AN APPROACH FOR DESIGNING REAL-TIME EMBEDDED SYSTEMS FROM RT-UML SPECIFICATIONS

Wehrmeister, M. A.[1], Becker, L. B.[2], Pereira, C. E.[3]

[1] Computer Science Institute, UFRGS, Brazil
[2] Automation and Control Systems Department, UFSC, Brazil
[3] Electrical Engineering Department, UFRGS, Brazil
Email: [1] mawehrmeister@inf.ufrgs.br; [2]lbecker@das.ufsc.br; [3]cpereira@eletro.ufrgs.br;

Abstract: The current work presents an API based on the Real-Time Specification for Java (RTSJ) that optimizes real-time embedded systems development. Using this API it is possible to state non-functional specifications, like time constraints, and guarantee its implementation in the selected platform. Moreover, it discusses how real-time requirements derived from the RT-UML standard can be mapped to the elements from the proposed API. .An integrated toolset is used to support the intermediate steps of this mapping process. The paper illustrates the mapping mechanism by means of a case study that implements the control system of an automated wheelchair. *Copyright © 2005 IFAC*

Keywords: embedded systems design, real-time, programming support.

1. INTRODUCTION

A recent study (Graff, *et al.*, 2003) shows that the current industry-practice of embedded software development is unsatisfactory, as more and more companies are having trouble to achieve sufficient product quality and timely delivery. It also depicts that development is mostly hardware-driven, i.e. software architecture often mirrors hardware architectures previously used by the company. Such practice limits innovations and optimizations on both software and hardware layers from new embedded systems projects.

Differently from the current industry practice, we defend that embedded systems design process should start focusing on systems requirements and software architecture. The hardware architecture should be addressed only afterwards, during the design phase, in conform to the system needs. To follows this proposal, high-level specification and modelling constructs should be used. A popular modelling notation within general-purpose systems is the Unified Modelling Language (UML) (Booch, *et al.*, 1999). Over the last years, UML and its real-time extension, namely UML-RT (OMG, 2003), has gained in popularity as a suitable tool for specification and modeling of embedded systems.

A previous work (Becker, *et al.*, 2002) has shown that it is possible to have a clear mapping from high-level constructs in UML-RT to the programming level of standard (not-embedded) real-time systems. In this paper we present ideas that extend the previous work aiming that such mapping can also address embedded systems requirements when using appropriated toolset. The related toolset is the Sashimi environment (Ito, *et al.*, 2001), which in combination with a special API to be presented allows the embedded systems generation directly from UML models, as further detailed along the paper.

The remaining of this paper is organized as follows. Section 2 gives an overview of previous work on mapping RT-UML to the programming level. Section 3 presents the Sashimi environment and the new API, which are both used to perform embedded systems

generation. Section 4 illustrates the proposed approach by means of a case study. Finally, section 5 draws the main conclusions and signals future works.

2. OVERVIEW OF PREVIOUS WORK

Some UML diagrams, like the class and the state-transition diagrams have a well-defined mapping to the programming level. However, there is no definition on how to map the timing constraints that can be associated to the model elements to the programming level.

In (Becker, *et al.*, 2002) it is presented an approach that defines a clear link between modelled real-time constraints and the programming entities that provide their implementation. The main idea is to enhance the traceability as well as readability of timing constraints from a model-based requirements structure to its implementation. Therefore, it focuses on the RT-UML standard (OMG 2003) and on the Real-time Specification for Java (RTSJ) API (*Bollela* 2001). This approach has been validated by several case studies.

A possible way of generating real-time Java code from the RT-UML model is following some conventions in the mapping process, as stated bellow:

- When applied to classes, the RT-UML stereotypes correspond to Java classes that may inherit from a RTSJ super class. Stereotype tags that are relevant in the context of the runtime application are mapped to RTSJ class attributes. Class constructors should accept initialization values for such attributes;
- When applied to class methods, the RT-UML stereotypes correspond to methods implemented in the generated class or in one of its attributes.

The current work follows the same approach (although the target system is completely different). More concrete examples on the proposed mapping are shown in section 4.

