A Reactive Approach to Component-Based Design of Resource-Constrained Embedded Systems

Jimmie Wiklander
A Reactive Approach to Component-Based Design of Resource-Constrained Embedded Systems

Jimmie Wiklander

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

Supervisors:
Professor Per Lindgren and Assoc. Professor Johan E. Carlson
Abstract

Embedded systems have become commonplace in today’s society and the number of functions and their complexity are steadily increasing. This can be attributed to the advances in the microprocessor technology and the continuous development 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.

Component-based design is an attractive approach for design of complex software systems. It reduces development time and cost and improves overall maintainability and robustness of the system. These benefits have been proven difficult to realize when designing software for embedded systems. In particular, it is a challenge to manage the extra-functional properties of such systems (e.g. timing).

This thesis presents a component-based software design methodology that enables constructing complex software for resource-constrained embedded systems, in particular systems with limited processing power and memory. The underlying component model is based on reactive objects, which are suitable for modeling embedded software. The component model along with the design methodology allows both functional and timing properties of a system model to be preserved during implementation process. Further, it enables the developer to offer platform-independent correctness for real-time systems, provided that the software can be scheduled on a given hardware platform.

An effective use of the methodology requires tools supporting it. An integrated development environment (IDE), REKO, has been developed and is presented in this thesis. REKO supports constructing system models graphically and generating C code from the model for execution on bare metal. Using C code makes it easier to integrate REKO with existing tool chains used in industry.

The thesis also includes a case study, in which REKO is used for designing a real-life system. The case study demonstrates the potential of the methodology to bridge the gap between state-of-the-art in component-based software design on the one hand, and the methods and tools currently used in the industry on the other.
# Contents

## Part I

<table>
<thead>
<tr>
<th>Chapter 1 – Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The reactive approach for developing software</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Scope</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Objectives</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Research questions</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Research work and method</td>
<td>6</td>
</tr>
<tr>
<td>1.6 Contributions</td>
<td>7</td>
</tr>
<tr>
<td>1.7 Thesis outline</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2 – Background</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Embedded software design</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Component-based design of embedded software</td>
<td>14</td>
</tr>
<tr>
<td>2.3 The reactive object model</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3 – Towards component-based design of embedded software</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Software design methodology</td>
<td>23</td>
</tr>
<tr>
<td>3.2 REKO - An IDE for development of embedded software</td>
<td>30</td>
</tr>
<tr>
<td>3.3 Developing embedded software for an autonomous car</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4 – Conclusions, related work, and future work</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Conclusions</td>
<td>35</td>
</tr>
<tr>
<td>4.2 Related work</td>
<td>36</td>
</tr>
<tr>
<td>4.3 Future work</td>
<td>37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5 – Summary of publications</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>39</td>
</tr>
<tr>
<td>5.2 Paper A</td>
<td>41</td>
</tr>
<tr>
<td>5.3 Paper B</td>
<td>41</td>
</tr>
<tr>
<td>5.4 Paper C</td>
<td>42</td>
</tr>
<tr>
<td>5.5 Paper D</td>
<td>43</td>
</tr>
<tr>
<td>5.6 Paper E</td>
<td>43</td>
</tr>
<tr>
<td>5.7 Paper F</td>
<td>44</td>
</tr>
</tbody>
</table>

References | 45 |
# Part II

**Paper A – Enabling component-based design for embedded real-time software**

1. Introduction ........................................... 51
2. Modeling Framework ................................ 53
3. Software Design Methodology ...................... 54
4. An Implementation Approach: The Timber Language . 60
5. An Example System: A Personal Alarm Device ..... 67
6. Related Work ........................................... 68
7. Conclusion ............................................. 72
8. Acknowledgment ....................................... 76

**Paper B – Personal alarm device: A case study in component-based design of embedded real-time software**

1. Introduction ........................................... 81
2. Methodology Overview ............................... 83
3. Personal Alarm Device (PAD) ....................... 88
4. PAD Design ............................................ 90
5. Timing Requirements for the Acceleration Sensor Component . 95
6. PAD Implementation ................................ 96
7. PAD Verification ...................................... 99
8. Related Work ......................................... 101
9. Conclusion ............................................. 103

**Paper C – An IDE for component-based design of embedded real-time software**

1. Introduction ........................................... 105
2. Modeling Framework ................................ 107
3. IDE for Embedded Real-Time Software .......... 108
4. Internal Data Format .................................. 111
5. Future Work .......................................... 113

**Paper D – Implementation of SRP-DM scheduling for embedded real-time software**

1. Introduction ........................................... 117
2. Background ............................................ 119
3. Translation of the CRO model ...................... 121
4. Example System ...................................... 123
5. Code Synthesis of Framework ...................... 125
6. Resource ceiling and Priority Extraction for CRO . 128
7. Kernel Design for efficient SRP DM scheduling of CRO . 130
8. Conclusion ............................................. 134
<table>
<thead>
<tr>
<th>Paper E – Utilizing an IDE for component-based design of embedded real-time software for an autonomous car</th>
<th>137</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction ..................................................................</td>
<td>139</td>
</tr>
<tr>
<td>2 IDE Overview ..................................................................</td>
<td>141</td>
</tr>
<tr>
<td>3 Related Work ..................................................................</td>
<td>143</td>
</tr>
<tr>
<td>4 Problem Statement ......................................................</td>
<td>145</td>
</tr>
<tr>
<td>5 Conceptual Model ..........................................................</td>
<td>146</td>
</tr>
<tr>
<td>6 Target Platform for the Software Implementation ................</td>
<td>147</td>
</tr>
<tr>
<td>7 Software Design for the Autonomous Car ...........................</td>
<td>149</td>
</tr>
<tr>
<td>8 Program analysis ............................................................</td>
<td>162</td>
</tr>
<tr>
<td>9 Conclusion .................................................................</td>
<td>163</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Paper F – Towards executing TinyOS models under hard real-time constraints</th>
<th>167</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction ..................................................................................</td>
<td>169</td>
</tr>
<tr>
<td>2 TinyOS Concurrency and Execution Model ........................................</td>
<td>171</td>
</tr>
<tr>
<td>3 CRC Concurrency and Execution Model ..........................................</td>
<td>174</td>
</tr>
<tr>
<td>4 TinyOS to CRC mapping ..................................................................</td>
<td>176</td>
</tr>
<tr>
<td>5 Conclusions ..............................................................................</td>
<td>179</td>
</tr>
</tbody>
</table>
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 Anders Högström for his encouragement and support as well as Jerry Lindblom for sharing his experience with me.

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
Västerås, October 2011
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 and complex software implementations. However, using the traditional methods and tools for development of complex software can be tedious and error-prone. Hence, new approaches for the specification, design, and development of embedded software are needed.

An attractive approach for managing software complexity is component-based design. The intuition behind it is to construct software systems by composing ready-made parts, or components. However, adoption of this approach in development of embedded software has been significantly slower than in development of general-purpose software. It can be argued that this is due to the long-established concept of separating computation (software) from physicality (platform and environment) [1]. This concept is central to software development and consequently to the existing component-based approaches as well. For this reason, the programmer can build applications focusing on accessing and processing data, not having to pay attention to the details of the underlying hardware platform or environment. Unfortunately, the separation between computation and physicality makes it very hard to address extra-functional requirements that depend on the underlying hardware or environment, such as power consumption or timing behavior. For embedded systems, the extra-functional requirements are often as important as the functional requirements. For example, the majority of embedded systems can be viewed as real-time systems. For real-time systems, 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 the computed values as such. Hence, the key to embedded systems design is to control the interplay between computations and physical constraints in such a manner that all requirements are met [1].

Based on this reasoning, any software design approach for embedded systems should try to bring computation and physicality closer together. In a component-based design approach, this can be achieved by using a model that supports expressing not only
Introduction

functional, but also extra-functional properties such as timing requirements. The model
must also support independent definition of components, that is, the properties defined
locally for a component should not be affected by external factors.

It is becoming increasingly common for embedded systems to perform multiple tasks
at the same time, often with real-time requirements. While hardware is inherently par-
allel, software is not. A parallel behavior of software can be emulated on a single CPU
system by switching between tasks; such execution is called concurrent. Concurrency is
a generic term that includes true parallel execution as well as emulation of such behav-
ior. Any model targeting software development for embedded systems should support
expressing concurrency. Further, the tasks must be able to interact with each other. This
gives rise to the problems of synchronization and state protection. These issues must be
addressed by the model.

The reactive object model [2] supports declaration of both functionality and timing
requirements. It is based on reactive objects which can be seen as units of concurrent
execution with an inherent state protection. The model supports both synchronous and
asynchronous communication in the form of messages. Hence, the objects supports spec-
ifying complex concurrent behaviors that includes synchronization and state protection.
Accordingly, the reactive object model is suitable for specifying both the functionality
and timing behavior of embedded real-time systems.

This thesis demonstrates how to utilize the reactive object model in a component-
based software design methodology for embedded systems. Special emphasis is given to
developing software for resource-constrained embedded systems with real-time require-
ments, in particular, systems with limited processing power and memory.

1.1 The reactive approach for developing software

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; a
reaction may produce additional events on its own, creating a chain of reactions. Stating
that the events are discrete, i.e. stating that each event occurs at 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 Section 2.3).

This approach is adopted by the reactive object model [2, 3]. A reactive object has a
state and one or several methods and the state is only accessible via the object’s methods.
Moreover, each object has a provided interface (the methods that can be invoked by other
objects) and a required interface (the methods in the object’s environment that it may
invoke). In the model, an output port in the required interface of an object can be linked
to an input port in the provided interface of another object, creating a communication
path between two different objects. A method’s code can perform computations on local
variables, read/mutate the object’s state, and invoke a method of the same or another object by sending a message to it. A more detailed description of the reactive object model is given in Section 2.3.

The desired functionality of an embedded system can be specified by combining reactive objects into a model of the system. However, in order to execute the model on a hardware platform, the objects must be implemented in a programming language (e.g. Timber [4,5], which fully supports the reactive object model). Execution on a hardware platform also requires infrastructure in the form of a lightweight kernel supporting scheduling of method execution and message passing between components.

1.2 Scope

Throughout the history there has been a gap between the theoretical and practical approaches in the development of software. Researchers tend to focus on the theoretical aspects and not on tool development. At the same time, the industry itself is not keen on adapting new software design approaches. One reason for this is that there is always a cost associated with changing already established methods and tools. Hence, in order for the company to “bite the bullet”, the benefits of the new approach must be considered substantial. In the academia, the benefits of a particular design approach are often presented through logical reasoning and theoretical results. In the industry, the benefits are better demonstrated through use cases involving real design problems.

As mentioned earlier, this thesis demonstrates how to utilize the reactive object model in a component-based software design methodology for resource-constrained embedded systems. However, the intention of this thesis is to bridge the gap between the proposed methodology on one hand, and the methods and tools currently used in the industry on the other. For that reason, special emphasis have been placed on developing a methodology that is attractive for use in industry. For example, the component model supports using C-based constructs for coding methods; the model also supports generating intermediate C code (for a substantial part of industry, C code is the de facto standard for programming embedded systems). Tools have been developed to facilitate the use of the methodology. For example, one of the tools enables constructing system models graphically.

As mentioned earlier, this thesis focuses on developing software for resource-constrained embedded systems with real-time requirements. For these systems it is crucial to ensure that the system will not run out of memory or fail to meet its deadline at run-time. By adapting existing resource analysis techniques to our model it is possible to statically predict the memory and processing requirements at design time. For this reason, the proposed component model has been restricted. In particular, it only supports static state and communication structures. As a further limitation of scope, this thesis only considers software development for single processor systems.
1.3 Objectives

This thesis has two main objectives. The first is to develop a component-based software design methodology, which should enable constructing complex software for resource-constrained embedded systems. This methodology should be based on the reactive object model. The model should support:

- specifying both functional behavior and timing requirements of an embedded system,
- specifying methods (i.e. specifying the functional behavior) using C-based constructs,
- generating C code that preserves the timing constraints.

The model should only support static state and communication structures. However, the methodology is not defined solely by the component model. The following elements are central to the methodology:

- a component model,
- a way to construct system models graphically,
- a way to generate C code from the model,
- a way to perform statical analysis of schedulability and memory requirements,
- a design process that details how to utilize the component model to develop software.

More specifically, for resource efficient execution onto lightweight target platforms, the design methodology and tools should support stack resource policy (SRP) scheduling and resource management [6]. Moreover, the purpose of the design process is to ensure a systematic development of software in accordance with the proposed component-based approach.

The second objective of this thesis is to develop tools to support the methodology. More specifically, the following tools should be developed:

- a tool for constructing system models graphically and for specifying method code,
- a tool for generating code from the model,
- a tool for performing statical analysis on the generated code.

The overall goal is to combine all the tools into an integrated development environment (IDE). Existing tool chains such as the GNU tool chain [7] can be used to compile the generated C code into an executable for a specific platform. Moreover, a run-time system must be provided at compile time in order to support SRP-based real-time scheduling and resource management.

1.4 Research questions

As mentioned earlier, one of the main objectives of this thesis is to develop a component-based software design methodology. However, there are some underlying issues that need
to be addressed in order to accomplish this. The research questions of this thesis are formulated accordingly, as presented next.

The reactive object model is a good starting point for formulating a component model. However, there are some issues that need to be addressed in order to define the component model. First of all, we need to devise a component definition based on reactive objects. This also requires defining composition of components as well as identifying how to verify composability. Hence, the first research question of this thesis can be stated as:

**Q1: How to define a component model using reactive objects?**

One intention of this thesis is that the methodology should be attractive for use in industry. For this reason, the objectives specify that the methodology should provide a way to generate C code from the model. This requires a solid semantics or at least an unambiguous translation from modeling constructs into C code. However, the resource constraints of the target systems impose certain requirements on code generation. Firstly, special emphasis should be placed on limiting the memory and CPU overhead. Secondly, the generated code should support static analysis targeting SRP-based scheduling. The third research question is stated accordingly:

**Q2: How to generate code from the component model so that the resource requirements can be predicted?**

As a way to further promote the use of the methodology it should support constructing models of embedded systems graphically. Graphical visualization is a powerful tool for supporting understanding of complex models. However, this requires a graphical representation of the model, which leads to the third research question:

**Q3: How should the component model be represented graphically?**

1.5 Research work and method

The research work has been conducted in joint research projects with industrial partners. This has given valuable insights regarding the challenges of designing embedded software as well as hands-on experience with the methods and tools used for addressing these challenges. Another aspect of the research work has been to answer the research questions by studying state-of-the-art approaches and models for design of embedded systems. These have been studied by reading papers in the area and taking relevant courses. Special attention has been payed to the component-based design approach and the reactive object model. Based on the findings, a methodology have been proposed. The methodology enables constructing embedded software in accordance with the component-based approach. An integrated development environment (IDE) supporting the methodology has been developed. The main objective of the IDE is to support the design of real-life embedded software applications. Further, the IDE should support integration with tools used in the industry. The IDE and the underlying methodology were tested in case studies involving real-life embedded applications. Quality has been assured by presenting the research results in peer-reviewed journals and conferences.
1.6 Contributions

In this thesis, the temporal properties are included as a central part of both the component model and the design process. The temporal properties in the model become a specification that the final system should conform to and this specification is not affected by the composition process and is preserved throughout the design process all the way to implementation. This is fundamentally different from the standard procedure which treats time separately and where the temporal properties are analyzed after the system is developed. More specifically, the research work have resulted in:

- A component model for embedded systems based on time-constrained reactions and concurrent reactive components that facilitates defining both functional and temporal properties of the system (joint work with Andrey Kruglyak).

- A software design process for embedded real-time systems that specifies how to define and preserve both functional and temporal properties throughout the design process.

- REKO – An IDE for development of embedded real-time software that enables constructing a system model using components and that allows generation of C code that can be executed efficiently on resource-constrained embedded systems (code generation and system analysis is joint work with Johan Eriksson).

The thesis also includes a case study, in which REKO is used for designing the software for a real-life system, an autonomous car. The case study demonstrates the potential of the proposed methodology to bridge the gap between state-of-the-art in component-based software design on the one hand, and the methods and tools currently used in the industry on the other.

1.7 Thesis outline

This thesis is divided into two parts. This introduction chapter is the first chapter of Part I. The remaining chapters of Part I are organized as follows:

- Chapter 2 provides a background related to embedded software design, component-based design of embedded software, and the reactive object model.

- Chapter 3 complements the papers in Part II by providing an overview of the research results, namely the component model, the software design process, and the REKO IDE. Moreover, it presents the aforementioned case study in which REKO is used for designing the software for an autonomous car.

- Chapter 4 provides conclusions for the thesis along with related work and future work.
• **Chapter 5** provides an introduction to the papers included in Part II as well as a short summary of each paper. The introduction gives an account of the differences between the various versions of the component model found in the papers.

The second part (Part II) includes the papers on which this thesis is based. All papers have been reformatted from their original layout to match the layout of this thesis.
Chapter 2

Background

This chapter provides an introduction to embedded software design. Moreover, it presents challenges and research approaches related to component-based design of embedded software. Further, it gives an overview of the reactive object model which is suitable for modeling embedded software.

2.1 Embedded software design

An embedded system is a computing system that has been designed to provide a specific function, often as part of a larger system. Their applications can be found in many different domains, such as telecommunication, automotive, medical equipment, aerospace, process industry, consumer electronics, etc. An embedded system consists of hardware and software and apart from performing computations, it interacts with its environment. In contrast to general purpose computing systems, embedded systems often manifest a tight integration between software and hardware. The environment includes everything that is considered external to the embedded system, e.g. the physical world or other systems. Their behavior can be seen as reactive, i.e. its behavior can be described in terms of reactions to events that originate either from the system’s environment or internally within the system. The computing platform of an embedded system is typically constrained with respect to processing power and memory. Interaction with the environment is provided via peripherals, such as sensors or communication interfaces. The peripherals may be integrated as separate hardware parts. However, embedded systems are often based on a configurable computing platform, such as a microcontroller or a digital signal processor (DSP). In these platforms, the processing core alongside peripherals are integrated on a single chip. The physical hardware of an embedded system (i.e. the computing platform, peripherals, electronic circuits, etc.) is often denoted hardware platform. Embedded systems are typically designed to provide a specific function, often as part of a larger system. Hence, they are often subject to strong implementation requirements which, in turn, means that they must be heavily optimized with respect to processing power, memory size, physical size, cost, power consumption etc. The major-
ity of embedded systems can be viewed as real-time systems, i.e. systems in which the 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 the computed value as such.

Designing software for embedded systems is complicated by such factors as the tight integration between software and hardware, scarceness of available resources, and hard real-time requirements. Moreover, the complexity and size of embedded software is steadily increasing. For example, the embedded software for a modern premium car may comprise nearly 100 million lines of code [8]. The increasing complexity of embedded software calls for a new, more efficient design approach. An attractive approach for managing software complexity is component-based design. Next section presents challenges and research approaches related to component-based design of embedded software. The remainder of this section presents existing design practices for embedded software and discusses two important properties often associated with embedded software, namely, concurrency and real-time.

2.1.1 Existing design practices

There are many different design approaches and most of them are top-down. However, they are usually based on an understanding of what is supported at the lower levels. The design process typically includes a series of transformations (which may include iterations between the different stages) from requirement specification to the executable code. In general, the following stages are included in the design process [9]:

- requirement specification,
- architectural design,
- detailed design,
- implementation,
- testing.

The input to the design process is the product specification which originates from the client commissioning the system. The product specification is often an informal specification of the desired behavior of the system (it is usually written in natural language). The first stage is devoted to specifying all the requirements that needs to be considered when constructing the system. This includes specifying a conceptual model of the system, i.e. determining how to utilize the hardware to get the desired system behavior (for real-time systems the temporal behavior is as important as the functional behavior). The second stage is devoted to defining the structure of the software system. The structure is typically expressed using some kind of model; the model provides an abstract view of the system containing (in theory) the relevant properties and features. Another technique often used together with abstraction is decomposition. Decomposition means partitioning the system into smaller parts iteratively to get more manageable parts. This is often supported by the models used. The third stage is devoted to detailing the content of the parts identified in the architectural design stage. From the detailed design it should
be possible to implement the software and the output from the implementation stage is
the executable code. Typically, this involves implementing the entities of the software
architecture (the model) using some programming language and compiling the code for
execution on a specific platform. The aim of the final stage, testing, is to verify that the
behavior of the system is correct and that all the requirements are met.

