An Object-Oriented Platform-based Design Process for Embedded Real-Time Systems

Marco A. Wehrmeister¹, Leandro B. Becker³, Flávio R. Wagner¹, Carlos E. Pereira²

¹Computer Science Institute, Federal University of Rio Grande do Sul (UFRGS), Brazil
²Electrical Engineering Department, Federal University of Rio Grande do Sul (UFRGS), Brazil
³Automation and Control Systems Department, Federal University of Santa Catarina (UFSC), Brazil

ABSTRACT

The growing complexity of today’s embedded real-time systems demands new methods and tools in order to manage the problems of design, analysis, integration and validation of complex systems. This paper describes an object-oriented platform-based design process for real-time embedded systems. The proposed approach promotes a smooth transition from high-level UML specification to implementation, which is composed by hardware and software components. The transition from higher to lower abstraction levels is facilitated by the use of an OO real-time API, whose underlying facilities can be optimized according to the application needs and selected platform. An integrated toolset is used to support the intermediate steps of the design process. In order to illustrate the proposed approach and related toolset the design of an embedded real-time automation system for an “intelligent” wheelchair is presented.

Keywords: real-time object computing, RT-UML, RT-Java, code generation.

1 Introduction

The design of real-time embedded systems is a complex task because these systems include hardware and software components that must be normally highly optimized for the application. In the past, hardware configuration dominated the implementation of these systems. Nowadays, a large part of embedded real-time systems are implemented as mixed configurations (hardware + software), where the software components have become key for a successful system [2]. The dominance of software in real-time embedded systems design caused the interest on methodologies with widely accepted notations in the software community, such as the Unified Modeling Language (UML) [5]. Approaches that use UML for embedded systems design are presented in [8], [1], [6] and [10]. Nevertheless, there are still gaps that should be tackled by new methodologies.

This paper introduces the Object-Oriented Platform-Based Design Process for Embedded Real-Time Systems, or simply SEEP (project acronym in Portuguese), which proposes a new method to design real-time embedded systems. This project deals with 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 FPGA-based architectural platforms.

The SEEP approach aims at a smooth transition from object-oriented models specified with RT-UML [11] to implementation. The transition from higher to lower abstraction levels is facilitated by the use of a real-time Java API [12] 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 [7].

The mapping from the RT-UML specs into the RTSJ-based API is based on the work presented in [3], which presents an approach that defines a clear link between real-time constraints and the programming entities that provide their implementation. The main idea of this approach is to enhance the traceability as well as the readability of timing constraints from the modeled requirements to its implementation.

The remainder of this paper is organized as follows. Section 2 describes the design methodology proposed in the SEEP project. Section 3 presents the toolset developed to support the proposed design methodology and respective embedded system generation. Section 4 shows the use of the design methodology for the development of a wheelchair control and automation system. Concluding remarks and future work directions are presented in Section 5.

2 The SEEP Methodology

The SEEP methodology proposes a complete and integrated approach for SoC (system-on-chip) design and ASIP (application-specific instruction set processor) generation. This methodology is defined to guide the system integrator and the core provider towards the development of embedded applications within a reduced design time. Therefore the reuse concept is assumed, and each design step aims at facilitating the development of reusable components and the rapid, but cost effective, design space exploration. An overview of the proposed design methodology is depicted in Figure 1.

According to the design steps in this methodology, the design of an embedded SoC starts with the definition and validation of a high-level, pure functional model, which is not influenced by architectural choices and does not consider how design requirements (power, performance) may be fulfilled.

The second step, namely system exploration, allows the evaluation of algorithms that can be used to implement parts or the whole application. As input, this process uses a library of pre-implemented algorithms and models, as well as estimation data regarding their implementation on available platforms. Thus, the algorithms can be analyzed, considering the available estimation data, in terms of efficiency, power consumption, memory occupa-
tion, and area costs. Test issues are also taken into account during this phase. As a result, the chosen algorithms are coded in some executable language, such as SystemC, Java, C++, etc. With this description, the architectural exploration can take place.

As previously mentioned, the SEEP approach promotes a smooth transition from object-oriented models specified with RT-UML to the final embedded system. Such transition is facilitated by the use of an object-oriented real-time API, which is currently implemented for the RT-Java programming language (it is based on the RTSJ standard [4]).

3 Toolset for Embedded System Generation

As mentioned in the previous section, an executable description is used as input for the architectural exploration, where alternative hardware and software solutions that fulfill the system requirements should be considered and evaluated. After compiling all the available information, the final system generation is performed, resulting in a micro-architecture and a software description for a dedicated system.