3. RELATED TOOLS

A key tool in the proposed design flow is the Sashimi environment (*Ito*, 2001). Therewith, designers develop their application directly in Java, although they must follow some programming restrictions in order to fulfill the environment constraints. For example, they must use only APIs provided by the Sashimi environment rather than the standard Java-API. Once application programming is finished, the source code is compiled using the standard Java compiler. The generated classes can be tested using libraries that emulate the Sashimi API in the development host. Afterwards the application can be deployed into the FemtoJava processor, which is a stack-based microcontroller that natively executes Java bytecodes. It is designed specifically for the embedded system market. Therefore, the Java bytecodes of the application are analyzed, and a customized control unit for the FemtoJava processor is generated, supporting only the opcodes used by that application. The size of its control unit is directly proportional to the number of different opcodes utilized by the application software.

Another related component is an API developed to fulfill the problems related to using the scheduling mechanism in the Sashimi environment (avoiding low-level system calls) and also to facilitate the definition of timing constraints within the embedded application. This API is based in the RTSJ. It adopts the concept of schedulable objects, which are instances of classes that implement the *Schedulable* interface, like the *RealtimeThread*. It also specifies a set of classes to store parameters that represent a particular resource demand from one or more schedulable objects. For example, the *ReleaseParameters* class (super class from *AperiodicParameters* and *PeriodicParameters*) includes several useful parameters for the specification of real-time requirements. Moreover, it supports the expression of the following elements: time values (absolute and relative time), timers, periodic and aperiodic tasks, and scheduling policies. The term 'task' derives from the scheduling literature, representing a schedulable element within the system context. It is also a synonym for schedulable object. The class diagram of the developed API is shown in Figure 1. Follows a brief description from the available classes:

- **RealtimeThread:** extends the default class java.lang.Thread and represents a real-time task in the embedded system. The task can be periodic or aperiodic, depending on the given release parameter object. If it receives a *PeriodicParameters* object then the task is periodic, otherwise if the object is an instance of *AperiodicParameters* or *SporadicParameters* class then the task is aperiodic.
- **ReleaseParameters:** base class for all release parameters of a real-time task. It has attributes like cost (required CPU processing time), task deadline, and others. Its subclasses are *PeriodicParameters* and *AperiodicParameters*, which represent release parameters for, respectively, periodic and aperiodic tasks. *PeriodicParameters* has attributes like the start and end time for the task, and the task execution period. The *SporadicParameters* class is a subclass from *AperiodicParameters* class and, as the name suggests, represents a sporadic task that may occur at any time after a minimum interval between two occurrences.
- **SchedulingParametes:** base class from all scheduling parameters that are used by the *Scheduler* object. *PriorityParameters* is a class that represents the task priority and that can be used by scheduling mechanisms like the *PriorityScheduler*.
- **Scheduler:** abstract class that represents the scheduler itself. In the API, the sub-classes *PriorityScheduler*, *RateMonotonicScheduler*, and *EDFScheduler* represent, respectively, fixed priority, rate monotonic and earliest deadline first scheduling algorithms.

- **HighResolutionTime:** base class from all classes that represent a time value. It has three 32 bits integer attributes that represents days, milliseconds and nanoseconds. The subclass *AbsoluteTime* represent an absolute instant of time which is based in the same date/time reference as specified in java.util.Date class (01/01/1970 00:00:00) (Sun, 2004). The subclass *RelativeTime* represents a time relative to other instant of time that is given as parameter. The days, milliseconds and nanoseconds attributes represent, respectively, the quantity of days, milliseconds and nanoseconds relative to a given time instant.
- **Clock:** represents a global clock reference. This class returns an *AbsoluteTime* object that represents the current date and time of the system.
- **Timer:** abstract class that represents a system timer. The derived class *OneShotTimer* represents a single occurrence timer, and the derived class *PeriodicTimer* represents a periodic one.