In general, different tools are used at different stages of the design process. For
example, a requirement management tool may be used for managing the requirement
specification and the implementation stage may require tools for writing, compiling, and
debugging the software. The tools used for developing the actual code are often bundled
together into a single application with a single graphical user interface. This type of
application is known as an integrated development environment (IDE). An IDE normally
includes:

- a source code editor,
- a compiler and/or an interpreter,
- build automation tools,
- a debugger.

The IDE typically supports a specific programming language. For embedded systems,
the IDE is often provided by the hardware platform vendor (e.g. AVR Studio from
Atmel [10] or MPLAB IDE from Microchip Technology [11]). Its purpose is to simplify
the development of software for the hardware platforms supported by the IDE (i.e. the
hardware platforms included in the product range of the vendor). For instance, the IDE
may include software libraries providing typical functionality, such as interaction with
peripherals or math functions optimized for the target platform.

There are also many open source tools that are used extensively such as the tool chain
from the GNU Project [7] (which includes the widely used tools gcc and gdb), and the
Eclipse IDE [12].

2.1.2 Concurrency and real-time

Concurrency is inherent in hardware and is unavoidable in more complex embedded soft-
ware systems that have to perform multiple tasks (react to multiple events) at the same
time. The conventional view of a concurrent program is that it consists of a number of
sequential processes that may be executed in parallel. Further, concurrent programs rely
on synchronization and communication between processes. There are different methods
for constructing concurrent programs. One method is to use a programming language
with an inherent support for defining concurrent programs such as Ada [13] or Erlang [14].
Another method is to define the system in a model that supports the notion of concurre-
cy. The model can then be translated into a programming language, either manually
or automatically.

Most methods (and languages) require some kind of run-time system for executing
the resulting software. The main responsibility of the run-time system is to operate as
a scheduler, i.e. to manage how the processes are executed on the CPU. For real-time
applications, the scheduler must ensure that the processes can meet their deadlines. In real-time systems, the scheduler is usually part of what is often referred to as a real-time operating system (RTOS). There are many real-time operating systems available for embedded systems, both commercial and open source (see [15]).

2.2 Component-based design of embedded software

The intuition behind component-based design is to construct (software) systems by composing ready-made parts (or components). This is an attractive approach for managing software complexity and it is seen to maximize reuse, improve quality and to achieve shorter time to market. However, adoption of this approach in development of embedded software has been significantly slower than in development of general-purpose software. Even though many of the challenges relating to component-based design of embedded software have already been addressed by researchers, few of the suggested research approaches have achieved widespread use in the industry.

2.2.1 Challenges

We rely on embedded systems in our everyday life and in many cases the operations performed by embedded systems are safety-critical. A safety-critical system may pose a potential threat to human health in the case of a malfunction. Therefore, special measures have to be taken in order to guarantee that the system behaves in accordance with its specification. Today, the system specification is often written in a natural language and a common method for verifying the behavior of the system is to rigorously test the constructed system against the specification. There are several problems with this approach. First of all, the cost of correcting the design increases exponentially throughout the development process [16]. Therefore, the system behavior should be verified at various points throughout the development process instead of just in the final testing stage. Secondly, writing a system specification in a natural language can be cumbersome. Hence, the specification is often incomplete which, in turn, complicates testing. Finally, most embedded systems are too complex to allow verifying their behavior through testing. This calls for alternative methods for the specification and verification of embedded software.

A potential benefit of component-based design is the possibility to re-use components in future designs. Therefore, it makes sense to try to anticipate how the components can be utilized in future designs already at design time and design them accordingly. Then, at a later stage, individually constructed components can be composed with other components into new software systems. However, a major challenge relating to component composition is how to ensure compatibility between components. By component compatibility we mean that the components should be able to interact with each other in a meaningful way. Consequently, the model should provide a mechanism for verifying composability.

Any component-based approach rely on an underlying component model that is used for defining the functionality of the system. However, embedded systems manifest a tight
integration between functionality implemented in software and functionality of hardware parts. Hence, the component model should support describing the functionality of software as well as hardware functionality (or at least support describing the interaction with hardware). This also applies to the implementation techniques and tools that should be provided in order to support a systematic development of software in accordance with the approach.

When developing software for general purpose computing systems the emphasis is usually on defining the functional characteristics of the system, that is, what the system is supposed to do. For embedded systems, the extra-functional characteristics are often critical for proper system operation. In fact, for embedded systems the extra-functional requirements are often as important as the functional requirements themselves. As a result, it is of vital importance that these characteristics can be predicted at design-time. This is perhaps even more important in the realm of the component-based approach where systems are constructed from ready-made components. In this context it is desirable to predict the extra-functional properties not only for individual components but for the resulting component composition. For embedded systems this means predicting aspects such as timing behavior or memory consumption.

### 2.2.2 Approaches targeting embedded systems

There are many component-based approaches aiming to provide solutions for the aforementioned challenges. Common for these approaches is that they rely on an underlying component model. Hence, there are many different component models targeting different domains (such as AUTOSAR [17] that targets the automotive domain and Koala [18] that targets the consumer electronics domain). Some of the approaches that targets component-based design of (resource-constrained) embedded systems are presented next.

**Ambient RT**

AmbientRT [19] is a real-time operating system for embedded devices with resource-constraints in terms of memory, processing, and energy resources. It supports real-time preemptive EDF scheduling [20] on systems with scarce resources. Moreover, task priorities do not have to be assigned to task at beforehand, instead they are dynamically assigned based on the timing properties and the point in time. This approach does not offer any tools for composing components to a software model. However, it supports loading blocks of precompiled software and executing the blocks dynamically. AmbientRT also supports the data centric architecture [21] which means that the components of an embedded application can be enabled or disabled, or mutually rearranged. Moreover, the connections between components are centrally coordinated data streams. Hence, the architecture can be reconfigured after it is deployed and not only at compile time. In AmbientRT protection of shared resources is enforced by the kernel without requiring the programmer to manually infer explicit monitors. Instead, all the programmer has to do, is to list the resources that are used by a particular task.
COMDES-II

COMDES-II (COMponent-based design of software for Distributed Embedded Systems) is a component-based software framework supporting development of distributed embedded control systems with real-time constraints [22]. It defines two modeling levels: system level and actor level. At the system level, the system is modeled in terms of a network of communicating actors; they communicate by sending labeled messages. An actor is a software artifact containing multiple I/O drivers and a single actor task (execution thread). The I/O drivers handle the interaction with the corresponding network or physical units and communicate locally with the actor through signals. The signals from the I/O drivers (control, data) are processed by the actor tasks. The functional behavior of a task is specified by a composition of function block instances. COMDES-II defines different kinds of function blocks that offers various kinds of system functionality (e.g. a state machine).

The computational model of COMDES-II follows a split-phase execution pattern as described next. An actor is activated by an execution trigger (periodic or sporadic event). When the kernel receives a triggering event, it notifies the corresponding actor that it is eligible for activation. At activation, the input drivers acquire the input signals. The task may start processing the data after being released by the execution trigger. Note that in order for the task to start execution, it has to become the highest priority task among the released/preempted tasks. The processed data are sent to the output drivers that generates the output signals. However, the output drivers are not activated until the actor deadline expires. This model ensures that the actor output will be free from jitter as long as the actor tasks can finish their operations before their deadline expires (this holds even in the case when actor tasks are preempted).

The underlying real-time kernel of COMDES-II implements fixed-priority timed multitasking scheduling. Further, a development toolset designed to supports embedded control system development in accordance with the COMDES framework has been presented [23].

ProCom

The ProCom [24] component model targets the development of control-intensive distributed embedded systems. It defines a two-layer component model. At the upper layer, called ProSys, models are constructed by composing concurrent architectural elements, called subsystems. At the lower layer, called ProSave, the subsystems are internally modeled in terms of functional components. There are two kinds of subsystems: primitive subsystems and composite subsystems. The primitive subsystems constitute the smallest modeling elements in ProSys. Such elements are either a subsystem originating from the ProSave layer, or a non-decomposable unit of composition (e.g. legacy code). The composite subsystems support constructing hierarchical models by composing both primitive and composite subsystems. Each subsystem may define typed input and output ports that are used for communication with other subsystems (they communicate by sending messages). Message ports are connected via message channels.
At the ProSave layer, single subsystems can be created by composing hierarchically structured functional components. A functional component provides an abstraction of tasks and control loops. The subsystems defined at this level typically interact with the system environment by reading sensor data and controlling actuators. The ProSave component model is based on the pipes-and-filters architectural style. It distinguishes between two kinds of interaction, control flow and data flow. Each ProSave component may encapsulate one or several services. The service is defined by an input port group that defines the triggers and data required for using the service. Further the service defines a set of output port groups for delivering the data produced by the service. ProSave components are passive in the sense that they do not initiate activities on their own. However, a component can switch to an active state if it receives a trigger signal on its input trigger port. After being activated, the component first reads (atomically) the data input ports. Then, it performs computations on the data and sends the results to its output ports. It must write to all its output groups (only once) before switching back to its passive state. Input data ports can receive data when the component is in its active state. However, the data will not be used until the following component activation. Hence, the model ensures that once a service has been activated it is functionally independent from other components that execute concurrently.

The implementation for a functional component may be given by programming code (with services implemented as non-suspending C functions), or it may be provided by the programming code of inner components (these have C implementations at some level). A construct called connection is used for connecting inner components. Further, there are connectors that can be placed in between components in order to acquire additional control over the data and control flow. This setting makes it possible to explicitly specify and then analyze the control flow, timing properties, and system performance [24].

**CORBA/e (CORBA for embedded)**

Real-Time CORBA [25, 26] is a standard defined by the Object Management Group (OMG) for development of distributed real-time systems. However, the relatively extensive resource demands of CORBA makes it unfitting for development of embedded systems. As an attempt to overcome this problem, OMG have fairly recently (2008) defined two new profiles (Compact profile and Micro Profile) under the banner CORBA/e (CORBA for embedded)\(^1\) The CORBA/e Compact Profile merges key features from CORBA and Real-time CORBA into a compact middleware package. Applications developed in accordance with this profile can be deployed on 32-bit platforms running a standard Real-time Operating System (RTOS). The CORBA/e Micro Profile is a subset of the Compact Profile targeting resource-constrained platforms such as low-powered microprocessors or DSPs. However, since this is a fairly new standard it remains to be seen whether it will become accepted in the domain of embedded systems.

---

1 An earlier example of a OMG standard targeting embedded systems is minimumCORBA. It can be seen as a cut-down version of CORBA and it can be combined with Real-Time CORBA to enable development for distributed embedded real-time systems [27, 28]. This approach has not achieved widespread use.
Rubus

The Rubus Component Model (RCM) [29,30] targets development of resource-constrained embedded systems. In particular, it supports modeling single node embedded systems. In RCM, the smallest architectural element is called a Software Circuit (SWC). It is used for encapsulation of software functions. Each SWC has a state and it supports interaction with other SWCs through the use of ports. The RCM also supports creating hierarchical models through the use of a separate architectural element, called assembly. An assembly is used for encapsulating several SWCs and like SWCs it uses ports for interaction. The model distinguishes between two kinds of interaction, control flow and data flow. More specifically, the execution semantics of a software component is defined as [30]:

1. Upon triggering, read data on data in-ports,
2. Execute the function,
3. Write data on data out-ports,
4. Activate the output trigger.

Control flow (or triggering events) includes interrupts, internal periodic clocks, and internal and external events.

The architectural elements can be composed into a model of a particular system. The functions encapsulated by the SWCs can then be mapped to run-time entities (tasks). Each task corresponds to an external event trigger and its reaction chain in terms of SWCs and connections between them. Reaction chains can also be triggered by clocks. Such reaction chains are allocated to automatically generated static schedule. The schedule fulfills the precedence order and timing requirements for the reaction chain.

Different optimization criteria (e.g. minimizing memory usage) can be applied to the mapping of SWCs to tasks and to the allocation of reaction chains to the static schedule. Further, an efficient use of memory is ensured by the run-time system since it supports executing all tasks on a shared stack. The RCM is integrated to the commercially available development environment Rubus-ICE (Integrated Component development Environment) [31].

TinyOS

TinyOS [32] is an open source component-based operating system targeting wireless sensor networks. It is not an OS in the traditional sense, it is a framework with an underlying component model that can be used to build an application-specific OS into each embedded software application. It targets building concurrent, reactive applications for resource-constrained platforms. The resulting TinyOS application integrates its own application specific OS. The underlying component model is supported by the nesC language [33] which is a dialect of C.

In TinyOS, each component encapsulates a specific set of services, specified by interfaces. The components are connected using a wiring specification that defines the complete set of components that the application uses. The component interfaces describe how to interact with the component. Components can be of two kinds: modules
2.3. The reactive object model

Modules are the primitive components that provide the implementation in terms of nesC code. Each module has state variables and data buffers that only itself can access. Configurations on the other hand may encapsulate a number of connected modules and additional configurations. Hence, they support constructing hierarchical models. A TinyOS application is defined by a top-level configuration that encapsulates all the components of the application.

The execution model in TinyOS is based on run-to-completion tasks and event handlers (event handlers are triggered by hardware). The notions of tasks are included in the nesC language and a program can post a task to the scheduler. The scheduler may execute the tasks in any order but must obey the run-to-completion criteria. The standard TinyOS scheduler is a FIFO scheduler but others have been implemented (such as earliest-deadline first). Tasks are atomic with respect to each other, however, they are not atomic with respect to interrupt handlers or to commands and events they invoke. This may result in race condition. Therefore, in order to facilitate the detection of race conditions, TinyOS distinguishes between synchronous code (SC) and asynchronous code (AC). Synchronous code is only reachable from tasks and asynchronous code is reachable from at least one interrupt handler.

There is no risk of race conditions among tasks (a task may not preempt another task). However, there is still the possibility of race conditions between SC and AC, as well as between AC and AC. In general, there is always the possibility of race conditions when updating a shared state that is reachable from AC. However, in TinyOS the race conditions can be avoided by enclosing the state updates inside atomic sections. An atomic section guarantees that its content is executed just as it would execute if it was the only process in the system.

2.3 The reactive object model

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. 2.1); it is denoted as after \( t_{\text{after}} \) before \( t_{\text{before}} \), doSmth. Here \( t_{\text{after}} \) is the baseline offset (period of time between the triggering event and the baseline), \( t_{\text{before}} \) is the period of time between the baseline and the deadline, and doSmth is the invoked procedure. A reaction with a permissible execution window defined for it will be called a time-constrained reaction [2].

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].

**Reactive objects**

Each object has a state and one or several methods and the state is only accessible via the object’s methods. The object is reactive in the sense that it reacts to an incoming event by executing one of its methods. Each object has a provided interface (the methods that can be invoked by other objects) and a required interface (the methods in the object’s environment that it may invoke). In the model, an output port in the required interface of an object can be linked to an input port in the provided interface of another object, creating a communication path between two different objects. Multiple output ports can be connected to a single input port. Each object is either idle (maintaining its state) or executes a method. Methods execute run-to-end, that is, once a method has started execution, no other method of the same object may preempt it. However, any two methods of different objects may execute concurrently. A method’s code can perform computations on local variables, read/mutate the object’s state, and invoke a method of the same or another object by sending a message to it. Messages sent between objects can be of two kinds: asynchronous, which are executed concurrently with the caller and can be delayed by a certain amount of time, and synchronous, with the caller blocked until the invoked method completes execution, optionally returning a value (such methods cannot be delayed). The permissible execution window of an asynchronous message can be either inherited or explicitly specified in the code relatively to the caller’s baseline; in either case, it is viewed in the model as a separate reaction. A synchronous message, on the other hand, always inherits the caller’s time constraints and is viewed as a part of the original time-constrained reaction. Both asynchronous and synchronous messages can carry data (the values of arguments of the invoked methods).

**Classes and Instances**

Every object has a class definition from which instances can be created at run-time. The class definition can be defined at the top level or locally within another class; in the latter case, it can only be used inside that class and the classes that are hierarchically encompassed by the class. A class definition defines the object’s state variables, methods, and the provided and required interfaces of the object. In the method code we may:
2.3. The reactive object model

1. define local variables,
2. read/update the object’s state,
3. perform calculations on the object’s state, method argument, and local variables,
4. invoke an asynchronous method (see Fig. 2.1) of this object or of another object using the name of an output port or a port of the provided interface of an encapsulated object: \( \text{ASYNC} \text{(PortName, BaselineOffset, RelativeDeadline, Argument)} \),
5. invoke a synchronous method of another object using the name of an output port or a port of the provided interface of an encapsulated object: \( \text{SYNC} \text{(PortName, Argument)} \)
6. return a value (only meaningful if the method is synchronous).

To conclude, the reactive object model supports declaration of both functionality and timing requirements. It is based on reactive objects which can be seen as units of concurrent execution with an inherent state protection. The model supports both synchronous and asynchronous communication in the form of messages. Hence, the objects support specifying complex concurrent behaviors that includes synchronization and state protection. Accordingly, the reactive object model is suitable for specifying both the functionality and timing behavior of embedded real-time systems.
This chapter provides an overview of the research results of this thesis. These results are part of the component-based software design methodology and the integrated development environment (IDE) presented next. Moreover, this chapter presents a case study in which the IDE is used for developing the software for a real-life system, an autonomous car.

3.1 Software design methodology

The proposed methodology supports development of software for resource-constrained embedded systems. It provides: a component model, a code generation framework, a graphical representation of the model, and stack resource policy (SRP) scheduling and resource management.

3.1.1 Component model

The developed component model is based on the reactive object model (see Section 2.3). It supports constructing a hierarchical model of a system by composing *concurrent reactive components* (CRC). Each component may have a state, one or several methods and it may encapsulate a number of other components, creating a hierarchical structure. It is reactive in the sense that it reacts to an incoming event by executing one of its methods or invoking a method of an underlying component. Each CRC has a *provided interface* (the methods, or input ports, of the component that can be invoked by other components) and a *required interface* (the methods, or output ports, in the component’s environment that it may invoke). In the model, an output port in the required interface of a component can be linked to an input port in the provided interface of another component, creating a communication path between two different CRCs.
Methods execute run-to-end, that is, once a method has started execution, no other method of the same CRC may preempt it. However, any two methods of different CRCs may execute concurrently. This also holds for components that are hierarchically related, an inner component have no access to the state of the component that encapsulates it. The functional behavior of a system is specified in the method code of the CRCs. A method’s code can perform computations on local variables, read/mutate the component’s state, and invoke a method of the same or another component by sending a message to it.

Messages sent between components can be of two kinds: asynchronous, which are executed concurrently with the caller and can be delayed by a certain amount of time, and synchronous, with the caller blocked until the invoked method completes execution, optionally returning a value (such methods cannot be delayed). The permissible execution window (see Section 2.3) of an asynchronous message can be either inherited or explicitly specified in the code relatively to the caller’s baseline; in either case, it is viewed in the model as a separate reaction. A synchronous message, on the other hand, always inherits the caller’s time constraints and is viewed as a part of the original time-constrained reaction. Both asynchronous and synchronous messages can carry data (the values of arguments of the invoked methods).

The interaction with the environment is specified using a special construct, an environment interface, that defines an interface to the environment. The model does not make any assumptions regarding the behavior of the environment. Currently, the environment interfaces of the REKO IDE (see Section 3.2 below) supports interaction with the hardware platform for a single CPU system through ports (reading from registers and writing to registers) and interrupts (interrupts can invoke methods of CRCs). More specifically, this model supports:

- expressing concurrency using CRC,
- specifying both functional behavior and timing requirements of an embedded system,
- specifying methods (i.e. specifying the functional behavior) using C-based constructs,
- generating C code that preserves the timing constraints.

