LICENTIATE THESIS

# Component-Based Software Design of Embedded Real-Time Systems

Jimmie Wiklander

LULEÅ UNIVERSITY OF TECHNOLOGY

# Component-Based Software Design of Embedded Real-Time Systems

**Jimmie Wiklander**

Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå, Sweden

**Supervisors:**

Professor Per Lindgren and Assoc. Professor Johan E. Carlson

Printed by Universitetstryckeriet, Luleå 2009

ISSN: 1402-1757
ISBN 978-91-7439-021-6

Luleå 2009

www.ltu.se

# Abstract

Embedded systems have become commonplace in today's society and their complexity and number of functions are steadily increasing. This can be attributed to the unceasing advances in the microprocessor technology and the continuous delivery of more powerful and power-efficient microprocessors, which, in turn, allows more elaborate software implementations. Consequently, there is a strong interest in finding methods and tools that support flexible and efficient development of embedded software. Since these qualities are typically attributed to component-based design it makes sense to develop new design techniques targeting embedded systems based on components. This thesis aims to adapt the traditional component-based design approach for development of embedded real-time software.

Component-based design relies on the existence of consistent and coherent models of individual components that can be composed to model the whole system. However, it can be argued that the special characteristics of embedded systems make such modeling challenging. One reason is that embedded systems typically exhibit a strong integration between hardware and software, which leads to a need for a common design space, or at least the possibility to create consistent models of both hardware and software components of an embedded system. Another reason is that the majority of embedded systems can be viewed as real-time systems and therefore it is necessary to express timing requirements alongside functional properties in the model.

In order to overcome these difficulties, we adopt a reactive perspective, in which the functionality of both hardware and software is described in terms of *time-constrained reactions* of *reactive objects*. This enables capturing the complete functionality of the system (hardware and software) along with timing requirements in a single model.

The reactive view lies behind the modeling framework for embedded real-time systems and the component-based software design methodology presented in this thesis. The methodology allows both functional and timing properties of a system model to be preserved during implementation process by means of a seamless transition between a model and an implementation, whereas the modeling framework enables the developer to offer platform-independent correctness for real-time systems, provided that the software can be scheduled on a given hardware platform. Further, this thesis includes a case study, in which the methodology is used for designing a real-life system. The case study demonstrates the potential of the methodology to bring the benefits of classical component-based design to the realm of embedded systems.

# Contents

# Acknowledgments

There are many who deserve my sincere thanks. First of all, I would like to thank my supervisor Per Lindgren for his guidance, support, and patience. Many thanks also to my assistant supervisor Johan Carlson.

I would like to give special thanks to my colleague and friend Andrey Kruglyak, for his guidance and support. Further, I would like to thank Jerry Lindblom, for gladly sharing his expertise on electronics and embedded systems design, and Anders Högström for his encouragement and support.

The work in this thesis was funded by The Knowledge Foundation in Sweden under a research grant for the research school SAVE-IT, the EU Interreg III A North Programme (grant no. 304-13723-2005), the EU Interreg IV A North Programme (grant no. 304-15591-08), and the ESIS project (European Regional Development Fund, grant no. 41732).

Finally, I would like to express my gratitude to all of you people who have encouraged me along this path; friends, colleagues and family.

*Jimmie Wiklander*
*Luleå, October 2009*

# Part I

# Chapter 1

# Introduction

Embedded systems have become commonplace in today's society and their complexity and number of functions are steadily increasing. This can be attributed to the unceasing advances in the microprocessor technology and the continuous delivery of more powerful and power-efficient microprocessors, which, in turn, allows more elaborate software implementations. This increase in software complexity together with the special characteristics of embedded systems calls for new approaches to their specification, design, and implementation.

## 1.1 Component-Based Design

Component-based design is an attractive software design approach that facilitates re-use, separate development of components, and can improve the overall maintainability and robustness of the system. The underlying concept is to construct systems by composing ready-made parts (or components). These components are generally defined in accordance with a *component model*. Typically, a component model defines component types, their interaction scheme, and constructs used for component composition. A model of a system is produced by choosing and combining ready-made components in a process that is driven by the system specification. Once the system model has been designed, its functionality must be verified against the specification, either formally or by simulation of the model. In case the model fails to meet the requirements, the design must be revised. Otherwise, the model can proceed to the synthesis stage in which it is translated into executable code. Typically, a run-time system is needed in order to realize the behavior expressed by the executable.

## 1.2 Modeling of Embedded Systems

As described above, the component-based approach relies on the existence of consistent and coherent models of individual components that can be composed to model the whole

system. Modeling embedded systems, however, is not straightforward. In fact, it can be argued that the special characteristics of embedded systems turn the development of such systems into an challenging task [1].

To begin with, unlike most general-purpose computing systems, embedded systems often perform computations subject to various constraints, such as processor speed, amount of memory, and power consumption. This makes it virtually impossible to bring about the powerful separation between computation (software) and physicality (platform and environment), which is evident in general-purpose computing systems. Instead, embedded systems often manifest a tight integration between functionality implemented in software and functionality of hardware parts. In many embedded systems, hardware components cannot be viewed as part of the environment since the software has to be developed "around" the available hardware resources, relying on their timing and other properties. Moreover, the timing requirements are often of special importance for embedded systems, especially for safety critical systems. In fact, the majority of embedded systems can be viewed as real-time systems, i.e. systems in which correctness of system behavior (for hard real-time systems) or quality of service (for soft real-time systems) relies on the time when results are delivered to the environment as well as on the computed values as such.

## 1.3 Research Problems

Currently, there is a strong interest in finding methods and tools that support flexible and efficient development of embedded software. Since these qualities are typically attributed to component-based design, it is compelling to develop new design techniques for embedded systems, based on components. This thesis aims to adapt the traditional component-based design approach for development of embedded real-time software.

In order to approach this problem it is important to identify the factors that make it difficult to incorporate components as part of the design process. First of all, modeling is a natural ingredient in most of the embedded systems design approaches. But although it is clear that embedded systems exhibit a strong integration between hardware and software, there have been few attempts to unify them in the same model. What is needed is a common design space, or at least the possibility to create consistent models of both hardware and software components of an embedded system.

The behavior of an embedded system can be naturally described in terms of reactions to events that originate either from the system's environment or internally within the system. This view is manifested through the notion of *reactive objects* that can be used to describe the functionality of software as well as hardware [2]. It is therefore attractive to envision a component-based design approach based on reactive objects as stated in the following research question:

*Q1: Is it possible to define a component model based on reactive objects?*

In addition, for real-time systems, the timing requirements are of utmost importance. The standard procedure is to treat time separately and analyze temporal properties when the system is ready. An alternative approach, which is advocated in this thesis, is

based on combining both functional and timing requirements in the same model. This gives rise to the following research question:

> *Q2: Can both functional and timing requirements be consistently modeled in a component-based modeling framework?*

As mentioned earlier, component-based design facilitates re-use of components. This is not easy to achieve for embedded systems considering the fact that embedded systems often have an inherent coupling between functionality implemented in software and functionality of hardware parts. Today, the functionality of hardware parts are typically combined into a *hardware platform*, a platform designed for a specific class of applications as a way to limit the development cost, facilitate re-use etc. This makes replacing certain hardware functionality difficult, not only because of the need to redesign the platform, but also because it often necessitates rewriting the software, which in the end may affect the overall system functionality. These issues can be summarized in the following research question:

> *Q3: How can we use the component-based design to enhance reusability of individual components as well as hardware platforms?*

The reasoning above has its main focus on the modeling aspects of the design process, which should not come as a surprise. The proposed modeling framework should support modeling of components that may contain hardware, software, or a combination of the two. Moreover, the component model should also facilitate describing timing requirements. However, in order to make use of such a modeling framework it must be accompanied by a software design methodology that describes how the models should be defined, how to translate the models into executable code, and also how to perform verification. That is why this thesis also involves devising a clear methodology for development of embedded real-time systems based on the component-based approach.

## 1.4 Research Approach

Adaptation of the component-based approach to design embedded real-time software can be separated into a number of different activities. The first activity is to investigate how to utilize reactive objects for modeling of components. The second activity is to formulate a modeling framework supporting component-based design based on the results from the first activity. The modeling framework and the design process must be compatible with each other, and therefore the third activity involves using different examples to study different ways to form a system model using the modeling framework. The results will be used to define a component-based methodology that is compatible with the modeling approach. The final activity is to test the methodology in real-life design examples for evaluation and refinement purposes.

## 1.5 Thesis Outline

This thesis consists of two parts. The first part starts with an introduction to component-based design alongside a discussion of modeling embedded systems (Chapter 1). The introduction also presents the research problems and describes the research approach. Chapter 2 introduces reactive objects and defines the notion of a *reactive component*. Chapter 3 gives an overview of the *SensorBand* project which includes a description of a personal alarm system developed in close cooperation with the industry. Summaries of publications related to the project are also included in the chapter. Chapter 4 provides summaries of the papers included in the second part of the thesis and Chapter 5 presents related work. Conclusions and future work are found in Chapter 6, the last chapter of the first part. The proposed component-based design approach for embedded real-time systems is presented in detail in the papers included in the second part of the thesis.

# Chapter 2

# The Reactive Approach

This chapter introduces the concept of reactive objects and provides a definition of reactive components. Together, they provide a foundation for the modeling framework for embedded real-time systems and the component-based software design methodology presented in the second part of this thesis (Paper A and Paper B). A summary of Paper A and Paper B can be found in Chapter 5.

## 2.1 Reactive Objects

The interaction between a system and its environment, as well as between components of the system can be described in terms of *events*. Following the reactive view, this allows the functionality of the system to be described in terms of *reactions* to such events; reactions that may produce additional events on its own, creating a chain of reactions. Stating that the events are discrete, i.e. stating that each event must relate to a specific point in time, makes it possible to impose time constraints on the reactions. The simplest way to specify such constraints is by defining the earliest and the latest reaction time (*baseline* and *deadline*) relative to the time of the input event triggering the reaction. The time window between the reaction baseline and its deadline is called a *permissible execution window* for this reaction (see Fig. 1 on page 25), and it is denoted as after ***after*** $t_{after}$ ***before*** $t_{before}$ *doSmth*. Here $t_{after}$ is the baseline offset (period of time between the triggering event and the baseline), $t_{before}$ is the period of time between the baseline and the deadline, and *doSmth* is the invoked method of the object *obj*. A reaction with a permissible execution window defined for it will be called a *time-constrained reaction*.

In general, functionality of a program is often organized using objects. Uniting objects with time-constrained reactions (by assigning each method a permissible execution window and by coupling each method to a certain external or internal event) and using objects for modeling hardware as well as software makes it possible to provide a consistent modeling for the whole system [2].

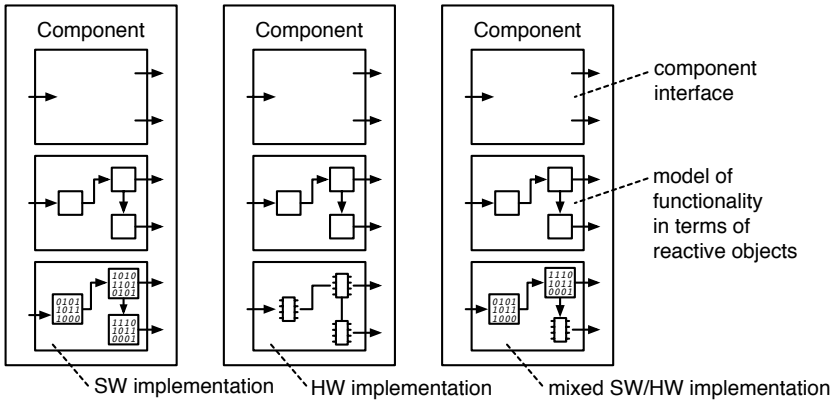



*Figure 2.1: Possible implementation schemes for a component.*

## 2.2 Reactive Components

Many of the terms and concepts that are used in the component-based software engineering domain are relatively general and are not always fully agreed on [3]. The key terms *component* and *interface* are based on software abstractions that might not be the most applicable when it comes to design of embedded software. It can be argued that in order to realize component-based design of embedded systems, it is necessary to take on a component definition adapted for embedded systems.

There are many different definitions of components. A hardware component can be seen as a hardware part with a clear interface (way of interacting with it) and functionality, and a software component can be seen as a software unit with an interface (collection of operations or services) and an implementation. In the component-based software engineering domain the Szyperski definition [4] is widely accepted. From this definition it follows that a software component is a unit of composition with contractually specified interfaces that can be deployed independently. This very general definition is not practical for embedded systems whose functionality strongly depends on the underlying hardware. Hence, another definition of component is needed, one that includes a model that facilitates describing the functionality of both hardware and software. The following component definition is denoted *reactive component* and it is based on reactive objects.

A reactive component is defined by its:

- **interface** where interaction with its environment (system environment and/or other components) is expressed as discrete events;
- **model** expressing the functionality of the component in terms of reactive objects;
- **implementation** of the model in software (code in a programming language), hardware (one or more hardware parts), or both.

The reactive component encapsulates a part of system state and/or hardware resources and has a clearly defined interface and functionality. Its implementation may consist of software, hardware, or a combination of software/hardware as depicted in Fig. 2.1.

# Chapter 3

# The *SensorBand* Project

The research presented in this thesis targets development of embedded systems, an area that has gained a lot of interest from the academia, but even more from the industrial sector. It is therefore important to connect the research in this thesis to the reality, an opportunity provided by my participation in the *SensorBand* project. The project involved development of an alarm device for elderly people. The device was developed in close cooperation with the industry and the problems targeted in this thesis were observed during the development process. This system has been used later on as a test bench for ideas arising during the research. The case study presented in Paper B describes the design, implementation and verification of software for the alarm device, all performed in accordance with the software design methodology presented in this thesis (Paper A). The system is also used in Paper A, as a demonstration of the design process.

## 3.1 Project Overview

The *SensorBand* project was launched in 2006 and ended in 2007. The aim of the project was reliable identification of falls of elderly people. One of the more concrete goals was to develop a *Personal Alarm Device* (PAD); a wearable system, which can detect if the person who wears it falls and sends an alarm if such an incident takes place. The system should also contain an alarm button and send an assistance alarm when the button is pressed.

The first task was to construct a fall detector prototype, which was needed to validate the concept of fall detection using a triaxial accelerometer worn at the waist. The prototype was used in studies where acceleration data from simulated falls as well as activities of daily living (ADL) were collected and different fall detection algorithms were analyzed using the collected data. The results can be found in Section 3.2 below, which provides a summary of selected publications relating to the conducted studies. Based on the results of the studies, one of the algorithms was chosen for implementation. The outcome of the development was a PAD fulfilling the requirements defined above.

The specification, design, and implementation of the PAD manifested many of the

issues considered in this thesis and has therefore been a valuable source of understanding. However, the development of the PAD was not carried out in accordance with the approach suggested in this thesis since, at that point, the approach was not yet formalized.

Subsequent to the SensorBand project, the PAD software has been revised in accordance with the approach presented in this thesis. The new PAD design is presented in Paper A as an illustrative example of the approach. Paper B presents a case study describing the design, implementation and verification of the PAD.

## 3.2 Related Publications

This section provides a summary of selected publications connected to the *SensorBand* project.

### SensorBand fall detector prototype : validation through data collection and analysis

**Authors:** Maarit Kangas, Jimmie Wiklander, Irene Vikman, Lars Nyberg, Per Lindgren, and Timo Jämsä
**Publication:** *Proceedings of the 2nd International Symposium on Medical Information and Communications Technology (ISMICT 2007)*, Oulu, Finland, December 2007
**Summary:** This paper presents the *SensorBand* fall detector prototype which is used to validate the concept of fall detection using a triaxial accelerometer worn at the waist. The prototype is used to collect acceleration data from intentional falls. Different fall detection algorithms are analyzed using the collected data as input. The results of this pilot test shows that the suggested concept and the implemented hardware are efficient for fall detection. The tested algorithms detected maximally 100% of the falls.
**Contribution:** The SensorBand fall detector prototype was designed and implemented by Jimmie Wiklander. He was also responsible for the technical set-up used for collecting acceleration data during the tests.

### Sensitivity and specificity of fall detection in people aged 40 years and over

**Authors:** Maarit Kangas, Irene Vikman, Jimmie Wiklander, Per Lindgren, Lars Nyberg, and Timo Jämsä
**Publication:** *Gait & Posture*, Volume 29, Issue 4, 2009
**Summary:** The aim of the study presented in this paper is to validate the data collection of a fall detector prototype and to define the sensitivity and specificity of different fall detection algorithms with simulated falls from 20 middle-aged (40-65 years old) test subjects. Activities of daily living (ADL) performed by the middle-aged subjects, and also by 21 older people (aged 58-98 years) from a residential care unit, were used as a reference. The results shows that the hardware platform and algorithms used can discriminate various types of falls from ADL with a sensitivity of 97.5% and a specificity of 100%. This suggests that the present concept provides an effective method for automatic

fall detection.

**Contribution:** The fall detector prototype described in this work is identical to the one used in the pilot study described above. As in the pilot study, Jimmie Wiklander was responsible for the technical set-up used for collecting acceleration data during the tests.

# Chapter 4

# Summary of appended Papers

This chapter introduces the papers that are included in the second part of this thesis. Paper A presents a modeling framework which enables component-based design of embedded real-time systems together with a detailed software design methodology exploiting the framework. Paper B describes the implementation and verification of a real-life system that has been developed in accordance with the methodology presented in Paper A.