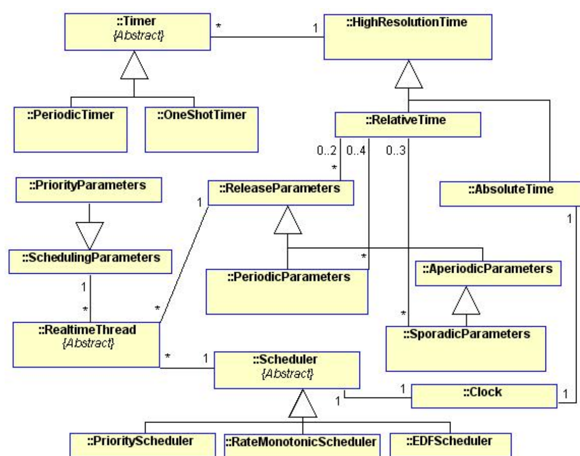

Figure 1. Class diagram of the proposed API

The implementations from some of the proposed API classes have slightly differences in comparison to the RTSJ. This is due to constraints in the FemtoJava architecture and also for clarity matters. An example of such differences appears in the *RealtimeThread* class. In the proposed implementation it has two abstract methods that have to be implemented in the derived subclasses: *mainTask()* and *exceptionTask()*. They represent, respectively, the task body – equivalent to the *run()* method from a normal Java thread – and the exception handling code applied for deadlines misses. The latter substitute the use of an *AsyncEventHandler* object, which should be passed to the *ReleaseParameters* object, as specified in the RTSJ. If the task deadline is missed, the task execution flow deviates to the *exceptionTask()* code. After the exception handling code execution, the execution flow may deviate to the *run()* method or even terminated, depending on the real-time task characteristic. If the task is periodic, than the *run()* method should be restarted. This difference was proposed to provide support to scheduling algorithms that use the concept of task-pairs, like the Time-Aware Fault-Tolerant (TAFT) scheduler (*Nett*, 2001), and also to enhance the entire task source code readability providing more legibility to source code of the entire task code.

Other class that has a different implementation is the *Timer* class. It has a abstract method named *runTimer()* that must to be implemented in the subclass and represents the code executed as the timer expires. Note that this method appears both in *OneShotTimer* and *PeriodicTimer* classes.

To support the utilization of the proposed API, some development tools must be modified and extended. For example, the Sashimi tool and the analysis of the Java class files were adapted to support the synthesis of objects. Moreover, four new opcodes were added in the FemtoJava microprocessor to provide support to objects.

4. MAPPING FROM RT-UML TO EMBEDDED JAVA

This section illustrates the mapping from a RT-UML specification to Java source code using the proposed API. Thereby, it presents the RT-UML model of a real-time embedded automation and control system for an "intelligent" wheelchair, used to support people with special needs. The main functions of the system are: movement control, collision avoidance, navigation, target pursuit, battery control, system supervision, task scheduling, automatic movement, driver's health supervision, and self testing. It also includes calendar-based activation of tasks (for instance, every day at 6 p.m. wheelchair user has to be brought to a specific place to get some medicine. Hard-real time requirements must be accomplished for safety reasons.

Our design starts with the construction of a high-level object modelling using UML together with the profile for Schedulability, Performance and Time (SPT) (OMG, 2003), also known as RT-UML. The most important UML diagrams used in the model are: Use Cases, Collaboration, and Class Diagrams. Specially the last two diagrams are decorated with the stereotypes and tag-values coming from the SPT profile. Due to space limitations, the discussions in this paper concentrate in the wheelchair movement control, which is essential to the system and incorporates critical hard real-time constraints. The wheelchair movement control functionality is presented in the use case diagram of Figure 2.