In the SEEP methodology, we are currently limited to accept Java source code for representing the architecture-independent description. Using the SASHIMI environment [7], both a VHDL description for a dedicated Java processor and the respective program memory code (application code) are generated. This CAD environment automatically synthesizes a dedicated microprocessor (named FemtoJava) for a target application, using only a subset of instructions required by the application. This Java processor implements an execution engine for Java in hardware through a stack machine that is compatible with the Java Virtual Machine (JVM) specification. A customized control unit for the FemtoJava processor is generated, supporting only the opcodes used by the application. The size of its control unit is thus directly proportional to the number of different opcodes utilized by the application software.

In order to more clearly express the timing and other constraints in the source code of the real-time embedded application, an API based on the Real-Time Specification for Java (RTSJ) was developed. 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. Moreover, it supports the expression of the following elements: absolute and relative time values, 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. A brief description of the available classes follows (a more detailed description can be found in [12]):

RealtimeThread: represents a real-time task in the embedded system. It 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 and deadline. The periodicParameters subclass has attributes like the start and end time and also the execution period. The SporadicParameters class (a subclass from AperiodicParameters) has attributes like the minimum interval between two occurrences.

SchedulingParameters: base class for all scheduling parameters used by the Scheduler object. The task priority is repre-
Methods that have to be implemented in the derived subclasses: Thread class. In the proposed implementation, it has two abstract reasons. An example of such difference appears in the Realtime-constraints in the FemtoJava architecture and also for clarity their implementation when compared to the RTSJ. This is due to the instant of time. The subclass RelativeTime represents a time relative to other time instant that is given as parameter. Clock: a global clock reference. It returns an AbsoluteTime object that represents the current system date and time.

Timer: abstract class that represents a system timer. The derived class OneShotTimer represents a single timer occurrence, while the derived class PeriodicTimer represents a periodic one. Some of the proposed API classes have slight differences in their implementation when compared to the RTSJ. This is due to constraints in the FemtoJava architecture and also for clarity reasons. An example of such difference 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 when deadline misses occur. This simplification allows the exception handling code to be executed in the same object context of main thread code, thus reducing the memory usage when compared to the RTSJ approach. Another class that has a different implementation is the Timer class. It has an abstract method named runTimer() that must be implemented in the subclass and represents the code executed when the timer expires. This method appears both in the OneShotTimer and PeriodicTimer classes.

4 Case Study

To illustrate the use of the SEEP methodology, a case study is presented. It consists in the design of a real-time embedded automation and control system for an “intelligent” wheelchair to support people with special needs. The main function of the system is the movement control. Hard-real time requirements must be accomplished for safety reasons.

The design process starts with the construction of a high-level object modeling using RT-UML. Diagrams used in the model are: Use Cases, Collaboration, and Class Diagrams. Especially the last two diagrams are decorated with the stereotypes and tag-values coming from RT-UML. Figure 2 presents the object collaboration diagram for the ‘movement actuation’. It 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 RT-UML stereotypes that are relevant in the context of the movement actuation. Period and deadline information derives from tags specified in the collaboration diagram in Figure 2. They are represented, respectively, by the «SAScheRes» and «SAscheRes» stereotypes of RT-UML. The code contains two important methods: mainTask() and exceptionTask(). The former represents the task body, i.e., the code executed when the task is activated. This is a periodic task, in which periodic activation is implemented as a loop whose execution frequency is controlled by calling the waitForNextPeriod() method. This method uses the task release parameters for interacting with the scheduler and controlling the correct execution of the method. The exceptionTask() method represents the exception handling code that is triggered in case of a deadline miss, i.e., if the mainTask() method does not finish on time.

After the source code is generated using the RTSJ-based API, it 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 and analyzed in order to generate VHDL files that will customize the FemtoJava processor. It is important to highlight that the generated FemtoJava’s control unit is proportional to the number of different Java opcodes used by the application software. In the final step, the resulting binary is loaded into the FPGA.
of different design decisions in the proposed methodology. It is important to highlight that the evaluation of alternative implementations of the wheelchair mapping process is not unique and that other APIs offer different alternatives to implement given timing requirements. Nevertheless, the developed API is highly optimized for the Femto-Java platform.

The paper showed the integration of the design phases proposed within the SEEP design process. It described the requirements modeling stage using RT-UML as well as the further design stages, including the system implementation using an API and the available Java platform. It is important to highlight that the mapping process is not unique and that other APIs offer different alternatives to implement given timing requirements. Nevertheless, the developed API is highly optimized for the Femto-Java platform.

As future work, the authors intend to integrate the proposed methodology into a CASE tool, thus increasing the automation level for embedded systems design. Other issue to be tackled is the evaluation of alternative implementations of the wheelchair control using different platforms, in order to compare the impact of different design decisions in the proposed methodology.

6 Acknowledgments

Thanks are given to the Brazilian funding agency CNPq, which is the project sponsor. The authors also thank the other SEEP-project researchers for the valuable discussions.

References