For a more detailed presentation of this model, see Papers E and F. Note that the model presented here is similar to the models presented in Papers A and B, and Papers C and D as described in Section 5.1.

3.1.2 Code generation framework

The code generation framework is depicted in Fig. 3.1. It is centered around the CRC model which is a principal element of the Reko IDE (see Section 3.2). How to generate code from the CRC model is described next.

---

1Paper F investigates how to map TinyOS models (see Section 2.2.2) onto the CRC model as a way to enable resource efficient execution of TinyOS models.
3.1. Software design methodology

Figure 3.1: The code generation framework used for translating the CRC model into an executable.

Requirements on the methods

All methods are written in the C language. However, the C language allows us to specify behavior not allowed by the model (such as entering an infinite loop, waiting for input, etc.). Thus, in order for a method written in C to be a valid CRC method it must comply with the following rules, it must

1. be run-to-end (complete execution within a finite amount of time),
2. not access any global memory (state) outside of its object, and
3. not invoke a method of another object directly (without using the ASYNC or SYNC primitive).

Compliance with two and three are partly enforced by name scoping, i.e. the framework will generate local defines for names in the current scope (method). However, the system designer can still force an incorrect behavior by directly calling methods (of other objects) or accessing global memory (e.g., using pointers). Strictly enforcing two and three would require implementing a parser for a subset of the C language (disallowing pointer arithmetic, extern keyword, etc.). The first rule is currently not enforced in any way and compliance must be ensured by the system designer. One approach to enforcing the first rule is to perform worst-case execution time analysis [34] on the methods (see Paper E). Below is an example of emitted C code, showing local defines for names in the current scope.

```c
// local name bindings
#define get_feedback ....
#define control_out ....
#define process ....
// method implementation
int controller_process(OBJ* self, int arg){
    int fb = SYNC(get_feedback, 0);
    // Controller state update etc, eg:
    self->state.out = 10;
    SYNC(control_out,self->state.out);
```
ASYNC(process, MS(10), MS(1), 0);
}
#undef get_feedback
#undef control_out
#undef process

Code generation

The CRC model contains definitions, instances, methods, and states. From this, C functions and object definition structures (C typedefs) can be emitted. Additional information required to compile (using a C-compiler) the system into an executable binary is:

- a static component structure (see below),
- defines for preemption levels and resource ceilings (see Paper D),
- a Kernel (a kernel design is presented in Paper D).

Static component structure

The static component structure is generated by transforming the CRC model to a model consisting only of component instances (CRC instances). From this model it is possible to generate one C-struct containing all instances in the system. The timing specification of the model is preserved during code synthesis and later used by the run-time kernel (for reference, see the generated C-function above).

3.1.3 Graphical representation

The component model (see Section 3.1.1) has a corresponding graphical representation. This representation is presented next in terms of screen shots from the integrated development environment (IDE) described in Section 3.2. These screenshots are from a real design, an autonomous car. The design of the autonomous car discussed in detail in Paper E.

The model distinguishes between CRC definitions and CRC instances. A CRC definition is represented graphically by a box with the definition name at the top (see Fig. 3.2). The ports that belong to its provided and required interfaces are shown as smaller boxes at the left and right side respectively. The ports name is printed inside the box. The interface ports are internally connected to methods or inner components as will be discussed later.

An environment interface definition is represented in a similar way but with a different color (see Fig. 3.2). However, its interface ports have a different meaning semantically. Each port in its provided interface represents a set of registers of the hardware platform of the modeled system. Further, each port of its required interface represents a specific interrupt in the hardware platform. Note that the model does not make any assumptions regarding the behavior of the environment so the assumptions about hardware registers
3.1. Software design methodology

Figure 3.2: The graphical representations for the elements of the component model: the CRC definitions in blue and yellow and environment interfaces in orange (the IDE displays the CRCs in different colors depending on their content). Each definition has its name at the top. The ports that belong to the provided or required interface is indicated as small boxes at the left and right side of the definition respectively, with the port name displayed inside each box.

and interrupts are external to the model. However, for a graphical representation targeting software development for resource-constrained embedded systems this is a reasonable assumption.

Any CRC may contain methods and instances of CRCs. Methods and state of a CRC are not considered part of its model structure and is therefore not described here (they can easily be presented in a text editor). However, the internal structure of a CRC can be represented graphically (see Fig. 3.3). Internally, the provided interface is represented by a vertical list of small boxes that correspond to ports in the external interface. The port name is displayed inside each box and the word *Provided* is displayed above the list. However, the ports that trigger method invocations are not included in the set. The required interface is represented in the same way but with the word *Required* instead of *Provided*. The set includes all the ports that should be reachable from the internal CRC instances. The CRC instances have the same graphical representation as their definition. However, the internal structure of a CRC instance is given by its definition. Ports can be connected to other ports in the following way:

- a provided interface port of a CRC can internally be connected to a provided interface port of an inner CRC,
- a required interface port of a CRC can internally be connected to a required interface port of an inner CRC,
- a required interface port of an inner CRC instance can be connected to a provided interface port of another inner CRC instance (or to a port in its own provided interface).

A connection is represented graphically by an arrow that indicate the message path. Different kinds of arrows can be used to distinguish different types of messages (synchronous/asynchronous).

One of the CRC definitions must be defined as the root CRC. A root CRC has no state, methods, or interface ports. Instead it defines everything that should be instantiated by the run-time system at startup (itself and hence all of its content).

The root CRC is the only place where environment interface instances are allowed (see Fig. 3.4). Their provided interface ports (denoted *regport*) and required interface
Towards component-based design of embedded software

Figure 3.3: The graphical representation for the internal structure of a CRC definition: the provided and required interfaces indicated by their respective name (Provided/Required) and the CRC instances in blue and yellow (the IDE displays the CRCs in different colors depending on the content of their corresponding definition). The ports that belong to the provided or required interface is indicated as small boxes with the port name displayed inside each box. Arrows indicate message paths (the small box on each line is used for selecting the line when working in the IDE).

ports (denoted intport) can be connected with ports of CRC instances in the following way:

- a regport can be connected to a required interface port of a CRC instance,
- an intport can be connected to a provided interface port of a CRC instance.

A connection between an intport and a required interface port of a CRC instance is represented graphically by an arrow that indicate the message path between the interrupt
vector and the CRC. However, a connection between a regport and a required interface port of a CRC instance is represented graphically by a line with a circle in the end instead of an arrow. The circle indicates access to the hardware registers of the modeled system as defined by the regport.

3.1.4 Stack resource policy (SRP) scheduling and resource management

Stack resource policy (SRP) is a policy for scheduling real-time tasks with shared resources that permits tasks with different priorities to share a single run-time stack [6, 35, 36]. SRP applies directly to scheduling policies with dynamic and static priority, including e.g., Earliest Deadline First (EDF) and Deadline Monotonic (DM) for which interrupt hardware of commonplace platforms can be utilized efficiently. SRP scheduling offers a number of advantages, mainly deadlock-free execution and memory savings due to the shared runtime stack, but it also bounds the number of preemptions to at most two for each job instance. The bounded number of preemptions together with early blocking (a job is not allowed to start execution until all resources are available) allows for simple and sufficient schedulability test for EDF [6], and DM [37]. The traditional version of SRP only addresses single-core systems, however, SRP has also been extended to multi-core and multi-processor systems (see, for example, [38] and [39]).

The SRP scheduling and resource management of CRC models are manifested by:

- a mapping between the undertaken CRC model to the notions of SRP,
- an algorithm that from a CRC model extracts resource ceilings and interrupt priorities for SRP-DM based scheduling,
- algorithms that for a large subset of CRC models derives periods (minimum interarrival times) and offsets of tasks/jobs for internal events (the inter-arrival times for external events must be specified by the developer),
- an SRP-DM kernel design supporting cooperative hardware- and software-scheduling utilizing the derived resource ceilings and interrupt priorities at runtime.

All these subjects are detailed in Paper D.

3.1.5 Design process

The different stages of the design process are presented in Fig. 3.5. 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.

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. This step is repeated iteratively for all components until no more components can be defined. At every step, the model of each component is matched against a repository of previously developed components, which should contain reactive models of components alongside their implementation.

The fourth step is implementation of components in some programming language. The fifth 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 the component level. 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.

A more detailed presentation of the design process is given in Paper A. Note that the description given here has been modified slightly to match the component model presented in this thesis (see Section 5.1). However, the general principles should apply.

### 3.2 REKO - An IDE for development of embedded software

REKO is an integrated development environment that enables component-based software design for resource-constrained embedded systems. A screenshot of the REKO IDE is shown in Fig. 3.6. REKO supports the component model presented in Section 3.1.1. Further, it provides a graphical user interface (GUI) where the user can create, browse, and edit the CRC models which are represented graphically in the IDE. States and method code is specified in C code. Models are presented in accordance with the representation presented in Section 3.1.3. A model is stored as an XML file that captures both system structure and implementation (the internal format of the XML file is presented in Paper C). Further, REKO supports code generation in accordance with the code generation framework presented in Section 3.1.2. Fig. 3.1 shows a schematic that detail the model transformations in terms of the XML representation, the CRC model, C code, and the executable. The code generation includes implementations of the algorithms listed in Section 3.1.4. Consequently, the resulting C code supports SRP scheduling and resource management on the target system provided that it is compiled together with a kernel.
3.2. REKO - An IDE for development of embedded software

Figure 3.5: Stages in the design process: from a specification to a ready product.

Figure 3.6: Screenshot showing the GUI of the REKO IDE.
implementation similar to the one described in Paper D. GCC [7] can be used to compile the C code to an executable for a specific target platform.

3.3 Developing embedded software for an autonomous car

The REKO IDE was utilized in a case study for developing the software for an autonomous car. The specification for the autonomous car comes from a mechatronics project course at the university\(^2\). In this course the students are divided into teams and each team design and build their own autonomous car by modifying a standard radio controlled model car. The function of the autonomous car is to track an IR signal coming from a beacon and then direct its way towards the beacon and make a full stop in front of it. The course is concluded with a single-elimination tournament where the cars race each other on who arrives first to the beacon. The target platform for the software design was one of the student cars (including its onboard sensors, servos, and microcontroller). This system is representative for the type of systems addressed by the IDE, i.e. resource-constrained embedded systems with hard real-time requirements. The problem statement as defined in the mechatronics course is presented below.

A conceptual model was formulated based on the problem statement. The IDE was used for constructing the structure of the model graphically, and to specify the functionality and timing requirements in terms of method code. Code was generated from the IDE and the timing properties of the resulting code was analyzed. The results of the analysis was that the timing requirements will always be met. The code was compiled together with a kernel implementation for execution on the target platform. The compiled code supports SRP scheduling and resource management on the target system. As a concluding remark the car operates according to its specification, that is, it drives autonomously from the starting line to the beacon and stops in front of the beacon as intended. The case study is presented in detail in Paper E.

3.3.1 Problem statement

The team should design and build an autonomous car. The car should track an IR signal coming from a beacon and then direct its way towards the beacon and make a full stop in front of it. The car should be equipped with an optical sensor which can detect the IR signal from a distance of 10 meter in daylight. The sensor signal should be processed by electronic hardware designed and built by the team. The processed signal should then be feed to the ADC in the onboard AVR microcontroller. The whole sensor unit will be mounted on a standard RC servo which enables the sensor to be panned in the horizontal plane. The beacon has two vertical rows of IREDs which emit IR radiation with a wavelength of 900 nm. The horizontal distance between the rows is 95 mm. The horizontal angle between the IR emitting rows (\(\delta\) in Fig. 3.7) can be measured by the

\(^{2}\)Luleå University of Technology.
3.3. Developing embedded software for an autonomous car

Figure 3.7: A schematic of the two competing cars and the race track. Each car has an optical sensor that can be panned which enables measuring the horizontal angle $\delta$ between the IR emitting rows. The angle $\delta$ is used to determine the direction and distance to the beacon relative to the car.

panning optical sensor. The panning servo, the speed and front wheel steering angle of the car is controlled by the microcontroller. The cars compete in a single elimination tournament. In every round two cars compete in a best two-of-three format. The cars are placed side by side behind a starting line at a 10 m distance from the beacon (see Fig. 3.7) facing the beacon\(^3\). On a given start signal the cars are activated by the team. The cars should drive autonomously to the beacon as fast as possible. The goal position is defined as a circle with one meter in diameter placed between the start line and the beacon in such a way that the vertical projection of the emitting rows lies on the circle. The winner is the car which first enters the circle with a wheel and stops (the wheel must remain within the circle after the car has stopped). The beacon must not be touched by the car.

\(^3\)In the case study we wanted the autonomous car to be able to locate and track the beacon even if it is not facing the beacon to start with.
Towards component-based design of embedded software
Conclusions, related work, and future work

This chapter summarizes the research contributions and discusses their relation to the research questions stated in Section 1.4. Moreover, in related work, the proposed approach is compared to other approaches targeting resource-constrained embedded systems. The chapter is concluded with suggestions for future work.

4.1 Conclusions

The main results of this thesis are a component-based software design methodology and an integrated development environment (IDE) – REKO. The methodology enables constructing complex software for resource-constrained embedded systems and REKO supports this by ensuring a systematic development of software in accordance with the approach. REKO generates C code, hence it can be integrated with a majority of the tools used in the industry (for a substantial part of industry, C code is the de facto standard for programming embedded systems). Let us now discuss the results of this thesis with respect to the research questions stated in Section 1.4.

The methodology comprises a component model that is based on the reactive object model. As a result, it supports specifying both functional behavior and timing requirements of an embedded system. It can be used for constructing software models by composing ready-made components. Components are composed by connecting the ports specified in their interface. Each port has a name and a type and the types enables verifying the composability of the constructed model. The functionality of components (their methods) is specified using C-based constructs. Furthermore, the model supports generating C code that preserves the timing constraints. The component model provides an answer to question Q1.

The methodology also includes a code generation framework. It supports generating code that manifest a static component structure. This enables using more extensive compiler optimizations at compile time. Further, it enables predicting the resource requirements by making static analysis targeting SRP-based scheduling. This answers question Q2.
The methodology also proposes a graphical representation of the component model. This representation is effectuated in REKO. On that account, the user can create, browse, and edit the software models, which are represented graphically in REKO’s graphical user interface (GUI). This answers question Q3.

The thesis also includes a case study, in which REKO is used for designing a real-life system, an autonomous car. The case study demonstrates the potential of the proposed methodology to bridge the gap between state-of-the-art in component-based software design on the one hand, and the methods and tools currently used in the industry on the other.

4.2 Related work

There is a strong interest for component-based design of embedded software. As a result, various approaches and component models have been proposed. Section 2.2.2 describes a selection of approaches that targets software development for resource-constrained embedded systems. In this section, some of the described approaches are compared against the approach taken in this thesis.

Resource-constrained embedded systems are typically interrupt driven, i.e. they must respond to events that originate in their environment (i.e. the physical world), often with real-time requirements. Moreover, it is becoming increasingly common for these systems to perform multiple tasks at the same time. Hence, the normal operation for such systems is the reactive execution of concurrent events.

Our model for developing systems is reactive, which means that the natural behavior of the modeling constructs (CRCs) is to react to events. Moreover, the model allows to express concurrency simply by specifying functionality in terms of separate CRCs. Hence, the model eliminates the need for manually specifying concurrency (e.g. threads, mutexes, etc.) which often lead to incorrect code with bugs that are hard to decipher (race conditions, deadlocks, etc.).

TinyOS [32] targets building concurrent, reactive applications for resource-constrained systems. Its component model is similar to the CRC model. A module in TinyOS can be seen as a CRC that is prohibited to encapsulate inner CRCs. Moreover, a configuration can be seen as a CRC that is prohibited to contain anything other than inner CRCs. Hence, the TinyOS model supports constructing hierarchical models in the same manner as the CRC model. Moreover, TinyOS modules encapsulates state, as does CRCs. In contrast to the CRC model, the concurrency is not modeled in terms of components, instead TinyOS expresses concurrency in terms of asynchronous code that can be invoked by interrupt handlers. The execution model in TinyOS is based on run-to-completion tasks and event handlers. The standard TinyOS task scheduler uses a non-preemptive, FIFO scheduling policy. TinyOS application are well suited for execution on resource-constrained systems. However, TinyOS offer limited support for developing real-time applications.

The Rubus Component Model (RCM) [29,30] targets development of resource constrained embedded real-time systems. Like the CRC model, it supports constructing a
model of a system by composing modeling constructs graphically. However, as in the TinyOS case, RCM distinguishes between the basic component which is called a Software Circuit (SWC) and the *assembly* which enables constructing hierarchical models. SWCs are used for encapsulating software functions. The software functions are mapped to tasks at a later stage of the design process. Similar to our approach, RCM supports expressing timing requirements and analyzing the resource demands for a particular hardware platform. Further, the run-time system supports executing all tasks on a shared stack. This is supported in our methodology as well through SRP resource management. In contrast to the CRC model and the REKO IDE, RCM and its tool suite has been used in the industry for the development of embedded software for many years.

ProCom [24] component model targets the development of control-intensive distributed embedded systems. It defines a two-layer component model. The upper layer is called ProSys and this model is similar to the CRC model. Like in the CRC model it models a system in terms of concurrent subsystems. Moreover, the communication structure used in ProSys is similar to the one used in the CRC model. Like the previous models, ProSys supports constructing hierarchical models. The components used for constructing hierarchical systems are called *composite subsystems* and the basic components are called *primitive subsystems*. The lower level is called ProSave and it is similar to RCM. For example, both models separate data flow from control and they use the same communication mechanism [30].

It is also worth mentioning AmbientRT [19] which is a real-time operating system for embedded devices with resource-constraints in terms of memory, processing, and energy resources. This approach does not offer any tools for composing components to a software model. However, it supports loading blocks of precompiled software and executing the blocks dynamically. Hence, it enables creating new software configurations online. Like in our approach, a guarantee on meeting the real-time constraints for each task can be given through analysis. Moreover, the AmbientRT kernel supports pre-emptive real-time scheduling of tasks whereas our kernel supports pre-emptive real-time scheduling of C-code generated from a CRC model.

### 4.3 Future work

The applicability of the software design methodology presented in this thesis is very much dependent on supporting tools. There are many things that can be done in order to improve the REKO IDE in terms of its usability and the quality of the software that is produced using REKO. One improvement would be to add support for using data structures in the communication between objects (e.g. conceptually sending a struct to another component). However, this should be provided in the context of static memory and without inducing substantial overhead. More visualization could be provided, such as distinguishing between synchronous/asynchronous communication or displaying event chains (including deadlines and inter-arrival times).

In addition, more features can be added to the methodology. For example, it is attractive to add support for predicting the memory consumption of the system (including
the memory consumption of messages in the system). Further, it should be possible to improve the analysis (make it less conservative) by using offset based analysis. After all, REKO already supports extracting offsets between tasks.

It is important to perform additional case studies in order to further evaluate the methodology. In particular, the case studies should target real-life systems with dissimilar hardware platforms.
This chapter provides an introduction to the papers included in the second part of this thesis as well as a short summary of each paper. Further, the relevant contributions of each paper is ascribed to the corresponding author.

5.1 Introduction

The component model presented in this thesis (Section 3.1.1) has undergone some changes during the course of this work. This is manifested by the papers included in the second part of this thesis, which describe three different versions of the component model. This introduction details how the different versions of the model compare to each other.

