ASSESSING THE USE OF RT-JAVA IN AUTOMOTIVE TIME-TRIGGERED APPLICATIONS

Marco A. Wehrmeister[1], Fernando H. Athaide[2], Fabiano C. Carvalho[1], Carlos E. Pereira[2]

[1]*Computer Science Institute, Federal University of Rio Grande do Sul, Brazil*
{mawehrmeister, fhataide, fccarvalho}@inf.ufrgs.br

[2]*Electrical Engineering Department, Federal University of Rio Grande do Sul, Brazil*
{cpereira}@eletro.ufrgs.br

Abstract: Distributed embedded real-time systems are becoming widely used in several areas of application, including automotive systems. Most of these applications have severe timing constraints and are safety critical. A key component in distributed embedded real-time systems is to ensure a deterministic and reliable communication among distributed embedded systems. Especially in the automotive area, which is considering the possibility of replacing the major part of mechanical and/or hydraulic systems for electronic systems, the importance of a correct behavior in the electronic communication system plays a key role. This paper presents an platform-based embedded real-time system design methodology which can be used in automotive embedded real-time system design. The use of the proposed method is demonstrated through a steer-by-wire case study implemented using a Java platform.

Keywords: Embedded Real-Time Systems, RT-UML, RT-Java, Time-Triggered Architecture, Steer-By-Wire

1. INTRODUCTION

Recent advances in the areas of electronics and informatics have enabled the use of embedded computational systems in different application areas. In the automotive sector, the so called "x-by-wire" systems have been investigated as a cost-effective alternative to replace mechanical and

hydraulic systems. The "x" in "x-by-wire" represents the basis of any safety related application, such as steering, braking, suspension control or multi-airbag systems. These applications should greatly increase overall vehicle safety by assisting the driver to find solutions in critical situations. X-by-wire systems have hard timing constraints and must present a deterministic predictable behavior[1]. Most of x-by-wire systems are implemented as distributed embedded real-time systems. The design of this kind of system is recognized as a very complex activities. It has to deal with the (co-)design of hardware and software components – including a communication infra-structure – as well as due to the safety and reliability constraints from the application domain. Thus the entire distributed embedded real-time systems must be deterministic and reliable. The time-triggered architecture - TTA (see for instance Kopetz and Bauer[5]) has been successfully applied to some automotive applications and can be considered a interesting deployment architecture.

From a high-level perspective, the object-oriented paradigm appears as an interesting alternative to tackle the complexity in the design of distributed real-time embedded systems. Object oriented methodologies using the Unified Modeling Language (UML)[12] are widely accepted, specially in the development of software-intensive systems [7,8,9,10]. However, the application of object-oriented concepts in the area of distributed embedded real-time systems is still a research area.

This paper presents the Platform-Based Embedded Electronic Systems design methodology, or simply SEEP (project acronym in Portuguese), applied to a time-triggered distributed embedded system design. The SEEP methodology encompasses all development phases, including modeling, analysis, validation, and synthesis tools to support the development of optimized real-time embedded systems. The approach is based on the reuse of hardware and software components and on the configuration of predefined FPGA-based architectural platforms.

The remaining of this paper is organized as follows. Section 2 compares event-triggered and time-triggered systems. Section 3 gives a brief overview on the design methodology proposed in the SEEP project. Section 4 shows the application of the SEEP's design methodology to the development of a steer-by-wire system. Finally, Section 5 draws conclusions and signals future work directions.

2. EVENT-TRIGGERED VS. TIME-TRIGGERED SYSTEMS

There has been an intense debate in the real-time community on which of the two approaches, event-triggered or time-triggered, has more advantages in the design of distributed embedded real-time systems (see for instance, Kopetz and Bauer[5]). In the event-triggered model, all communications and processing activities initiates when a significant event occurs (for instance, the system enters into a new state). In the time-triggered approach, all activities are started periodically at predetermined instants.

Event-triggered systems usually require lower network bandwidth since a new message transmission is only needed if the value to be transmitted is distinct from the previous one. A drawback of this approach is that missing messages may cause synchronization problems, which can be detected only at the sender. In time-triggered systems, messages are transmitted periodically so, both sender and receiver can detect a communication fault. In this case, although missing messages may temporarily lead to synchronization loss, they can be detected at the receiver because all message transmission instants are known *a priori*. Time-triggered systems are considered to be predictable and more composable than event-triggered due to the temporal regularity of its communication pattern. However event-triggered systems are recognized as more flexible than time-triggered systems.