## 4.1 Paper A - Enabling Component-Based Design for Embedded Real-Time Software

**Authors:** Jimmie Wiklander, Jens Eliasson, Andrey Kruglyak, Per Lindgren, and Johan Nordlander
**Publication:** In press (Journal of Computers, 2010).
**Summary:** This paper presents a modeling framework based on the notions of reactive objects and time-constrained reactions, which enables component-based design of embedded real-time systems. Within this framework, functionality of both hardware and software components is defined in terms of reactions to discrete events, and timing requirements are specified for each reaction relative to the event that triggered it. Each component is modeled using reactive objects, which for the software part of the system form an executable model. The paper also presents a detailed software design methodology for real-time embedded systems exploiting the presented modeling framework. It can be used both in the case when software is developed alongside a hardware platform (the latter being assembled from existing hardware parts) and in the case when such a platform is given from the start. In both cases, a platform is instantiated using some implementation of hardware and/or software components depending on performance, power consumption, and other non-functional requirements. The approach also allows to clearly define the notion of a *resource platform* as a combination of hardware and software resources. A resource platform can be designed to serve as a basis for a whole range of

related applications, decreasing the overall development cost and time to market.
**Contribution:** The concepts of reactive objects and time-constrained reactions used in the thesis follows the work of Johan Nordlander[2, 5]. The development of the technique for modeling components (software and hardware) is a result of joint research by Andrey Kruglyak and Jimmie Wiklander. Development of a concrete software design methodology as well as the notion of a *resource platform* is due to Jimmie Wiklander.

## 4.2 Paper B - Personal Alarm Device: A Case Study in Component-Based Design of Embedded Real-Time Software

**Authors:** Jimmie Wiklander, Andrey Kruglyak, Per Lindgren, and Johan Nordlander
**Publication:** To be submitted.
**Summary:** This paper describes a real-life system developed using the methodology (presented in Paper A) and discusses its advantages. The system is a personal alarm device that should be worn at the waist of a person and that should detect his or her (accidental) fall and send an alarm signal. The underlying fall detection algorithm is based on measuring two types of acceleration, static (gravity and tilt) and dynamic. The system was implemented using the programming language Timber. The implementation was verified using a Simulink-based simulator for Timber. During simulation, the software operated according to its specification. The simulation also demonstrated that, even though calculation of acceleration was simplified to allow for an efficient execution on a resource-constrained platform, the fall detection is satisfactory. This makes it possible to utilize a lightweight microcontroller, which in turn implies a lower power consumption, a smaller physical size of the device, and a lower price. These qualities are very important for portable systems and are crucial for the system's applicability in real life. The case study demonstrates the advantages of the proposed software design methodology, including the fact that functional and timing properties of a system model can be preserved during implementation process by means of a seamless transition between a model and an implementation.
**Contribution:** The design and implementation of the personal alarm device should be accounted to Jimmie Wiklander. Jimmie Wiklander has developed the software design methodology used for designing the system.

# Chapter 5

# Related Work

There is a strong interest for component-based design of embedded systems and as a result, various techniques and component models have been proposed. Here follows a presentation of some techniques that are relevant for comparison against the approach taken in this thesis.

The Rubus component model [6] shares many of the goals and implementation approaches with this work. It provides constructs for encapsulating software functions (called software circuits) and can be used to express interaction between them in single- and multi-node systems in terms of control flow as well as data flow. However, the software model has to be translated into executable threads, whose execution has to be controlled by a specialized run-time system such as Rubus-RTOS [7].

Another approach is Time-Triggered Architecture [8] which requires that each component is fully specified, including in the time domain, and can thus be verified separately from the rest of the system. Originally targeting distributed systems, this approach can easily be applied to componentization of a single-node system provided that components either do not share any resources, or utilize them according to a statically pre-defined schedule (including a shared CPU). This approach is very robust and can be used for safety-critical systems, but robustness comes at the cost of flexibility of the design and leads in most cases to a below-optimal utilization of resources.

Apart from well-developed approaches with well-defined semantics such as Rubus and TTA, there exist a number of other modeling frameworks and design tools that can be used for component-based development of real-time systems. This encompasses real-time synchronous languages (Esterel, SCADE, Lustre, etc. [9]); RT-CORBA [10], which is a enhancement for the middleware architecture CORBA for real-time and embedded systems; time-triggered languages such as Giotto [11]; the Koala component technology [12], which uses a thread-sharing technique to limit resource usage; the Ptolemy framework for assembly of concurrent components [13], which is particularly suitable for modeling distributed systems; and tools such as Rhapsody [14], ARTISAN Studio [15], and Rose-RT [16].

We should also mention work on specification of real-time systems, such as RT-

UML [17] and MARTE [18]. In contrast to this work, these approaches offer numerous, often highly specialized ways to define timing properties, which are difficult to preserve (with consistent semantics) throughout the design process.

# Chapter 6

# Conclusions and Future Work

This chapter summarizes the research contributions and discusses their relation to the research questions stated in Section 1.3. Moreover, future work is discussed.

## 6.1 Conclusions

The main results of this thesis are a component-based modeling framework for embedded real-time systems and a concrete software design methodology utilizing the framework. The methodology can be used both in the case when software is developed alongside a hardware platform (the latter being assembled from existing hardware parts) and in the case when such a platform is given from the start. The modeling framework enables the developer to offer platform-independent correctness guarantees for hard real-time systems (and quality of service guarantees for soft real-time systems), provided that the software can be scheduled on a given hardware platform so that all reaction deadlines are met.

Let us now discuss the results of this thesis with respect to the research questions stated in Section 1.3. An important aspect of this work is the notion of a *reactive component*. A reactive component is defined by its interface, with interaction expressed as discrete events; its model, expressing the functionality of the component in terms of reactive objects; and its implementation, representing the functionality expressed by the model. The reactive component supports re-use since it is possible to construct new systems by combining such components. The reactive component provides an answer to question *Q1*. Moreover, inclusion of timing requirements in a functional model in the form of time-constrained reactions allows us to specify, reason about, and verify real-time properties of embedded systems. Uniting object-oriented approach with time-constrained reactions and using reactive objects for modeling of both software and hardware enables modeling of both functionality and timing requirements within a single modeling framework and that answers question *Q2*. Apart from addressing the issues of software complexity, interdependency between software and hardware, and complying with the timing requirements, our approach also allows to clearly define the notion of a *resource platform* as a combination of hardware and software resources. A resource

platform can be designed to serve as a basis for a whole range of related applications, decreasing the overall development cost and time to market. Reusability of individual components and hardware platforms is naturally supported by the use of consistent and systematic models, and the notion of a resource platform, respectively. This answers question *Q3*.

Let us now turn to the aim of this thesis, that is, *to adapt the traditional component-based design approach for development of embedded real-time software*. The presented results are a first step towards reaching this goal. Moreover, the case study (presented in Paper B) demonstrates the potential of the presented methodology to bring the benefits of component-based design to the realm of embedded systems. However, more work is needed in order to fully reach this goal.

## 6.2 Future Work

When designing software for embedded systems it is of utmost importance to comply with not only functional, but also non-functional requirements. Temporal requirements are the most prominent non-functional requirements when it comes to real-time systems and these have successfully been integrated into the modeling framework and software design methodology presented in this thesis. However, in order to further improve the design approach, the role of other requirements such as power consumption needs to be investigated. Another point of interest is the present lack of design tools whose existence would support the designer during the design process. Development of such tools is therefore advocated.

# References


[1] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *Proc. of the 14th Int. Symp. on Formal Methods (FM)*, ser. Lecture Notes in Computer Science, 2006, pp. 1–15.

[2] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black, "Reactive objects," in *Fifth IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2002, pp. 155–158.

[3] I. Crnkovic, B. Hnich, T. Jonsson, and Z. Kiziltan, "Specification, implementation, and deployment of components," *Communications of the ACM (CACM)*, vol. 45, no. 10, pp. 35–40, 2002.

[4] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1998.

[5] J. Nordlander, M. P. Jones, M. Carlsson, and J. Jonsson, "Programming with time-constrained reactions," Luleå University of Technology, Tech. Rep., 2005. [Online]. Available: http://pure.ltu.se/ws/fbspretrieve/441200

[6] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck, "The Rubus component model for resource constrained real-time systems," in *3rd Int. Symp. on Industrial Embedded Systems*, 2008, pp. 177–183.

[7] Arcticus Systems Homepage. [Online]. Available: http://www.arcticus-systems.com

[8] H. Kopetz, "Component-based design of large distributed real-time systems," *Control Engineering Practice*, vol. 6, no. 1, pp. 53–60, 1998.

[9] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[10] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, pp. 294–324, 1997.

[11] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[12] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[13] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2007, pp. 259–268.

[14] E. Gery, D. Harel, and E. Palatshy, "Rhapsody: A complete life-cycle model-based development system," in *Proc. of the Third Int. Conf. on Integrated Formal Methods*, ser. Lecture Notes in Computer Science, vol. 2335. London, UK: Springer-Verlag, 2002, pp. 1–10.

[15] Artisan Software Tools. [Online]. Available: http://www.artisansw.com

[16] Rational Rose Technical Developer. [Online]. Available: http://www.ibm.com/software/rational

[17] S. Graf, I. Ober, and I. Ober, "A real-time profile for UML," *Int. J. on Software Tools for Technology Transfer*, vol. 8, no. 2, pp. 113–127, 2006.

[18] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard, "MARTE: Also an UML profile for modeling AADL applications," in *12:th IEEE Int. Conf. on Engineering Complex Computer Systems*, 2007, pp. 359–364.

# Part II

# Paper A

# Enabling Component-Based Design for Embedded Real-Time Software

**Authors:**
Jimmie Wiklander, Jens Eliasson, Andrey Kruglyak, Per Lindgren, and Johan Nordlander

**Reformatted version of paper accepted for publication in:**
Journal of Computers (JCP)

# Enabling Component-Based Design for Embedded Real-Time Software

Jimmie Wiklander, Jens Eliasson, Andrey Kruglyak, Per Lindgren, Johan Nordlander

**Abstract**

The increasing complexity of embedded software calls for a new, more efficient design approach. A natural choice is to use well-established component-based design; however, its adoption to design of embedded software has been slow and riddled with difficulties. It can be argued that these problems are due to the following peculiarities of embedded systems. Firstly, the tight integration between hardware and software, typical for embedded systems, makes it virtually impossible to model and implement software separately from hardware. Secondly, it is difficult to express timing requirements, an intrinsic part of functionality of many embedded systems, in dataflow abstractions traditionally used in component-based design.

We propose to overcome these difficulties by introducing a uniform, consistent modeling of both hardware and software and by integrating timing requirements into the model. We present a modeling framework based on the notions of reactive objects and time-constrained reactions, which enables component-based design of embedded real-time systems. Within this framework, functionality of both hardware and software components is defined in terms of reactions to discrete external events, and timing requirements are specified for each reaction relative to the event that triggered it. We also present a detailed software design methodology for embedded real-time systems based on our modeling framework.

## 1 Introduction

In recent years, the complexity of embedded systems has been steadily increasing, and the number and complexity of functions performed by embedded software has also grown. This calls for introduction of new, more efficient design methods[1]. An attractive approach is component-based design, which facilitates component re-use, separate development of components, and improves overall maintainability and robustness of the system.

However, adoption of this approach to embedded software development has been significantly slower than to software development in general. It can be argued that the problem lies in the fact that embedded systems manifest a tight integration between functionality implemented in software and functionality of hardware parts. In many

[1] A good overview of existing design practices and research trends in embedded system design is given in [1] and [2].

embedded systems, hardware components cannot be viewed as part of the environment external to the software system since the software has to be developed "around" the available hardware resources, relying on their timing and other properties. This requires a uniform, consistent modeling of both hardware and software. The situation is further complicated by the fact that embedded systems, unlike most general-purpose computing systems, often perform computations subject to various constraints, such as processor speed, amount of memory, power consumption, and reaction time. The timing requirements are often of special importance, especially for safety-critical systems. In fact, the majority of embedded systems can be viewed as real-time systems, i.e. systems in which correctness of system behavior (for hard real-time systems) or quality of service (for soft real-time systems) relies on the time when results are delivered to the environment as well as on the computed values as such.

We conclude that it is necessary to modify the traditional component-based approach to software development so that (a) a tight integration between software and hardware is taken into account, and (b) timing requirements can be clearly defined at both system and component level and used to guide implementation.

In this article we present a modeling framework that allows to uniformly model both hardware and software and to incorporate timing requirements into the model (Section 2). We also present a step-by-step methodology for embedded software design based on our modeling framework (Section 3) and demonstrate it in the design of a small embedded system, a personal alarm device (Section 5), implemented in the programming language Timber (Section 4). A short overview of related work is given in Section 6.

## 2 Modeling Framework

Component-based design relies on the existence of consistent and coherent models of individual components that can be composed to model the whole system. We propose a modeling paradigm based on a combination of event-based, reactive, concurrent, and object-oriented programming models that provides a natural framework for specifying the behavior of hardware, software, and mixed hardware/software components of an embedded system.

*Event-based modeling* implies that interaction between the system and its environment, as well as between components of the system is conducted by means of discrete events occurring at specific times. The *reactive approach* allows us to specify functionality in terms of reactions to such events, and since both input and output events are discrete, it is possible to impose time constraints on these reactions, effectively integrating timing requirements into functional specification [3]. The simplest way to specify such constraints is by defining the earliest and the latest reaction time (*baseline* and *deadline*) relative to the time of the input event triggering the reaction. We will call the time window between the reaction baseline and its deadline a *permissible execution window* for this reaction (Fig. 1) and denote it as ***after*** $t_{after}$ ***before*** $t_{before}$ *doSmth*, where $t_{after}$ is the period of time between the event and the baseline, $t_{before}$ is the period of time between the baseline and the deadline, and *doSmth* is the invoked method.

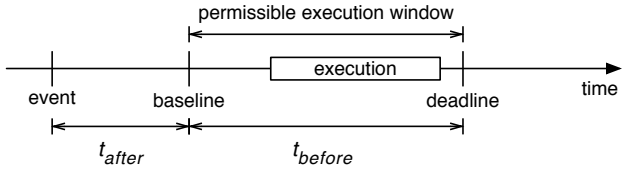



*Figure 1: Permissible execution window for a reaction to an event.*

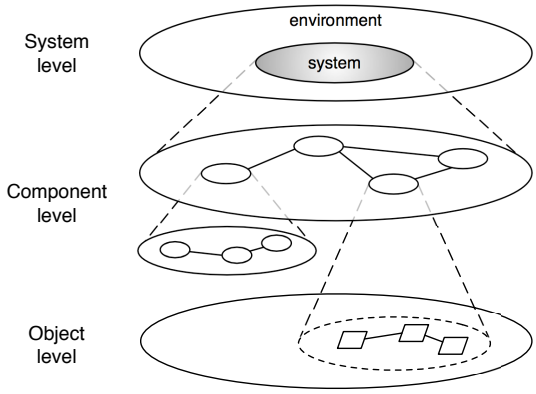



*Figure 2: Three abstraction levels of modeling: system level, component level (including multiple sublevels to accommodate a component hierarchy), and object level. A system is realized in terms of components, and each component is realized in terms of objects.*

*Concurrency* is inherent in hardware and is unavoidable in more complex software systems that have to perform multiple tasks (react to multiple events) at the same time. It is important to reflect this concurrency in the model of an embedded system. This gives rise to the problems of synchronization and state protection. We address these issues by modeling and implementing components using *reactive objects*[2] [4]; we define that all mutable state variables have to be encapsulated within an object and only accessible via its methods. Reactive objects can be used as units of concurrency by specifying that no two methods of the same object can execute concurrently while any two methods of different objects can.

Modeling complex systems requires using multiple levels of abstraction. We will distinguish the following abstraction levels: system level, component level (which can include multiple sublevels to accommodate a hierarchy of components), and object level, as depicted in Fig. 2. In our model, we will not try to include all information at each level; instead, the relationship between the layers is one of a gradual refinement of the model where each next level contains more details.

[2]We will be using executable models which means that reactive objects are preserved in the implementation.

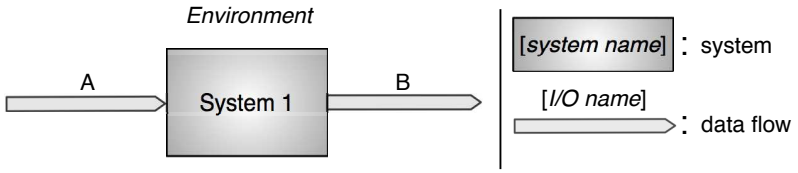



*Figure 3: Data flow model of system interaction with its environment.*

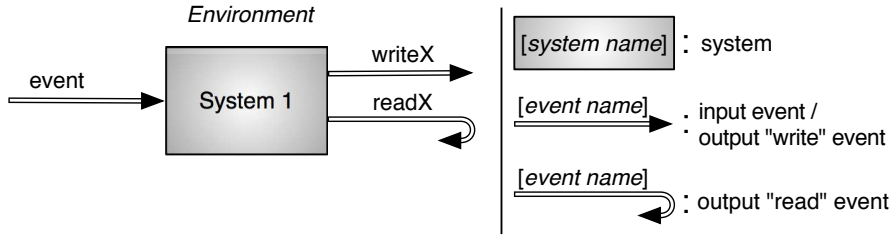



*Figure 4: Event-based system-level model.*

## 2.1 System-Level Model

At system level, the system is viewed as a black box, and the focus is on defining the boundary between the system and its environment. Since embedded systems typically manifest a tight integration between software and hardware, the system model should include both software and hardware, even if the hardware is given and is not developed as part of the design process. From the modeling perspective, existing hardware parts can be considered either as a part of the system or as a part of its environment. In this case, the system boundary should be defined so that it is easy to specify system functionality in terms of reactions to input events as described below.