The initial version of the component model is described in papers A and B. Its purpose is to model the functionality of both hardware and software, i.e. the whole embedded system, in a single model. Further, by using a language that uses the same primitives as the model (i.e. the Timber language [40] as described in Paper A) the translation of a model into programming code becomes straightforward. The fundamental modeling construct of the model is a reactive object. Each reactive object has a state and one or several methods, and it is reactive in the sense that it reacts to an incoming event by executing one of its methods. The reactive object has a provided and a required interface (in papers A and B these are referred to as inputs and outputs). The provided interface include the methods of the object that can be invoked by other objects and the required interface specify the methods that the object may invoke. Methods can be of two kinds (which corresponds to two kinds of messages sent between objects): asynchronous, which are executed concurrently with the caller and can be delayed by a certain amount of time, and synchronous, with the caller blocked until the invoked method completes execution, optionally returning a value (such methods cannot be delayed). Thus the permissible execution window of an asynchronous message can be either inherited or explicitly specified in the code relatively to the caller’s baseline (see Section 2.3). A synchronous message, on the other hand, always inherits the caller’s time constraints
and is viewed as a part of the original time-constrained reaction. The reactive objects create a flat, non-hierarchical structure of a system. To support efficient development of complex software systems, the model includes the notion of component. A component contains no own state or methods, but instead encapsulates a number of objects and/or other components, creating a simple hierarchical structure. Like objects, each component has a provided interface (the methods, or input ports, of the component that can be invoked by other components) and a required interface (the methods, or output ports, in the component’s environment that it may invoke). The input ports are connected to the interface (input ports) of inner components or to the interface (methods) of inner objects. In this model, each reactive object can have one or several hardware or software implementations.

The component model described in papers E and F (and in Section 3.1.1) enables modeling the software of an embedded system based on interaction with the system platform (e.g., a microcontroller and its peripherals in terms of sensors, motors etc.). It is utilized in an integrated development environment (IDE) where the structure of the software system is defined graphically and the functionality is defined using a subset of C. From the model it is possible to generate C code that can be compiled for execution on a hardware platform assisted by a real-time kernel. Unlike the aforementioned model, the hardware functionality is not included in the model. Instead, the interaction with the environment (i.e., the hardware platform) is modeled using a special construct, an environment interface, that defines an interface to the hardware. The model does not distinguish between reactive objects and components. Instead, the reactive objects themselves are viewed as components and they are denoted concurrent reactive components (CRC). Similar to a reactive object, a CRC may have a state and one or several methods. In addition, a CRC may encapsulate a number of other components, creating a hierarchical structure. A CRC reacts to an incoming event by executing one of its methods or invoking a method of an underlying component. Each CRC has a provided interface (the methods, or input ports, of the component that can be invoked by other components) and a required interface (the methods, or output ports, in the component’s environment that it may invoke). In the model, an output port in the required interface of a CRC can be linked to an input port in the provided interface of another CRC, creating a communication path between two different components. The components communicate by passing messages. The messages are either asynchronous or synchronous as described in the aforementioned model. Every CRC has a class definition from which software instances can be created at run-time. A CRC definition defines the object’s state variables, methods, and the provided and required interfaces of the component. Moreover, a CRC definition defines the component instances that the CRC encapsulates as well as the communication paths between the ports of the provided/required interface of the CRC and the encapsulated components.

Papers C and D targets an early version of the IDE that uses a component model (denoted CRO model) similar to the previous model based on components (CRC). The difference is that the CRO model distinguishes between objects (each object has a state and one or several methods) and components (each component may encapsulate other
objects or components) denoted concurrent reactive objects (CRO) and concurrent reactive components (CRC) respectively. The CRO model has been set aside on behalf of the previous model.

All the reactive entities defined by the different models (reactive object, CRC, and CRO) conform to the more general definition of reactive object given in Section 2.3.

5.2 Paper A

**Title:** Enabling Component-Based Design for Embedded Real-Time Software  
**Authors:** Jimmie Wiklander, Jens Eliasson, Andrey Kruglyak, Per Lindgren, and Johan Nordlander  
**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 are 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 follows the work of Johan Nordlander [2,3]. The presented component model 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. During the work, numerous discussions involving Jimmie Wiklander, Jens Eliasson, Andrey Kruglyak, Per Lindgren, and Johan Nordlander have contributed to both content and presentation. The preparation of the manuscript is joint work by Jimmie Wiklander and Andrey Kruglyak.

5.3 Paper B

**Title:** Personal Alarm Device: A Case Study in Component-Based Design of Embedded Real-Time Software  
**Authors:** Jimmie Wiklander, Andrey Kruglyak, Johan Nordlander, and Per Lindgren

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. The underlying component model is a result of joint research by Andrey Kruglyak and Jimmie Wiklander. During the work, numerous discussions involving Jimmie Wiklander, Andrey Kruglyak, Johan Nordlander, and Per Lindgren have contributed to both content and presentation. The preparation of the manuscript is joint work by Jimmie Wiklander and Andrey Kruglyak.

5.4 Paper C

Title: An IDE for Component-Based Design of Embedded Real-Time Software

Authors: Jimmie Wiklander, Johan Eriksson, and Per Lindgren


Summary: This paper describes work in progress on an Integrated Development Environment (IDE) for component-based design of embedded real-time software (the IDE is presented in Section 3.2). The IDE supports graphical modeling of software systems using concurrent reactive objects and components (see Section 5.1), as well as generation of C code from the model. The resulting application code can be combined with a lightweight kernel for execution on bare metal.

Contribution: The design and implementation of the IDE (apart from the backend) should be accounted to Jimmie Wiklander. The development of the backend that enables code generation should be accounted to Johan Eriksson. Andrey Kruglyak have contributed to the development of the IDE through numerous discussions about the
graphical representation. The development of the underlying model (the CRO model) is a result of joint research by Jimmie Wiklander, Johan Eriksson, and Per Lindgren.

5.5 Paper D

**Title:** Implementation of SRP-DM Scheduling for Embedded Real-Time Software  
**Authors:** Johan Eriksson, Simon Aittamaa, Jimmie Wiklander, Pawel Pietrzak, and Per Lindgren  
**Summary:** This paper focuses on code synthesis of CRO models (see Section 5.1) and extraction of information for scheduling (during run-time) as well as offline schedulability analysis. In particular, the paper shows how the timing requirements specified in the CRO model are preserved and translated into resource ceilings and priority levels for Stack Resource Policy (SRP) scheduling. The paper presents an informal mapping from the undertaken CRO model to the notions of SRP and detail the algorithms that for a particular CRO model derives periods (minimum inter-arrival times) and offsets of tasks/jobs. Further, an example system (a process controller) is used for demonstrating how system state, functionality and temporal properties are captured and abstracted in terms of the CRC model, and how timing requirements from the model is translated into resource ceilings and priorities for the scheduler. Additionally, the paper presents the design of an efficient SRP-DM kernel, and demonstrate how the derived resource ceilings and priorities are utilized at run-time.  
**Contribution:** The development of the CRO model is a result of joint research by Jimmie Wiklander, Johan Eriksson, and Per Lindgren. The extraction of resource ceilings and priority levels for Stack Resource Policy should be accounted to Johan Eriksson. The mapping from the CRO model to SRP is the result of joint research between Simon Aittamaa and Pawel Pietrzak. The development of the SRP-DM kernel should be accounted to Simon Aittamaa.

5.6 Paper E

**Title:** Utilizing an IDE for Component-Based Design of Embedded Real-Time Software for an Autonomous Car  
**Authors:** Jimmie Wiklander, Johan Eriksson, and Per Lindgren  
**Publication:** To be submitted.  
**Summary:** This paper presents the software design for an autonomous car constructed using an integrated development environment (IDE) (an overview of the IDE is given in Section 3.2). The specification for the autonomous car comes from a mechatronics course at the university where the students build their own car by modifying a standard radio controlled model car. Its function is to track an infrared (IR) signal coming from a beacon and then direct its way towards the beacon and make a full stop in front of it.
The tracking of the beacon is based on panning an optical sensor from side to side and use the sensor signal to derive the position of the beacon relative the car. In the IDE, the structure of the software system for the autonomous car was defined graphically and its functionality was defined using a subset of C. As supported by the IDE, the timing specification for the software system was included in the software design (e.g. a delay was added that allows the sensor to reach its new position before acquiring a sensor sample). The IDE was used for generating C code from the software model of the autonomous car. Further, the generated code was compiled for the target platform (an AVR microcontroller integrated as part of the autonomous car) together with a lightweight kernel. The kernel utilizes the timing specification preserved in the code to schedule method execution and message passing between components at run-time. The car operated according to its specification, that is, it was able to track the beacon and autonomously guide its way towards the beacon and stop in front of it.

**Contribution:** The design and implementation of the software system for the autonomous car should be accounted to Jimmie Wiklander. The design and implementation of the IDE (apart from the backend) should be accounted to Jimmie Wiklander. Johan Eriksson has developed the backend that enables code generation from the IDE. During the work, numerous discussions involving Jimmie Wiklander, Johan Eriksson, and Per Lindgren have contributed to both content and presentation.

### 5.7 Paper F

**Title:** Towards Executing TinyOS Models Under Hard Real-Time Constraints  
**Authors:** Per Lindgren, Johan Eriksson, Henrik Mäkitaavola, and Jimmie Wiklander  
**Submitted to:** Design, Automation and Test in Europe (DATE), 2012.  
**Summary:** This paper investigates how TOS models can be executed under hard real-time constraints. It presents an alternative execution model for TOS programs that exploits additional concurrency while maintaining race free execution. Moreover, it presents a mapping from TOS models to Concurrent Reactive Components (CRC), and show that the resulting CRC model is able to express the aforementioned execution model. It is also shown how timing requirements can be added to allow TOS models to be executed under hard real-time constraints. This paper also discusses the implications of the CRC model to implementation efficiency and offline resource and performance analysis.  
**Contribution:** The original idea, and writing is due Per Lindgren. During the work, numerous discussions involving Johan Ericsson, Jimmie Wiklander and Henrik Mäkitaavola have contributed to both content and presentation. Examples in the camera ready version is due to Henrik Mäkitaavola with support from Jimmie Wiklander for the CRC models and tools.
REFERENCES


Part II
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 originally published in:

© Academy Publisher 2009, Reprinted with permission.

¹http://dx.doi.org/10.4304/jcp.4.12.1309-1321
Enabling component-based design for embedded real-time software

Jimmie Wiklander, Jens Eliasson, Andrey Kruglyak, Per Lindgren, and 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\(^2\). 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

\(^2\)A good overview of existing design practices and research trends in embedded system design is given in [1] and [2].
functionality implemented in software and functionality of hardware parts. In many 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 \( t_{after} \) before \( t_{before} \) doSmth, where \( t_{after} \) is the period of time between the event and the baseline, \( t_{before} \) is the period
2. Modeling Framework

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.

of time between the baseline and the deadline, and doSmith is the invoked method.

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\(^3\) [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.

\(^3\)We will be using executable models which means that reactive objects are preserved in the implementation.
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
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
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 6:** Partitioning of a system into a resource platform and a software application.
2. Modeling Framework

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 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 can be translated into messages.
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
3. Software Design Methodology

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 comprehensible, 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 suite 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-constrained 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*, 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.

*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.
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.

- **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 $t_{\text{period}}$ 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 $t_{\text{alarm}}$ 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-
5. AN EXAMPLE SYSTEM: A PERSONAL ALARM DEVICE

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 $t_{\text{alarm}}$ 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 $t_{\text{period}}$ 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 $t_{\text{lag}}$ 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 $t_{\text{period}}$; the lag between impact detection and posture evaluation $t_{\text{lag}}$; and the maximum period of time between fall detection and sending an alarm $t_{\text{alarm}}$.

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 $t_{\text{period}}$ milliseconds, and so forth.

The timing requirements on the first reaction consist of a relative deadline $t_{\text{alarm}}$ 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 $t_{\text{period}}$ 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
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.

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 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 \( \text{sampleAcc} \), and an output that delivers the acceleration value once it has been acquired \( \text{consumeAcc} \). 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 \( \text{sendMsg} \), 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\(^5\). 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}`:

\[
\text{newBaseline} = \text{currentBaseline} + t_{period}.
\]

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.

\(^5\)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.
5. An Example System: A Personal Alarm Device

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.
(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 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

Germany: Springer-Verlag, 2005.


Personal alarm device: A case study in component-based design of embedded real-time software

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

Reformatted version of paper originally published in:
Personal alarm device: A case study in component-based design of embedded real-time software

Jimmie Wiklander, Andrey Kruglyak, Johan Nordlander, and Per Lindgren

Abstract

Designing software for embedded systems is complicated by such factors as the tight integration between software and hardware, scarceness of available resources, and 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 software design methodology for embedded real-time systems.

Here we describe a 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 fall and send an alarm signal. The implementation of the system was verified using a Simulink-based simulator. The simulation demonstrated that, even though calculation of acceleration was simplified to allow for an efficient execution on a resource-constrained platform, fall detection remained satisfactory.

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. 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 dif-
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 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 hard-
ware 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 [1].

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 com-
ponents 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 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 $t_{after}$ 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 \textit{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 interaction 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 \textit{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.

An object to be realized by an underlying hardware implementation should be viewed as a singleton instance (reflecting that it is indeed just an interface to a single hardware instance). Such an object typically provides an interface consisting of synchronous read/write methods (i.e., I/O port operations) and can be seen as part of the software environment. This view allows us to restrict access to hardware resources (e.g., as being in scope of just the controlling software/driver.)
2. Methodology Overview

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

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

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 [5] 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 elderly people were used as a reference. For each algorithm, sensitivity and specificity were calculated using Eqs. 1 and 2:

\[
sensitivity = \frac{TP}{TP + FN},
\]

\[
specificity = \frac{TN}{TN + FP},
\]

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 [5]) 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 [5].
Description of the Algorithm

The fall detection algorithm is based on impact detection and posture evaluation. The acceleration is monitored periodically (with period $t_{\text{period}}$) 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

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

(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 $t_{\text{lag}}$. 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 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. For this particular system, the focus was on the functional requirements, which had to be expressed in terms of time-constrained reactions. For the application, two such reactions were identified: sending an assistance alarm within $t_{\text{alarm}}$ msec after the push button has been pressed, and sending a fall alarm within $t_{\text{alarm}}$ 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 (internal) timing requirements – $t_{\text{period}}$ msec (the time between acceleration samplings) and $t_{\text{lag}}$ msec (the time between impact detection and posture evaluation). This illustrates that timing requirements can either stem from the desired (observable) behavior or from the selected solution approach.

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
4. PAD Design

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).

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, showing the observable timing requirements of the application.

The fall detector component is activated periodically (with period time $t_{\text{period}}$) 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
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.

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 (\texttt{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 (\texttt{consumeAcc}).

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

The push button component is activated on the arrival of an interrupt from the button. 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 $t_{\text{alarm}}$ \text{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 input 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 $t_{\text{alarm msec}}$ is defined. It is inherited by the assistance alarm sender and later on by the message sender components.

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 com-
Figure 8: 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.
ponents 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; and fall detection controller, which controls the fall detection procedure.

The fall detection procedure starts with the fall detection controller being invoked 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 is triggered in the same manner by the push button component.

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:

$$adController setChan convert read accAnalyzer = class$$

$chan := 1 \{ \text{number of the channel to read from} \}$

$values := [] \{ \text{a list of acceleration values} \}$

sampleAcc = before $t_{setChan}$ action

$after (t_{setChan} + t_{wait})$ initConversion

$setChan$ chan $\{ \text{set the channel} \}$

$chan := chan + 1 \{ \text{update channel state} \}$

initConversion = before $t_{convDl}$ action

$convert \{ \text{initiate conversion} \}$

to be continued...

Here we are using the notation from the Timber language that reflects the concepts of time-constrained reactions and reactive objects 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.
Figure 9: Timing requirements for A/D conversion described as permissible execution windows for the methods of the A/D controller object.

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 \((t_{\text{setChDL}} + t_{\text{wait}})\) ensures that at least \(t_{\text{wait}}\) seconds 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 \texttt{adIRQHandler} method):

\begin{verbatim}
...continued from above
adIRQHandler = before t_{adIRQDL} action
val ← read \{ - read the value -\}
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
\end{verbatim}

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.

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 ex-
ecution 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 [6–8] 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

Timber allows to express the model in terms of Timber objects, communicating through
asynchronous and synchronous messages.

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.

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

```turing
accSampler adController = class
    sample = before $t_{\text{sampleDl}}$ action
        adController.sampleAcc
    after $t_{\text{period}}$ sample
```

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

```turing
accAnalyzer fallDetectionController = class
    a := []
    analyze xyz = before $t_{\text{analyzexDl}}$ action
        { - low-pass filter the \textit{y} input (vertical acceleration) - }
        if { - look for impact in \textit{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:

```timber
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 [8] and linked with the Timber run-time system [9] (written in C) for the target platform. The Timber runtime system provides scheduling of messages directly based on the timing constraints from the model preserved throughout implementation 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 [10] 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 [5] 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.

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 [11] 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 [12].

Another approach is Time-Triggered Architecture [13] 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. [14]); time-triggered languages such as Giotto [15]; the Ptolemy framework for assembly of concurrent components [16], which is particularly suitable for modeling distributed systems; and tools such as Rhapsody [17], Artisan Studio [18], and Rose-RT [19].

<table>
<thead>
<tr>
<th></th>
<th>Simulink simulations of the Timber implementation, $t_{lag} = 0.9$ sec</th>
<th>LabView simulations of the algorithm, taken from [5], $t_{lag} = 2.0$ sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No. of rec.</td>
<td>Specificity*/Sensitivity**</td>
</tr>
<tr>
<td>ADL (no falls)</td>
<td>160</td>
<td>100.0% *</td>
</tr>
<tr>
<td>Fall forw. (syncope)</td>
<td>38</td>
<td>100.0% **</td>
</tr>
<tr>
<td>Fall forw. (tripping)</td>
<td>37</td>
<td>100.0% **</td>
</tr>
<tr>
<td>Fall backw. (attempting to sit down)</td>
<td>39</td>
<td>97.4% **</td>
</tr>
<tr>
<td>Fall backw. (from standing position)</td>
<td>40</td>
<td>97.5% **</td>
</tr>
<tr>
<td>Lateral fall</td>
<td>39</td>
<td>100.0% **</td>
</tr>
<tr>
<td>Fall from bed</td>
<td>40</td>
<td>100.0% **</td>
</tr>
<tr>
<td><strong>Falls, total</strong></td>
<td><strong>233</strong></td>
<td>**99.1% **</td>
</tr>
</tbody>
</table>

Table 1: Simulation results.
We should also mention work on specification of real-time systems, such as RT-UML [20] and MARTE [21]. 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 developed using the component-based design methodology presented in [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 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.

Future work includes development of an IDE supporting the suggested software design methodology, with component repository management, code synthesis and validation tools.

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 AESOP project.
References


An IDE for component-based design of embedded real-time software

Authors:
Jimmie Wiklander, Johan Eriksson, and Per Lindgren

Reformatted version of paper originally published in:

© 2011 IEEE, Reprinted with permission.

1http://dx.doi.org/10.1109/SIES.2011.5953677
An IDE for component-based design of embedded real-time software

Jimmie Wiklander, Johan Eriksson, and Per Lindgren

Abstract

This paper describes work in progress on a tool for component-based design of embedded real-time software. The tool supports graphical modeling of software systems using concurrent reactive objects and components, as well as generation of C code from the model. The resulting application code can then be combined with a lightweight kernel for execution on bare metal.

1 Introduction

Tool support is instrumental for efficient and correct design of embedded software. In this paper we present ongoing work on an Integrated Development Environment (IDE) for Component-Based Design (CBD) of embedded real-time software, supporting the design methodology presented in our earlier work [1]. The aim is to construct system models based on concurrent reactive objects and components in a graphical environment and to generate efficient C code from the model for execution on bare metal. More specifically, the system structure is defined graphically and the actual method code is written manually in C. The tool ensures that this code does not violate the properties of the model, such as state encapsulation in objects.

Distinctive to our approach is that timing requirements are captured as part of the model and are preserved throughout the design process; they can then be utilized by our kernel for scheduling at run-time. This stands in contrast to other modeling approaches (such as RT-UML [2] and MARTE [3]), which are typically based on successive transformations between different models, an approach that increases complexity of the design process. Due to space limitations, this work does not include any comparison to other modeling approaches.

The rest of this paper is organized as follows. In the next section (Section 2) we describe the underlying modeling framework. In Section 3 we focus on the functionality and intended use of the tool. In Section 4 we present the internal format of the tool. Finally, in Section 5 we discuss imminent future work.
2 Modeling Framework

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 and time-constrained reactions. A more detailed presentation of the framework can be found in [1]. In this section we highlight the key aspects of time-constrained reactions, reactive objects, their abstraction to hierarchical concurrent reactive components, and discuss modeling interaction between components/objects as well as a system’s interaction with its environment.