The Time Triggered Architecture (TTA) (Kopetz and Bauer[5]) proposes a communication system that is executed autonomously, controlled by a time-triggered schedule (Time Division Multiple Access - TDMA). In TTA, the tasks read/write state messages from/to Communication Network Interface (CNI), at predefined instants known *a priori*. Therefore, the communication controller reads the messages from CNI and transmit them, also at instants known a priori, for all other nodes in the network. This transmission overwrites the previous received messages on all other CNIs in the cluster. Execution times of the periodic fetch and delivery actions are stored into the message scheduling table (Message Descriptor List – MEDL) on each communication controller.

An important feature of the CNI, is that it separates the local processing within a node from the global interactions among the nodes. It consists of two unidirectional data-flow interfaces, one from the host computer to the communication system and the other one in the opposite direction.

2.1 Time-Triggered Scheduling

In a time-triggered scheduling strategy, the static scheduler activates tasks in a strictly sequential order, which is stored in a dispatcher table. The dispatcher table is executed cyclically, providing a periodic task execution scheme. A time-triggered task does not have a static priority, its activation time is configured in the dispatcher data structure. If a task is running and another task becomes active (its activation instant arrives), the current task will be preempted and the other task will run. After termination, the preempted task will continue its execution. All time instants related the activation of tasks are stored into a dispatch table that follows the temporal characteristics of the application domain and the precedence relations among tasks. The dispatch table is defined during system configuration phase and takes into account the specified timing requirements. That means that task scheduling is done offline and therefore all time instants related to task activations are know *a priori*.

An example of a scheduler that follows these characteristics is the OSEKtime[3]. According to the OSEKTime specification, there is no task synchronization via blocking mechanisms. An RTOS that implements this offline schedule approach is the TTP-OS[6], which is a time-triggered operating system with TTP communication system support. An example of time-triggered tasks activations, showed in figure 5, illustrates six time-triggered tasks (TTTask) distributed on different nodes, which are activated at different times.

3. SEEP PROJECT

The SEEP project proposes a complete and integrated methodology for the analysis, design, implementation, and test of real-time embedded systems. An overview of the proposed design methodology, which extensively supports design space exploration activities, is depicted in figure 1 (interested readers on a detailed description of the SEEP project are referred to Wehrmeister *et al.*[13]).

The SEEP approach aims to ensure a smooth transition between development phases, from RT-UML[2] specification to implementation. The transition from higher to lower abstraction levels is facilitated by the use of a real-time Java API[14], whose underlying facilities are customizable and optimized according to the application requirements and available platforms. This API includes high-level real-time constructs and therefore avoids the

use of low-level system calls to implement the specified temporal behavior. Furthermore, using the provided API it is possible to design concurrent real-time Java applications and synthesize them into a dedicated Java processor[15].

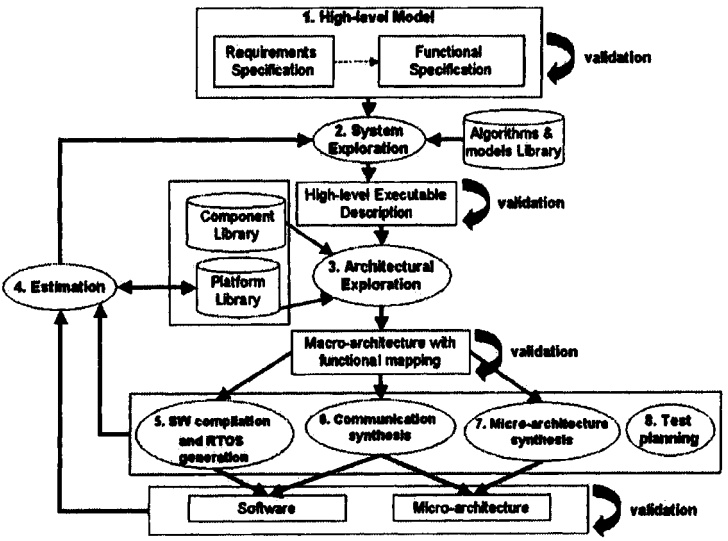

Figure 1. The SEEP design methodology

The mapping from RT-UML specification diagrams to the RTSJ-based API is discussed in Becker *et al,*[11] on which a clear link between real-time requirements and the programming entities that provide their implementation is established. The main idea of this approach is to enhance the traceability as well as the readability of timing constraints from a model-based requirement model to implementation.