In component-based design, the system's interaction with the environment is typically described in dataflow terms as input from the environment and output from the system (Fig. 3). However, to be able to define timing properties of the system, input and output should be expressed as discrete events occurring at specific times, resulting in a reactive event-based model. Then system functionality can be defined as reactions to input events and timing requirements can easily be described as constraints on these reactions. Output events constitute part of a system reaction to an input event and can be divided into asynchronous ("write") and synchronous ("read") events (Fig. 4). Note that if some parameter in the environment is sampled by the system, this can be reflected as an input in the dataflow model but as a "read" output event in the event-based model.

## 2.2 Component-Level Model

At component level, we use components to model the system. A component is defined as an encapsulation of a part of system state and/or hardware resources, with a

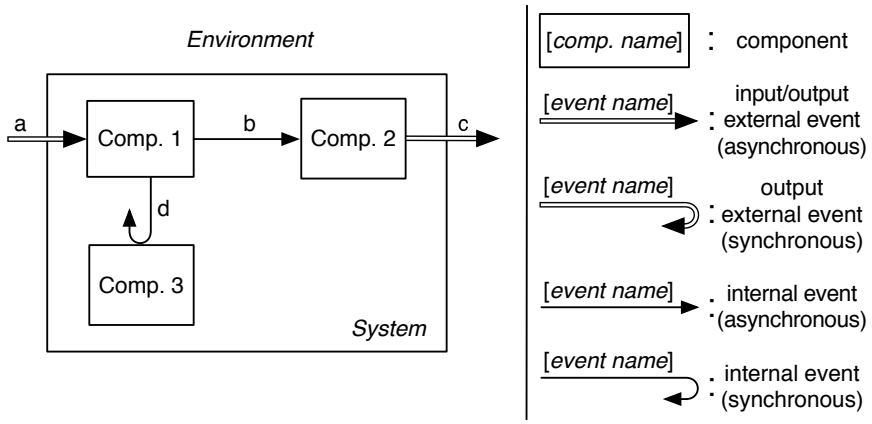



*Figure 5: Event-based component-level model.*

clearly defined interface and functionality. Importantly, state variables and hardware resources must belong to only one component and cannot be shared by two or more components (the question of allocating CPU resources, i.e. processing time, to components will be addressed later). This definition allows for hardware, software, and mixed hardware/software components.

Each component can be specified independently of the rest of the system in terms of time-constrained reactions to input events. Input events can either be external events originating outside the system, or internal events originating in another component. Both input and output events can be asynchronous ("write" events, one-way interaction) or synchronous ("read" events, synchronization events, etc.). Note, however, that external input events are always asynchronous (Fig. 5). Unlike reactions to asynchronous events, reactions to synchronous events cannot have a permissible execution window of their own, as they have to complete before the deadline for the component that posted the synchronous event and awaits a response.

Components can be organized hierarchically, when a component is partitioned into subcomponents. Partitioning is governed by considerations such as composability, reusability, ease of understanding, etc. as will be described later in this article.

## 2.3 Resource Platform

A useful abstraction that can be built upon system partitioning into components is the notion of a *resource platform*. The intuition behind it is that a number of components taken together can present a certain basic functionality with a clearly defined interface that can be utilized by a whole range of applications (Fig. 6). A resource platform typically includes all hardware components of the system, since they are the most difficult to change and may often have somewhat limited composability, but it can equally well include mixed hardware/software components or software-only components that are used

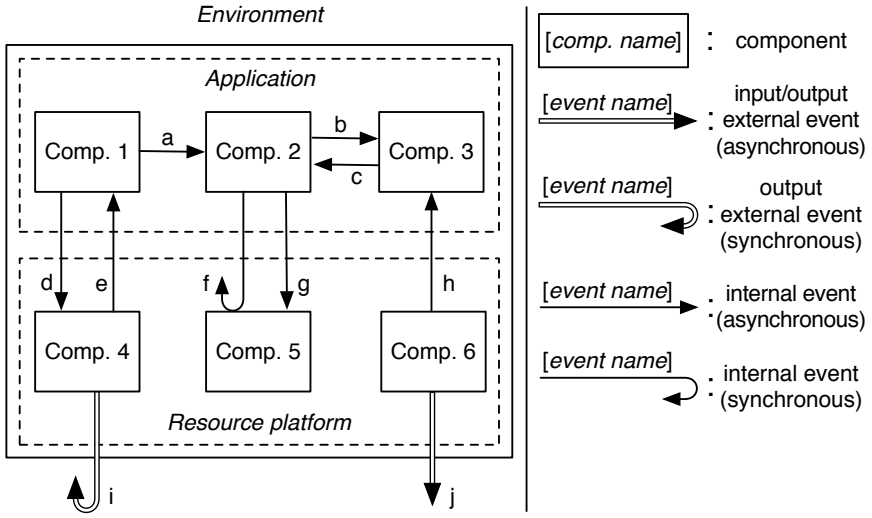



*Figure 6: Partitioning of a system into a resource platform and a software application.*

as resources by the applications. This gives us a clear separation of the system into a resource platform and an application (embedded systems typically have only one application, but it is possible to consider several applications sharing the same platform at runtime).

Note that since the separation into a platform and an application is performed at a relatively high level of abstraction, a platform may have multiple instances, differing in the choice of specific hardware and/or specific implementation of software. This approach allows for a fast and efficient development of a number of applications for a certain platform while leaving enough flexibility in platform implementation to perform optimizations in device size, cost, power consumption, and performance.

## 2.4 Object-Level Model

At the lowest level, each component is modeled using *reactive objects*. A reactive object is a model that can have one or several hardware and/or software instances; however, it cannot be instantiated as a mixed hardware/software entity. The choice of implementation is made at this level, so an object-level model clearly specifies which objects should be implemented in hardware and which in software (Fig. 7).

Both software and hardware objects react to external and internal input events and for each reaction a permissible execution window can be specified relative to the time of the event (Fig. 1). External input events originate in the environment, and each type of event triggers a method of a specific object. Internal input events originate within the system; in the case when such events are both produced and consumed by software objects, they can be viewed and implemented as messages. Even events originating

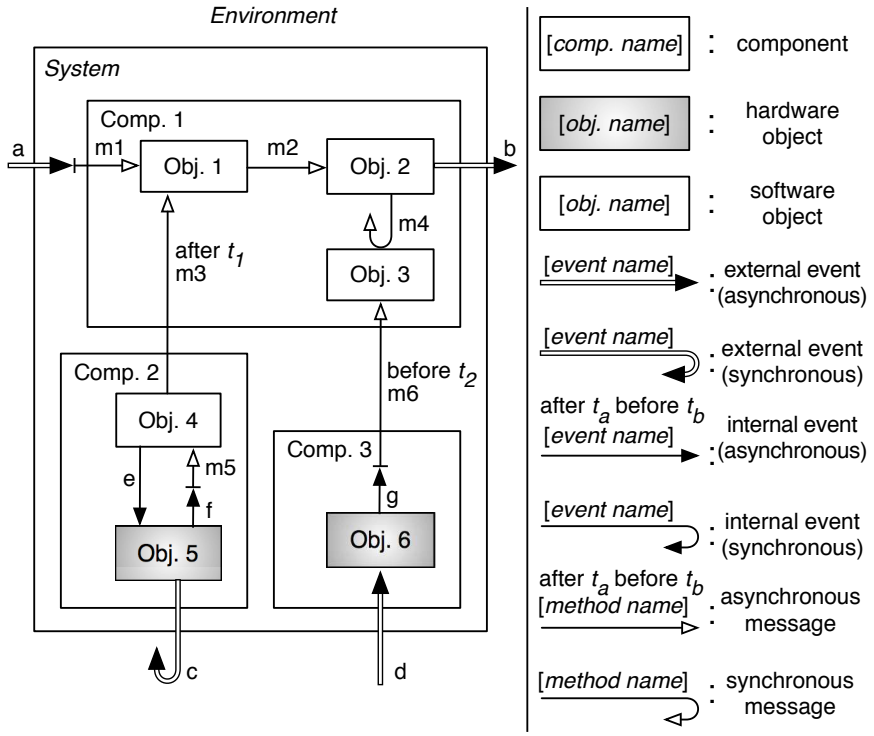



*Figure 7: Event-based object-level model. For each reaction, a permissible execution window can be specified using the* ***after*** $t_a$ ***before*** $t_b$ *notation; absence of such notation indicates inheritance of timing constraints. Input events to software objects originating in other software objects are marked as messages; input events to software objects originating outside the system or in hardware objects are translated into messages.*

outside the system or in hardware objects can be translated into messages if they are consumed by a software object.

As any events, messages can be either synchronous or asynchronous. In the latter case, a software object can also post a message to itself. Asynchronous messages can be delayed by a certain amount of time defined relative to the baseline of the object sending the message. This also allows to encode a periodic behavior by letting an object post a delayed asynchronous message to itself. Synchronous messages return a value, and the execution of the sender object is blocked until then; that is why reactions to synchronous messages cannot be delayed and always inherit the permissible execution window of the sender.

A software object encapsulates its state and provides methods to operate on it; a *reactive* software object cannot block during method execution waiting for input. In our model, state protection is absolute – all mutable variables have to be state variables in some object, and no access to state variables is allowed except via methods of the object.

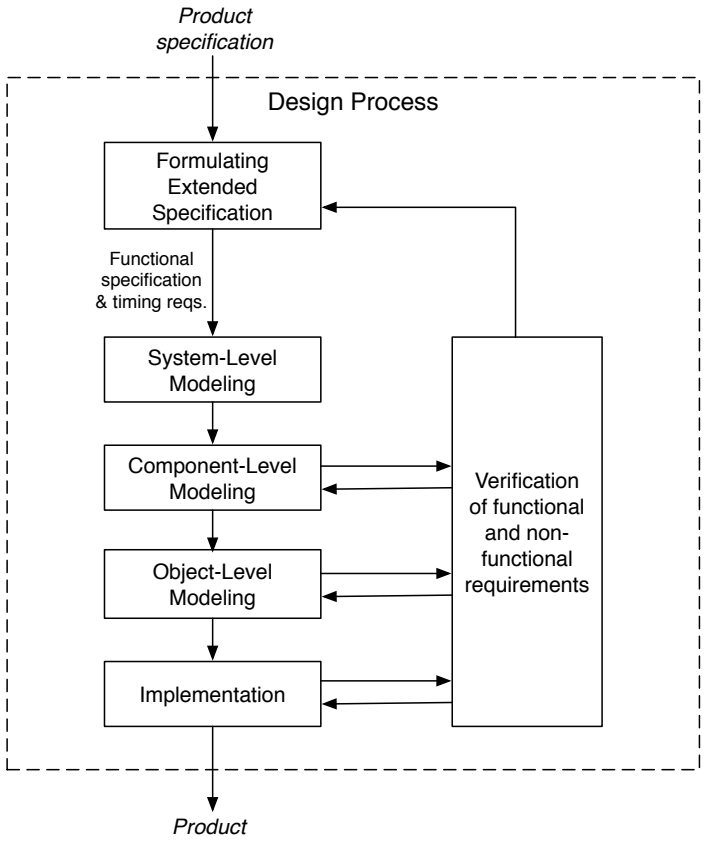



*Figure 8: Stages in the design process: from a specification to a ready product.*

Besides, no two methods of the same object are allowed to execute concurrently but methods of two different objects can, resulting in an object-level concurrency model.

# 3 Software Design Methodology

Here we present a methodology for embedded software design based on the modeling framework described in the previous section. This framework allows us to model both software and hardware parts of an embedded system, and a complete model of the system is essential for designing embedded software and verification of the system as a whole.

The different stages of the design process are presented in Fig. 8. The input to the design process is the product specification which originates from the client commissioning the system and which we assume to be static during the development of the system. This specification is usually written in a natural language, is often incomplete and imprecise. Hence the first step is drawing up a complete specification with a clear division into

functional and non-functional parts. In our case, functional specification is integrated with timing requirements and is used throughout system modeling and implementation. Non-functional specification lists the remaining system properties and constraints, such as system size and power consumption, and is primarily used during verification. It can also be used to guide selection of ready-made components, especially those including hardware.

The second step is formulating a system-level model where system interface to its environment is defined in terms of external input events triggering time-constrained system reactions and system output events which are part of such reactions. In the third step, this model is elaborated by identifying system components and interfaces between them. Such components are key to facilitating software re-use and maintenance, as well as system verification. In the fourth step, the components are realized using reactive objects, and a decision is made on which reactive objects should be implemented in software and which represent models of (existing) hardware parts. At every step, the model of each component is matched against a repository of previously developed components (either software or hardware), which should contain reactive models of components alongside their implementation.

The fifth step is implementation of software objects in some programming language. The sixth and final step is system verification, which can be done by simulation of the model, by testing of the implementation, or by formal methods. Both functional and non-functional requirements can be verified, and a failed verification forces a return to an earlier development step, making the development process iterative. Verification of new components should also be performed at earlier stages of development to verify certain properties at object and component levels. At the final step, verification of component integration and of the system as a whole is conducted. Note that verification of system schedulability on a particular hardware platform is separate from verification of the model itself.

## (1) Defining Extended System Specification

The extended specification has to be complete, verifiable, and contain a clear separation into functional and non-functional specification; it is obtained by refining the original product specification. Our approach requires that the functional specification should be expressed in terms of time-constrained reactions to external events representing input to the system or changes in the environment. Thus timing requirements are integrated into functional specification. It has to be noted that not all systems let their functionality to be naturally defined in terms of time-constrained reactions, which should be seen as a limitation of applicability of our approach. However, it is our belief that the majority of embedded systems can be specified in such a way.

A prominent example is a system with a time-continuous input; a system which as such does not conform to the notions of events and reactions. However, on closer inspection, any discrete realization of such a system would indeed require a sampling strategy with its corresponding timing requirements.

In other cases timing requirements can be implicit, defined by the rate of incoming events and the necessity to keep up with them. A typical example is routing of packets in a network; while the maximum forwarding delay may be omitted from the specification, it can be derived from a packet buffer length together with the allowed drop rate for a given traffic profile.

## (2) Formulation of System-Level Model

In this step a system-level model should be formulated from the functional and timing specification by determining the system's boundary with its environment and its interface. To achieve a clear-cut separation between the system and its environment, the system should be defined to encompass all the functionality that we have to develop, and it should be taken to include the hardware that the developed software will execute on. Such hardware should be notionally included in the system even if it is given and cannot be changed during the development process. Note that the environment includes both natural phenomena the system will interact with and the infrastructure that is being developed or has been developed separately. Thus all "external" services used by the system, especially those shared between the system under development and other systems, are considered to be part of the environment rather than the system proper.

## (3a) Partitioning into Components

Although component-based design has been studied for several decades, partitioning of a system into components (as well as partitioning of a component into subcomponents), remains more of an art than an exact science. However, it is possible to identify the main guiding principles.

Each component should have a clearly defined role in the system, and a one-to-one mapping between components and system functions is always preferable. This means that any two independent tasks, triggered by independent external events and resulting in independent outputs, should be realized by two separate components. The same is true for the case when a system should perform two activities in parallel, with little or no state sharing and/or interaction between them.

A special type of components (often associated with hardware or mixed hardware/software components) are resources which can be used by one or several activities and which usually enforce some kind of exclusion or sharing protocol to guarantee consistency of system output and/or its internal state. Several resources are often bundled together in one component when they are used jointly to perform one task or cannot operate in parallel.

Apart from these main principles, a number of other considerations can affect the design of a particular system, such as:

- *composability* – to facilitate system composition from newly-designed or ready-made components, it is important for each component to have a clear purpose (role in the system) and a clearly defined interface. It is also advantageous to have

as few interdependencies between components as possible.

- *reusability* – functionality common to a class of (possible) applications can be effectively assigned to a separate component, facilitating component re-use.
- *robustness* – to make better use of ready-made components, and to enhance system verification while shortening the development time, it is important that each component is designed with regard to future verification (testing, simulation, and possibly formal verification) at component level as well as at system level. Robustness can also be improved if components are used as fault-containment regions, which requires detectability of errors at component boundaries.
- *ease of understanding* – an extremely important consideration that is often overlooked is that partitioning into components should enhance the ability of the original developer(s) of the system as well as those who may work with it in the future to clearly understand the functions and structure of the system. This calls for the components to be small enough to be easily comprehendable, but at the same time large enough to keep the structure of the higher-level component simple. Experience shows that following this principle leads to fewer mistakes (and hence shorter development times and increased robustness) and facilitates re-use and maintenance.

An important issue of component-based design is what kind of interactions are allowed between components. It is advantageous to make components as independent of each other as possible since it simplifies component specification, enhances composability, and facilitates verification of individual components. We therefore strongly discourage synchronous communication across component boundaries. Synchronous communication between components should normally be used for predictably quick interaction, such as reading a value (as opposed to waiting for a value to be computed), or performing a hardware operation that takes a known time to complete under certain operational conditions.