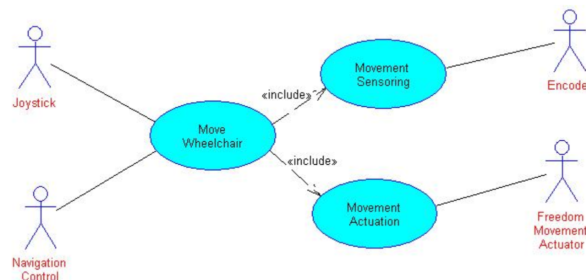

Figure 2. Use Case diagram for the wheelchair movement control

Figure 3 presents the object collaboration diagram that refines the behavior from the 'movement actuation' use case. This diagram contains three classes representing, respectively, the interface class for the joystick used in the wheelchair control, the interface class for the motor activation drive, and the class that represents the movement controller itself. Readers should attempt to the stereotypes derived from the SPT profile that are used to decorate this diagram. For example, the stereotype «SAtrigger» in A.1 is used to represent a periodic activation for the joystick data sampling. Moreover, «SAresponse» in A.1.2 is used to assign a deadline for the execution of wheelchair motors control. The complete set of classes that constitute the wheelchair movement control are depicted in the class diagram of Figure 4.

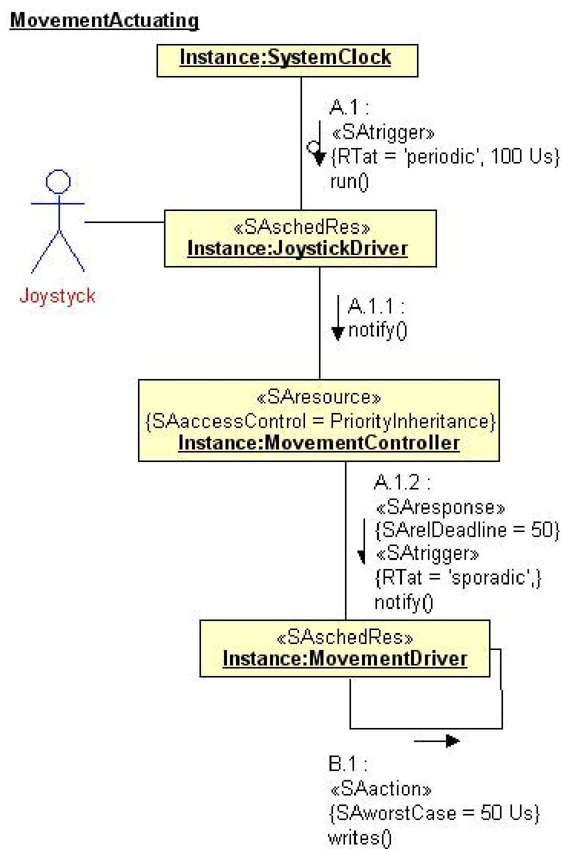

Figure 3. Movement control collaboration diagram.

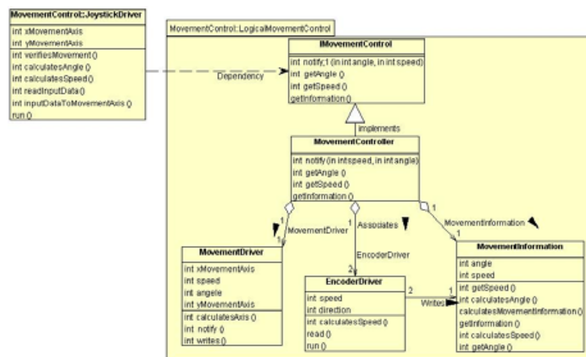

Figure 4. Wheelchair movement control class diagram

Figure 5 represents the source code of the main class from the wheelchair system. In this code one can observe that the system has two real-time concurrent objects, which are represented in the RT-UML diagram by the «*SAscheRes*» stereotype. These objects are instance from *JoystickDriver* and *MovementDriver* classes. As it can be observed in the code, the objects activation is triggered by the *start()* method call.

```
public class Wheelchair {
 // Application objects allocation:
 public static MovementController
   movementCtrl = new MovementController();

 // periodic tasks
 public static JoystickDriver
   joystickDriver = new JoystickDriver();
 public static MovementDriver
   movementDriver = new MovementDriver();
 ... //allocates remaining objects

 public static void initSystem() {
   ... //initializes remaining objects

   // Real-time tasks startup:
   Wheelchair.joystickDriver.start();
   Wheelchair.movementDriver.start();
   ... //startup remaining tasks

   while (true) {
      FemtoJava.sleep();
   }
};
```

Figure 5. Source code from the main class of the wheelchair system.