3.1 Target Hardware Platforms

Currently, two platforms are being considered within the SEEP project. The first one is based on a family of customizable Java processors, implemented on commercial FPGAs, while the second one is a commercial platform, containing a large FPGA with two PowerPC processors cores. In this paper we will focus on the customizable Java platform.

In SEEP methodology, one of the possibilities is to use Java source code to represent the high-level executable description. Using the SASHIMI environment[15], both a VHDL description for a dedicated Java processor and the respective program memory code (application code) are generated. This CAD environment automatically synthesizes an Application Specific

Instruction Set Processor (ASIP) microprocessor (named FemtoJava) for a target application, using only the subset of instructions used by the designed application. This Java processor implements a Java execution engine in hardware through a stack machine compatible, but restricted, Java Virtual Machine (JVM) specification. A customized control unit for the FemtoJava processor is generated, supporting only the opcodes used by that application. The control unit of the synthesized processor is thus directly proportional to the number of different opcodes needed by the application software, therefore optimizing the final footprint based on application requirements.

In order to express more clearly timing constraints in the source code of real-time embedded application, an API based on the Real-Time Specification for Java (RTSJ) was developed[14]. This specification introduces the concept of schedulable objects, which are instances of classes that implement the Schedulable interface, such as 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 (superclass from AperiodicParameters and PeriodicParameters) includes several useful parameters for the specification of real-time requirements. Additionally, RTSJ supports the expression of other useful concepts for real-time systems development, such as 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. For a detailed description of the RTSJ-base API see Wehrmeister *et al.*[14].

4. CASE STUDY

This paper presents a case study, on which the SEEP methodology is applied to the development of a steer-by-wire system that follows the time-triggered model of computation. This application presents hard real-time requirements that must be accomplished for safety reasons. Johannessen[4] presents a steer-by-wire system that has six nodes interconnected through a TTP/C network (see Figure 2). This steer-by-wire system has three steering modes: two-wheel steering (normal), four-wheel steering and parallel steering.

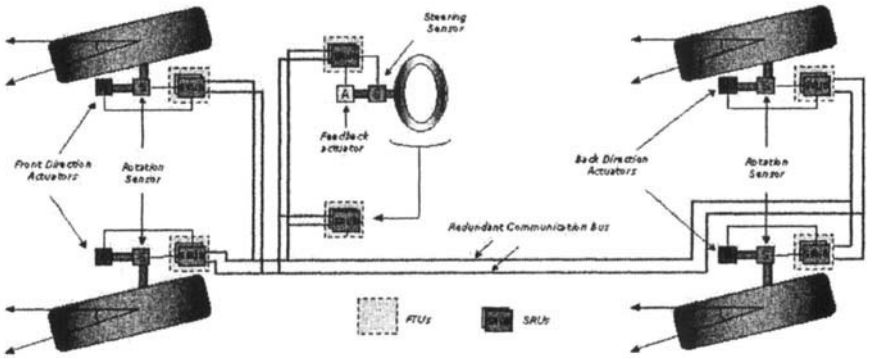

Figure 2. Steer-By-Wire Systems Architecture [Johannessen[4]]

The steer-by-wire system proposed in this case study includes two subsystems: the steering wheel subsystem that runs on the computational node located in the steering wheel; and 4 wheel subsystems that run on each wheel. The first subsystem is responsible for sampling the steering wheel rotation angle and disseminating this information to the wheel nodes through the communication network. This subsystem also has to provide feedback force to the steering wheel, based on the angle and the velocity of the wheels. The wheel subsystems must sample the angle and speed, and then send these two values to the steering wheel node. Also, the wheel computational nodes have a controller task that controls the rotation angle according to the steering wheel position and selected steering mode. The wheel angle depends directly on the steering wheel angle applied (by the driver) and the selected steering mode. It is important to highlight that due to limitations in papers' length, the discussion on this paper will focus in the timing aspects only, despite the fact that authors are aware that fault-tolerant aspects represent a fundamental issue in safety-critical applications.

The design process started with the construction of a high-level object model using RT-UML. Diagrams used in the model are: Use Cases, Collaboration, and Class Diagrams. Particularly the last two diagrams are decorated with the stereotypes and tag-values define by the RT-UML profile[2].Figure 3 and 4 depicts two collaboration diagrams for the steer-by-wire system: one for the steering wheel node (figure 3) and other for the wheel node (figure 4).