Once defined and implemented, components can be stored in some repository for future use. It is important to preserve not only the actual implementation, but also a model of the component (see section *3b*) alongside its testing and verification results. It may also be useful to preserve a testing suit for a component so that the tests can be re-run in a new setting. If the implementation of a component is protected as intellectual property and will not be accessible for system verification, the timing properties of the component also have to be stored in the repository. These would have to include execution time and maximum blocking time (per hardware resource) for each reaction defined in the interface of the component.

## (3b) Search for Ready-Made Components

In this step, models of defined components are matched against models of earlier developed components from the repository. Comparison between the models requires that they are of the same kind. In our case, it means that a component model should have its functionality expressed in terms of time-constrained reactions to events external to the

component. Identity of modeling principles should lead to a straightforward integration of a matched component into the system model.

There might be components in the repository that do not match the specification, but can be either adapted by introducing an intermediate layer, or can be modified to fit the specification. The downside of component modification is that it may require substantial work on re-implementing the component as well as invalidate the testing and verification results.

## (3c) Hierarchical Refinement of Component Structure

One of the strengths of component-based design is the possibility of hierarchical refinement of component structure. Partitioning of a component into subcomponents closely mirrors partitioning of a system into components as described above; the same principles and guidelines apply. Since one and the same component can (at least theoretically) be used in different systems, partitioning into subcomponents should be performed independently for each component and should not be influenced by a wider context in which the component is used. However, it is possible that identical subcomponents are identified as parts of different components, and those can be viewed as separate instances of the same component class.

If any new subcomponents have been identified in this step, a return to search for matching components in a repository is warranted. The process is repeated until no further refinement of component structure can be justified.

## (4) Realization Using Reactive Objects

The last step in the modeling process is component realization using concurrent reactive objects. This step involves partitioning of the component into reactive objects and identifying hardware and software parts. Similarly to partitioning into subcomponents, it is performed on each component independently of its context. Note that at this level hardware parts are modeled as reactive objects, which allows for a certain flexibility when several hardware parts are modeled using the same object model if they only differ in, for example, power consumption.

For each component, it is necessary to identify: hardware resources; object state in terms of state variables; and object functionality in terms of methods. Partitioning of a component into objects is governed by slightly different principles than partitioning of a system into components. These principles can be obtained by adaptation of well-known object-orientation strategies to the concept of concurrent reactive objects. The following has to be taken into consideration:

Each object encapsulates its state that can only be accessed by methods of the same object. At the same time, the objects are units of concurrency, meaning that any two methods of the same object cannot be executed concurrently but any two methods of two different objects can. A notable exception is the case when an object posts a synchronous message to another object; then the caller remains blocked until the invoked method returns.

The guiding principles of partitioning into objects aim to maximize schedulability of the system while maintaining state consistency. Component state, seen as a collection of state variables, should be partitioned and assigned to objects in such a way that

- state duplication (when the same state is duplicated as state variables in two or more objects, leading to synchronization problems) is avoided;
- state variables routinely modified together are encapsulated in one object;
- otherwise, state is maximally distributed between different objects to allow for a better schedulability of the system.

Functionality should be assigned to methods, and methods to objects in such a way that

- methods using the same state variables are assigned to the same object;
- methods using different parts of component state are assigned to different objects together with corresponding state variables, in order to maximize schedulability of the system; an exception to this rule is the case when consistency between several state variables has to be guaranteed;
- a special attention is paid to the consequences of mutual exclusion between methods of the same object, when an object remains blocked and cannot execute any other method while an earlier invoked method is executing. For example, in some cases a single reaction should be split into two methods, one calling the other asynchronously, thus creating a window of opportunity for a reaction with a shorter deadline to execute on the same object in between the two methods.

The issue of software interaction with hardware parts is of utmost importance and has to be considered separately. This interaction is often governed by complicated protocols that are not relevant to the application at large. Hence it is a good idea to have a single software object controlling access to specific hardware. Apart from providing a useful abstraction of the software-hardware interface, such objects can be used to explicitly control sharing of the hardware resource by enforcing arbitration or queuing if so required.

## (5) Implementation

The next step is the implementation process in which the system model is instantiated. The hardware platform is built using identified hardware parts (COTS components, SoC blocks, etc.), and software reactive objects are implemented in some programming language. In the case when some of the software components are re-used from the repository, the issue of code integration has to be addressed. The complexity of code integration will depend on the language used in the implementation of the re-used components.

An example of a programming language, Timber, fully supporting the described modeling framework and thus suitable for use together with the present software design methodology, will be described in section 4.

### (6) Verification

The final step in embedded system design is verification (see Fig. 8). We will distinguish between verification of the model and verification of the implementation; both should be conducted at component as well as system level.

Verification of the model is done against system specification and specification of individual components. This includes verification of component composition at system level and verification of functional specification (including timing requirements), which can be performed using simulation or with formal methods (see, for example, the work on UPPAAL [5–8]). Importantly, verification of the model is independent of its feasibility, i.e. whether or not it can be implemented in a specific programming language and on a specific hardware platform in such a way that the functional and timing requirements are met.

Verification of the implementation should also be conducted at both component and system level and, unlike verification of the model, it involves verification of both functional and extra-functional requirements. At component level, it is only necessary to verify that the implementation corresponds to the model. At system level, both component integration and system feasibility have to be verified. System feasibility refers to the ability of a specific implementation (software and hardware) to meet the functional and timing specifications of the model under extra-functional constraints such as energy consumption; an important part of feasibility verification is schedulability analysis (see [9]). Note that schedulability analysis requires a full knowledge of the system implementation. In the case when the implementation of a particular component is not available for analysis, at least the list of resources used by each reaction of the component should be known together with the execution time and maximum blocking time for each resource. Schedulability analysis should be the preferred way of system verification since it allows to prove system correctness for all inputs and in all situations, as do other formal methods. However, verification of system implementation can also be conducted using simulation and testing.

Let us separately consider verification of a resource platform. A clear division into a platform and an application allows to verify them separately, so that an already verified platform with known properties can be used for development of other applications. It should be noted, however, that system-level verification such as schedulability analysis has to be performed on the system as a whole, including both the application and the resource platform, even if the platform has previously been verified.

## 4 An Implementation Approach: The Timber Language

The presented model is sufficiently general to allow a variety of possible implementations. For example, since we can model a complete system including both hardware and software parts, the border between hardware and software can be adjusted even after the model has been completed. Hardware components together with hardware parts of

mixed (hardware/software) components can be realized by e.g. selecting existing COTS hardware parts and integrating them into a single hardware platform, whereas software components together with software parts of mixed components have to be implemented in some programming language, typically combined with a minimal operating system or a kernel that will provide scheduling, I/O, etc.

While it is fully possible to implement the model described above in, for example, C/C++ or Esterel, the translation itself would be far from trivial. The problem is to preserve the properties of individual components and of the system as a whole, to maintain composability of defined components, and to be able to verify that functionality and timing of the resulting code reflect those of the model. Using mainstream programming languages often results in a gap opening up between the model and its implementation. For example, reaction deadlines may have to be translated into thread priorities and as a result, the system's behavior would depend on other tasks and the scheduling policy; hence the correspondence between the model and its implementation becomes very difficult to verify.

Another possible implementation approach is to use the recently developed modeling and programming language Timber, which targets real-time systems ([10–12]). Timber is a high-level programming language that uses the same primitives as the proposed model, including reactive objects and time-contrained reactions. Hence translation of a model into Timber code is straightforward and preserves system structure and timing specification, closing the gap between the model and its implementation. Timber code can be compiled into a subset of C and executed on any target platform in combination with a Timber kernel[3], which uses permissible execution windows preserved in the code for deadline-based scheduling.

Timber is both a high-level programming language for real-time systems and a formalism that can be used to verify a system's functional behavior, timing properties (complying with deadlines), liveness (absence of deadlocks), and termination of computations. Let us briefly describe the relevant properties of the language:

- *inherent support for reactivity*: the system functionality is expressed in terms of reactions to external events, with reaction defined as a combination of internal state updates and/or system outputs. Each reaction can be comprised by a chain of reactions executed by different objects, some of them executed concurrently. Execution of a system reaction must be non-blocking, i.e. it cannot block waiting for an external input.
- *time-constrained reactions*: each reaction has a baseline (the earliest time when execution can start) and a deadline (the latest time by which execution must have finished); it is possible to schedule a reaction to start at some point of time in the future by setting its baseline relative to the baseline of the reaction being executed. The timing requirements are preserved in the application code at run-time and can be used to guide scheduling.

[3] A prototype version of a Timber kernel has so far been implemented for a generic POSIX environment and for an ARM platform, but thanks to its minimalistic nature it can be ported to other platforms relatively easily.

- *object-orientation*: while constants (including global functions) can be defined at the top level, mutable variables are only allowed within objects as state variables. State encapsulation and protection are achieved by limiting access to these variables to the methods of the object, and state consistency is easily guaranteed by always enforcing mutual exclusion between the methods of the same object.
- *object-level concurrency*: Timber is a highly concurrent language with concurrency achieved by allowing methods of any two different objects to be executed in parallel.
- *message passing between objects*: Timber objects communicate by passing messages, synchronous (when the sender remains locked and waits for the message to return), and asynchronous (when the sender posts a message to another object or to itself, possibly with a postponed baseline, and continues execution). Asynchronous messages lead to concurrent execution of reactions.

# 5 An Example System: A Personal Alarm Device

The software design methodology described above has been tested in the development of a personal alarm device, used here to demonstrate different stages in the design process. Some details have been omitted for presentation purposes. The following functional specification of the device was given in the beginning of the design process:

The personal alarm device is a battery-driven system worn by a person on his or her body, for example, by an elderly person at a care facility. The device is capable of detecting the person's fall by analyzing acceleration. Once a fall has been detected, a fall alarm is sent wirelessly to an external receiver. The analysis requires that acceleration is sampled periodically every $t_{period}$ milliseconds. The device also includes an assistance call button that can trigger a separate kind of alarm sent in the same manner. An alarm must be sent within $t_{alarm}$ milliseconds after a fall has been detected or after the button has been pressed.

## (1) Defining Extended System Specification

An extended system specification should include both functional and non-functional requirements. The functional requirements have to be expressed in terms of time-constrained reactions. Two such reactions can be identified by analyzing the original specification.

The first reaction is sending an assistance alarm when the push button has been pressed. There is a timing requirement that the alarm is to be sent within $t_{alarm}$ milliseconds. The second reaction is sending a fall alarm, which is triggered by fall detection. This is realized using a fall detection algorithm that requires sampling acceleration at regular intervals equal to $t_{period}$ milliseconds. The algorithm distinguishes two stages in fall detection: impact detection, with impact detected by acceleration exceeding a threshold value; and posture evaluation (see [13, 14] for a detailed description of the algorithm). Posture evaluation is performed $t_{lag}$ milliseconds after an impact has been detected, and is used to establish if the person is lying down, in which case a fall has been detected. The

acceleration is sampled with the same periodicity both for impact detection and posture evaluation. Hence the following timing requirements can be given for the second reaction: the acceleration sampling period $t_{period}$; the lag between impact detection and posture evaluation $t_{lag}$; and the maximum period of time between fall detection and sending an alarm $t_{alarm}$.

Both an assistance alarm and a fall alarm are sent using a radio transceiver and are received by external infrastructure which is outside the scope of the system. Therefore, the communication protocol (with its timing requirements) has to be part of the extended specification.

Non-functional requirements for the system include a relatively small size (since the system has to be worn on the body, for example, at the hip), and a low power consumption (as the device is to be powered by a battery).

## (2) Formulation of System-Level Model

Analyzing system specification, we can distinguish two events that the system should react to: an assistance call realized as an interrupt from a button; and the person's fall. The interrupt from a button can be modeled as an external input event. The person's fall, however, is something that is detected by the fall detection algorithm which is internal to the system and hence it is not an external event. However, we can encode a periodic sampling of acceleration by the system as a reaction to a reset (an external input event) that starts up the system and triggers a reaction that includes sampling the acceleration (an external output "read" event) and posting a message with a delayed baseline that invokes another sampling after $t_{period}$ milliseconds, and so forth.

The timing requirements on the first reaction consist of a relative deadline $t_{alarm}$ milliseconds; the timing requirements on the second reaction are defined for each sampling that has a baseline equal to the baseline of the previous sampling plus $t_{period}$ milliseconds.

Note that while the hardware for the button, the accelerometer, and the radio transceiver are clearly a part of the system, the receiver of the alarm transmission is outside the developer's remit and should be viewed as an external service, not a system component. Thus the interface between the system and its environment is comprised on one hand, by reset interrupts and call button interrupts, and on the other hand, by the radio protocol used for communicating the alarms alongside the codes used to distinguish an assistance alarm from a fall alarm (see Fig. 9).

## (3a) Partitioning into Components

Let us now consider partitioning into components of our device. Analyzing the specification and the system-level model (Fig. 9) we can see that the application will need the following independent resources: an *acceleration sensor*, a *message sender* (containing a radio transceiver), and a *push button*. Their independence warrants creating three separate components, each of them including both hardware and software parts (Fig. 10).

The next step is to define the interface of these components, bearing in mind that it should be complete but at the same time sufficiently abstract to accommodate various

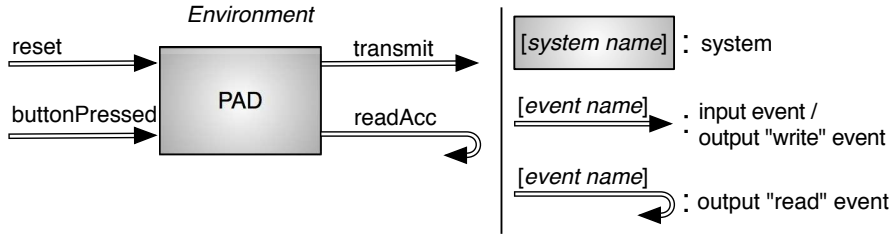



*Figure 9: System-level model for personal alarm device. Input events: reset and buttonPressed. Output "read" event: readAcc (reading acceleration). Output "write" event: transmit (sending a fall alarm or an assistance alarm).*

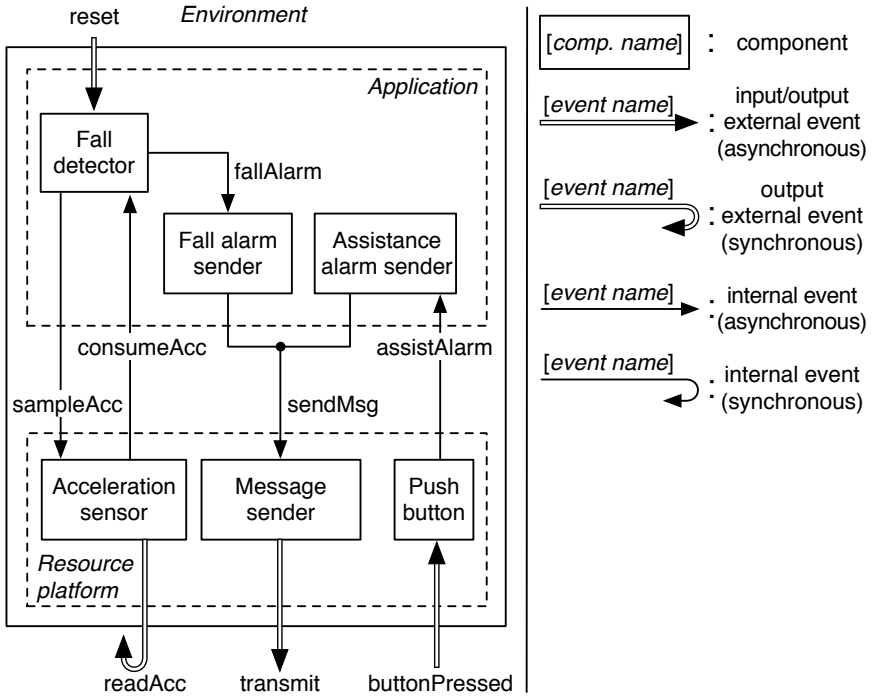



*Figure 10: Component-level model for personal alarm device, with separation into a resource platform and a software application.*

component implementations, which may possibly use different hardware to support the same functionality. The interface to the acceleration sensor should contain an input that can trigger sampling (*sampleAcc*), and an output that delivers the acceleration value once it has been acquired (*consumeAcc*). Note that to preserve reactivity and component independence, we cannot allow the caller to block waiting for the sampling to complete. It is therefore necessary to implement callback functionality in the acceleration sensor to

specify to which component the measured acceleration should be delivered. This can be done either when the acceleration sensor is instantiated (a static callback), or by passing a pointer to a function each time sampling is triggered (a dynamic callback). Similarly, to achieve the desired level of generality, the interface of the message sender should only contain one input – sending a message (*sendMsg*), and one output – delivery of a received message, but the latter is superfluous for our application. Note that the message sender represents a clear example of a shared resource – it can be used by any of the independent tasks of (a) fall detection, and (b) handling an assistance call. As such, it will have to include either message queuing or some kind of arbitration to synchronize access to the resource transparently to the components that may want to use it simultaneously. The interface of the last resource component – the button – is very simple, as it only needs one output to deliver the button event and the target component can easily be set statically. These three components naturally form a platform with clearly defined functionality and interface between it and any possible application.

