RTSJ-based API for Real-Time Embedded Systems

Marco A. Wehrmeister
Computer Science Institute
Federal University of Rio Grande do Sul, Brazil
mawehrmeister@inf.ufrgs.br

Leandro B. Becker
Automation and Control Systems Department
Federal University of Santa Catarina, Brazil
lbecker@das.ufsc.br

Carlos Eduardo Pereira
Electrical Engineering Department
Federal University of Rio Grande do Sul, Brazil
cpereira@eletro.ufrgs.br

Abstract

Java's popularity in the real-time embedded systems domain has grown significantly over the last years. This popularity influenced the definition of the Real-Time Specification for Java (RTSJ), which is a high-level programming interface for real-time applications written in Java. This paper describes an RTJS-based API, which aims to facilitate the development of real-time embedded systems by including high-level constructs for concurrent real-time computation.Further, the developed application can be synthesized and optimized (in terms of footprint and timing requirements) to a customizable Java processor. The use of the proposed API is illustrated in the paper by means of a case study that implements a crane control system. This case study highlights the benefits and advantages on using the proposed API.

1. Introduction

Real-time embedded systems (RTES) are ubiquosly present in our daily life. It is expected that new generation of these systems will have more functions, increasing considerably its complexity. A prominent solution for handling complexity is the use of object-orientations concepts. Therefore, Java has gained popularity in embedded real-time systems development.

The Real-Time Specification for Java (RTSJ) [1] is an Application Programming Interface (API) that allows the creation,verification, execution, and management of Java-based real-time applications. Quite recently, some Java Virtual Machines (JVM) that fully support the RTSJ have been made available. The TimeSys RTSJ Reference Implementation (RI) [10] was the first RTJVM that implements all mandatory features in the RTSJ and is based on the Java 2 Micro Edition (J2ME) JVM running on Linux OS. Sun Labs' Mackinac [9] aims to allow the use of Java in physical systems control. A third example is the JRate [11] that is an open-source RTSJ-based extension to GNU Compiler for Java (GCJ) runtime systems. This proposal is a little different than the afore mentioned ones, because the Java application is ahead-of-time compiled into native code that means there is no JVM.

These implementations, however, are not targeted for the embedded systems domain, on which footprint requirements are of equal importance as real-time requirements. The Sashimi environment (see [7]) is an example of JVM optimization for embedded systems. The main concept of this proposal is the use of a configurable Java processor, called FemtoJava [7], which natively executes Java bytecodes and is optimized to execute only opcodes effective needed by application. The hardware is modified to fit application (software) requirements and not the other way around, as usual in most programming environments

However, as originally proposed, the Sashimi environment lacks a programming model for representing concurrency and real-time constraints. The goal of the current work is to overcome this limitation by providing an RTSJ-based API that supports the specification of concurrent tasks and also the specification of timing constraints. Using the provided API together with Sashimi environment, programmers can develop concurrent real-time applications and synthesize them into the FemtoJava processor.

The remaining of the paper is divided as follows: section 2 gives an overview on the Sashimi environment and the FemtoJava processor. Section 3 details the proposed API by describing its class hierarchy and section 4 presents a case study that elucidates the use of the proposed API, remarking the achieved benefits. Finally, section 5 highlights some conclusions and signals future work directions.

2. Sashimi Environment

Sashimi adopts Java as specification language for deploying embedded systems. In order to fulfill the environment constraints, some programming restrictions must be followed. For example, only integer numbers are allowed, and programmers should only use APIs provided by the Sashimi environment rather than the standard Java-API. Additionally,

designers should use only static methods and attributes, since there is no support for object allocation. Additionaly, the use of inheritance in class hierarchy and method polymorphism, key concepts in the object oriented development, are also not supported.

Using the Sashimi environment, Java-based specifications are translated into bytecodes by using standard Java compilers. The generated classes can be tested using libraries that emulate the Sashimi API in the development host. Based on the generated bytecodes, the application and the FemtoJava processor are synthesized. The 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, making it suitable for embedded applications. Different scheduling algorithms are supported by Sashimi. An evaluation of the impact of these algorithms in terms of footprint, power consumption and real-time performance are described in [5]. A drawback of the Sashimi environment is the lack of high-level constructs, forcing designers to use low-level system calls to generate concurrent processing and to interact with the scheduler. Additionally, there is no mechanism to express clearly the tasks timing constraints.

3. The Proposed API

The main purpose of the developed API is to overcome drawbacks of the Sashimi environment related to the use of low level constructs to schedule concurrent processes. Moreover, it should facilitate the fulfillment of timing constraints.