2.1 Time-Constrained Reactions

Time-constrained reactions [4] lie at the heart of our model. 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 a reactive approach, functionality is specified in terms of time-constrained reactions to such events.

Timing requirements on system operation can be specified by defining the earliest and the latest reaction time (baseline and deadline) relative to the time of the 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).

2.2 Concurrent Reactive Objects

The fundamental modeling construct of the framework is a concurrent reactive object (CRO) [5]. Each object has a state and one or several methods, and it is reactive in the sense that it reacts to an incoming event by executing one of its methods. Thus a CRO is either idle (maintaining its state) or executes a method. Methods execute run-to-end, that is, once a method has started execution, no other method of the same object may preempt it. However, any two methods of different objects may execute concurrently. A method’s code can perform computations on local variables, read/mutate the object’s state, and invoke a method of the same or another object by sending a message to it.

Methods can be of two kinds (which corresponds to two kinds of messages sent between objects): asynchronous, which are executed concurrently with the caller and can be delayed by a certain amount of time, and synchronous, with the caller blocked until the invoked method completes execution, optionally returning a value (such methods cannot
be delayed). Thus the permissible execution window of an asynchronous message can be either inherited or explicitly specified in the code relatively to the caller’s baseline; in either case, it is viewed in the model as a separate reaction. A synchronous message, on the other hand, always inherits the caller’s time constraints and is viewed as a part of the original time-constrained reaction. Both asynchronous and synchronous messages can carry data (the values of arguments of the invoked methods).

Each CRO has a provided interface (the methods, or input ports, of the object that can be invoked by other objects) and a required interface (the methods, or output ports, in the object’s environment that it may invoke). In the model, an output port in the required interface of an object can be linked to an input port in the provided interface of another object, creating a communication path between two different CROs. Multiple output ports can be connected to a single input port.

2.3 Concurrent Reactive Components

Concurrent reactive objects create a flat, non-hierarchical structure of a system. To support efficient development of complex software system, we introduce the notion of concurrent reactive components (CRC), which contain no own state or methods but instead encapsulate a number of objects and other components, creating a simple hierarchical structure. Like objects, components have provided and required interfaces, but the ports in the interfaces are connected not to methods, but to ports in the interfaces of inner objects or components. Thus communication across component boundary is in fact communication between two objects belonging to two different components, which allows for an efficient implementation where a component hierarchy has no overhead at run-time.

It is worth noting that while a CRO can only execute one method at a time with concurrency existing between different objects, a CRC can be concurrent in itself as it may encapsulate multiple objects.

2.4 Classes and Instances

Every instance of a CRO/CRC belongs to a class, which can be defined at the top level in a module or locally within another class; in the latter case, it can only be used inside that class. A CRO definition defines the object’s state variables, methods (these are written manually in C and the tool verifies that the C code complies with the model’s requirements as outlined below), and the provided and required interfaces of the object. In the C code of a method we may:

1. define local variables,
2. read/update the object’s state,
3. perform calculations on the object’s state, method argument, and local variables,

---

2No object or component can be part of two different components.
Figure 2: The complete set of CRC/CRO class definitions (including an interface to the environment) for the controller system (CRC definitions in blue, CRO definitions in yellow, and an environment interface in orange). The top label shows the class name and the labels at the left/right side of each class represent its provided/required interfaces. Note that only the system component (i.e., the model of the whole system that encapsulates an instance of the environment interface) does not require any other CRC/CRO in order to be instantiated (it has no provided/required interface).

4. invoke an asynchronous method of this object or of another object using the name of an output port (see Fig. 1):
   \[ \text{ASYNC(PortName, BaselineOffset, RelativeDeadline, Argument)} \]

5. invoke a synchronous method of another object using the name of an output port:
   \[ \text{SYNC(PortName, Argument)} \]

6. return a value (only if the method is synchronous).

A CRC definition defines the provided and required interfaces and specifies instances of CROs and CRCs encapsulated in the component.

2.5 Kernel support and interaction with the environment

A system model (i.e. the system’s structure in terms of CROs and communication paths between them, and C code written for each method) can be used to generate C code for the application. Execution on a hardware platform also requires infrastructure supporting the model in the form of a lightweight kernel supporting scheduling of method execution and message passing between objects.

Embedded systems interact with their environment and this must be reflected in the model. This is modeled using a special construct, an environment interface component that defines an interface to the hardware. An instance of such interface component enables reading from hardware registers and writing to them and is also used to connect specific hardware interrupts to asynchronous methods of particular objects. The model does not make any assumptions regarding the behavior of the environment.

Sometimes it is necessary to include legacy software that is not based on the CRO model (typically, external software libraries). Such libraries can be treated as part of the system’s environment, and we can use an environment interface component as a wrapper to legacy code.
3. IDE for Embedded Real-Time Software

In this section we focus on the functionality and intended use of the tool, that is, constructing system models in a graphical environment. An example system (a process controller) is also presented.

A model of a particular system along with configuration settings is stored in a project file in XML format. Each project window initially contains two tabs: a configuration tab and a system tab.

3.1 System Tab

The system tab supports creating a graphical model of a system using modeling constructs described in the previous section. Definitions of objects and components can be created and edited in the tool, and instances of objects and components can be included in
a component definition. The CRC/CRO definitions for the process controller example are shown in Figure 2. The content of the system and app CRC definitions from the example are shown in Figures 3 and 4, respectively. Connecting one interface port to another enables message passing (sending a synchronous or an asynchronous message) between instances at run-time. The content of a CRO definition (state and methods) can be edited in a built-in text editor by opening a separate CRO tab for the object in question.

3.2 Configuration Tab

Certain settings have to be specified to generate code for a particular hardware platform. This is done in the configuration tab. In addition, it is necessary to specify the “root” CRC (the CRC that contains the whole system and that should be instantiated at system startup). The root component cannot have any ports in its interfaces and all interface ports of the inner objects/components must be connected for code generation to succeed.

3.3 Control system example

This example implements a simple control application, where the root component (called system) defines the environment interface envinst and the CRC instance appinst, encapsulating the CROs implemented in software. Thus the interfaces of the app component describe interaction with the environment, in this case there are two outputs (control_out, get_feedback) and three inputs, or entry points: the startup routine (start) and two interrupt handlers (inc, dec). The application component instantiates three objects (adapter, controller, and logger).

The adapter object definition provides three methods:
- a startup routine for setting up the local state, and initializing the controller and logger by calling their initialization methods, and
- inc and dec methods for increasing/decreasing the local state variable (these also invoke setpoint).

The controller object definition is specified in a similar way:
- init initializes the local state and invokes a private (not visible in the interface) method process,
- setpoint updates the internal state variables, and
- process, a private method that samples the AD, performs control calculations and sets a new output value periodically (the definition of a private method is that it is not part of the object’s interface, so it cannot be invoked from outside the object; private methods of an object can be edited by opening a CRO tab).

The logger is similar to the controller in that it has an initialization method and a method that periodically samples the AD for logging purposes.
4 Internal Data Format

A model is stored in an XML file capturing the hierarchical structure of the system and method implementations. At the top level, it contains one or several modules:

```
<module name="mymodule">
  [CONTENT]
</module>
```

In this paper we have restricted us to using only one module, but combining multiple modules in one project is useful for creation of commodity libraries for component re-use.

Each module contains one or several class definitions for CROs and CRCs. A class definition specifies a required interface (the list of output ports \texttt{arg}) and a provided interface (the list of input ports \texttt{res}). Assignments to the provided interface ports of the class are defined in \texttt{con}. Here is an example of a class definition (\texttt{CONTENT} stands for inner modeling constructs included in the class definition):

```
<class name="controller"
  arg="control_out,get_feedback"
  res="init,setpoint">
  [CONTENT]
  <result con="init=init,setpoint=setpoint"/>
</class>
```

A class definition of a CRO may contain:

1. One state structure with C code declaring state variables of the object, e.g.:

```
<state>
  <![CDATA[ int i; ]]>
</state>
```

2. One or several methods with C code implementation, written according to guidelines discussed in in Section 2, e.g.:

```
<method
  name="init"
  arg="int arg"
  ret="void">
  <![CDATA[ASYNC(process, 100, NO_ARG);]]>
</method>
```

And a class definition of a CRC may contain:

1. Zero or more class definitions (as described above) only visible in the scope of the component and any sub-component(s) (referred to as local class definitions).
2. One or more instances of objects or components, created within an instance of this class, e.g.:

```xml
<inst
   name="contrinst"
   class="controller"
   classarg="get_feedback->get_val,
             control_out->control_out" />
```

Here `name` is the instance name, `class` selects the class definition to instantiate, and `classarg` are the argument(s) for instantiation.

5 Future Work

There are a number of obvious extensions that will be pursued in the immediate future. In particular, we will also add support for combining a number of ports into a single port, which is important to avoid clutter in graphical representation of larger embedded systems. We will also work on introducing a graphical representation of timing constraints for method execution. Finally, we will add support for component repositories (the tool currently supports creating multiple instances from a class definition but there is no support for re-use of definitions between projects).

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 AESOP project.

References


time-constrained reactions,” Luleå University of Technology, Tech. Rep., 2005,
TimedReact.pdf

objects,” in Fifth IEEE Int. Symp. on Object-Oriented Real-Time Distributed Com-
puting (ISORC), 2002, pp. 155–158.
Implementation of SRP-DM scheduling for embedded real-time software

Authors:
Johan Eriksson, Simon Aittamaa, Jimmie Wiklander, Pawel Pietrzak, and Per Lindgren

Reformatted version of paper originally published in:
Implementation of SRP-DM scheduling for embedded real-time software

Johan Eriksson, Simon Aittamaa, Jimmie Wiklander, Pawel Pietrzak, and Per Lindgren

Abstract

Model and component based design is an established means for the development of large software systems, and is starting to get momentum in the realm of embedded software development. In case of safety critical (dependable systems) it is crucial that the underlying model and its realization captures the requirements on the timely behavior of the system, and that these requirements can be preserved and validated throughout the design process (from specification to actual code execution). To this end, we base the presented work on the notion of Concurrent Reactive Objects (CRO) and their abstraction into Reactive Components.

In many cases, the execution platform puts firm resource limitations on available memory and speed of computations that must be taken into consideration for the validation of the system.

In this paper, we focus on code synthesis from the model, and we show how specified timing requirements are preserved and translated into scheduling information. In particular, we present how ceiling levels for Stack Resources Policy (SRP) scheduling and analysis can be extracted from the model. Additionally, to support schedulability analysis, we detail algorithms that for a CRO model derives periods (minimum inter-arrival times) and offsets of tasks/jobs. Moreover, the design of a micro-kernel supporting cooperative hardware- and software-scheduling of CRO based systems under Deadline Monotonic SRP is presented.

1 Introduction

Model and Component Based Design (CBD) has over the years proven to be an effective means for the development of large software systems. Key drivers behind are to increase the efficiency of the design process (mainly through the re-use of components) and to improve the product quality (mainly by facilitating the validation process through allowing for separate verification of components).

With the ever increasing complexity of embedded systems, CBD of embedded software is starting to gain momentum.

In case of embedded, safety critical (dependable) systems, it is crucial that the underlying model and its realization can capture the requirements on the timely behavior of the system, in terms of both external and internal interactions, and that these requirements can be preserved and validated throughout the design process (from specification
to actual code execution). In many cases, the execution platform puts firm resource limitations on available memory and speed of computations, that must be taken into consideration for the validation of the system. Thus, a straight forward transition of traditional CBD models [1, 2] and tools does not suffice.

We undertake the component model and accompanying design methodology presented in our earlier work [3]. With the outset from a reactive system view, the behavior of the system can be observed as its output to incoming events. In general, system output also depends on previous events. Thus, embedded systems of scale are stateful. To deal with complexity and allow for CBD of such systems, we partition state and functionality into a hierarchy of Concurrent Reactive Components (CRC). Components are specified in terms of Concurrent Reactive Objects (CRO) instances and component instances [4]. CRO allows the intended behavior of the system to be expressed in terms of Time-Constrained Reactions (TCR) [5]. A CRO instance is either implemented in software (e.g., synthesized from the CRO model) or implemented by the system’s environment. This allows incorporating hardware interactions and legacy code (typically external software libraries) in the model, as long as their interface is compliant with the CRO model.

In this paper, we focus on code synthesis of CRO models and extraction of information for scheduling (during run-time) as well as offline schedulability analysis. To this end, we show how specified timing requirements (in the CRO model) are preserved and translated into scheduling information, specifically resource ceilings and priority levels for Stack Resource Policy (SRP) scheduling. Moreover, to support schedulability analysis, we detail algorithms that for a CRO model derives periods (minimum inter-arrival times) and offsets of tasks/jobs. Additionally we present the design of a micro-kernel supporting SRP Deadline Monotonic (SRP-DM) scheduling of CRO based systems, exploiting the interrupt hardware for efficient scheduling.

Section 2 gives the necessary background. We detail the underlying CRO model (and its abstraction to the CRC model) and briefly recapture the notions and key features of SRP. In section 3, we give an informal mapping from the undertaken CRO model to the notions of SRP. In Section 4, we present an example system (a process controller), and show how system state, functionality and temporal properties are captured and abstracted in terms of the CRC model. In Section 5, we propose a method for code synthesis of CRO models, and show how the timing requirements from the specification is preserved. In Section 6, we propose an algorithm that from a CRO model extracts resource ceilings and priorities for SRP-DM based scheduling. The algorithm is exemplified on the process controller, showing how timing requirements from the model specification is translated into resource ceilings and priorities for the scheduler. Furthermore, in Section 7, we present the design of an efficient SRP-DM kernel, and demonstrate how the derived resource ceilings and priorities are utilized at run-time. The paper is concluded in Section 8, where we summarize the presented proposals and results, and give directions for future research in the field.
2 Background

2.1 Underlying Model

Concurrent Reactive Object Model

The concurrent reactive object (CRO) model is the execution and concurrency model of the Timber programming language, a general-purpose object-oriented language that primarily targets real-time systems [6–8]. A subset of C, TinyTimber [9], implements the core features of Timber and uses CRO as its execution model.

In this section we briefly describe the main features of the CRO model and its abstraction to Concurrent Reactive Components, and discuss its implementation. We present an informal mapping from the CRO model to the notions of SRP. We focus on reactivity; object-orientation with complete state encapsulation; object-level concurrency with message passing between objects, the ability to specify timing behavior of a system, and the abstraction to components.

Reactivity

Reactivity is the defining property of the CRO model, which makes it particularly suitable for embedded system, since functionality of most, if not all, embedded systems can be expressed in terms of reactions to external stimuli and timer events. A reactive system can be described as follows: initially the system is idle, an external stimulus (originating in the system’s environment) or a timer event triggers a burst of activity, and eventually the system returns to the idle state. A reactive object is either actively executing a method in response to an external stimulus or a message from another object, or passively maintaining its state. Since initially the system is idle, some external stimulus is needed to trigger activity in the system.

Objects and state encapsulation

The CRO model specifies that all system state is encapsulated in objects $O_1, \ldots, O_n$. Each object has a number of methods, and the encapsulated state is only accessible from the object’s methods. This is also known as a complete state encapsulation. A name of a method $m$ can be fully expanded as $O_i : m$, where $O_i$ is the object of the method. Methods of two objects can be executed concurrently, but each method is granted an exclusive access to its object’s state, so only one method of an object can be active at any given time. Coupled with a complete state encapsulation, this provides a mechanism for guaranteeing state consistency under concurrent execution. The source of concurrency in a system can either be two (or more) external stimuli that are handled by different objects, or an asynchronous message sent from one object to another (more about message passing below).

To ensure that execution is reactive in its nature, each method must follow run-to-end semantics [10], i.e., it is not allowed to block execution awaiting external stimulus or a message. An example of this would be an object representing a queue: if a $\text{enqueue}$
method is invoked on an empty queue, it is not allowed to wait until data becomes available, but it must instead return a result indicating that the queue is empty.

Message passing and specification of timing behavior

In the CRO model objects communicate by passing messages. Each message specifies a recipient object (O) and a method (m) of this object that will be invoked. A message is either synchronous (SYNC(O, m)) or asynchronous (ASYNC(O, m)). The sender of a synchronous message blocks waiting for the invoked method to complete (with a possible result), while the sender of an asynchronous message can continue execution concurrently with the invoked method. Thus, asynchronous messages introduce concurrency into the system. An asynchronous message can also be delayed by a specific amount of time.

Timing behavior of a system can be specified by defining a baseline and a deadline for an asynchronous message (a synchronous message always inherits the timing specification of the sender). The baseline specifies the earliest point in time when a message becomes eligible for execution, which for an external stimulus corresponds to its “arrival time” and for a message sent from one object to another is defined directly in the code. If the defined baseline is in the future, this corresponds to delaying the delivery of the message. The deadline specifies the latest point in time when a message must complete execution, which is always defined relative to the baseline. Together, baseline and deadline form a permissible window of execution for a message (see Figure 1). Whenever we talk about timing behavior of asynchronous messages, we shall extend the notation to \(\text{ASYNC}(O, m, B, D)\), where \(B\) and \(D\) are respectively a baseline and a relative deadline of the message. As mentioned above, synchronous messages always inherit their timing specification from the initial asynchronous message.

Concurrent reactive objects can be used to model the system itself and its interaction with its environment (e.g., via sensors, buttons, keyboards, displays). Events in the physical world (such as pushing a button) results in an asynchronous message being sent to a handler object, and system output (e.g., flashing an LED) is represented as messages sent from an object to the environment.

Implementation of CRO

The implementation of an object instance can be either in software (e.g., synthesized from the class definition in the Timber language, or from a TinyTimber definition in C) or provided by the environment. This allows incorporating hardware interactions and
Concurrent Reactive Components

To deal with complexity, system state and functionality is partitioned into a hierarchy of concurrent reactive components (CRCs) encapsulating component instances and CRO instances (see [11]). Thus, interaction in between components will always be grounded by interaction in between objects (belonging to different components). This allows us to apply a consistent modeling for component interaction, component implementation and interaction with the environment (since a CRC is simply a collection of one or more CROs).

2.2 Stack Resource Policy

Stack resource policy (SRP) is a policy for scheduling real-time tasks with shared resources that permits tasks with different priorities to share a single run-time stack [12–14]. SRP applies directly to scheduling policies with dynamic and static priority, including e.g., Earliest Deadline First (EDF) which is used by current Timber and TinyTimber kernels, and Deadline Monotonic (DM) for which interrupt hardware of commonplace platforms can be utilized efficiently. SRP scheduling offers a number of advantages, mainly deadlock free execution and memory savings due to the shared runtime stack, but it also bounds the number of preemptions to at most two for each job instance. The bounded number of preemptions together with early blocking (a job is not allowed to start execution until all resources are available) allows for simple and sufficient schedulability test for EDF [12], and DM [15]. The traditional version of SRP only addresses single-core systems, however, SRP has also been extended to multi-core and multi-processor systems (see, for example, [16] and [17]).

3 Translation of the CRO model

In order to allow a CRO system to be represented in notions of SRP, we must first translate the CRO model into jobs and resources. A straightforward translation is possible:

- Each object $O_i$ is treated as a single-unit resource.
- Each asynchronous message $M = \text{ASYNC}(O, m, B, D)$ is treated as a job request, where the resulting job instance initially performs a resource request for $O$, invokes $m$, and releases $O$.
- Each synchronous message $M = \text{SYNC}(O, m, B, D)$ as a resource request for $O$, invocation of $m$ and release of $O$.
- The baseline $B$ of a message $\text{ASYNC}(O, m, B, D)$, corresponds to the arrival time of a job request and the deadline $D$ to the relative deadline of the job instance.
• Priority of an asynchronous messages \( M \) is denoted by \( p(M) \). Consider \( M_1 = \text{ASYNC}(O_1, m_1, B_1, D_1) \) and \( M_2 = \text{ASYNC}(O_2, m_2, B_2, D_2) \). In the DM scheduling priorities are defined so that \( p(M_1) > p(M_2) \) iff \( D_1 < D_2 \).