It now remains to partition the rest of the system – the application – into components. Here two independent activities can be identified: *fall detection* and *assistance call handling*, resulting in two separate components. At the same time, it is appropriate to de-couple the fall detection algorithm from how the system should react to a detected fall. For our application, this involves creating a message and forwarding it to the message sender, which can be done by a separate component – a fall alarm sender. If assistance call detection in the application is similarly de-coupled from the reaction to it, we will have two very similar components – a *fall alarm sender* and an *assistance call sender*. A possible implementation is to create them as two instances of the same component, a general *alarm sender*, with some parameter set to different values at initialization. Alternatively, they can be viewed as two different components.

Timing requirements can be part of component specification as time constraints on the reactions. In this case, however, we skip this step and define the timing requirements directly at the object level.

## (3b) Search for Ready-Made Components

In our example, the personal alarm device is developed from scratch and there are no components that can be re-used in the design. However, let us consider what components could be used in the future in similar applications.

The first candidate for future use is, of course, the platform, consisting of an acceleration sensor, a message sender, and a push button (all components combining hardware and software). This is most natural because a platform is always defined as a collection of hardware and software resources that can be used by a range of possible applications. At the same time, it is not inconceivable that such components as an acceleration sensor, a message sender, or an alarm sender can be used separately in other designs.

## (3c) Hierarchical refinement of Component Structure

In the case of the example system, there is no room for hierarchical refinement of component structure due to the system's simplicity.

## (4) Realization Using Reactive Objects

The object-level model of the example system is presented in Fig. 11. The hardware parts have been identified and are shaded in the figure (their interfaces have been significantly simplified for presentation purposes).

It is clear that all resource components in our example require a mixed hardware/software implementation. In the acceleration sensor, the A/D controller object is used to abstract from the specific hardware interface of the A/D converter and to perform deserialization[4]. In the message sender, several objects are used to implement the network protocol, and a transparent sharing of the message sender between multiple components is provided by queuing incoming messages before sending. In the push button, a button controller functions as a simple interrupt handler.

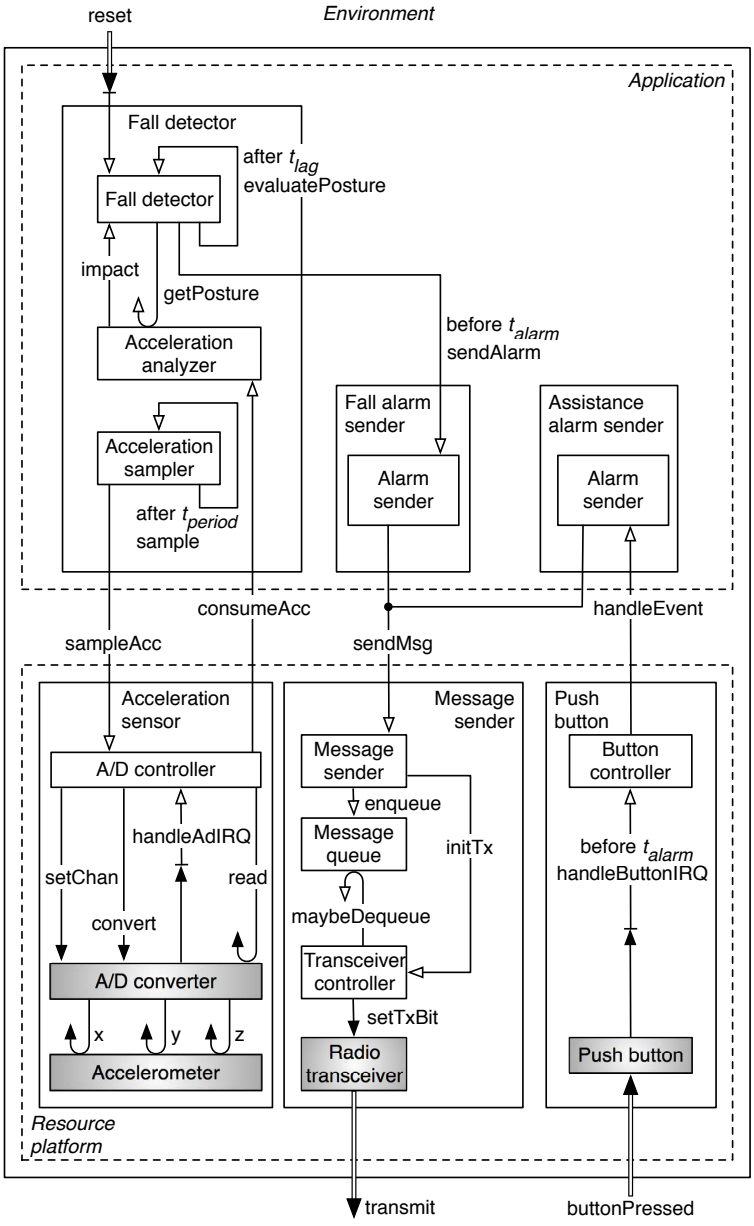
The only purely software component that consists of more than one reactive object is the fall detector. The acceleration sampler object triggers sampling by posting an asynchronous message to the *sampleAcc* method of the acceleration sensor component. Sampling at pre-determined intervals is achieved by the acceleration sample posting an asynchronous message to its own method *sample* with baseline delayed by $t_{period}$: $newBaseline = currentBaseline + t_{period}$.

The acceleration analyzer and fall detector objects cooperate to detect a fall. The acceleration analyzer posts an asynchronous message to the fall detector on detection of impact upon which the fall detector updates its internal state and posts an asynchronous message to its own method *evaluatePosture* delayed by $t_{lag}$. Once the *evaluatePosture* method is invoked, the person's posture is requested from the acceleration analyzer and if he or she is lying down, the fall detector posts an asynchronous message to the fall alarm sender specifying $t_{alarm}$ as deadline.

Assistance alarms are handled in a similar way. The application's *handleEvent* method is directly linked to the *sendAlarm* method in the assistance alarm sender. This method inherits the permissible execution window defined for the *handleButtonIRQ* method in the push button component.

*Figure 11: [Next page] Object-level model for personal alarm device. Note the absence of the output "read" event readAcc, which has been redefined as three internal events x, y, z reading different channels of an analog accelerometer.*

[4]Deserialization is required since A/D conversion can only be performed on one channel at a time, but values from all three channels are sent to the application for analysis.


reset
Environment
Application
Fall detector
after $t_{lag}$
evaluatePosture
Fall detector
impact
getPosture
Acceleration analyzer
before $t_{alarm}$
sendAlarm
Fall alarm sender
Alarm sender
Assistance alarm sender
Alarm sender
Acceleration sampler
after $t_{period}$
sample
consumeAcc
handleEvent
sampleAcc
sendMsg
Acceleration sensor
A/D controller
handleAdIRQ
setChan
read
convert
A/D converter
x
y
z
Accelerometer
Message sender
Message sender
enqueue
Message queue
initTx
maybeDequeue
Transceiver controller
setTxBit
Radio transceiver
Push button
Button controller
before $t_{alarm}$
handleButtonIRQ
Push button
Resource platform
transmit
buttonPressed
[comp. name] : component
[obj. name] : hardware object
[obj. name] : software object
[event name] . external event (asynchronous)
[event name] . external event (synchronous)
after $t_a$ before $t_b$
[event name] . internal event (asynchronous)
[event name] . internal event (synchronous)
after $t_a$ before $t_b$
[method name] . asynchronous message
[method name] . synchronous message

## (5) Implementation in Timber

The software part of the system was implemented in the programming language Timber (see section 4). As expected, the system structure presented in the model was complete and did not require any modifications; each reactive object in the model was implemented as such in Timber. Thus parallelism between system reactions was expressed at the object level in the model and preserved in the implementation. The implementation stage also involved writing Timber code for each method. All algorithms and functions were implemented; for example, a buffer holding sampled acceleration values was defined as a state variable in one of the objects, and a function was defined for filtering accelerometer data to remove noise in the signal.

The hardware platform defined in the model has not been implemented as yet. However, the software can be executed on any hardware platform that matches the presented model of the system, and we have verified that there exist COTS hardware parts that correspond to each reactive object in the model that should be implemented in hardware (acceleration sensor, radio transceiver, etc.).

## (6) Verification

The Timber implementation of the system software was verified, partly by typechecking performed by the Timber compiler, and partly by simulation of the software in a Simulink-based Timber simulator. In the simulated environment, the functionality and the timing specification (preserved in the Timber code of the implementation) were tested by feeding the software simulation with real sensor data of recorded falls of several human subjects as well as their normal daily activity [14].

When software is executed in a simulated environment and not on a real hardware platform, no meaningful execution times are available. However, since the timing specification is preserved in the implementation in form of permissible execution windows for each reaction, it is possible in the simulation to choose any point within this window, which is correct in the sense that it corresponds to the timing behavior as expressed in the model. A natural choice is to let each reaction to execute (with zero execution time) at its baseline; this approach was used in our simulations. Note that verification of whether the worst-case execution times on a particular hardware platform allow the system to always meet the required deadlines is a separate issue and has not been part of the verification performed so far.

The verification of the software implementation demonstrated the validity of both the algorithm and its implementation, as the falls of the subjects were accurately detected in the simulation.

A detailed account of implementation and verification of the system will be published elsewhere.

# 6 Related Work

The modeling and implementation approach realized in the Timber language can be compared to other solutions such as real-time synchronous languages (Esterel, SCADE, Lustre, etc. [15]) and time-triggered languages such as Giotto [16]. However, they are substantially different even if they can be seen as addressing the same design problems. For example, in synchronous languages, concurrency in system behavior is eliminated in the course of implementation, leading to a further separation between specification and model on one hand and implementation on the other hand. In Giotto, software is defined in terms of periodically executed tasks, reading inputs and writing outputs at pre-determined times, which is not particularly suitable for many embedded systems that exhibit a clearly reactive behavior and is not applicable to modeling hardware.

Our design approach is also different from that developed in the Ptolemy project [17]. The Ptolemy approach is a framework for assembly of concurrent components particularly suitable for modeling distributed systems. Essential to it is the notion of an *actor*, embodying the concept of active objects (as opposed to reactive objects). It can also be argued that the Ptolemy approach does little to bridge the gap between models and implementation, which is achieved in Timber by using executable models.

Component-based approach to design of (not necessarily embedded) real-time systems has been promoted by various extensions of real-time UML profile [18], with a number of tools already on the market (the most well-known probably are Rhapsody [19], ARTISAN Studio [20], and Rose-RT [21]). However, these solutions do not feature a true integration of timing requirements into functional specification, and do not completely solve the problem of modeling of mixed hardware/software systems.

Another design approach specifically targeting embedded systems is platform-based design [22]. This approach cannot really be considered component-based, as it concentrates on the methodology for separate development of a hardware platform, a system platform comprised by a hardware platform and hardware-software middleware (an API platform), and a software application that is developed for a given system platform. This can be contrasted with our approach where both a resource platform and a software application can be composed of multiple components, and where a resource platform can be designed together with an application; even if a platform is given, a model of the whole system is used to develop the software application. Our definition of a resource platform includes not only hardware components, but also software and mixed hardware/software components that can be utilized as resources by a range of applications.

# 7 Conclusion

The presented modeling framework allows for a unified, consistent modeling of both hardware and software. Integration of these models is beneficial for development of embedded systems as they often exhibit a great degree of interdependency between hardware and software, and the specification often describes the system as a whole rather than only its software part. At the same time, inclusion of timing requirements in a functional

specification in the form of time-constrained reactions allows us to specify, reason about, and verify real-time properties of embedded systems. Moreover, our modeling framework enables the developer to offer platform-independent correctness/quality of service guarantees for hard/soft real-time systems, provided that the software can be scheduled on a given hardware platform so that all reaction deadlines are met.

By combining this modeling framework with component-based design techniques and by expressing system functionality using reactive objects, our approach draws from the strengths of component-based design as well as from event-based, reactive, concurrent, object-oriented programming models. It facilitates software re-use and maintenance as well as separate development of parts of the system. This approach is realized in the concrete software design methodology presented above.

Apart from addressing the issues of software complexity, interdependency between software and hardware, and complying with the timing requirements, our approach also allows to clearly define the notion of a *resource platform* as a combination of hardware and software resources. A resource platform can be designed to serve as a base for a whole range of related applications, decreasing the overall development costs and time to market.

The presented software design methodology can be used both in the case when software is developed alongside a hardware platform (the latter being assembled from existing hardware parts) and in the case when such a platform is given from the start. In both cases, a platform is instantiated using some implementations of hardware and/or software components depending on performance, power consumption, and other non-functional requirements.

The design approach presented in this article requires further development in the following directions: formalization of component structure; creating design tools supporting the methodology; and investigating the role of other requirements, such as power consumption, in the design process. The power consumption issue is especially challenging since it introduces new constraints and modeling parameters to deal with. However, with today's rapid increase of battery-powered embedded systems, this is a very important issue to address. A holistic view of both timing and and power consumption will offer new and interesting possibilities in the area of embedded system design.

## 8 Acknowledgment

This work was supported in part by the Knowledge Foundation in Sweden under a research grant for the project SAVE-IT, the EU SOCRADES project, and the EU Interreg III A North Programme grant 304-13723-2005.

## References

[1] B. Bouyssounouse and J. Sifakis, *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, ser. Lecture Notes in Computer Science. Berlin,

Germany: Springer-Verlag, 2005.

[2] C. Atkinson, C. Bunse, H.-G. Gross, and C. Peper, *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, ser. Lecture Notes in Computer Science. Berlin, Germany: Springer-Verlag, 2005.

[3] J. Nordlander, M. P. Jones, M. Carlsson, and J. Jonsson, "Programming with time-constrained reactions," Luleå University of Technology, Tech. Rep., 2005. [Online]. Available: http://pure.ltu.se/ws/fbspretrieve/441200

[4] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black, "Reactive objects," in *Fifth IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2002, pp. 155–158.

[5] W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving," in *Proc. of the 7th Int. Conf. on Formal Description Techniques*, 1994, pp. 223–238.

[6] K. G. Larsen, P. Pettersson, and W. Yi, "Model-Checking for Real-Time Systems," in *Fundamentals of Computation Theory*, ser. Lecture Notes in Computer Science, no. 965. Springer-Verlag, 1995, pp. 62–88.

[7] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Uppaal—a tool suite for automatic verification of real-time systems," in *Proc. of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control*. Secaucus, NJ, USA: Springer-Verlag, 1996, pp. 232–243.

[8] T. Amnell *et al.*, "Uppaal — Now, next, and future," in *Proc. of the 4th Summer School on Modeling and Verification of Parallel Processes*, ser. Lecture Notes in Computer Science. London, UK: Springer-Verlag, 2001, pp. 99–124.

[9] L. Sha *et al.*, "Real-time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 28, no. 2, pp. 101–155, 2004.

[10] P. Lindgren, J. Nordlander, L. Svensson, and J. Eriksson, "Time for Timber," Luleå University of Technology, Tech. Rep., 2005. [Online]. Available: http://pure.ltu.se/ws/fbspretrieve/299960

[11] M. Carlsson, J. Nordlander, and D. Kieburtz, "The semantic layers of Timber," in *First Asian Symp. on Programming Languages and Systems (APLAS)*, ser. Lecture Notes in Computer Science, vol. 2895. Berlin, Germany: Springer-Verlag, 2003, pp. 339–356.

[12] The Timber language homepage. [Online]. Available: http://www.timber-lang.org

[13] M. Kangas, J. Wiklander, I. Vikman, L. Nyberg, P. Lindgren, and T. Jämsä, "Sensorband fall detector prototype: Validation through data collection and analysis,"

in *The 2nd Int. Symp. on Medical Information and Communication Technology (ISMICT'07)*, 2007.

[14] M. Kangas, I. Vikman, J. Wiklander, P. Lindgren, L. Nyberg, and T. Jämsä, "Sensitivity and specificity of fall detection in people aged 40 years and over," *Gait & Posture*, vol. 29, no. 4, pp. 571–574, 2009.

[15] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[16] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[17] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2007, pp. 259–268.

[18] S. Graf, I. Ober, and I. Ober, "A real-time profile for UML," *Int. J. on Software Tools for Technology Transfer*, vol. 8, no. 2, pp. 113–127, 2006.

[19] E. Gery, D. Harel, and E. Palatshy, "Rhapsody: A complete life-cycle model-based development system," in *Proc. of the Third Int. Conf. on Integrated Formal Methods*, ser. Lecture Notes in Computer Science, vol. 2335. London, UK: Springer-Verlag, 2002, pp. 1–10.

[20] Artisan Software Tools. [Online]. Available: http://www.artisansw.com/

[21] Rational Rose Technical Developer. [Online]. Available: http://www.ibm.com/software/rational

[22] A. Sangiovanni-Vincentelli, "Defining platform-based design," *EEDesign of EE-Times*, 2002.

# Paper B

# Personal Alarm Device: A Case Study in Component-Based Design of Embedded Real-Time Software

**Authors:**
Jimmie Wiklander, Andrey Kruglyak, Per Lindgren, and Johan Nordlander

**To be submitted.**

# Personal Alarm Device: A Case Study in Component-Based Design of Embedded Real-Time Software

Jimmie Wiklander, Andrey Kruglyak, Per Lindgren, Johan Nordlander

**Abstract**

Designing software for embedded systems is complicated by such factors as the tight integration between software, hardware and scarceness of available resources (power, memory, etc.), and the fact that system operation is often subject to hard real-time requirements. In our earlier work we proposed a component-based approach based on modeling both hardware and software using reactive objects and time-constrained reactions, which should allow us to overcome these difficulties. We also presented a step-by-step software design methodology for embedded real-time systems.