In Figure 6 it is presented the source code for the real-time class *JoystickDriver*, which represents objects responsible to read periodically the joystick hardware and, if necessary, to notify the *MovementController* about the necessity of acting in the movement. The period and deadline information derive from tags related, respectively, to the «*SAscheRes*» and «*SAscheRes*» stereotypes of the RT-UML diagram.

```
import saito.sashimi.realtime.*;

public class JoystickDriver extends
                           RealtimeThread {
 private static RelativeTime
     _100_us = new RelativeTime(0,0,100000);
 private static PeriodicParameters
     schedParams = new PeriodicParameters(
                        null,  // start time
                        null,  // end time
                        _100_us, // period
                        null,  // cost
                        _100_us);// deadline
   // Attributes
   private int m_angle;
   private int m_intensity;
   ... // other attributes
   public JoystickDriver() {
      super(null, schedParams);
      // do other initializations
   }
   ... //continues
};
```

Figure 6. JoystickDriver class

Figure 7 depicts the remaining parts from the *JoystickDriver* class. It contains two important

methods: *mainTask()* and *exceptionTask()*. The former represents the task body, that is, the code executed when the task is activated by calling the *start()* method. Since this task is periodic, there must be a loop which denotes the periodic execution. The loop execution frequency is controlled by calling the *waitForNextPeriod()* method. This method uses the tasks release parameters to interact with the scheduler and control the correct execution for the method. The *exceptionTask()* method represents the exception handling code that is triggered in case of deadline miss, that is, if the *mainTask()* method does not finish up to the established deadline.

```
public class JoystickDriver extends
                              RealtimeThread {
 ... //continuation
 public void mainTask() {

    while(isRunning == true){//periodic loop
       // read analogic joystick commands
       this.readValuesFromHardware();

       Wheelchair.movementCtrl.notify(
                          this.m_angle,
                          this.m_intensity);

       this.waitForNextPeriod();
    }
 }

 private void readValuesFromHardware(); {
   ... // reading hardware
 }

 public void exceptionTask() {
   // handle deadline missing
 }
};
```

Figure 7. JoystickDriver class continuation

The sporadic task that is responsible to control the motor activation of the wheelchair is depicted in Figure 8. As showed in Figure 2, the task representing the motor driver will be executed each time joystick is used.

```
import saito.sashimi.realtime.*;

public class MovementDriver extends
                              RealtimeThread {
 private static RelativeTime
       _50_us = new RelativeTime(0,0,50000);
 private static SporadicParameters
     schedParams = new SporadicParameters(
             null, // min. interarrival time
             null, // cost
             _50_us);// deadline

   // Attributes
   private int m_angle;
   private int m_intensity;
   private int speed;
   ... // other attributes

   public MovementDriver() {
      super(null, schedParams);
      // do other initializations
   }

   ... //continues
};
```

Figure 8. MovementDriver class

The code structure for the *maiTtask()* and *exceptionTask()* is similar to the JoystickDriver class. Note that this approach (two methods representing a task) is a difference in comparison with the RTSJ. The *mainTask()* and the *exceptionTask()* represent, respectively, the task body – equivalent to the *run()* method from a normal Java thread – and the exception handling code applied for deadlines misses. The latter substitute the use of an *AsyncEventHandler* object, which should be passed to the *ReleaseParameters* object, as specified in the RTSJ. If the task deadline is missed, the task execution flow deviates to the *exceptionTask()* code. After the exception handling code execution, the execution flow may deviate to the *run()* method or even terminated, depending on the real-time task characteristic. If the task is periodic, than the *run()* method should be restarted. This difference was proposed to provide support to scheduling algorithms that use the concept of task-pairs, like the Time-Aware Fault-Tolerant (TAFT) scheduler (Nett, *et al.* 2001), and also to enhance the entire task source code readability.