• In SRP, to every a message \( M \) is assigned a preemption level \( \pi(M) \), which should satisfy \( \pi(M_1) < \pi(M_2) \) iff \( D_1 > D_2 \). Values of preemption levels are natural numbers. For our case, with the DM scheduling, we can assign \( \pi(M) = p(M) \) for every \( M \) (since this assignment satisfies the condition for preemption levels, see (P1) in [18]).

This translation into notions of SRP is possible, since methods in the CRO model are run-to-end (blocking for future events to the system is prohibited), thus the execution of a message method can be seen as corresponding to the execution of a job instance.

### 3.1 Resource ceilings

Assume a message \( M_1 = \text{SYNC}(O_1, m_1, B_1, D_1) \) (or \( M_1 = \text{ASYNC}(O_1, m_1, B_1, D_1) \)). If the execution of \( m_1 \) can give rise to sending a synchronous message \( M_2 = \text{SYNC}(O_2, m_2, B_2, D_2) \) then we write \( M_1 \rightarrow M_2 \). Let \( \rightarrow^* \) be a transitive closure of \( \rightarrow \).

The initial message in a path defined by \( \rightarrow^* \) may be synchronous or asynchronous, the subsequent messages must be synchronous. The set of resources (objects) potentially requested by a message \( M_0 = \text{ASYNC}(O_0, m_0, B_0, D_0) \) is defined as

\[
\text{Objs}(M_0) = \{O_0\} \cup \{O \mid M_0 \rightarrow^* \text{SYNC}(O, m, B, D)\}
\]

We also say that \( M \) can lock objects \( \text{Objs}(M) \). A current resource ceiling \([O]\) is defined as

\[
\lfloor O \rfloor = \max \{\pi(M) \mid M \in \mathcal{M}, O \in \text{Objs}(M)\}
\]

where \( \mathcal{M} \) stands for all messages in the system. Note that \([O]\) can be computed statically (if \( \mathcal{M} \) is statically known). An algorithm for computing \([O]\) is discussed in Section 6. The current system ceiling \( \Pi \) is

\[
\Pi = \max(\{0\} \cup \{\lfloor O \rfloor \mid O \text{ is locked}\})
\]

The SRP states (cf. [18]) that a message \( M = \text{ASYNC}(O, m, B, D) \) is blocked from starting execution until \( \pi(M) > \Pi \). In addition to that, in order to be scheduled for execution \( M \) must have highest priority of all jobs, which in the case of DM follows directly from \( \pi(M) > \Pi \).

The derived resource ceilings, along with the base and deadlines of the messages provide sufficient information for the run-time scheduling under SRP, and will ensure that the system passing the analysis will be deadlock-free during execution.
4. Example System

Figure 2: The complete set of definitions of the controller system (components in blue and objects in yellow). Note that only the system definition is complete in the sense that it does not require any other components/objects in order to be instantiated (it has no provided/required interface).

Figure 3: The internals of the system component definition (object instances in yellow and component instances in blue) with arrows indicating the message paths.

Figure 4: The internals of the app component definition (provided/required interfaces in white and object instances in yellow) with arrows indicating the message paths.
4 Example System

The component model relies on message passing between components/objects and messages can only be sent between the provided/required interface of an component/object. Every component/object can define a provided and/or a required interface. The provided interface defines ports that can receive messages from other ports. For an object, a port corresponds to a method within the object and, for a component, a port corresponds to a method of an object defined at some level inside the component (components can be hierarchical). The required interface defines ports used for sending messages to the port of an other component/object. However, this requires that the sending port is connected to (has a reference to) a receiving port. Let us now present a small controller system and show how it is abstracted in terms of the CRC model.

At top level in the model we have two component definitions (app and system) and four object definitions (env, adapter, controller, and logger), see Figure 2. The env definition encapsulates specific hardware functionality and functions as a gateway between the app and the environment. It has a provided interface with two ports (get_ad, set_pwm) and a required interface with three ports (reset, int1, int2). The app component consists of a single instance of adapter, controller, and logger definition respectively, see Figure 4. The env interrupts (reset, int1, int2) are passed to the app via its provided interface (start, inc, dec). Internally, these are connected to the provided interface of the adapter. The role of the adapter is to forward the interrupts from the environment to the controller and logger in an application specific format (e.g. both inc and dec is forwarded to the controller’s port setpoint but with different arguments that will result in an increase/decrease of the setpoint). The controller functions as a simple feedback controller attempting to minimize the error (i.e. the difference between the measured process variable and the desired setpoint), by adjusting the control signal (control_out). The controller acquires the process variable through the get_feedback port of its required interface. It is connected to get_ad of the env via the required interface port get_val of the app. In a similar manner the controller’s control_out interface port is connected to set_pwm of the env.

The system component consists of a single instance of app and env respectively, see Figure 3. The system component definition is selected as the root for instantiation of the CRC model.

Each method (of a object definition) is implemented in C-code and message passing is done either asynchronously using the ASYNC primitive, or synchronously using the SYNC primitive (see Section 2). This is used to specify the timing behavior of the system, e.g. the controller defines the method process:

```c
int fb = SYNC(get_feedback, NO_ARG);
//Controller state update etc
SYNC(control_out, outval);
ASYNC(process, MS(10), MS(1), NO_ARG);
```

The process periodically acquire feedback values and writes control values as specified by the ASYNC primitive (with a period of 10 milliseconds and a deadline of 1
millisecond). Since baseline and deadline of synchronous messages are always inherited, there is no need to supply them when using the \texttt{SYNC} primitive in the code.

4.1 SRP Levels for the Example

Auto-generated output from the analysis for the given example. Shows all jobs with corresponding calltrees and deadlines (preemption levels is statically assigned according to deadline). Also all acquired resources and corresponding resource ceilings (rc).

```plaintext
#Starting points:
JOBREQUEST, d1: 10000
  -entry1 rc: 15 entry1 [envinst]
  --start rc: 15 adapter_start [adapterinst]

JOBREQUEST, d1: 50
  -entry2 rc: 15 entry2 [envinst]
  --inc rc: 15 adapter_inc [adapterinst]
  ---set_bor rc: 15 controller_set_bor [controllerinst]

JOBREQUEST, d1: 50
  -entry3 rc: 15 entry3 [envinst]
  --dec rc: 15 adapter_dec [adapterinst]
  ---set_bor rc: 15 controller_set_bor [controllerinst]

#Detected job requests (internal ASYNCs):
JOBREQUEST, d1: 10000
  -init rc: 20 logger_init [loginst]

JOBREQUEST, d1: 10000
  -init rc: 15 controller_init [controllerinst]

JOBREQUEST, d1: 20
  -log rc: 20 logger_log [loginst]
  --get_ad rc: 15 env_get_ad [envinst]

JOBREQUEST, d1: 15
  -process rc: 15 controller_process [controllerinst]
  --get_ad rc: 15 env_get_ad [envinst]
  --set_pwm rc: 15 env_set_pwm [envinst]
```

Here the resource ceiling is given by its deadline, this is not suitable for scheduling since interrupt hardware typically expects priorities given as integers \([0, 1, 2, \ldots]\) where 0 is most urgent. So all resource ceilings must be sorted and numbered, for this example we generate the following defines:
#define __[envinst]_rc 0
#define __[controllerinst]_rc 0
#define __[adapterinst]_rc 0
#define __[loginst]_rc 1

This means that [loginst] will be allowed to be preempted by any of the other job’s, but no other preemption is possible (preemption is not allowed between job’s having the same resource ceiling). In this example, all interrupts are messages to the same object ([envinst]), thus all other objects in the transitive closure of the highest priority message (to [envinst]) will have a resource ceiling equal to the preemption level of the highest priority message (i.e., 0). However, interrupt priorities are equal to corresponding message priority. Calculation of DM priorities is trivial and emitted in a similar fashion as resource ceilings (i.e., as defines). Note that names such as [loginst] are auto-generated by the compiler, but renamed in the paper for clarity.

5 Code Synthesis of Framework

The code synthesis framework is depicted in Figure 5. The Reko IDE [11] enables design of CRC models. It provides a GUI where the user can create, browse, and edit the CRC models which are represented graphically in the IDE. The illustrations of the example system presented in Section 4 are screenshots from the Reko IDE.

Code generation is an integrated part of the Reko IDE, and is covered in section 5.

5.1 XML Format

A model is stored in as an XML file that captures both system structure and implementation. For a description of the internal format see [11]

5.2 Requirements on the methods

All methods are written in the C language. However, the C language allows us to specify behavior not allowed by the CRO model (such as entering an infinite loop, waiting for
Thus, in order for a method written in C to be a valid CRO method it must comply with the following rules, it must

1. be run-to-end (complete execution within a finite amount of time),
2. not access any global memory (state) outside of its object, and
3. not invoke a method of another object directly (without using the ASYNC or SYNC primitive).

Compliance with two and three are partly enforced by name scoping, i.e. the framework will generate local defines for names in the current scope (method). However, the system designer can still force an incorrect behavior by directly calling methods (of other objects) or accessing global memory (e.g., using pointers). Strictly enforcing two and three would require implementing a parser for a subset of the C language (disallowing pointer arithmetic, extern keyword, etc.). The first rule is currently not enforced in any way and compliance must be ensured by the system designer. One approach to enforcing the first rule is to perform worst-case execution time analysis \[19\] on the methods, but this is outside the scope of this paper.

Below is an example of emitted C code, showing local defines for names in the current scope.

```c
// local name bindings
#define get_feedback ......
#define control_out ......
#define process ......
// method implementation
int controller_process(OBJ* self, int arg){
    int fb = SYNC(get_feedback, 0);
    // Controller state update etc., eg:
    self->state.out = 10;
    SYNC(control_out, self->state.out);
    ASYNC(process, MS(10), MS(1), 0);
}
#undef get_feedback
#undef control_out
#undef process
```

5.3 Code Synthesis

The CRC model contains definitions, instances, methods, and states. From this C functions and object definition structures (C typedefs) can be emitted. Additional information required to compile (using a C-compiler) the system into an executable binary is:

- A static object structure (see below).
• Defines for preemption levels and resource ceilings. (covered in: 6 and 4.1)
• A Kernel (7)

Static object structure

Is generated by transforming the CRC model to a model consisting only of object instances (CRO instances). From this model it is possible to generate one C-struct containing all instances in the system.

The timing specification of the model is preserved during code synthesis and later used by the run-time kernel. (For reference see the generated C-function above):

6 Resource ceiling and Priority Extraction for CRO

In this section we give algorithms for calculating resource ceilings, interarrival time (period) and offsets of jobs (asynchronous messages).

We define the message passing graph of the program as

\[ G = (S, A, N) \]

where

\[ N = \{ m_1, \ldots, m_{\text{method}} \} \] (\( m_i \) denotes a method)
\[ S = \{ \frac{1}{\text{sync}}, \ldots, \frac{1}{\text{sync}} \} \subseteq N \times N \] (SYNC(\( . \)) messages.)
\[ A = \{ \frac{1}{\text{async}}, \ldots, \frac{1}{\text{async}} \} \subseteq N \times N \] (ASYNC(\( . \)) messages.)

\( m \) represents a method, each \( m \) has one unique receiving object, represented as \( O(m) \) (i.e. one method cannot have two different receiving objects).

\( S \) and \( A \) is not initially known. Defining \( \kappa(m_i) \) as the set synchronous messages that \textbf{may} be sent from a method \( m_i \) and \( \alpha(m_i) \) as the set of asynchronous messages (that may be sent), then \( S \) and \( A \) can be calculated as

\[ S = \bigcup_{i=1}^{\text{method}} \kappa(m_i) \]
\[ A = \bigcup_{i=1}^{\text{method}} \alpha(m_i) \]

For a CRO program to be valid, \( S \) must be acyclic (otherwise the program may contain deadlocks).

Let \( \beta(\frac{1}{\text{async}}) \) denotes the baseline and \( \delta(\frac{1}{\text{async}}) \) the deadline of an asynchronous message \( \frac{1}{\text{async}} \). If invoking \( m_i \) \textbf{may} result in more than one asynchronous message to \( m_j \), then \( \alpha(m_i) \) will return a message \((m_i, m_j)\) with the shortest baseline and deadline (baseline and deadline may come from two different messages). The resource ceiling of an object can then be calculated as

\[ \lceil O_i \rceil = \min(C \bigcup V) \]
\[ V = \{ \delta(\frac{1}{\text{async}}) | \frac{1}{\text{async}} \rightarrow (m_j, m_i) \in A, O(m_i) = O_i \} \]
\[ C = \{ \delta(\frac{1}{\text{async}}) | \frac{1}{\text{async}} \rightarrow (m_j, m_k) \in A, (m_k, m_i) \in S, O(m_i) = O_i \} \]
where $\mathcal{S}^*$ denotes the transitive closure of $\mathcal{S}$. $V$ denotes the set of asynchronous messages to any method $m_l$ where $O(m_l) = O_i$, and $C$ the set of asynchronous messages to any method that may (transitively) send a synchronous message to $m_l$ where $O(m_l) = O_i$.

Additionally, for a program to be analysed for periods and offsets, the following must hold:

- Any Strongly connected component (SCC) of the graph $G$ may not be reachable from another SCC of $G$
- Any SCC of $G$ may only contain one cycle
- Multiple syns/asyncs from $m_i$ to $m_j$ is not allowed

Then we can define $\Gamma$ as a function that calculates the period of any given $\xrightarrow{\cdot \cdot \cdot}$, $\Gamma(\xrightarrow{\cdot \cdot \cdot}) = \sum_j \beta(\xrightarrow{\cdot \cdot \cdot})$ for all $j$ part of a SCC that can reach $\xrightarrow{\cdot \cdot \cdot}$.

For calculating the offset for a $\xrightarrow{\cdot \cdot \cdot}$ relative to method $m_j$ we define a function $\Delta(m_j, \xrightarrow{\cdot \cdot \cdot})$ that will return a set of all possible offsets. To calculate this function we must first find all paths from $m_j$ to (the called method of) $\xrightarrow{\cdot \cdot \cdot}$, for each path we calculate: $\sum_k \beta(\xrightarrow{\cdot \cdot \cdot})$ for all $k$ part of the path. The set of results of this calculation is the result of the function $\Delta$.

7 Kernel Design for efficient SRP DM scheduling of CRO

The goal of the kernel design is to provide an efficient and predictable implementation of the CRO semantics (see Section 2.1). In order to achieve this we have decided to use the deadline monotonic (DM) scheduling policy with stack resource policy (SRP). The benefits of SRP are well known (see Section 2.2), and DM was primarily chosen to allow for an efficient usage of the priority based interrupt hardware of the target architecture (i.e. Cortex-M3 [20]), more on this in Section 7.3. The scheduler requires that the priorities and preemption levels of messages, and resource ceilings of objects are known. In our case, these are automatically generated from timing specifications in the model during code synthesis (see Section 5).

7.1 Message Passing

The kernel must support two types of messages, asynchronous and synchronous. An asynchronous messages must be queued until the time it becomes eligible for execution (see permissible window of execution, Figure 1). Queued messages are stored in the timer-queue (sorted by ascending baseline), once the baseline of a message is passed (i.e., the baseline expires) the message is transferred to the active-queue (sorted by descending priority), and considered when scheduling decisions are made (see Section
7.2). The transfer of messages from the timer-queue to the active-queue is initiated by a timer interrupt, i.e. a hardware timer is configured to generate an interrupt when the earliest baseline in the timer-queue expires. When an asynchronous message is scheduled for execution the recipient object-resource (see Section 3) is requested, the method is invoked and (when it returns) the object-resource is released. A synchronous message is executed similarly (i.e., object-resource requested, method invoked, and object-resource released).

7.2 Scheduling

The scheduler only considers messages that are in the active queue, and it is invoked when either a baseline expires (timer interrupt) or the system ceiling is lowered (can only happen when a method of a message returns). Since messages are transferred from the timer-queue to the active-queue, a message with a higher priority than the currently executing message may be eligible for execution. If the head of the active-queue, $M_i$, has the highest priority, it is only allowed to execute if $\pi(M_i) > \Pi$ (see Section 3). Assuming $\pi(M_i) > \Pi$, $M_i$ is transferred from the active-queue to the running-stack (containing all messages that have been allowed to start execution) and executed. If we instead assume $\pi(M_i) \leq \Pi$ then $M_i$ can only become eligible for execution when the system ceiling is lowered, i.e. when a message completes execution and releases the object-resource, thus a new scheduling decision must be made when execution of a message completes. Similarly, if $M_i$ does not have the highest priority it can only become eligible for execution when an asynchronous message completes execution. Synchronous messages that are generated by an executing message are always scheduled immediately, since SRP guarantees that all resources (objects) are available and the priority of a synchronous message is always inherited from the sending message.

7.3 Implementation of data structures

The data structures required by the kernel:

- **timer-queue**: messages with a baseline in the future, sorted by ascending baseline
- **active-queue**: messages with an expired baseline, sorted by descending priority
- **running-stack**: messages that have been allowed to start execution
- **system ceiling**: the maximum of object ceilings of locked objects

The active- and timer-queues are currently implemented as sorted lists. While there are more suitable data structures (e.g., heaps or balanced search trees), sorted lists are easier to implement and when the number of items in the list is small the performance is comparable to more advanced structures [21]. The running-stack is simply a last-in-first-out stack, and the system ceiling is implemented as an unsigned integer. To allow for efficient usage of common interrupt hardware (priority-based), some values of the system ceiling are implemented in hardware.
Example scheduling using interrupt hardware

A subset of the values of the system ceiling are implemented in hardware, i.e. the interrupt mask register of the processor. To demonstrate the benefits, we consider a simple example: let

$$\mathcal{M} = [M_1 = \text{ASYNC}(\text{serial}, \text{mread}, B_{\text{inherited}}, 10\text{ms}),$$
$$M_2 = \text{ASYNC}(\text{serial}, \text{mwrite}, B_{\text{inherited}}, 50\text{ms})]$$

From the definitions in Section 3 follows: $p(M_1) > p(M_2) \implies \pi(M_1) > \pi(M_2)$, and $[\text{Oserial}] = \pi(M_1)$. $M_1$ is sent from the data-ready interrupt of the serial port (with interrupt priority $p(M_1)$), and $M_2$ is sent from the ready-to-send interrupt (with interrupt priority $p(M_2)$). Since $B_{\text{inherited}}$ in the interrupt context corresponds to the time when the interrupt handler is invoked, the messages can be placed directly into the active-queue.

Let all possible values of the system ceiling that can be represented in hardware be defined as

$$\mathcal{H}, \mathcal{H} \subset \mathbb{N}_0$$

and the hardware system ceiling as

$$\Pi, \Pi \in \mathcal{H}$$

Then, assuming $\{\pi(M_1), \pi(M_2)\} \subset \mathcal{H}$, the messages can be placed directly into the running-stack. This follows from the one-to-one mapping between priority and preemption level, i.e. $p(M) = \pi(M)$, and for any messages $M_j$ and $M_k$ where $\{\pi(M_j), \pi(M_k)\} \subset \mathcal{H}$. If $M_j$ is currently executing and an interrupt handler (with priority $p(M_k)$) that generates $M_k$ is invoked, then $\Pi < \pi(M_k)$ (otherwise the interrupt would be masked by the hardware system ceiling) and $p(M_k)$ must be the highest priority since if $M_j$ is executing then $\Pi \geq \pi(M_j)$, and hence $p(M_k) > p(M_j)$. Thus, an interrupt handler is only invoked (and corresponding message generated) if it has the highest priority and all resources are available.

Discussion and limitations of implementation

In the previous example, we demonstrate how interrupt hardware is exploited to perform scheduling of messages. This hardware scheduling is limited by the number of interrupt priorities of the hardware. Thus, if no one-to-one mapping exists between message priorities and interrupt priorities, then co-operative software and hardware scheduling is required, i.e. two or more messages with different priorities must share a single interrupt priority. Whenever an interrupt priority is shared by different message priorities, the software scheduler is invoked to determine if a generated message should be executed.