In this work we describe a real-life system developed using this methodology and discuss its advantages. The system is a personal alarm device that should be worn at the waist of a person and that should detect his or her (accidental) fall and send an alarm signal. The underlying fall detection algorithm is based on measuring two types of acceleration, static (gravity and tilt) as well as dynamic. The system was implemented using the programming language Timber. The implementation was verified using a Simulink-based simulator for Timber. During simulation, the software operated according to its specification. The simulation also demonstrated that, even though calculation of acceleration was simplified to allow for an efficient execution on a resource-constrained platform, the fall detection has been shown to be satisfactory. This makes it possible to utilize a lightweight microcontroller which in turn implies a lower power consumption, a smaller physical size of the device, and a lower price. These qualities are very important for portable systems and are crucial for the system's applicability in real life.

The case study demonstrates the advantages of the proposed software design methodology, including the fact that functional and timing properties of a system model can be preserved during implementation process by means of a seamless transition between a model and an implementation.

## 1 Introduction

Embedded systems possess certain qualities that turn embedded software design into a delicate task. First of all, embedded systems typically have limited resources at their disposal (CPU, memory, power, etc.) that have to be utilized efficiently in order to meet system requirements. As a consequence, embedded systems often exhibit a tight

integration between software and hardware. This makes it virtually impossible to model and efficiently implement software separately from hardware. Most embedded systems can also be classified as real-time systems, i.e systems in which correctness of system behavior (for hard real-time systems) or quality of service (for soft real-time systems) depends on the time when results are delivered to the environment as well as on the computed values as such. The traditionally used techniques do not have an inherent support to express and model temporal behavior, which makes designing software for such systems a challenging task. Moreover, the ongoing advances in the microprocessor technology leave room for more elaborate software implementations, adding complexity to the design, which requires more mature design methods than those in use today. In [1] we proposed a component-based approach to cope with these difficulties. We presented a modeling framework based on the notions of reactive objects and time-constrained reactions, which facilitates component-based design of embedded real-time systems. Within this framework, functionality of both software and hardware components is defined in terms of reactions to discrete events, and timing requirements are specified for each reaction relative to the event that triggered it. We also presented a detailed software design methodology for embedded real-time systems based on our modeling framework.

In this work we describe design of a real-life system developed using this methodology and discuss its advantages. The system in question is a personal alarm device (PAD) and its underlying fall detection concept is based on using an acceleration sensor. In our earlier work [1], this system was used as an example to demonstrate some stages of the design process. Here we present the system design process in full, including the choice of the the fall detection algorithm, system implementation and verification, omitted in [1]. A special focus is given to specification and implementation of timing requirements.

An overview of the software design methodology is given in Section 2. It is followed by a description of the PAD, including the prescribed fall detection algorithm (Section 3). The design of the PAD is discussed in Section 4, with the timing requirements discussed separately in Section 5. Section 6 deals with the implementation of the PAD in the Timber programming language, while Section 7 describes verification of the design and implementation in a Simulink-based simulator. A discussion of related work follows in Section 8, and Section 9 concludes the presentation by discussing the advantages of the methodology as demonstrated by the presented case study.

## 2 Methodology Overview

Our software design methodology relies on a unified, consistent modeling of both hardware and software. The modeling framework is based on the notions of reactive objects [2] and time-constrained reactions [3]. A more detailed presentation of the methodology and the modeling framework can be found in Wiklander et al. [1].

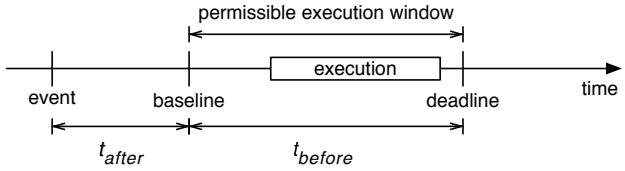



*Figure 1: Permissible execution window for a reaction to an event.*

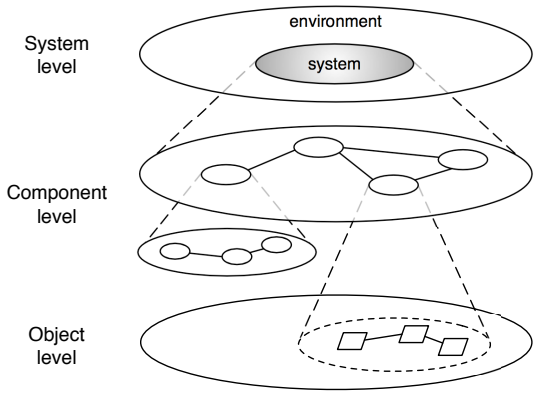



*Figure 2: Three abstraction levels of modeling: system level, component level (including multiple sublevels to accommodate a component hierarchy), and object level. A system is realized in terms of components, and each component is realized in terms of objects*

## 2.1 Modeling Framework

The modeling framework supports describing the functionality of both hardware and software. Interaction between the system and its environment, as well as between components of the system is modeled as discrete events occurring at specific times. Following the reactive approach, the functionality is specified in terms of reactions to such events. Embedded systems must often conform to specific timing requirements which can be specified by defining the earliest and the latest reaction time (*baseline* and *deadline*) relative to the time of the input event triggering the reaction. The time window between the reaction baseline and its deadline is called a *permissible execution window* for this reaction (Fig. 1), and it is denoted as after ***after*** $t_{after}$ ***before*** $t_{before}$ *doSmth*. Here $t_{after}$ is the baseline offset (period of time between the triggering event and the baseline), $t_{before}$ is the period of time between the baseline and the deadline, and *doSmth* is the invoked method of the object *obj*. A reaction with a permissible execution window defined for it will be called a *time-constrained reaction*.

In order to model complex systems there is a need to model the system at various levels of abstraction. The modeling framework distinguishes the following abstraction levels: system level, component level (which can include multiple sublevels to accommodate a hierarchy of components), and object level, as depicted in Fig. 2.

At the highest level (system level) the system is viewed as a black box and the inter-

action between the system and its environment is expressed as discrete events occurring at specific times. At the next level (component level) the system is partitioned into components and the interaction between components is again expressed as discrete events. Typically, a component encapsulates a part of system state and/or hardware resources and has a clearly defined interface and functionality. We extend this definition by requiring that functionality of each component is expressed in terms of time-constrained reactions to these events, which gives us what can be described as *reactive components*.

A component hierarchy allows us to structure the system, but ultimately, all system functionality has to be expressed in terms of reactive objects, the smallest building blocks in our model. So at the lowest level of abstraction, each component is modeled using reactive objects. A reactive object is defined by its interface (its methods), encapsulated state, and one or more implementations. A reactive object is a model that can have one or several hardware or software implementations, differing in their non-functional (but not timing) properties. In contrast to a component which may contain both software and hardware implementations of objects, a reactive object is implemented either in hardware, or in software (written in some programming language).

## 2.2 Software Design Methodology

The different stages of the design process are presented in Fig. 3. The input to the design process is the product specification which originates from the client commissioning the system. This specification is usually written in a natural language, is often incomplete and imprecise. Hence the first step is drawing up an extended specification with a clear division into functional and (optionally) non-functional parts. In our case, functional specification is integrated with timing requirements and is used throughout system modeling and implementation. Non-functional specification lists the remaining system properties and constraints, such as system size and power consumption, and is primarily used during verification.

The second step is formulating a system-level model where the interface between the system and its environment is defined in terms of external input events triggering time-constrained reactions and output events as a part of such reactions.

In the third step, this model is elaborated by identifying system components and interfaces between them. In the fourth step, the components are realized using reactive objects, and a decision is made on which reactive objects should be implemented in software and which represent models of (existing) hardware. At every step, the model of each component is matched against a repository of previously developed components (either software or hardware), which should contain reactive models of components alongside their implementation.

The fifth step is implementation of software objects in some programming language (which might require providing the infrastructure necessary for real-time execution on a hardware platform, e.g. an operating system supporting scheduling). This step might also involve building a hardware platform from identified hardware COTS components, SoC blocks, etc. depending on whether the platform was available in the beginning of

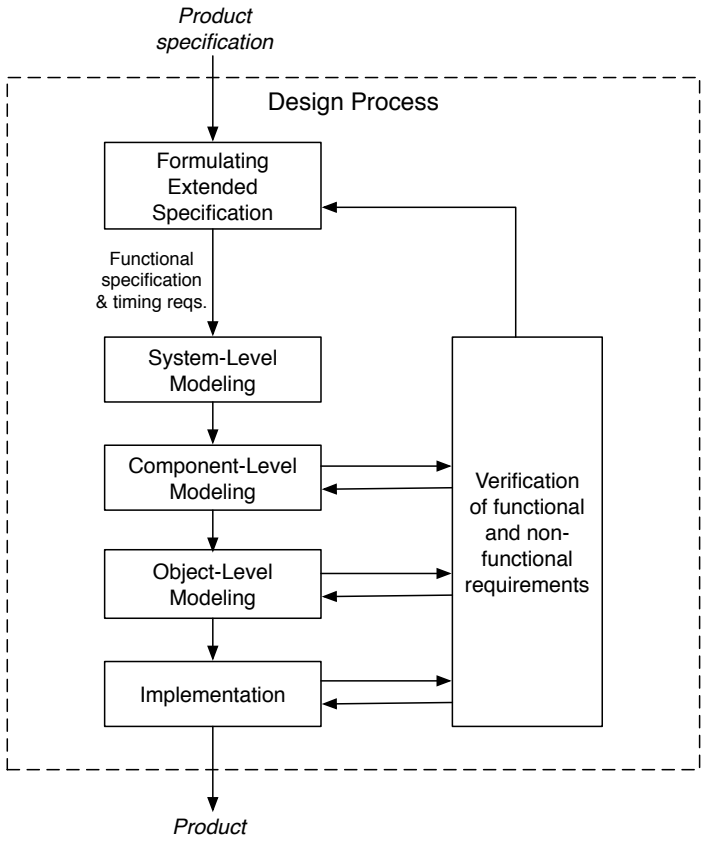



*Figure 3: Stages in the design process: from a specification to a ready product.*

the design process or not.

The sixth and final step is system verification, which can be done by simulation of the model and/or implementation, by testing of the implementation, by formal methods, etc. Both functional and non-functional requirements can be verified, and a failed verification forces a return to an earlier development step, making the development process iterative.

## 2.3 Maximizing Component Re-use: Introducing a Resource Platform

The component-based design methodology presented here describes how a system is modeled and designed when it is developed from scratch, possibly re-using individual components from previous designs. However, in reality the hardware platform (the assembly of hardware components) can often be given from the start, which should come as no surprise since the cost of its development can be substantial. This hardware platform may, apart from a CPU and memory, contain sensors, I/O and communication hardware

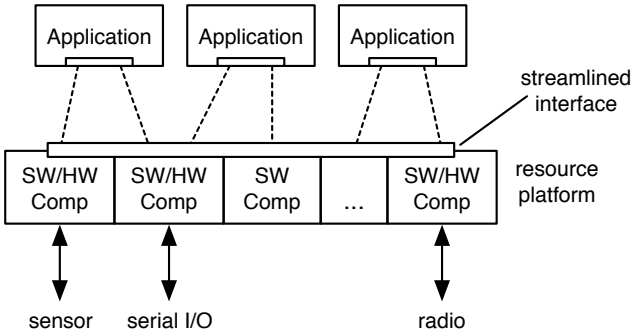



*Figure 4: An example of a resource platform used by several applications in parallel.*

and the like, which in our model will be parts of different components. Such hardware often has complicated and vendor-specific interfaces, so it makes sense to add a layer of software that would encapsulate the details of software-hardware interaction and present a streamlined interface to the main application. The resulting component containing a hardware resource and the interfacing software will be called a resource component.

The collection of resource components will form a resource platform, which in essence encapsulates the hardware platform and can be viewed as providing services (such as radio communication or sensor readings) to the application (Fig. 4). Note that apart from facilitation application development, this approach facilitates component re-use in more than one way. The resource platform with its clearly defined functionality and streamlined interface can be used with different applications, decreasing the cost of hardware platform development; moreover, it can be used as a shared resource for multiple applications running simultaneously. Parts of the resource platform can be upgraded or changed to satisfy non-functional requirements of the system (e.g., power consumption, device size) without changing its interface to the applications and thus without modifying the applications themselves.

We further extend the notion of a resource platform by allowing it to include pure software components that can be shared between applications running on the platform and that can be viewed as offering services to them, such as a database component. Such components will also be classified as resource components. Inherent in the notion of a resource component is its ability for offering services to multiple applications (or components). To facilitate component composition, clients of resource components should be able to operate unaware of each other. This can be achieved if the run-time system provides an exclusion mechanism where only one request can be handled at any particular time, or if the resource component itself holds a specialized internal queue of requests.

This approach is similar to what is known as "platform-based design" ([4], [5]). Note, however, that we introduce the notion of a resource platform as a part of a wider component-based approach rather than in isolation.


*Figure 5: The prototype device with axis orientation notations of the internal acceleration sensor.*

# 3 Personal Alarm Device (PAD)

The Personal Alarm Device (PAD) should be worn by a person who might require assistance in the case of a fall; the aim of a PAD is to detect such falls and automatically send an alarm, as well as to enable the wearer to manually trigger an alarm by pressing a button on the device. In order to allow for the wearer's mobility, the device should be battery-driven, and the alarms should be sent wirelessly to an alarm receiver covering a certain area.

## 3.1 Fall Detection: the Concept

Fall detection is based on using an acceleration sensor. Such a sensor would typically measure static acceleration (acceleration generated by the earth's gravity), and dynamic acceleration (generated by sensor movements) in three perpendicular directions. The device contains an acceleration sensor, a radio, and a microprocessor. It can be fastened at the waist so that the vertical axis of the human body is aligned with a predetermined axis of the acceleration sensor. As a result, the posture of the body (standing up or lying down) can be determined by evaluating the body's alignment relative to the gravitational field of the earth. The dynamic acceleration can be used to analyze parameters typical for a fall, such as velocity towards the ground, fall-related impacts (when the body abruptly hits something), etc. Combining all or some of these parameters with posture evaluation makes it possible to detect falls.

## 3.2 Finding a Suitable Fall Detection Algorithm

The concept described above requires some kind of fall detection algorithm for analyzing acceleration and making decisions based on the results of analysis. However, formulating a suitable fall detection algorithm is not straightforward.

In a previous study [6] different fall detection algorithms where evaluated using acceleration data recordings from intentional falls and ADL (activities of daily living). The data was recorded using a prototype device (Fig. 5) containing an internal acceleration

sensor. In the study, the y-axis of the sensor was aligned with the vertical axis of the body. Besides evaluation of different fall detection algorithms, the study also had an aim to validate the data collection of the prototype device.

The algorithms were analyzed in a LabView environment using fall data collected from middle-aged test subjects. Data representing activities of daily living collected from middle-aged and older people were used as a reference. For each algorithm, sensitivity and specificity were calculated using Eqs. 1 and 2:

$$sensitivity = \frac{TP}{TP + FN}, \tag{1}$$

$$specificity = \frac{TN}{TN + FP}, \tag{2}$$

where TP = true positives (detected falls), FN = false negatives (undetected falls), FP = false positives (ADL samples giving false fall alarm), and TN = true negatives (ADL samples not giving fall alarm).

Threshold values for the algorithms were adjusted for optimal detection of falls with as few false alarms as possible; recordings of ADL were used as a reference. The best performing algorithm (referred to as *Algorithm 1* in [6]) discriminated various types of falls from activities of daily living, with a sensitivity of 97.5% and a specificity of 100% using floating point simulations (presented in Table 1 in Section 7, where we compare the results of simulation from the previous study with the results of simulation of our implementation).

Realizing the aforementioned concept in a PAD requires implementing the fall detection algorithm on a suitable platform (such as the prototype device platform that may not support floating point operations). It is therefore important to test the effects of different representations of real values found in the mathematical definition of the algorithm (floating point, 32-,16-, or 8-bit integers). Simulations of *Algorithm 1* with 16-bit data format resulted in a preserved fall detection sensitivity and specificity compared to floating-point simulations. In contrast, data processing in 8-bit format resulted in a preserved fall detection sensitivity but only limited fall detection specificity. Based on the results from the study, *Algorithm 1* can be seen as a viable choice when it comes to implementing the fall detection functionality on a 16-bit platform [6].

### Description of the Algorithm

The fall detection algorithm is based on *impact detection* and *posture evaluation*. The acceleration is monitored periodically (with period $t_{period}$) which enables detection of impacts to the body (not necessarily fall-related). Detection of an impact triggers the posture to be evaluated after a specific time period. If the posture is categorized as lying, a fall has been detected.

Impacts are detected by calculating the sum vector

$$SV = \sqrt{{A_x}^2 + {A_y}^2 + {A_z}^2}, \tag{3}$$

where $A_x$, $A_y$, and $A_z$ represent the acceleration (dynamic and static) in the $x$-, $y$-, and $z$-direction of an acceleration sensor, respectively. If $SV$ exceeds the experimentally established threshold, it is assumed that an impact has occurred. An impact event triggers posture evaluation to take place after a predetermined time period $t_{lag}$. However, if an additional impact occurs within this time, posture evaluation is re-scheduled relative to the latest impact event. In posture evaluation the tilt of the vertical axis of the human body relative to the gravitational field of the earth is evaluated. A lying posture is presumed if the static acceleration of the vertical axis of the human body is less than or equal to the experimentally established threshold.