As previously mentioned, this new API is based in the RTSJ [1]. It makes use of the concept of schedulable objects, which are instances of classes that implement the Schedulable interface, such as RealtimeThread. The RTSJ-based API provides support to the following concepts: 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, on other words, a schedulable object. Follows a brief description of the main classes:

- **Real-timeThread:** extends the default class "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.
- **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 parameter for periodic and aperiodic tasks.
- **SchedulingParametes:** represents 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 as the PriorityScheduler.
- **Scheduler:** abstract class that represents the scheduler itself. Its subclasses "PriorityScheduler", "RateMonotonicScheduler", and "EDFScheduler" represent, respectively, fixed priority, rate monotonic and earliest deadline first scheduling algorithms.
- **HighResolutionTime:** base class for all classes that represent a time value. The subclass "AbsoluteTime" represents an absolute instant of time which is based in the same date/time reference as specified in Java Date class [8]. The subclass RelativeTime represents a time relative to other time instant that is given as parameter.
- **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.

The implementations for some of the proposed API classes have slightly differences to the proposed in RTSJ. This is due to constraints in the FemtoJava architecture and also some implemented extensions. An example of such differences is in the RealtimeThread class which has two abstract methods that must 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, 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 [3], which allows the implementation of adaptive behavior.

As mentioned in section 2, the original version of the Sashimi environment provided no support for object creation. Therefore, some extensions were introduced in order to provide full support for the proposed API in the FemtoJava platform. Firstly it was

necessary to extend the Sashimi environment with support to the synthesis of objects. According to the performed modifications, applications objects are statically allocated at synthesis time. In other words, all objects in the system are defined a priori, allowing the determination of the total memory necessary to store them into the RAM. Although such practice may incurs into higher memory usage, it is a suitable practice in real-time development, as it avoids the use of the garbage collector, which usually introduces non tolerable overheads that can not be tolerated by real-time applications.

The FemtoJava microprocessor also needed to be expanded to support the proposed API: four new opcodes were introduced: getfield, putfield, invokevirtual and invokspecial. The first two opcodes are related to object fields' access. They are used, respectively, for getting and setting values. The other two opcodes are related to method invocation. Another required change in the FemtoJava microprocessor is the addition of a real-time clock, used to provide the notion of time in the embedded system. This clock should be used both by the API elements and also by the scheduling layer.

4. Case Study

The crane control system, proposed as a benchmark for system level modeling [2] has been used as case study to validate the proposed Sashimi extensions. The design solution to be presented along the section follows the UML model of the crane system that is presented in [4].

An important aspect from this diagram is that it is decorated with stereotypes derived from the UML profile for performance, schedulability, and time, or simply UML-RT [6]. An example from such stereotypes is the «SASchedulable», which denotes a concurrent task in the system (see Controller class). Analyzing the object collaboration diagram presented in Fig. 1, we can observe the Controller class is also decorated with the stereotypes «SATrigger» and «SAResponse», which indicate, respectively, that this task is triggered periodically every 10 ms, with a deadline of 10 ms.

For the sake of paper length's limitation, although the system includes several classes (see Table 1) only the "CraneInitializer" and "Controller" classes will be discussed. Nevertheless, these classes are representative enough to depict the use of the API elements.

The main class from the crane system implementation is named "CraneInitializer". This class is responsible for objects allocation, initialization, and starting (applied for the real-time tasks). Its source code is depicted in Table 1.and one can observe that only static objects are allocated.

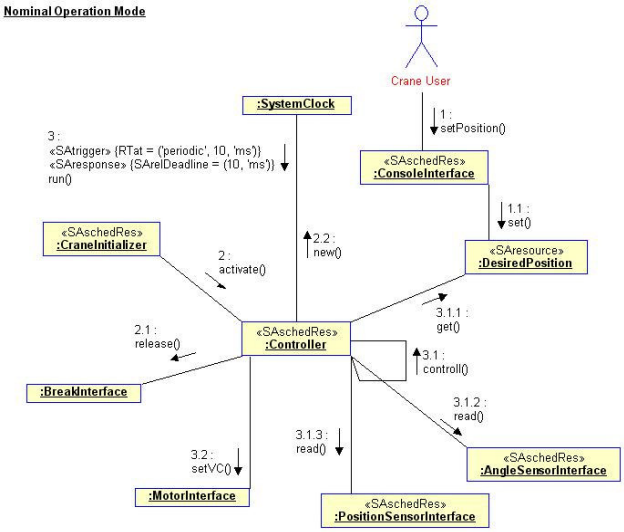

Fig. 1. Object Collaboration Diagram

Another important aspect from the code of Table 1 relates to the initSystem() method, which represents the starting point from the application execution flow and provides objects initialization and real-time tasks startup. The real-time tasks startup can be identified by the call to the start() method.

Table 1. Main class for the Crane System

```
public class CraneInitializer {
   // Application objects allocation
   public static Controller nominalCtrl = new
      Controller();
   public static ConsoleInterface
      ConsoleInterface = new
      ConsoleInterface();
   public static DesiredPosition
      desiredPosition = new DesiredPosition();
   public static BreakInterface
      breakInterface = new BreakInterface();
   public static void initSystem() {
      ... //Object initializations
      // Real-time tasks startup:
      Crane.nominalCtrl.start();
      ... //startup remaining tasks
      while (true) FemtoJava.sleep();
   }
};
```

The following discussion relates to the Controller class, on which the associated stereotype denotes a concurrent real-time task in the system. As previously mentioned, this task must be periodically activated every 10 ms, with a deadline of 10 ms. To implement such features using the provided API, the Controller class needs to inherit from "RealtimeThread", as shown in Table 2. as well as to make use of the class

PeriodicParameters from the API, whose instance is passed as parameter for the constructor. A "RelativeTime" object is used to represent the 10 miliseconds time for the task period and deadline. The mainTask() method represents the task body, that is, the code executed when the task is activated by calling the start() method. The exceptionTask() method represents the exception handling code that is triggered in case of deadline miss.