In the current implementation, transfer of time-delayed messages from timer-queue to active-queue is initiated by a timer interrupt. The priority of this interrupt must be set to that of the highest priority message in the timer-queue. Let $p_l$ be the lowest priority of messages in the timer-queue, and $p_h$ be the highest, then all messages (in
the system) with priority $p_m$, $p_i < p_m < p_h$ will be subject to scheduling overhead (i.e. transfer of message from timer-queue to active-queue). However, if a system contains $n$ time-delayed messages with $p$ unique priorities ($n \geq p$), then this scheduling overhead can be mitigated by using $p$ different timers, i.e. by minimizing the set of priorities in each timer-queue (one for each timer).

8 Conclusion

In this paper we have given an informal mapping from the undertaken CRO model to the notions of SRP. And shown how a CRC model can be translated into a CRO model. We have shown for an example system (process controller) how state, functionality and temporal properties are captured and abstracted in terms of the CRC model. We have proposed a method for code synthesis of CRC models, and shown how the timing requirements from the specification was preserved. We have proposed an algorithm that from a CRO model extracts resource ceilings and interrupt priorities for SRP-DM based scheduling. Additionally, to support schedulability analysis, we detail algorithms that for a CRO model derives periods (minimum inter-arrival times) and offsets of tasks/jobs. The algorithm was exemplified on the process controller, showing how timing requirements from the model specification is translated into resource ceilings and interrupt priorities for the scheduler. Furthermore, we have presented the design of an SRP-DM kernel supporting cooperative hardware- and software-scheduling utilizing the derived resource ceilings and interrupt priorities at run-time.

8.1 Current and future work

Current and future work includes SRP based scheduling analysis for the presented model[22], with the aim to make safe schedulability estimations by taking the kernel overhead into consideration. Additionally we are investigating methods to derive and minimize the total memory requirement of a CRO system.

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 AESOP project (grant no. 258682).

References

References


Utilizing an IDE for component-based design of embedded real-time software for an autonomous car

Authors:
Jimmie Wiklander, Johan Eriksson, and Per Lindgren

To be submitted.
Utilizing an IDE for component-based design of embedded real-time software for an autonomous car

Jimmie Wiklander, Johan Eriksson, and Per Lindgren

Abstract

Embedded systems are inherently reactive and they often operate under resource and real-time constraints. Hence, a model supporting these fundamental properties is desirable for an efficient and correct design process. To this end, we have developed a component model and design methodology that enables component-based design of embedded software. Moreover, we have developed an Integrated Development Environment (IDE) that ensures a systematic development of software in accordance with the methodology. In this paper we review the IDE and its underlying modeling framework and demonstrate its applicability through a use-case with typical properties - a control system with real-time constraints and hardware interaction.

This paper describes the whole design process – requirements specification, conceptual model, component-level design, component implementation, code synthesis, testing, and verification. In particular, we show how the modeling framework enforces a static component and communication structure reflecting the concurrent behavior of the system and its interaction with the environment (target platform). Our results (a working prototype) indicate applicability of the presented design approach, and show the IDE’s ability to generate efficient executables. Additionally, we show how to prove the schedulability of the target system in the IDE using standard scheduling theorems and machine assisted WCET measurements.

In conclusion, this paper demonstrates both the applicability of the software design methodology and the functionality of the developed IDE.

1 Introduction

Tool support is instrumental for efficient and correct design of embedded software. In this paper we show how to design the software for an autonomous car using an Integrated Development Environment (IDE) (Fig. 1).

The IDE supports component-based software design for resource-constrained embedded systems with hard real-time requirements. The underlying modeling framework [1] is based upon a reactive view in which the functionality of the system is modeled as reactions to events that originate in the systems environment or internally within the system itself. Moreover, the modeling framework supports specifying the timing constraints for the reaction to an event.
Figure 1: The IDE illustrated in terms of a screenshot of the autonomous car design presented in this paper.

The IDE supports defining a model of the software system based on interaction with the system platform (e.g., a microcontroller and its peripherals in terms of sensors, motors, etc.). The structure of the software system is defined graphically and the functionality is defined using a subset of C. From the model, it is possible to generate C code that can be compiled for execution on a hardware platform assisted by a real-time kernel. Distinctive to our approach is that timing requirements are captured as part of the model and are preserved throughout the design process; they are utilized by our kernel for scheduling at run-time. A brief overview of the tool can be found in [2].

The specification for the autonomous car comes from a mechatronics project course at the university. In this course, the students are divided into teams and each team designs and builds their own autonomous car by modifying a standard radio-controlled model car. The function of the autonomous car is to track an IR signal coming from a beacon and then direct its way towards the beacon and make a full stop in front of it. The course is concluded with a single-elimination tournament where the cars race each other on who arrives first to the beacon. In this case study, the target platform for the software design is one of the student cars (including its onboard sensors, servos, and microcontroller). This system is representative for the type of systems addressed by the IDE, i.e., resource-constrained embedded systems with hard real-time requirements.

The rest of the paper is organized as follows. In the next section (Section 2) we present an overview of the IDE. It is followed by related work (Section 3). The problem statement as defined in the mechatronics course is presented in Section 4. In Section 5 we outline the conceptual model of the autonomous car and in Section 6 we present the target platform for the software implementation. The software design for the autonomous car is presented in Section 7 and the details of the program analysis are presented in Section 8.
Finally, in Section 9 we present the conclusions of this case study.

2 IDE Overview

The aim of the IDE is to construct a model of the software system based on its interaction with the environment and to generate efficient C code from the model for execution on bare metal. More specifically, the system structure is defined graphically by composing components and the functionality of the components is defined using C code. An important aspect of the model is that it supports specifying not only the functionality of each component but also its intended timing behavior. Moreover, the model supports independent definition of components, that is, the properties defined locally for a component are not affected by external factors. For a real-time system to be correct its run-time behavior should comply with the specification. This is achieved by preserving the model and its timing requirements in the generated code all the way to the executable and allowing for the scheduler to utilize the timing requirements for scheduling at run time. Generating code from a static component composition enables using static verification and optimization techniques to achieve low overhead and predictability of the system behavior at run-time.

2.1 Modeling framework

The modeling framework is based on the notions of components and time-constrained reactions. A more detailed presentation of the framework can be found in [1]. Note that in [1] we make a distinction between reactive objects [3] and components. In this work we do not make this distinction and we denote the reactive objects themselves as concurrent reactive components. In this section we highlight the key aspects of time-constrained reactions, concurrent reactive components, and discuss modeling interaction between concurrent reactive components as well as a system’s interaction with its environment.

Time-Constrained Reactions

Time-constrained reactions [4] lie at the heart of our model. Interaction between the system and its environment, as well as between components of the system is modeled as discrete events (i.e. events occurring at specific points in time). Following a reactive approach, functionality is specified in terms of time-constrained reactions to such events. Timing requirements on system operation can be specified by defining the earliest and the latest reaction time (baseline and deadline) relative to the time of the event triggering the reaction. The time window between the reaction baseline and its deadline is called a permissible execution window for this reaction (Fig. 2).

Concurrent Reactive Components

The fundamental modeling construct of the framework is a concurrent reactive component (CRC) [3]. Note that what is denoted CRC in this work is known as reactive object
Figure 2: A permissible execution window for a reaction to an event. Here $t_{after}$ is the baseline offset, $t_{before}$ is the time between the baseline and the deadline.

in [3]. Each component may have a state, one or several methods and it may encapsulate a number of other components, creating a hierarchical structure. It is reactive in the sense that it reacts to an incoming event by executing one of its methods or invoking a method of an underlying component. Each CRC has a provided interface (the methods, or input ports, of the component that can be invoked by other components) and a required interface (the methods, or output ports, in the component’s environment that it may invoke). In the model, an output port in the required interface of a component can be linked to an input port in the provided interface of another component, creating a communication path between two different CRCs. Multiple output ports can be connected to a single input port. Each CRC is either idle (maintaining its state) or executes a method. Methods execute run-to-end, that is, once a method has started execution, no other method of the same CRC may preempt it. However, any two methods of different CRCs may execute concurrently. A method’s code can perform computations on local variables, read/mutate the component’s state, and invoke a method of the same or another component by sending a message to it.

Messages sent between components can be of two kinds: asynchronous, which are executed concurrently with the caller and can be delayed by a certain amount of time, and synchronous, with the caller blocked until the invoked method completes execution, optionally returning a value (such methods cannot be delayed). The permissible execution window of an asynchronous message can be either inherited or explicitly specified in the code relatively to the caller’s baseline; in either case, it is viewed in the model as a separate reaction. A synchronous message, on the other hand, always inherits the caller’s time constraints and is viewed as a part of the original time-constrained reaction. Both asynchronous and synchronous messages can carry data (the values of arguments of the invoked methods).

Classes and Instances

Every CRC has a class definition from which instances can be created at run-time. The class definition can be defined at the top level or locally within another class; in the latter case, it can only be used inside that class and the classes that are hierarchically encompassed by the class. A CRC definition defines the component’s state variables, methods (these are written manually in C), and the provided and required interfaces of the component. In the C code of a method we may:

1. define local variables,
2. read/update the component’s state,

3. perform calculations on the component’s state, method argument, and local variables,

4. invoke an asynchronous method of this component or of another component using the name of an output port or a port of the provided interface of an encapsulated component (see Fig. 2):
   \[ \text{ASYNC(PortName, BaselineOffset, RelativeDeadline, Argument)} \]

5. invoke a synchronous method of another component using the name of an output port or a port of the provided interface of an encapsulated component:
   \[ \text{SYNC(PortName, Argument)} \]

6. return a value (only meaningful if the method is synchronous).

The current implementation of the tool does not verify that the C code complies with the model’s requirements as outlined above but this will be added later.

Kernel support and interaction with the environment

A system model (i.e. the system’s structure in terms of CRCs and communication paths between them, and C code written for each method) is used to generate C code representing the software system. Execution on a hardware platform also requires infrastructure supporting the model in the form of a lightweight kernel supporting scheduling of method execution and message passing between components.

Embedded systems interact with their environment and this must be reflected in the model. This is modeled using a special construct, an environment interface that defines an interface to the hardware. An instance of an environment interface enables reading from hardware registers and writing to them and is also used to connect specific hardware interrupts to asynchronous methods of particular components. The model does not make any assumptions regarding the behavior of the environment.

Sometimes it is necessary to include legacy software that is not based on the CRC model (typically, external software libraries). Such libraries can be treated as part of the system’s environment, and we can use an environment interface as a wrapper to legacy code.

3 Related Work

There is a strong interest for component-based design of embedded software. As a result, various approaches and component models have been proposed. We have selected a few of these approaches for a comparison against our approach as presented below. In
particular, we have selected approaches targeting software development for resource-constrained embedded systems. Such systems are typically interrupt driven, i.e. they must respond to events that originate in their environment (i.e. the physical world), often with real-time requirements. Moreover, it is becoming increasingly common for these systems to perform multiple tasks at the same time. Hence, the normal operation for such systems is the reactive execution of concurrent events.

Our model for developing systems is reactive, which means that the natural behavior of the modeling constructs (CRCs) is to react to events. Moreover, the model allows to express concurrency simply by specifying functionality in terms of separate CRCs. Hence, the model eliminates the need for manually specifying concurrency (e.g. threads, mutexes, etc.) which often lead to incorrect code with bugs that are are hard to decipher (race conditions, deadlocks, etc.). In our approach, this view can be maintained even for systems with scheduling.

TinyOS [5] targets building concurrent, reactive applications for resource-constrained systems. Its component model is similar to the CRC model. A module in TinyOS can be seen as a CRC that is prohibited to encapsulate inner CRCs. Moreover, a configuration can be seen as a CRC that is prohibited to contain anything other than inner CRCs. Hence, the TinyOS model supports constructing hierarchical models in the same manner as the CRC model. Moreover, TinyOS modules encapsulates state, as does CRCs. In contrast to the CRC model, the concurrency is not modeled in terms of components, instead TinyOS expresses concurrency in terms of asynchronous code that can be invoked by interrupt handlers. The execution model in TinyOS is based on run-to-completion tasks and event handlers. The standard TinyOS task scheduler uses a non-preemptive, FIFO scheduling policy. TinyOS application are well suited for execution on resource-constrained systems. However, TinyOS offer limited support for developing real-time applications.

The Rubus Component Model (RCM) [6, 7] targets development of resource-constrained embedded real-time systems. Like the CRC model, it supports constructing a model of a system by composing modeling constructs graphically. However, as in the TinyOS case, RCM distinguishes between the basic component which is called a Software Circuit (SWC) and the assembly which enables constructing hierarchical models. SWCs are used for encapsulating software functions. The software functions are mapped to tasks at a later stage of the design process. Similar to our approach, RCM supports expressing timing requirements and analyzing the resource demands for a particular hardware platform. Further, the run-time system supports executing all tasks on a shared stack. This is supported in our methodology as well through SRP resource management. In contrast to the CRC model and the REKO IDE, RCM and its tool suite has been used in the industry for the development of embedded software for many years.

It is also worth mentioning AmbientRT [8] which is a real-time operating system for embedded devices with resource-constraints in terms of memory, processing, and energy resources. This approach does not offer any tools for composing components to a software model. However, it supports loading blocks of precompiled software and executing the blocks dynamically. Hence, it enables creating new software configurations online. Like in
Figure 3: A schematic of the two competing cars and the race track. Each car has an optical sensor that can be panned which enables measuring the horizontal angle $\delta$ between the IR emitting rows. The angle $\delta$ is used to determine the direction and distance to the beacon relative to the car.

our approach, a guarantee on meeting the real-time constraints for each task can be given through analysis. The AmbientRT kernel supports real-time scheduling of tasks using SRP-EDF. In contrast, the REKO IDE targets SRP-DM scheduling which is less flexible compared to SRP-EDF but allows for more efficient implementation onto commonplace hardware. However, for the autonomous car it was sufficient to use non-preemptive DM scheduling (similar to the scheduling provided by TinyOS).

4 Problem Statement

The team should design and build an autonomous car. The car should track an IR signal coming from a beacon and then direct its way towards the beacon and make a full stop in front of it. The car should be equipped with an optical sensor which can detect the IR signal from a distance of 10 meter in daylight. The sensor signal should be processed by electronic hardware designed and built by the team. The processed signal should then be feed to the ADC in the onboard AVR microcontroller. The whole sensor unit will be mounted on a standard RC servo which enables the sensor to be panned in the horizontal plane. The beacon has two vertical rows of IREDs which emit IR radiation with a wavelength of 900 nm. The horizontal distance between the rows is 95 mm. The horizontal angle between the IR emitting rows ($\delta$ in Fig. 3) can be measured by the panning optical sensor. The panning servo, the speed and front wheel steering angle of the car is controlled by the microcontroller.

The cars compete in a single elimination tournament. In every round two cars compete in a best two-of-three format. The cars are placed side by side behind a starting line at a 10 m distance from the beacon (see Fig. 3) facing the beacon\(^1\). On a given start signal the cars are activated by the team. The cars should drive autonomously to the beacon as fast as possible. The goal position is defined as a circle with one meter in diameter placed between the start line and the beacon in such a way that the vertical projection of the emitting rows lies on the circle. The winner is the car which first enters the circle with

\(^1\)In this work we want the autonomous car to be able to locate and track the beacon even if it is not facing the beacon to start with.
a wheel and stops (the wheel must remain within the circle after the car has stopped). The beacon must not be touched by the car.

5 Conceptual Model

Following the problem statement the car should continuously track the IR signal and drive towards the beacon (a.k.a. guidance). Guidance is usually implemented in accordance with some routing strategy. From the problem statement we formulated the following conceptual model.

5.1 Target tracking

The beacon has 14 IREDs positioned in two vertical columns with a horizontal distance of 95 mm. The IREDs emit an IR pulse with 900 nm wavelength. The optical sensor of the car measures the intensity of the signal while being panned from side to side.

The angular direction of the target ($\beta$ in Fig. 4) is defined as the angular direction between the centerline of the car and the imaginary line from the midpoint of the rear axis to the midpoint of the IRED columns. This angle is derived by monitoring the panning angle of the sensor when the sensor registers the outer IRED column edges during a panning cycle (panning from one side to the other). The angular direction is equivalent to the intermediate angle between the two registered angles. When the car is far from the target the column edges will be recognized as a single entity making the angular direction equivalent to the panning angle.

Note that in order to get the accurate panning angle the sensor should be mounted at the midpoint of the rear axis. In our case there will be a deviation in the registered panning angle due to the sensor being positioned at the midpoint of the front axis instead of the midpoint of the rear axis. However, the deviation was considered to be acceptable for our purposes.

The distance $d$ to the target (see Fig. 4) is calculated using the following equation

$$d = \frac{b}{\cos(\delta)}$$  \hspace{1cm} (1)

where $b$ is the horizontal distance between and $\delta$ is the horizontal angle between the two IRED columns (see Fig. 3).

5.2 Routing Strategy

The routing strategy is based on following a circular arc passing through the midpoint of the rear axis of the car and the target point where the centerline of the car is as tangent to the circular arc (see Fig. 4). Hence, the radius of the circular arc depends on the positioning of the car relative to the target. The following equation computes the steering angle ($\alpha$) that the car should take on in order to follow the circular arc.

$$\alpha = \arctan \frac{2a \sin \beta}{d}$$  \hspace{1cm} (2)
where $\beta$ is the angle between the centerline of the car and the imaginary line from the midpoint of the rear axis to the target, $a$ is the distance between the front and rear axis, and $d$ is the distance between the midpoint of the rear axis and the target.

5.3 Guidance

Guidance uses the targeting information to control the steering and speed of the car. It continuously calculates the steering angle (Eq. 2) based on the latest sensor readings. The speed of the car depends on the distance between the car and the target. When the target is far away the speed is set to maximum and when it approaches the target the speed is decreased. When the target is not visible by the sensor the speed is set to low and the steering is set to turn full left aiming to locate the target.

5.4 Modes of Operation

The car should have two different modes of operation: neutral and drive. The car should start in the neutral mode and it should be possible to switch between the modes manually by pushing a button on the car. In the neutral mode the car should track the target and calculate the route but it should not move. In the drive mode the car should direct its way towards the beacon and make a full stop in front of it.

6 Target Platform for the Software Implementation

The RC car has an onboard microcontroller (AVR – AT90CAN128) that enables controlling the servos of the car and reading data from the optical sensor. However, a software implementation must be provided in order to enable the car to autonomously track the target and direct its way towards the target. More specifically, our software implementation should read data from the optical sensor, calculate the position of the target (heading and distance), calculate a route to the target following a specific routing strategy, and have the car to follow the anticipated route by setting the control signals of the servos.
<table>
<thead>
<tr>
<th>Autonomous car function</th>
<th>AVR port</th>
<th>Signal type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR sensor signal</td>
<td>PF0 (ADC0)</td>
<td>DC input</td>
</tr>
<tr>
<td>IR sensor position servo</td>
<td>PE3 (OC3A)</td>
<td>PWM output</td>
</tr>
<tr>
<td>steering servo</td>
<td>PE4 (OC3B)</td>
<td>PWM output</td>
</tr>
<tr>
<td>speed servo</td>
<td>PE5 (OC3C)</td>
<td>PWM output</td>
</tr>
<tr>
<td>neutral/drive button</td>
<td>PE6 (INT6)</td>
<td>interrupt (falling edge)</td>
</tr>
</tbody>
</table>

Table 1: Mapping between the functions of the autonomous car and the ports of the AVR.

6.1 AVR Peripherals

The car has three electrical motors: one main motor that control the speed of the car and two servo motors that control the steering of the car and the panning of the optical sensor. The main motor is controlled by an engine controller servo. Each servo is connected to an output port of the AVR and can be operated using the Pulse Width Modulation (PWM) capabilities of the AVR (see Table 1). The PWM output is configured by setting the appropriate control registers as specified by the data sheet [9].

