of generation rules included in lower layers, but they do not need to know about their specific implementations. Lower layers of generators know nothing about upper levels.

We have introduced many complex problems and their solutions, but we have not addressed some additional problems. We can derive RTSJ implementations based on UML models. UML models can include Java *OpaqueExpressions* and *OpaqueBehaviors* with Java code. In solutions proposed, the execution time cost in MARTE designs and RTSJ implementations are simple estimations. A precise evaluation of execution times can be done at model level (the model integrates all information needed to derive the implementation), but we must solve some problems (some of these problems have not been solved yet in RTSJ environments): i) to integrate in execution costs the cost of generated code, that is not part of application code, ii) to customize the analysis of execution times for the specific virtual machine execution methods (e.g. Java byte code interpretation, JIT, ahead-of-time) iii) to integrate execution times of RTSJ libraries into execution costs.

All solution proposed are applicable in RTSJ systems that do not support dynamic allocation of threads or handlers, the main limitation are the consistency of analysis results and runtime executions. Analyses are done statically and because of that dynamic allocations of real-time threads and handlers are not supported. The code generations for threads and handlers are equivalent to specific analysis scenarios.

RTSJ does not address time predictability in specific runtime environments such as distributed and multicore, and it is not well integrated yet with component-based Java frameworks. The integration of some UML and MARTE concepts (e.g. UML and MARTE components and communication) in the software development process require the extension of Java RTSJ execution platform.

Most of solutions and problems addressed in this paper have been implemented. Problems identified and not implemented are the issues identified at the end of section 5. The implementations supports UML L0 and additional transformations are needed to support L1, L2 and L3, but some of these implementations would require improvements in RTSJ and its integration with some other Java frameworks (e.g. OSGi and EJB).

8. ACKNOWLEDGMENTS

This work has been partially developed in the context of FP7 European projects CHESS and MultiPARTES, and Spain national project RTModel.

9. REFERENCES

[1] Avionics Architecture Description Language Standards Document (AADL), http://www.aadl.info

[2] Pablo Basanta-Val, Marisol Garcia-Valls, Iria Estevez-Ayres, Enhancing OSGi with Real-time Java Support. Accepted in Practice and Experience. November, 22, 2011.

[3] Bollella, G.; Gosling, J. The real-time specification for Java. *Computer*. Volume: 33 , Issue: 6, DOI: 10.1109/2.846318. June 2000 , Page(s): 47 - 54

[4] A. Burns and A. Wellings. HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems. ELSEVIER, 1995.

[5] de Miguel, M; Alonso, A; de la Puente, J. Object-oriented design of real-time systems with stereotypes. 9th Euromicro Workshop on Real Time Systems, 1997, IEEE Computer.

[6] de Miguel, M. "QoS-Aware Component Frameworks. In The 10th International Workshop on Quality of Service (IWQoS 2002), May 2002.

[7] Eclipse JET (Java Emitter Templates) project. http://www.eclipse.org/modeling/m2t/?project=jet#jethttp://www.eclipse.org/modeling/m2t/?project=jet#jet

[8] Eclipse Modisco project. http://www.eclipse.org/MoDisco/

[9] Eclipse Model2Model project. http://www.eclipse.org/m2m/

[10] Obeo, Acceleo 3.1.0 User Guide. http://www.obeonetwork.com/page/acceleo-310-user-guide

[11] Gosling, J.; Bollella, G.; Dibble, P.; Furr, S.; Turnbull, M. The Real-Time Specification for Java. Addison Wesley Longman. January 15, 2000 | ISBN-10: 0201703238

[12] gforge.enseeiht.fr/docman/view.php/52/4445/A6-CNES-Obeo-v2.pdf

[13] MAST, Modeling and Analysis Suite for Real-Time Applications, http://mast.unican.es/

[14] Ming Ho, W., Jezequel, J.-M. ; Le Guennec, A. ; Pennaneac'h, F. UMLAUT: an extendible UML transformation framework. 14th IEEE International Conference on Automated Software Engineering, 1999.

[15] Object Management Group. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems. Version 1.1. ptc/2010-08-32

[16] Object Management Group. UML Profile for Schedulability, Performance, and Time Specification. Version 1.1. 2005. OMG document: formal/05-01-02.

[17] Object Management Group. Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, formal/2011-08-06.

[18] Object Management Group. MOF Model to Text Transformation Language, Version 1.0, formal/08-01-16

[19] Richardson, T.; Wellings, A.; Dianes, J.; Díaz, M. Providing temporal isolation in the OSGi framework. Proceeding JTRES '09 Proceedings of the 7th International Workshop on

Java Technologies for Real-Time and Embedded Systems Pages 1-10. ACM.

[20] Safety Critical Java Technology Specification, JSR-302, Version 0.78, 15 October 2010. http://download.oracle.com/otndocs/jcp/safety_critical-0.78-edr-oth-JSpec

[21] Saksena, M.; Ptak, A. ; Freedman, P. ; Rodziewicz, P. Schedulability analysis for automated implementations of real-time object-oriented models. In The 19th IEEE Real-Time Systems Symposium, 1998. Dec 1998. IEEE Computer.

[22] Selic, B. A generic framework for modeling resources with UML. *Computer*. Volume: 33 , Issue: 6, DOI: 10.1109/2.846320. June 2000 , Page(s): 64 - 69

[23] Stevens, P. On the interpretation of binary associations in the Unified Modelling Language. Software and Systems Modeling. Volume 1, Number 1. 2002, Pages: 68-79, DOI: 10.1007/s10270-002-0002-x

[24] Wellings, A. Concurrent and Real-Time Programming in Java. Wiley, 2004. ISBN 0-470-84437-X