The timing constraints ($t_{period}$ and $t_{lag}$) form a critical part of the algorithm, and our methodology allows to express and preserve them at all steps of the design and implementation process, as will be demonstrated below. This distinguishes our methodology from numerous other approaches to designing real-time systems.

# 4 PAD Design

Design of the PAD was performed in accordance with the methodology presented in [1] and summarized in Section 2. The first stage is to define an extended system specification on basis of the product specification (found at the beginning of Section 3). For this particular system, the focus was on the functional requirements, which had to be expressed in terms of time-constrained reactions. Two such reactions were identified: sending an assistance alarm within $t_{alarm}$ *msec* after the push button has been pressed, and sending a fall alarm within $t_{alarm}$ *msec* after a fall has been detected. Fall detection is based on the fall detection algorithm described in Section 3, which brings with it additional timing requirements – $t_{period}$ *msec* (the time between acceleration samplings) and $t_{lag}$ *msec* (the time between impact detection and posture evaluation).

### System-Level Model

The next stage in the design process is to define a model of the system. Modeling is performed successively at three different levels of abstraction. At the highest level (system level) the interaction between the system and its environment is defined in terms of input events and output events. Drawing from the intended functionality of the PAD (as described in Section 3) it is possible to identify the different forms of interaction that the system should support. The overall PAD interaction was formulated in terms of two input events (reset and pressing of the button) and two output events (alarm transmission and acceleration reading). Note that an acceleration reading is initiated by the system, not its environment, thus it is considered an output event, even though the dataflow is from the environment to the system. The reaction to reset consists of initializing the system and starting up periodic reading of acceleration (with period time $t_{period}$); a reading may result in detection of a fall and then transmission of an alarm, which has its own deadline $t_{alarm}$. The reaction of pressing the button consists of sending another type of alarm with the same deadline $t_{alarm}$. The system-level model of the PAD

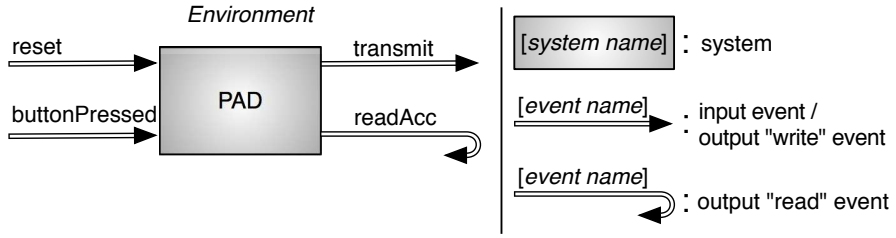



*Figure 6: System-level model for personal alarm device. Input events: reset and buttonPressed. Output "read" event: readAcc (reading acceleration). Output "write" event: transmit (sending an alarm).*

is presented in Fig. 6.

### Component-Level Model

The next modeling level is the component level. Analyzing the specification and the system-level model we conclude that the PAD application will need the following independent resources: an *acceleration sensor*, a *message sender* (containing a radio transceiver), and a *push button*. Their independence warrants creating three separate components, each of them including both hardware and software objects (Fig. 7). These components can be seen as resource components, and together they form the *resource platform* for the PAD application.

It now remains to partition the rest of the system – the application – into components. Here two independent activities can be identified: fall detection and assistance call handling, resulting in at least two separate components. At the same time, it is appropriate to de-couple the fall detection algorithm from how the system should react to a detected fall. For our application, this involves creating a message and forwarding it to the message sender, which can be done by a separate component. The resulting component structure is presented in Fig. 7.

The fall detector component is activated periodically (with period time $t_{period}$) and it triggers sampling of acceleration by acceleration sensor component. The sampling itself is a complicated interaction with hardware conducted in several steps, each with its own timing requirements; it will be discussed in detail in the next section. This is hardware-specific and is encapsulated in the acceleration sensor component, that only presents one method in its interface (*sampleAcc*), one that can be used to initiate acceleration sampling procedure. Once the sampling has been completed, the acceleration sensor component triggers analysis of the collected data in the fall detector component, using a method provided to it during initialization (*consumeAcc*).

Detection of a fall in the fall detector component constitutes a new (internal) event, which leads to invocation of fall alarm sender (*sendAlarm*) and message sender (*sendMsg*) components. Both reactions have a common deadline of $t_{alarm}$ *msec*.

The push button component is activated on the arrival of an interrupt from the button.

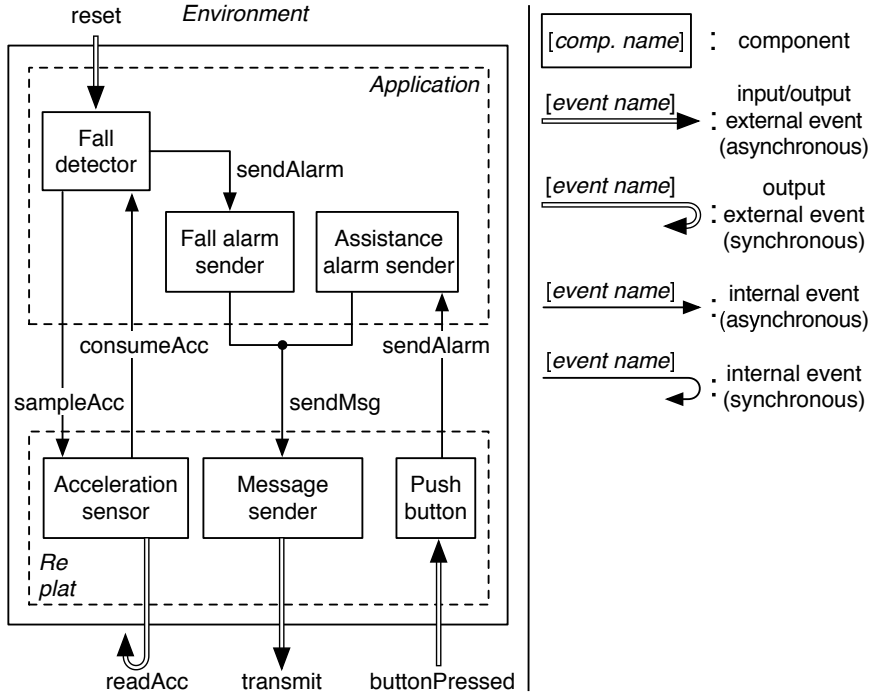



*Figure 7: Component-level model for personal alarm device, with separation into a resource platform and a software application. External events describe interaction between the system and the environment; internal events describe interaction between components of the system.*

The component takes care of filtering interrupts to eliminate the effect of "bouncing", when a single press results in multiple interrupts delivered to the microcontroller. Reaction to a button pressing event involves the three components (push button, assistance alarm sender, and message sender) with a common deadline of $t_{alarm}$ *msec*.

Note that the message sender represents a clear example of a shared resource – it can be used by any of the independent tasks of (a) fall detection, and (b) handling an assistance call. As such, it will have to include either message queuing or some kind of arbitration to synchronize access to the resource transparently to the components that may want to use it simultaneously. It is also worth noting that the message sender is defined in such a way that its timing behavior is specified by the "client" components rather than locally.

### Object-Level Model

The lowest level of abstraction in the modeling framework is the object level. At this level the internals of each component are modeled in terms of reactive objects. Similarly to partitioning into subcomponents, the process of defining the reactive object model is performed on each component independently of its context. For each component, it is

necessary to identify: hardware resources apart from CPU and memory; object state in terms of state variables; and object functionality in terms of methods. The object-level model should also contain additional information on which objects are implemented in software and which in hardware. The object-level model of the PAD is presented in Fig. 8.

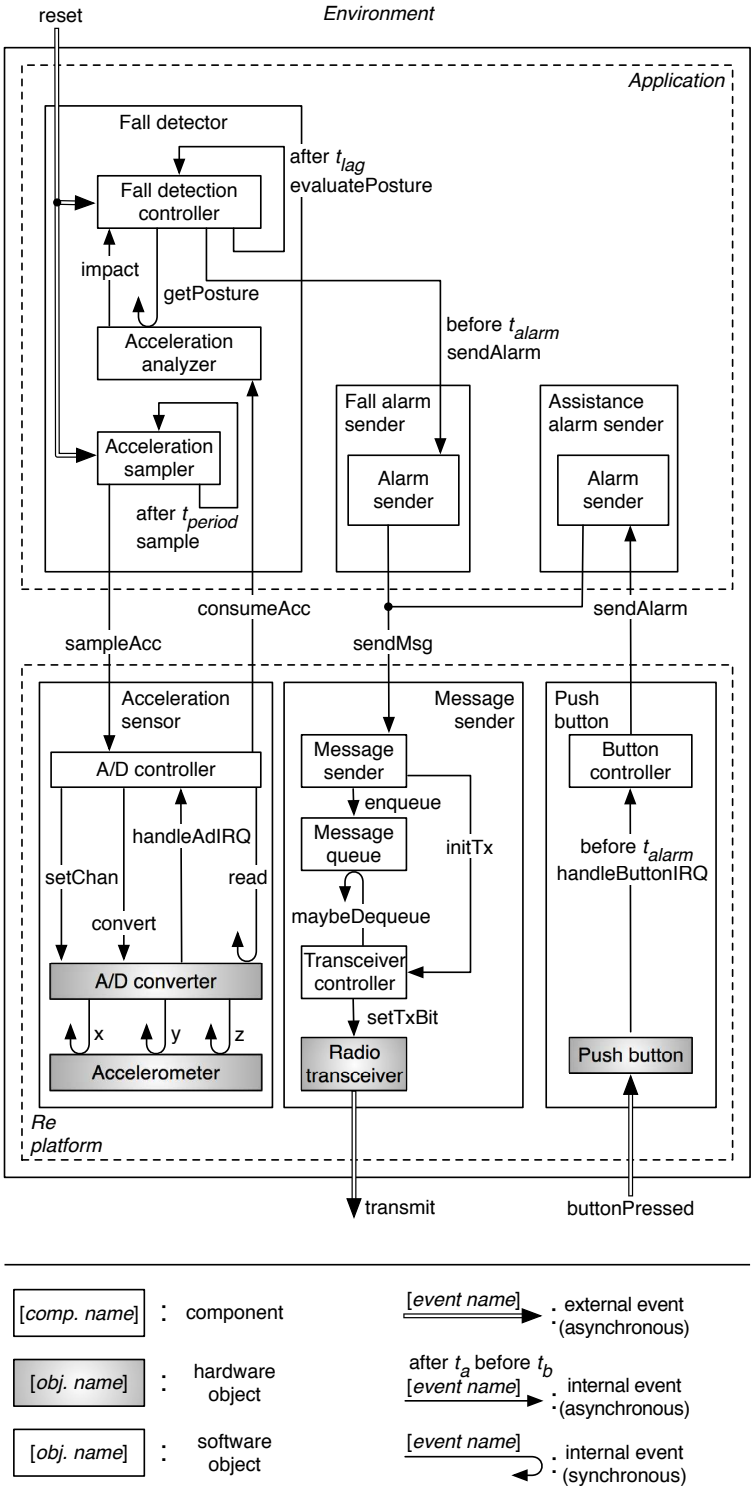
Interaction between objects is described in terms of internal events, just as interaction between components. In fact, interaction between components is nothing more than interaction between objects crossing component boundaries. Interaction between software objects can be implemented using messages, whereas interaction between software and hardware objects is implemented as writing to hardware registers, invoking interrupt handlers, etc.

Let us now take a deeper look at some of the internal of the components that constitute the PAD resource platform. The function of the *acceleration sensor component* is to deliver acceleration readings in three perpendicular directions. The component is passive in the sense that it reacts to sampling commands issued by the application. Specific hardware is needed in order to acquire acceleration readings from the environment, such as an analog accelerometer and an A/D converter. In Fig. 8 the hardware objects are shown as shaded boxes (the parts of their interfaces only used during system initialization have been left out). We choose an A/D converter that operates sequentially, i.e. the converter can only sample one channel (or acceleration direction) at a time. Hence, before triggering a conversion the appropriate channel must be selected. All A/D converter interaction, such as channel selection, is handled by the A/D controller software object. It is also responsible for delivering the results (in our case – to the fall detector component) after acquiring the three samples. As a result, all hardware-dependent timing requirements associated with A/D conversion must be handled by the A/D controller, which is described in the next section.

The function of the *message sender component* is to send messages over some media. Since the media in our case is radio it includes hardware in the form of a radio transceiver. The radio transceiver is controlled by a transceiver controller software object which encodes the message following a specific network protocol. In order to support transparent sharing of the message sender between multiple components the message sender component also includes two software objects that enable message queuing. The rather complicated timing constraints on the individual objects' reactions within this component are not discussed here.

In the *push button component*, the single reaction deadline $t_{alarm}$ *msec* is defined. It is inherited by the assistance alarm sender and later on by the message sender components.

*Figure 8: [Next page] Object-level model for personal alarm device. External events describe interaction between the system and the environment; internal events describe interaction between objects of the system.*


reset
Environment
Application
Fall detector
after $t_{lag}$
evaluatePosture
Fall detection controller
impact
getPosture
Acceleration analyzer
before $t_{alarm}$
sendAlarm
Fall alarm sender
Assistance alarm sender
Acceleration sampler
Alarm sender
Alarm sender
after $t_{period}$
sample
consumeAcc
sendAlarm
sampleAcc
sendMsg
Acceleration sensor
A/D controller
handleAdIRQ
setChan
read
convert
A/D converter
x
y
z
Accelerometer
Message sender
Message sender
enqueue
Message queue
initTx
maybeDequeue
Transceiver controller
setTxBit
Radio transceiver
Push button
Button controller
before $t_{alarm}$
handleButtonIRQ
Push button
Re platform
transmit
buttonPressed
[comp. name] : component
[obj. name] : hardware object
[obj. name] : software object
[event name] : external event (asynchronous)
after $t_a$ before $t_b$
[event name] : internal event (asynchronous)
[event name] : internal event (synchronous)

Let us not turn to the components comprising the PAD application. All of them are pure software components. The *fall detector* is the only one of these three components that consists of more than one reactive object. The objects are *acceleration sampler*, periodically triggering sampling of acceleration; *acceleration analyzer*, triggered by the acceleration sensor when the acceleration values from all three channels have been collected; and *fall detection controller*, which controls the fall detection procedure.

The fall detection procedure starts with the fall detection controller being invoked by the acceleration analyzer upon detection of an impact. Once invoked, the fall detection controller schedules posture evaluation to be carried out after $t_{lag}$ *msec*. When this time period has elapsed the fall detection controller reads the person's posture from the acceleration analyzer. If he or she is lying down, the fall detection controller triggers the fall alarm sender component with the deadline $t_{alarm}$ *msec* (relative to the time when the fall has been detected). This deadline is inherited by the message sender component. The assistance alarm sender component is triggered in the same manner by the push button component.

The reason for separating acceleration analysis from the fall detection is that the analysis might need to be adjusted depending on the target platform (e.g. available hardware arithmetics). Further, this reduces the complexity of the fall detector.

# 5 Timing Requirements for the Acceleration Sensor Component

Sampling of acceleration is initiated by invoking the *sampleAcc* method of the acceleration sensor component. Sampling acceleration involves multiple steps that are performed by the A/D controller object (implemented in software), which interacts with the A/D converter object implemented in hardware.

The A/D converter is only capable of sampling acceleration on one channel at a time, so each conversion must be preceded by setting the channel (*setChan*) with a certain minimum delay between that and initiating conversion (*convert*). A completion of the conversion is signaled by raising an interrupt which is handled by the *adIRQHandler* method of the A/D controller object. In the model, this behavior can be encoded as follows:

```
adController adConverter accAnalyzer = class
    {- number of the channel to read from next time -}
    chan := 1
    {- a list of acceleration values -}
    values := []
    sampleAcc = before t_setChDl action
        after (t_setChDl + t_wait) initConversion
        {- set the channel -}
        adConverter.setChan chan
        {- update state so that next reading will be from the next channel -}
```

```
        chan := chan + 1
initConversion = before t_convDl action
        {- initiate conversion -}
        adConverter.convert

{- to be continued -}
```

Here we are using the notation from the Timber language that reflects the concepts of reactive objects and time-constrained reactions that our model is built upon. In Timber, communication between software objects ("internal event" in our model) is accomplished using messages that can be sent synchronously (with the caller waiting for the callee to return) and asynchronously (allowing the callee to execute concurrently with the caller). An asynchronous message can be postponed by offsetting its baseline relative to the baseline of the caller, which corresponds to the semantics of our model.

In the code above, a "class" is a definition of an object constructor; an "action" is a method that is to be invoked asynchronously, the "before"-notation defines the deadline (relative to the baseline), and the "after"-notation shifts the baseline of the invoked method relative to the baseline of the method being executed. Note that posting the message to the method convert with a delay of $(t_{setChDl} + t_{wait})$ ensures that at least $t_{wait}$ *msec* elapses between setting the channel and initiating conversion, as required by the hardware.