The last discussion of the section relates to the ConsoleInterface class, which is not present in the presented UML diagram, but incorporates important features. This class is responsible for controlling the interaction between the wheelchair-user and the control system. It was chosen because it exemplifies the timer usage. The scenario exposed in this sample relates to an operation parameter change. The user chooses the parameter to change and, afterwards, has up to 15 seconds to save the new value. If he does not save the new value until the time limit, then the crane user interface will be restarted. In Figure 7 one can observe the *ParameterTimeOut* class that is responsible to signal the timeout. Note that the *ParameterTimeOut* extends the *OneShotTimer*, in other words, the timer will be executed just once per activation.

```
public class ParameterTimeOut extends
OneShotTimer {
   private ConsoleInterface m_owner;

   public ParameterTimeOut(ConsoleInterface
owner, HighResolutionTime timeout) {
      super(timeout);
      m_owner = owner;
   }

   protected void runTimer() {
      owner.abortUserInput();
   }
};
```

Figure 9. ParameterTimeOut class

The ConsoleInterface class can be analyzed in Figure 8. Note that in the *getParameterFromInterface()* method the timer will be started and the routine will remain in loop until the user saves the new value or until the operation is aborted because the timeout occurrence.

```
public class ConsoleInterface extends
RealtimeThread {
   // 15 seconds
   private RelativeTime _15_s = new
RelativeTime(0, 15000, 0);
   private ParameterTimeOut paramTimeOut =
new ParameterTimeOut(this, _15_s);
   private m_UserInputOK = false;
   private m_Abort = false;

   ... // continuation

   public void getParameterFromInterface() {
..// Print user interface and wait the input

      m_UserInputOK = false;
      m_Abort = false;

      paramTimeOut.start();
      while ((!m_UserInputOK) || (!m_Abort))
         FemtoJava.sleep();

      ... // make apropriate finalizations
   }

   public void abortUserInput() {
      m_Abort = true;
   }
};
```

Figure 10. ConsoleInterface class

5. CONCLUSIONS

The current work presented an API based on the RTSJ that optimizes real-time embedded systems development. Moreover, it discusses how RT-UML specifications can be mapped to the API elements. Using the proposed mapping it is possible to generate the embedded application directly from the UML level.

It is important to mention that the mapping process is not unique and other APIs offer different alternatives to implement a given specified timing requirement. The mapping process presented in this paper focused on the simplest constructors in order to increase readability by generating a modular and maintainable code. These features are obtained from the adequate structure of the proposed API.

For future work authors intend to implement the proposed mapping scheme into a case tool, increasing the automation level from embedded systems design-flow.

ACKNOWLEDGMENTS

This work has been partly supported by the Brazilian research agency CNPq within the scope of the SEEP research project. Thanks are also given to all researchers involved in the SEEP project for their valuable discussions, in special to Profs. Flavio Wagner and Luigi Carro, and also to Julio Mattos.

REFERENCES

Becker, L.B., Höltz, R., and Pereira, C.E. (2002). "On Mapping RT-UML Specifications to RT-Java API: Bridging the Gap". In prof. of International Symposium on Object-Oriented Distributed Real-Time Systems, Washington, USA. pp. 348-355.

Booch, I. Jacobson, and J. Rumbaugh (1999). The Unified Modeling Language User Guide. Addison-Wesley.

Bollela, G., et.al. (2001) The Real-Time Specification for Java. Addison-Wesley.

Graff, B., M. Lormans and H. Toetenel (2003). Embedded Software Engineering: The State of the Practice. IEEE Software. pp. 61-69.

Ito, S. A., L. Carro and R. P. Jacobi (2001). "Making Java Work for Microcontroller Applications". IEEE Design & Test of Computers, vol. 18, n. 5, pp. 100-110.

Nett E., Gergeleit M., and Mock M. (2001) "Enhancing OO Middleware to become Time-Awere", Special Issue on Real-Time Middleware in Real-Time Systems, pp. 211-228. Kluwer Academic Publisher.

OMG (2003). UML Profile for Performance, Schedulability, and Time. OMG document in. http://www.omg.org/technology/documents/formal/schedulability.htm.

Sun Microsystems (2004). Java 2 Platform Api Specification. Sun document in http://java.sun.com/j2se/1.5.0/index.jsp