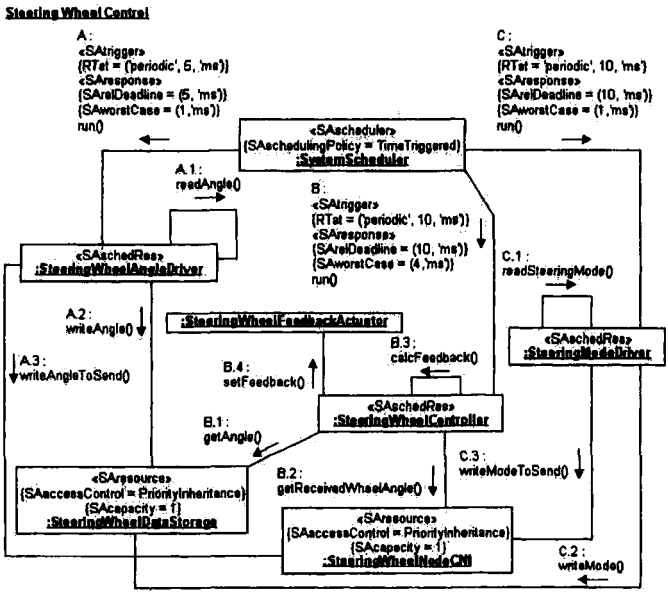

Figure 3. Collaboration Diagram for Steering Wheel Subsystem

In figure 3, the stereotype «SAtrigger» represents a periodic activation for active objects. Three active objects have been identified: the SteeringWheelAngleDriver that samples the steering wheel angle and stores this value in the shared memory (SteeringWheelDataStorage object); the second active object is the SteeringModeDriver which provides the steering mode selected by driver to the steering wheel subsystem, this value is stored in the shared memory and in the CNI; finally, the third active object is the SteeringWheelControler that is responsible to perform the control of this subsystem. It includes the control algorithm responsible for feedback force, the calculation of the steering wheel's rotation angle and the storage of the computed values in CNI for transmission to the other nodes. Additionally, a stereotype «SAresponse» specifies task's deadline and worst-case execution time (WCET). Similarly, in figure 4 the «SAtrigger» and «SAresponse» stereotypes identify the real-time constraints of active objects in the wheel node system, an information included in the time-triggered dispatch table.

In the "system design exploration" phase, algorithms and components libraries are selected to optimize the steer-by-wire system for a given platform (see for instance Mattos *et al.*[16] for further details). For the present case study, it is assumed that the Java programming language is chosen for coding the selected algorithm, so that programs that make use of the RTSJ-based API (see Wehrmeister *et al.*[14]) are generated.