Once conversion has been completed, an interrupt is handled by the A/D controller object (and the time-stamp of the interrupt becomes the natural baseline for its *adIRQHandler* method):

```
{- continued from above -}

adIRQHandler = before t_adIRQDl action
        {- read the value -}
        val ← adConverter.read
        values := val:values
        {- deliver the values when all three have been collected -}
        if chan == 4 then
                chan := 1
                {- send acceleration values for processing -}
                accAnalyzer.consumeAcc values
        else
                after (sec 0) sampleAcc
```

Here the special construct "after 0" resets the baseline of the invoked method to the time when the message was posted. The permissible execution windows for the methods of the A/D controller object are illustrated in Fig. 9.

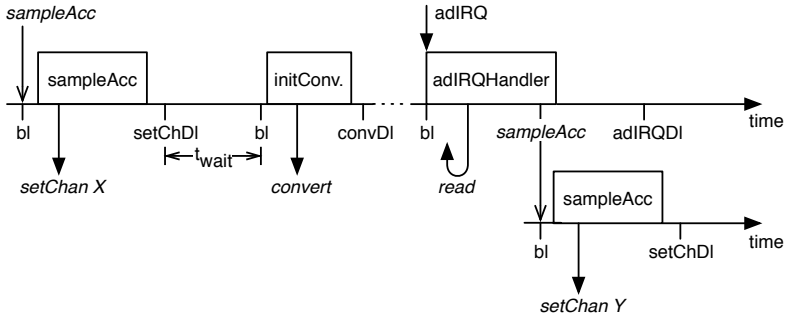



*Figure 9: Timing requirements for A/D conversion described as permissible execution windows for the methods of the A/D controller object.*

# 6 PAD Implementation

The aim of the implementation process is to realize the objects defined in the model (assuming that it is complete) in software and hardware, as defined by the object-level model. This involves implementing the software reactive objects in some programming language (which might require providing the infrastructure necessary for real-time execution on a hardware platform, e.g. an operating system supporting scheduling), and building a hardware platform from the identified hardware components. So far, only software part of the implementation has been completed, but we have checked that there exist hardware COTS (components-off-the-shelf) that correspond to each reactive object in the model that should be implemented in hardware.

The software must be implemented using a suitable programming language. We choose Timber [7–9] as programming language for the PAD software implementation which gives us many advantages compared to using standard programming languages. Timber fully supports the underlying modeling framework and therefore the reactive objects along with their timing constraints do not have to be translated into a separate programming model but can be described directly as Timber models, avoiding re-writing the model using different constructs and then verifying that the translation has preserved the essential properties of the system, such as timing properties.

## 6.1 Implementation in Timber

Each reactive object in the model of the PAD was encoded as such in Timber. To illustrate this we now present Timber code for the reactive objects that comprise the fall detector component: *acceleration sampler*, *acceleration analyzer*, and *fall detection controller*.

In the acceleration sampler object, the method *sample* triggers acceleration sampling by invoking the method *sampleAcc* of the acceleration sensor component. It also posts an asynchronous message to itself with a baseline delayed by $t_{period}$, which results in a

periodic sampling.

```
accSampler adController = class
    sample = before t_sampleDl action
        adController.sampleAcc
        after t_period sample
```

Once the acceleration values have been collected, the acceleration sensor component invokes the *consumeAcc* method of the fall detector component, which is realized as the *analyze* method of the acceleration analyzer object. This object interacts with the fall detection controller by invoking its method *impact* on detection of an impact:

```
accAnalyzer fallDetectionController = class
    {- state variable storing low-pass filtered acceleration data -}
    a := []
    analyze xyz = before t_analyzeDl action
        {- low-pass filter the y input (vertical acceleration) -}
        if {- look for impact in xyz input -} then
            fallDetectionController.impact
    getPosture = request
        if {- determine posture using filtered data -} then
            result LyingDown
        else
            result Upright
```

The synchronous method ("request" in Timber) *getPosture* returns the current posture of the person determined from the vertical acceleration. The result returned is encoded as one of two values, either *LyingDown* or *Upright*.

The partial Timber code of the fall detection controller is presented next:

```
fallDetectionController accAnalyzer alarmSender = class
    impact = action
        after t_lag evalPosture
    evalPosture = before t_evalDl action
        posture ← accAnalyzer.getPosture
        if posture == LyingDown then
            alarmSender.sendAlarm
```

According to the algorithm there should be a time delay between detection of an impact and checking the posture. In the model this is encoded as posting an asynchronous message to the method *getPosture* with the baseline delayed by $t_{lag}$ relative to the current baseline. The algorithm also states that posture evaluation should be re-scheduled relative to the latest impact event if additional impacts are detected.

### 6.2 Building an Executable

Before running the Timber model of the PAD as an executable on a target platform it needs to be compiled using the Timber compiler [9] and linked with the Timber run-time system for the target platform written in C [10]. The Timber runtime system provides EDF scheduling of messages and encodes interrupts coming from hardware as messages to handler objects (in our case, the push button and the A/D controller objects).

### 6.3 Optimizing for a lightweight microcontroller

Fall detection was implemented in conformance with the algorithm described in Section 3. The algorithm uses mathematical functions defined on the whole set of real numbers, so when implemented on a computer, especially a light-weight microcontroller with limited processing capabilities, it becomes an approximation. In fact, we have a tradeoff between precision of calculations and hardware requirements, where a more advanced microprocessor would be more expensive and presumably consume more power. In the implementation integer-only 16-bit data and operations were used, and multiplication was replaced by using a look-up table. The result of such approximation has to be verified.

## 7 PAD Verification

A traditional verification method is to use simulation. Unlike formal verification methods, simulation cannot produce any guarantees on system behavior. At the same time, simulation as opposed to testing allows to significantly lower design times and costs, by allowing to verify certain aspects of the design at an early stage of development. However, the main question that has to be asked is whether results of a simulation of a model are still valid for the final implementation of the system. Another key question is how it is possible to simulate timing properties (essential properties of any real-time system) when there is no specific hardware platform to measure WCETs on. We will try to answer these questions before we describe the simulations of the PAD that have been conducted.

The first problem is addressed in our approach by using the same formalism for modeling and for implementation. Indeed, as the model of the software only contains reactive objects, and an implementation in the Timber programming language is also based on them, the transition from a model to an implementation is no longer a translation. It should instead be viewed as filling in the model with more details, such as the actual code of the methods that have already been defined. Thus the properties of the model verified in simulation become the properties of the system, unless simulation relies on some assumptions not expressed in the model itself.

In general, the same is true about the timing properties, as permissible execution windows are defined in the model and are preserved in the implementation to be used at run-time (by the Timber kernel) to guide scheduling. However, there is an important

point to be made here. The permissible execution windows, defined for each reaction at system, component, and finally object level, are in fact timing specifications rather than timing properties; they define the multitude of allowed timing behaviors but do not in themselves guarantee that they will be adhered to at run-time. In conjunction with the Timber kernel, however, these timing specifications are guaranteed to be followed as long as the underlying hardware platform is fast enough to allow the software to meet all its deadlines. Thus simulation of the model (with timing specifications) aims to verify the validity of such specifications with respect to intended system behavior, and the question whether a particular platform allows for all deadlines to be met has to be studied separately based on WCETs and the chosen scheduling algorithm.

### Simulink-Based Timber Simulator

In the case of our system, the model is implemented in Timber, and the resulting code is compiled to C by the Timber compiler. This code can then be executed on a real hardware platform with an appropriate version of the Timber kernel; simulated on a PC under POSIX with Timber's POSIX run-time system; or simulated in Simulink [11] together with a specialized version of the Timber kernel. The significance of being able to run literally the same code both in simulations and on the real hardware platform cannot be overestimated; it allows us to verify the system's behavior and eliminate software bugs already in simulation.

Simulink is widely used in industry for modeling and simulation of various control systems, and numerous subject-specific simulators come with the ability to interface Simulink models. Being able to execute Timber code in Simulink allows us to easily create models of the environment for our embedded software. The specialized version of the Timber kernel provides communication between the Simulink world and the Timber world by emulating interrupts as input to the Timber system and output signals as output from it. It also schedules the execution of Timber objects in accordance with the defined permissible execution windows. Since the model does not represent the system running on any particular hardware platform, there are no execution times available to us and the virtual execution time of each reaction is set to zero. It is possible to schedule the execution of the reactions at any point within the permissible execution window; in our case, we chose to execute all reactions as early as possible, i.e. at their respective baselines. Note that simulation thus becomes an approximation of system behavior as the effect execution times on the actual scheduling is ignored, but the simulated behavior is one of those conforming to the system specification.

### Simulations of the PAD

The goal of Simulink simulations of the PAD was to eliminate possible design flaws and software bugs in the model and its Timber implementation as well as to verify that the system will operate as intended under realistic conditions. To this end, data recordings from simulated falls as well as ADL (activities of daily living) were used as input to the simulations. Apart from the actual execution times and the resulting scheduling, we also

ignore memory consumption.

It was not possible to utilize the whole set of recorded data because some of these recordings did not provide enough data for posture evaluation after an impact. In order to utilize as many recordings as possible, the parameter $t_{lag}$ (which determines the time interval between a detected impact and posture evaluation) was set to 0.9 sec. It was also necessary to exclude four recordings from the set of recorded ADL when the prototype device was accidentally misaligned.

The results of the simulations are shown in Table 1, with fall detection sensitivity and specificity calculated using Eqs. 1 and 2. These results can be compared to floating-point simulations of the algorithm in LabView (presented in a previous study [6] and in Table 1), conducted with the same recordings as input data. Note, however, that in the previous study absence of sufficient data after impact in certain recordings of falls was compensated for by using supplementary evaluation methods, and the recordings of ADL with misaligned prototype device were used after correcting the values of acceleration to counter the effect of misalignment. The necessary exclusion of these recordings from our simulations accounts for the differences in the number of simulations.

| | Simulink simulations of the Timber implementation. $t_{lag} = 0.9$ *sec* | | LabView simulations of the algorithm (taken from [6]). $t_{lag} = 2.0$ *sec* | |
|---|---|---|---|---|
| | No. of recordings | Specificity* / Sensitivity** | No. of recordings | Specificity* / Sensitivity** |
| **ADL (no falls)** | 160 | **100.0% *** | **164** | **100.0% *** |
| Fall forward (syncope) | 38 | 100.0% ** | 40 | 100.0% ** |
| Fall forward (tripping) | 37 | 100.0% ** | 40 | 100.0% ** |
| Fall backward (attempting to sit down) | 39 | 97.4% ** | 40 | 95.0% ** |
| Fall backward (from standing position) | 40 | 97.5% ** | 40 | 90.0% ** |
| Lateral fall | 39 | 100.0% ** | 40 | 100.0% ** |
| Fall from bed | 40 | 100.0% ** | 40 | 100.0% ** |
| **Falls, total** | **233** | **99.1% ** ** | **240** | **97.5% ** ** |

*Table 1: Simulation results.*

According to the results of the simulation, the PAD implementation preserves the sensitivity and specificity exhibited by the fall detection algorithm in previous simulations.

# 8 Related Work

The Rubus component model [12] shares many of the goals and implementation approaches with our methodology. It provides constructs for encapsulating software functions (called software circuits) and can be used to express interaction between them in single- and multi-node systems in terms of control flow as well as data flow. However, the software model has to be translated into executable threads, and the execution is controlled by a specialized run-time system such as Rubus-RTOS [13].

Another approach is Time-Triggered Architecture [14] which requires that each component is fully specified, including in the time domain, and can thus be verified separately from the rest of the system. Originally targeting distributed systems, this approach can easily be applied to componentization of a single-node system provided that components either do not share any resources, or utilize them according to a statically pre-defined schedule (including a shared CPU). This approach is very robust and can be used for safety-critical systems, but robustness comes at the cost of flexibility of the design and leads in most cases to a below-optimal utilization of resources.

Apart from well-developed approaches with well-defined semantics such as Rubus and TTA, there exist a number of other modeling frameworks and design tools that can be used for component-based development of real-time systems. Here we will only mention real-time synchronous languages (Esterel, SCADE, Lustre, etc. [15]); time-triggered languages such as Giotto [16]; the Ptolemy framework for assembly of concurrent components [17], which is particularly suitable for modeling distributed systems; and tools such as Rhapsody [18], ARTISAN Studio [19], and Rose-RT [20].

We should also mention work on specification of real-time systems, such as RT-UML [21] and MARTE [22]. In contrast to our work, these approaches offer numerous, often highly specialized ways to define timing properties, which are difficult to preserve (with consistent semantics) throughout the design process.

# 9 Conclusion

In this work we have presented a personal alarm device (PAD) developed using the component-based design methodology presented in Wiklander et al. [1] and summarized in Section 2 of this paper. The Timber implementation of the system was verified using a Simulink-based simulator. During simulation, the system operated according to its specification. The simulation also demonstrated that, even though the calculations were simplified in order to execute efficiently on a 16-bit platform, the ability to detect falls remained satisfactory. This makes it possible to utilize a lightweight microcontroller which in turn implies a lower power consumption, a smaller physical size of the device, and a lower price. These qualities are very important for portable systems and therefore they are also important for the future deployment of the system.

The case study demonstrates how our methodology allows us to model complex interaction between hardware and software, facilitating design of embedded systems. We have also demonstrated that complex timing behavior (e.g., due to interaction between

hardware and software) can be not only modeled but also preserved in the implementation.

Our component-based approach, used to partition this particular real-time system into reactive components, allowed us to specify and model system functionality as well as timing behavior at different abstraction levels. Thus the case study demonstrates the potential of our methodology to bring the benefits of classical component-based design (re-use of components, ease of system maintenance and modification, etc.) to the realm of embedded real-time systems.

Using Timber for implementation of the system allowed us to preserve the properties of the model by means of a seamless transition from model-based design to implementation. It also allowed us to simulate the actual implementation, when the same code can be used for simulation in a Simulink environment and on a target platform.

## 10 Acknowledgment

This work was supported in part by the Knowledge Foundation in Sweden under a research grant for the SAVE-IT project, the EU Interreg IV A North Programme (grant no. 304-15591-08), the ESIS project (European Regional Development Fund, grant no. 41732), and the EU SOCRADES project.

## References

[1] J. Wiklander, J. Eliasson, A. Kruglyak, P. Lindgren, and J. Nordlander, "Enabling component-based design for embedded real-time software," *Journal of Computers (JCP)*, 2010, In press.

[2] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black, "Reactive objects," in *Fifth IEEE Int. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2002, pp. 155–158.

[3] J. Nordlander, M. P. Jones, M. Carlsson, and J. Jonsson, "Programming with time-constrained reactions," Luleå University of Technology, Tech. Rep., 2005. [Online]. Available: http://pure.ltu.se/ws/fbspretrieve/441200

[4] A. Sangiovanni-Vincentelli, "Defining platform-based design," *EEDesign of EE-Times*, 2002.

[5] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design and Test of Computers*, vol. 18, no. 6, pp. 23–33, 2001.

[6] M. Kangas, I. Vikman, J. Wiklander, P. Lindgren, L. Nyberg, and T. Jämsä, "Sensitivity and specificity of fall detection in people aged 40 years and over," *Gait & Posture*, vol. 29, no. 4, pp. 571–574, 2009.

[7] P. Lindgren, J. Nordlander, L. Svensson, and J. Eriksson, "Time for Timber," Luleå University of Technology, Tech. Rep., 2005. [Online]. Available: http://pure.ltu.se/ws/fbspretrieve/299960

[8] M. Carlsson, J. Nordlander, and D. Kieburtz, "The semantic layers of Timber," in *First Asian Symp. on Programming Languages and Systems (APLAS)*, ser. Lecture Notes in Computer Science, vol. 2895. Berlin, Germany: Springer-Verlag, 2003, pp. 339–356.

[9] The Timber language homepage. [Online]. Available: http://www.timber-lang.org

[10] M. Kero, P. Lindgren, and J. Nordlander, "Timber as an RTOS for small embedded devices," in *Proc. of the First Workshop on Real-World Wireless Sensor Networks (REALWSN)*, 2005.

[11] Simulink homepage. [Online]. Available: http://www.mathworks.com/products/simulink

[12] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck, "The Rubus component model for resource constrained real-time systems," in *3rd Int. Symp. on Industrial Embedded Systems*, 2008, pp. 177–183.

[13] Arcticus Systems Homepage. [Online]. Available: http://www.arcticus-systems.com

[14] H. Kopetz, "Component-based design of large distributed real-time systems," *Control Engineering Practice*, vol. 6, no. 1, pp. 53–60, 1998.

[15] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[16] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003.

[17] Y. Zhao, J. Liu, and E. A. Lee, "A programming model for time-synchronized distributed real-time systems," in *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2007, pp. 259–268.

[18] E. Gery, D. Harel, and E. Palatshy, "Rhapsody: A complete life-cycle model-based development system," in *Proc. of the Third Int. Conf. on Integrated Formal Methods*, ser. Lecture Notes in Computer Science, vol. 2335. London, UK: Springer-Verlag, 2002, pp. 1–10.

[19] Artisan Software Tools. [Online]. Available: http://www.artisansw.com

[20] Rational Rose Technical Developer. [Online]. Available: http://www.ibm.com/software/rational

[21] S. Graf, I. Ober, and I. Ober, "A real-time profile for UML," *Int. J. on Software Tools for Technology Transfer*, vol. 8, no. 2, pp. 113–127, 2006.

[22] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard, "MARTE: Also an UML profile for modeling AADL applications," in *12:th IEEE Int. Conf. on Engineering Complex Computer Systems*, 2007, pp. 359–364.