Table 2. Controller Class

```
import saito.sashimi.realtime.*;
public class Controller extends RealtimeThread
{   private static RelativeTime
              _10_ms = new RelativeTime(0,10,0);
   private static PeriodicParameters
     schedParams = new PeriodicParameters(
                          null,   // start time
                          null,   // end time
                          _10_ms, // period
                          null,   // cost
                          _10_ms);// deadline
   ... // constructor and other methods
   public void mainTask() {
      Crane.breakInterface.release();
      // periodic loop
      while(isRunning == true){
         this.controll();
         Crane.monitorInterface.setVC(m_vc);
         this.waitForNextPeriod();
      }
   }
   private int controll() { ... }
   public void exceptionTask() {
      // handle deadline missing
   }
};
```

The next step in the project cycle is to synthesize the embedded system. The Sashimi enviroment that takes as input the Java class files, generated by the Java compiler, to generate the hardware from the FemtoJava processor (in form of VHDL files) and the software of the embedded system.

It is important to highlight that the hardware generated by Sashimi is optimized because it supports only the Java opcodes used by the embedded system software, economizing in area, which is a design constraint from actual embedded systems. Another point to be observed is that once the application objects are allocated at synthesis time, there is no need for using a garbage collector providing the required determinism from real-time embedded applications.

5. Conclusions and Future Work

The current work presented an API based on the RTSJ that optimizes real-time embedded systems development. This API is target to the FemtoJava processor, which is a stack-based processor specially design to execute Java bytecodes. Using the provided API, programmers can make use of high-level mechanisms to represent concurrency and timing constraints in their Java applications.

To keep the proposed API as close as possible to the RTSJ, minor modifications were provided. These changes relates basically on how one can define a timeout exception handler for the concurrent real-time tasks operations, which is applied as the operation violates its deadline. The provided method provides a clear code, which is considered easier to be understood.

For future work, authors intend to adapt the used modeling tool to provide code generation from the UML diagrams to Java using the API elements. Therefore, it will be possible to generate the embedded application directly from the UML level.

6. Acknowledgements

This work has been partly supported by the Brazilian research agency CNPq within the scope of the SEEP research project.

7. References

[1] Bollella, Greg; Gosling, James; Brosgol, Benjamin (2001). "The Real-Time Specification for Java", http://www.rtj.org/rtsj-V1.0.pdf

[2] E. Moser and W. Nebel. Case Study: System Model of Crane and Embedded Control. In: Proceedings of DATE'1999 – Design, Automation and Test in Europe, Munich, Germany, March 1999

[3] E. Nett, M. Gergeleit, and M. Mock "Enhancing OO Middleware to become Time-Awere", Special Issue on Real-Time Middleware in Real-Time Systems, 20(2): 211-228, March, 2001, Kluwer Academic Publisher. ISSN-0922-6443.

[4] L. Brisolara, L.B. Becker, L. Carro, F. Wagner, and C.E. Pereira. Evaluating High-level Models for Real-Time Embedded Systems Design. To be published in: IFIP Working Conference on Distributed and Parallel Embedded Systems. Toulouse, France. 2004.

[5] L.B. Becker, M.A. Wehrmeister, L. Carro, F. Wagner, and C.E. Pereira. Evaluating High-level Models for Real-Time Embedded Systems Design. To be published in: 29th Workshop on Real-Time Programming. Istanbul, Turkey, 2004.

[6] Object Management Group (2003). "UML Profile for Schedulability, Performance and Time Specification", http://www.omg.org/cgi-bin/doc?ptc/02-03-03

[7] S.A.Ito, L.Carro, R.P.Jacobi. "Making Java Work for Microcontroller Applications". IEEE Design & Test of Computers, vol. 18, n. 5, Sept/Oct. 2001, pp. 100-110

[8] Sun Microsystems, "Java 2 Platform Api Specification", http://java.sun.com/j2se/1.4.2/docs/api/

[9] J. Heiss. "From Rockets to Power Plants to Automobiles A Conversation with Real-Time Specification for Java Expert", http://java.sun.com/developer/technicalArticles/Interviews/Bollella_qa.html

[10] TimeSys. "Java Reference Implementation (RI)", http://www.timesys.com/index.cfm?bdy=java_bdy_ri.cfm

[11] A. Corsaro, D. Schmidt. "Evaluating Real-Time Java Features and Performance for Real-Time Embedded Systems", In: Proceedings of RTAS'02 - VIII Real-Time and Embedded Technology and Applications Symposium, p.90, September, 2002