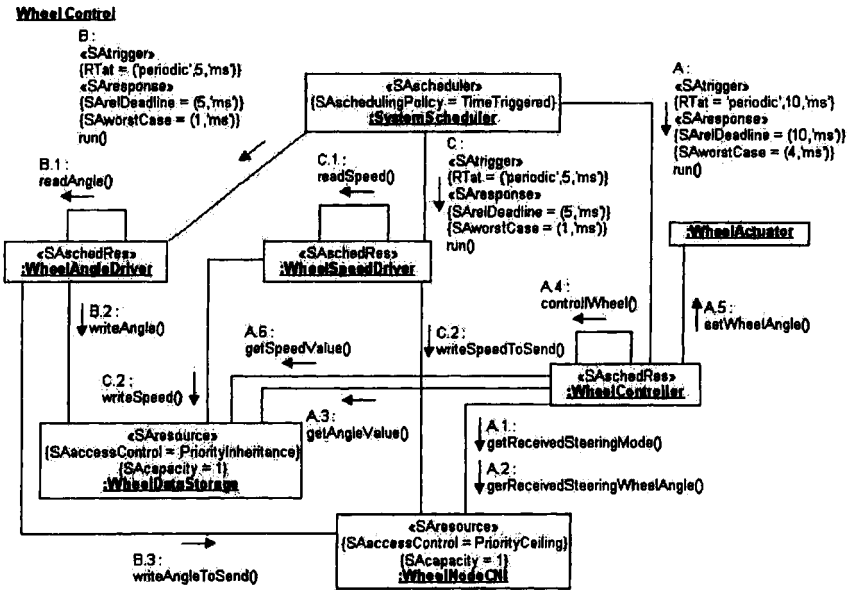

Figure 4. Collaboration Digrams for Wheel Subsystem

When applied to classes, the RT-UML stereotypes correspond to Java classes that may extend classes from a RTSJ-based API, e.g. «SAschedRes. Stereotype tags that are relevant in the context of the runtime application are mapped to RTSJ-based API class attributes. Class constructors should accept initialization values for such attributes. When applied to methods, stereotypes correspond to methods implemented in the generated class or in one of its attributes. The tasks characteristics, modeled in the collaboration diagrams presented in figures 3 and 4, are used as basis to build the dispatch table (as mentioned previously this is done offline and based on application time requirements). Figure 5 depicts the generated dispatch table where the predefined task activation instants can be observed.

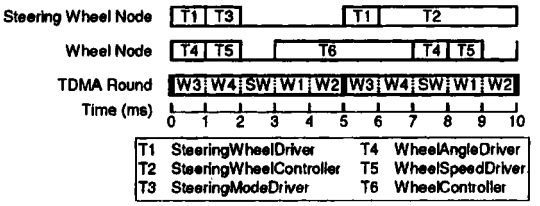

Figure 5. Task and comunication schedules

Figure 5 also shows the TDMA round that drives the communication among steer-by-wire nodes where SW stands for the steering wheel node and Wn for the other four wheel node. Figure 6 depicts the performed mapping,

showing the generated source code for the real-time class SteeringWheelControl, which includes objects responsible for the periodic force feedback control applied to the steering wheel. The period and deadline information derive from tags specified in the collaboration diagram in figure 4. They are represented, respectively, by the «SATriggers» and «SAResponse» stereotypes of the RT-UML. As it can be seen in the code, two distintic method: *mainTask()* and *exceptionTask()* are generated. The former represents the task body, i.e., the code executed when the task is activated. This is a periodic task, on which periodic activation is implemented as a loop with execution frequency being controlled by calling the *waitForNextPeriod()* method. This method controls the task execution based on the dispatch table generated by the offline scheduler. The *exceptionTask()* method represents an exception handling code that is triggered in case of deadline misses, i.e., if the mainTask() method does not finish within the established deadline.

```
01  public class SteeringWheelController extends RealtimeThread {
02    private static PeriodicParameters
03                                releaseParams = new
04  PeriodicParameters(
05                                      null, // start time
06                                      null, // end tTime
07                                      TimeObjects._10_ms, // period
08                                      TimeObjects._4_ms, // cost
09                                      TimeObjects._10_ms);
10  //deadline
11     // Other class attributes...
12    public void mainTask() {
13      while (true) {
14        // control actions
15        // ...
16        waitForNextPeriod();
17      }
18    }
19    public void exceptionTask() {
20      // exception handling code is inserted here
      }}
```

Figure 6. SteeringWheelControl class source code

After having generated the source code using the RTSJ-based API, this code is then compiled using a standard Java compiler and the synthesis of the embedded real-time system is performed. The resulting class files are used as input to the SASHIMI tool[15] and analyzed in order to generate VHDL files that will customize the FemtoJava processor. It is important to highlight that the size of the generated FemtoJava control unit is proportional to the number of distinct Java opcodes used by the application software. In the final step, the resulting binary is loaded into the FPGA. In the synthesis

process of the steer-by-wire application, the generated software occupies 410 bytes of RAM to store all objects in steering wheel node and 426 bytes on each wheel node. The code size needed by application objects for each node type is, respectively, 3259 and 3374 bytes. This case study was simulated using the Cycle Accurate COnfigurable Power Simulator (CACO-PS)[17] considering a FemtoJava processor running at 10 Mhz. The simulation results are showed in table 1 (timing information in milliseconds).

Table 1. Steer-By-Wire simulation data

Task	Predefined Activation Instant	Real Activation Instant	Delay	Task WCET
Steering Wheel Node				
SteeringWheelAngleDriver	0	0,2316	0,2316	0,4830
	5	5,2316	0,2316	0,4830
SteeringModeDrive	1	1,2316	0,2326	0,4805
SteeringWheelController	6	6,2338	0,2338	2,3789
Wheel Nodes				
WheelAngleDriver	0	0,2316	0,2316	0,4830
	7	7,2316	0,2316	0,4830
WheelSpeedDriver	1	1,2316	0,2316	0,4830
	8	8,2338	0,2338	0,4830
WheelController	3	3,2316	0,2316	2,9017

The simulation results present a satisfactory behavior of the steer-by-wire system over the FemtoJava processor without deadline misses that can be seen on real activations instants and WCET of all tasks. The response times are consistent in relation to temporal specification contained in the dispatcher data structure of the communication nodes.

5. CONCLUSIONS

In this paper a case study showing how the object-oriented methodology proposed within the SEEP project can be mapped to a time triggered architecture is presented. The paper shows the integration of the design phases, from a requirements specification using the RT-UML profile to an implementation that makes use of a RTSJ-based API and a customizable real-time Java processor. The obtained results indicate that the use of object orientated paradigm and the RT-UML to design time-triggered systems can be an interesting alternative to conventional project methods. Timing requirements can be extracted from RT-UML model, through the stereotypes and tagged values that decorate diagram elements and provide the required information to build the time-triggered application dispatch table as well as

to generate the application code. The results also indicate that a RTSJ-based programming language can be used in time-triggered systems.

REFERENCES

1. X-By-Wire Team, "X-By-Wire Safety Related Fault Tolerant Systems in Vehicles," Project No. BE 95/1329, Tech. Report, 1998.
2. OBJECT MANAGEMENT GROUP, "UML Profile for Schedulability, Performance, and Time Specification," http://www.omg.org/cgibin/doc?ptc/02-03-02, March 2002.
3. OSEK Group, "Time-Triggered Operating System Specification 1.0. OSEKtimeOS 1.0," OSEK/VDX group, July 2001.
4. Johannessen, P, "Project SIRIUS 2001: A University Drive-by- Wire Project ," Dept. of Computer Eng., Chalmers University of Technology, Goteborg, Sweden, , Tech. Report 01-14, 2001.
5. Kopetz H., Bauer Gunther, "The Time-Triggered Architecture," Proceedings of the IEEE Special Issue on Modeling and Design of Embedded Software, October 2002.
6. TTTech GROUP, "TTPOS Time-Triggered Operating System with TTP Support," White Paper http://www.tttech.com/technology/docs/general/TTTech-TTPOS.pdf, 2004.
7. Jong, G., "A UML-Based Design Methodology for Real-Time and Embedded Systems," Proc. of Design, Automation and Test in Europe Conference and Exhibition, France, March 2002.
8. Axelsson, Jakob, "Real-World Modeling in UML," Proceedings of International Conference on Software and Systems Engineering and their Applications, France, December 2000.
9. Chen, Rong; Sgroi, Marco; Lavango, Luciano; Martin, Grant; Sangiovanni-Vincentelli, Alberto; Rabaey, Jan, "UML and Platformbased Design. Norwell," Kluwer 2003. p. 107-126.
10. Nebel, Wolfgang; Oppenheimer, Frank; Schumacher, Guido, "Object-Oriented Specification and Design of Embedded Hard Real-Time Systems," Proceedings of Internation Conference on Chip Design, Laxenburg: IFIP, 2000. p. 285-296.
11. Becker, Leandro B.; Holtz, Rudy; Pereira, Carlos E, "On Mapping RT-UML Specifications to RT-Java API: Bridging the Gap," Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.
12. Booch, Grandy; Rumbaugh, James; JacobsonN, Ivar. The Unified Modeling Language User Guide. Reading, Massachusetts: Addison-Wesley, 1999.
13. Wehrmeister, Marco A.; Becker, Leandro B.; Wagner, Flavio R.; Pereira, Carlos E. "On Object-Oriented Platform-based Design Process for Embedded Real-Time Systems". Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005.
14. Wehrmeister, Marco A.; Becker, Leandro B.; Pereira, Carlos E. "Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API". *Proceedings of Workshop On Java Technologies* For Real-Time And Embedded Systems, 2004.
15. Ito, S.; Carro, L; Jacobi, R.P. "Making Java Work for Microcontroller Applications". IEEE Design & Test of Computers, vol. 18, n. 5, 2001, p. 100-110
16. Mattos, J. C. B.; Brisolara, L.; Hentschke, R.; Carro, L.; Wagner, F. "Design Space Exploration with Automatic Generation of IP-based Embedded Software". Proceedings of the IFIP Working Conference on Distributed and Parallel Embedded Systems, 2004.
17. Beck Filho, A.C.; Mattos, J.; Wagner, F.R.; Carro, L. "CACO-PS: A General-Purpose Cycle-Accurate Configurable Power Simulator". Proceedings of Symposium on Integrated Circuits and Systems Design, 2003.