The optical sensor is attached to a servo motor which enables panning the sensor in the horizontal plane (see Fig. 3). The sensor has a built in filter to distinguish the pulsed IR signal coming from the beacon from other signal sources. The output from the sensor is a DC signal that is proportional to the intensity of the pulsed IR signal. It is connected to the port ADC0 of the AVR (see Table 1) which enables software to access sensor data by triggering and reading the results of AD conversions. A single AD conversion can be started by writing a logical one to the ADC start conversion bit (see [9]). When the conversion is completed the hardware clears the start conversion bit. An efficient way to implement the conversion is to busy wait until this bit is cleared. The alternative would be to have the hardware signal the completion through an interrupt.

Further, the car has an onboard push button that is connected to one of the input ports of the AVR (see Table 1). The button should be used for switching between the two operation modes of the car (as described in Section 5.4).

6.2 Physical Constraints

The following constraints must be considered in the autonomous car software implementation for the target platform.

**Button events**

When pressing the button there will be a “bouncing” effect due to the mechanical construction of the button. This results in multiple interrupts being delivered to the microcontroller. This effect can be eliminated by disabling the interrupts for a short time upon receiving a button interrupt. We denote this time period $T_{bounce}$. From measurements we get $T_{bounce} < 1$ ms.
7. Software Design for the Autonomous Car

The software for the autonomous car was developed using the IDE described in Section 2. The IDE supports constructing software system models in a graphical environment and generating efficient C code from the model for execution on bare metal e.g. a microcontroller. In this section we present the software model for the autonomous car and we describe the process of generating an executable from the tool.

7.1 Software Model

The goal of the software design process is to partition the functionality of the system into a hierarchical model using the CRC modeling constructs. In order for the model to be able to interact with the environment it must also include environment interfaces. They provide an interface to the underlying target system (i.e. the microcontroller and its peripherals) and enables resource protection (a resource can only be accessed through its environment interface). Hence, the environment interfaces disclose all the resources used by the software. The design process is described in detail in [1]. Next, we present the software model for the autonomous car that was developed using the aforementioned design process.

The model must specify one CRC as the root (by setting it as root in the configuration pane of the IDE) and this CRC alongside its content will be instantiated at run-time. The root CRC serves as the container for all the hierarchical CRC instance structures in the model and it has no provided or required interface. Moreover, it encapsulates all the environment interfaces in the model. The environment interfaces defines the provided and required interface ports that provide access to specific services in the hardware platform (reading sensors, controlling motors etc.). The provided interface ports abstracts hardware registers of the target platform and the required interface ports abstracts events that originate in the target platform (e.g. the interrupt vector of the target platform may invoke a method in the model through the required interface port of the environment interface component as a reaction to a hardware interrupt). In order for the software system (i.e. CRC) to access a service (i.e. a set of registers) it must be linked with the provided interface of the environment interface (by connecting the required interface of the CRC to the provided interface port of the environment interface).

The root CRC in the autonomous car model has the name autonomous_car and its definition is depicted in Fig. 5. It defines three environment interfaces (env, timercounter3, etc.).

Optical sensor

The protocol used for panning and reading the optical sensor is described next. The panning of the sensor is done by stepwise changing the sensor angle one degree at the time and acquiring a sensor value after each move. There is a time period associated with moving the sensor one degree in either direction. We denote this time period $T_{move}$. From measurements we get $T_{move} < 3 \text{ ms}$. 
Figure 5: The definition of the autonomous car CRC named autonomous_car: the CRC instances in yellow (indicating that the CRC has no inner CRC instances) and blue (indicating that the CRC has inner CRC instances), and the environment interfaces in orange. The arrows indicate message paths and the circles indicate access to environment interface registers.
Table 2: The representations for speed, steering, and sensor position used by the guidance system.

<table>
<thead>
<tr>
<th>Function</th>
<th>type</th>
<th>range</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>drivemode</td>
<td>int</td>
<td>0 - 1</td>
<td>(no unit)</td>
</tr>
<tr>
<td>sensor position</td>
<td>int</td>
<td>-80 - 80</td>
<td>(degrees)</td>
</tr>
<tr>
<td>steering</td>
<td>int</td>
<td>-25 - 25</td>
<td>(degrees)</td>
</tr>
<tr>
<td>speed</td>
<td>int</td>
<td>0 - 6</td>
<td>(no unit)</td>
</tr>
</tbody>
</table>

and adc) and instances thereof (env\_inst, tc3\_inst, and adc\_inst). They act as abstractions for the functions listed in Table 1.

The env\_inst provides access to the registers needed for configuring the PE6 port of the AVR to trigger interrupts when pressing the button on the car (the functionality of specific ports of the AVR is described in [9]). Further, its required interface ports (startup\_int and button\_int) are used for delivering startup and button interrupt events originating in the environment. The startup events are handled by the startup handler CRC (sh\_inst) who initializes all the CRCs in the model that have init methods and then starts the guidance system. The env controller CRC (env\_ctrl\_inst) is activated on the arrival of an interrupt from the button. The object takes care of filtering interrupts to eliminate the effects of “bouncing”, when a single press results in multiple interrupts delivered to the interrupt port and it forwards the button press event to the guidance system CRC.

The tc3\_inst provides access to the registers needed for configuring the PE3, PE4, and PE5 ports of the AVR as PWM outputs and setting the output signals. The tc3 controller CRC (tc3\_ctrl\_inst) enable the guidance system to switch between the two drive modes (neutral/drive) by disabling or enabling the PWM output signal using the Data Direction Register for port E5 (setreg\_DDRE), to control the panning of the optical sensor (setreg\_OCR3A), and to control the steering and speed of the car (setreg\_OCR3B and setreg\_OCR3C). The guidance system uses an internal representation for the drive-mode, sensor position, steering, and speed values which simplifies its implementation (see Table 2). The setreg methods of the tc3 controller CRC converts these values to their (hardware specific) PWM representations.

The adc\_inst provides access to the registers needed for configuring the PF0 port of the AVR as an analog input to the A/D converter and acquiring samples of the input signal. The initialization is managed by the adc controller CRC and its read\_ad method is used for acquiring and delivering IR signal samples from the optical sensor.

The guidance system CRC (gs\_inst) enables target tracking and guidance based on data from the optical sensor. It provides different services to its environment: start which is invoked automatically by the environment when the system has been initialized and toggle\_drivemode which enables the user to toggle the modes of operation for the autonomous car by pressing the push button on the car.

The definition of the guidance system CRC is depicted in Fig. 6. It encapsulates an instance of the splitter CRC (splitter\_inst), guidance CRC (gs\_inst) and targeting system
Figure 6: The definition of the guidance system CRC: the provided/required interfaces of the guidance system CRC in white, the CRC instances in yellow (indicating that the CRC has no inner CRC instances) and blue (indicating that the CRC has inner CRC instances). The arrows indicate message paths.

Figure 7: The definition of the targeting system CRC: the provided/required interfaces of the targeting system CRC in white and the CRC instances in yellow (indicating that the CRC has no inner CRC instances). The arrows indicate message paths.

CRC $(ts_{\text{inst}})$. The splitter CRC forwards the events to its output ports. The guidance CRC guides the car along a route in accordance with the routing strategy described earlier (see Section 5). It controls the operation of the car ($set\_steering$ and $set\_speed$) based on the location of the target relative the car ($read\_distance$ and $read\_heading$). Moreover, it switches the drive mode ($set\_drivemode$) between neutral and drive upon invocation of its $toggle\_drivemode$ method. In case the target is out of sight the guidance forces the car to drive in a small circle at low speed. The target data is provided by the targeting system.

Fig. 7 depicts the CRC definition of the targeting system. It encapsulates instances of the splitter CRC ($splitter2\_inst$ and $splitter3\_inst$), the panning controller CRC ($pc\_inst$), and the beacon detector CRC ($bd\_inst$). The panning controller is responsible for panning the optical sensor from side to side by stepwise changing the sensor angle one degree at the time. The angle change event is sent to the splitter which forwards the event to the optical sensor as well as to the beacon detector. The beacon detector reacts to such events by investigating the sensor signal aiming to register the angles of the two columns. It samples the sensor signal and checks each sample to see if it is above a predetermined
threshold which indicates that the sensor is directed towards a column or towards a point in between the columns. For every pan from one side to the other the beacon detector registers the first and the last angle for which the sensor value exceeds the threshold. From the registered angles it calculates the horizontal angle between the columns which in turn is used to determine the distance to the target and the direction of the target relative to the car (see Section 5). When the target data changes it will notify the receiver through the event `targetdata_updated`. Moreover, the beacon detector is notified when the sensor is at the edge of the panning cycle and it uses this information to identify that the target is out of sight (no detection). The beacon detector automatically turns the direction of the panning (using the `force_turn` method of the panning controller) if no column edge is detected within 30 degrees from the latest column edge.

Next, we present the code for the CRC methods that define the functionality of the software model.

### 7.2 CRC Code

This section presents the method code (C code) for each CRC in the system model for the autonomous car. The macros `ASYNC` and `SYNC` are used to encode messages. The first parameter in the async and sync macros is the name of a port in the required interface and the last parameter is the argument to the receiving method. The two remaining parameters in the async macro is the baseline offset and the deadline (integers denoting time); these parameters define the execution window for the receiving method. The timing specification for this model is described in detail in the next section (Section 7.3).

All variables that are not defined in the method in which they are used should be seen as state variables. State variables are defined in the state section of the CRC, the same goes for constants (such as lookup tables etc.). However, since the state section is not crucial for understanding the method code it is omitted in this presentation.

The method code of the startup handler CRC is presented below. Upon receiving the startup event it first initializes all the objects connected to its `init` ports in the order given by the method code. Then it starts the guidance system by sending a synchronous message via its required interface `start`.

```c
startup_handler CRC
{
    SYNC(init1);
    SYNC(init2);
    SYNC(init3);
    SYNC(init4);
    SYNC(start);
}
```

The method code of the env controller CRC is presented below. Its purpose is to eliminate the effect of bouncing (pressing the push button results in a burst of button interrupts due to the effect of bouncing in the mechanical construction of the button). Upon receiving a button event to the method `button_int` the env controller disables the
button interrupts (by clearing a bit in one of the registers provided by env) and sends a synchronous message via its required interface button_press. Moreover, it makes sure that the button events will be enabled at a later time by sending a message to its method enable with a delayed baseline. Note that the method enable is not part of the provided interface of the button controller and can therefore only be invoked by the button controller itself.

The tc3 controller has access to the registers associated with the timer/counter unit 3 of the AVR (see [9]) and it controls the three PWM outputs (the PWM outputs are configured by the init method). It converts the values given as arguments to its methods (see Table 2) and sets the PWM registers accordingly. The setreg_OCR3A/B/C methods sets the output to the sensor servo, steering servo, and the speed servo respectively and the setreg_DDRE sets the drive mode to neutral or drive. The method code is presented next.
The method `read_ad` of the adc controller CRC triggers a conversion and returns the value once the conversion is finished as presented next.

```c
init(){
    ADMUX |= (1<<REFS1);
    ADCSRA |= (1<<ADEN);
}
read_ad(){
    ADCSRA |= (1<<ADSC);
    while(ADCSRA & (1<<ADSC));
    return ADC;
}
```

The method code of the guidance CRC is presented below. It has two methods:
Speed output value as a function of distance to the target as provided by the array speed_arr[distance].

Distance to target in terms of angle between column edges (degrees).

Figure 8: The guidance CRC uses a lookup table (speed_arr) to determine the speed output value relative to the distance to the target. Note that no efforts have been taken to investigate the performance of the car using different functions.

toggle_drivemode that toggles the mode of operation (neutral or drive), and update_output that calculates the speed and steering output values that control the servos of the autonomous car. When in the neutral mode the PWM output is disabled.

The speed and steering computations involve trigonometric functions as well as multiplications and divisions. Performing these calculations in real-time on the microcontroller consumes an extensive amount of processing power. This can be avoided through the use of lookup tables. Lookup tables typically provide a mapping between function parameters and precalculated function values. The guidance CRC defines two lookup tables. The array speed_arr provides the speed output value as a function of the distance to the target (see Fig. 8). Note that the distance correspond to the horizontal angle between the IRED columns (\(\delta\)). The array steering_arr provides the steering output value (\(\alpha\)) as a function of the horizontal angle between the IRED columns (\(\delta\)) and heading of the target (\(\beta\)) as defined by Eq. 2 (the constant \(a\) in Eq. 2 that represent the distance between the front and rear axis of the car was set to 0.27 m based on manual measurements on the car).

```
init()
{
    drivemode = NEUTRAL;
}
toggle_drivemode()
{
    switch(drivemode){
        case NEUTRAL:
            drivemode = DRIVE;
```

```
SYNC(set_drivemode, DRIVE)
    break;
    case DRIVE:
    drivemode = NEUTRAL;
    SYNC(set_drivemode, NEUTRAL)
    break;
}
}
update_output(){
    int distance = SYNC(read_distance, NULL);
    int heading = SYNC(read_heading, NULL);
    if(distance > ARR_LENGTH){
        distance = ARR_LENGTH;
    }
    switch(distance){
        case DISTANCE_UNKNOWN:
            SYNC(set_speed, LOW_SPEED);
            break;
        default:
            SYNC(set_speed, speed_arr[distance-1]);
            break;
    }
    switch(heading){
        case ON_TARGET:
            SYNC(set_steering, CENTER);
            break;
        case HEADINGUNKNOWN:
            SYNC(set_steering, MAX_LEFT);
            break;
        default:
            if(heading > 0){
                SYNC(set_steering, steering_arr[(heading-1)<<<3][distance-1]);
            }
            else{
                SYNC(set_steering, -steering_arr[(-heading-1)<<<3][distance-1]);
            }
            break;
    }
}

The splitter has a single method (in). The reaction to an event is to forward the event to the ports of its required interface (out1 and out2). Note that the splitter always forwards the argument to both of its external receivers which makes it more general. Hence, its instances can be used in multiple locations in the model (splitter1_inst, splitter2_inst, and splitter3_inst in Fig. 6 and Fig. 7).

```c
in(int arg){
    SYNC(out1.arg);
    splitter CRC
```
The method code for the panning controller is presented below. Its purpose is to control the panning of the optical sensor. The method $\text{pan}$ increases or decreases the angle based on the panning direction ($\text{dir}$) and sends it to the optical sensor ($\text{SYNC}(\text{sensor\_angle}, \text{angle})$). The $\text{pan}$ method contains an asynchronous call to itself which makes this a continuous process and the frequency is defined by the baseline offset ($\text{BLO\_PAN}$). The timing specification is presented in detail in the next section (Section 7.3). The $\text{pan}$ method is not exposed in the provided interface of the panning controller CRC so the only way to start the panning process is to invoke the method $\text{start}$. The methods $\text{force\_turn}$ and $\text{curr\_angle}$ switches the panning direction and returns the current angle respectively.

```c
init(){
  dir = RIGHT;
  angle = LEFT_EDGE;
}

start(){
  ASYNC(pan,BLO_PANSTART,DL_PANSTART,NULL);
}

force_turn(){
  switch(dir){
    case LEFT:
      dir = RIGHT;
      break;
    case RIGHT:
      dir = LEFT;
      break;
  }
}

pan(){
  ASYNC(pan,BLO_PAN,DL_PAN,NULL);
  switch(dir){
    case LEFT:
      if(angle == LEFT_EDGE){
        dir = RIGHT;
        SYNC(edge_reached,NULL);
      }
      else{
        angle--;
      }
      break;
    case RIGHT:
      if(angle == RIGHT_EDGE){
        dir = LEFT;
        SYNC(edge_reached,NULL);
      }
      else{
        angle++;
      }
  }
}
```
The purpose of the beacon detector CRC is to evaluate the distance and heading to the target. Its method code is presented below. It has two states: find_first_edge and find_second_edge. Initially it will be in its find_first_edge state where it will search for the angle were the first column becomes visible. The find_column_edges method is invoked by the panning controller on each change in the sensor angle. It contains an asynchronous call to sample with a baseline offset which enables the sensor to reach its new position before acquiring a sample from the optical sensor. The sample method reads a value from the optical sensor and processes the value. Upon finding the first column edge (by comparing the signal from the optical sensor against a threshold) it sets the start angle to the current angle and sets the counter value to SCAN_ANGLE (the scan angle denote the scanning range for finding the second column edge, that is, the last angle where the second column is visible by the sensor). In the find_second_edge state it continuously sets the end angle to the current angle (provided that the signal is above the threshold). When it reaches the scan angle (when the counter becomes zero) it evaluates the distance and heading and returns to the initial state (find_first_edge). Moreover, it sends a notification to the guidance CRO that the target data has been updated and changes the panning direction. The pan_edge method is invoked when the panning reaches a turning point. It calculates the distance and heading based on its state and notifies the guidance CRO that the target data has been updated. The guidance CRC reads the distance and heading (read_distance and read_heading) to calculate the route to the target.
7.3 Timing specification

The modeling framework is based on the notions of reactive objects and time-constrained reactions which allows us to define a timing specification for the software system. This entails specifying the permissible execution window for all the asynchronous messages in
the code (the synchronous messages always inherits the caller’s time constraints). The timing specification is preserved throughout the design process and is utilized by the scheduler at run-time. However, without knowing the worst case execution time for all the methods in the system it is not possible to guarantee that the timing behavior that follow from executing the software will conform to the specification. Still, as long as the processor can execute the method code within the specified deadline for the method the timing specification will be met. Since none of the methods contain any demanding computations (recall the lookup tables described in the previous section) it should be safe to set the deadlines for every asynchronous method to 0, 1 ms. The complete timing specification for the autonomous car is presented in Table 3.

Button interrupts are delivered to the button-int method which immediately disables the button interrupts. We want the bouncing effect to end before we enable the interrupts again and therefore we set $BLO_{enable} = 1$ ms (from Section 6.2 we know that $T_{bounce} < 1$ ms).

In the start method of the panning controller we want the pan process to start immediately. This is achieved by setting the baseline offset to zero ($BLO_{panstart} = 0$).

The behavior of the car rely on the number of route updates that is generated as the car approaches the target (a faster panning of the sensor increases the chance of column detections which results in supplementary route updates). The panning of the optical sensor is managed by the method pan in the panning controller. It contains an asynchronous call to itself which makes this process continuous and the period time is defined by the baseline offset $BLO_{pan}$. We want to set the period time of the panning process as low as possible while allowing for the servo to reach its new position before the next position change. However, since the signal of the sensor is sampled in the beacon detector we must examine the timing characteristics of the whole reaction chain in order to determine the shortest possible period time of the panning process. The timing characteristics for the panning process are shown in Figure 9. It illustrates the permissible execution windows for the reactions alongside physical constraints of the target platform (Section 6.2) that must be met in order to achieve a proper system behavior. From the

<table>
<thead>
<tr>
<th>Name</th>
<th>Value (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLO_enable</td>
<td>1</td>
</tr>
<tr>
<td>DL_enable</td>
<td>0.1</td>
</tr>
<tr>
<td>BLO_panstart</td>
<td>0</td>
</tr>
<tr>
<td>DL_panstart</td>
<td>0.1</td>
</tr>
<tr>
<td>BLO_pan</td>
<td>3.2</td>
</tr>
<tr>
<td>DL_pan</td>
<td>0.1</td>
</tr>
<tr>
<td>BLO_sample</td>
<td>3.1</td>
</tr>
<tr>
<td>DL_sample</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 3: The timing specification for the autonomous car in terms of baseline offsets and relative deadlines for the asynchronous method calls.
Figure 9: The timing characteristics of the panning process in terms of baseline offsets and deadlines for the permissible execution windows of the asynchronous methods (pan and sample) alongside the physical constraint of the target platform ($T_{move}$). Setting $BLO_{sample} \geq DL_{pan} + T_{move}$ assures that the sensor has reached its new position before issuing an AD conversion ($read_{val}$). The period time of the panning process ($BLO_{pan}$) is defined by $BLO_{sample}$ and $DL_{sample}$.

Accordingly, the following equations hold:

\[
BLO_{pan} \geq BLO_{sample} + DL_{sample} \tag{3}
\]

\[
BLO_{sample} \geq DL_{pan} + T_{move} \tag{4}
\]

Accordingly, the period time for the panning process becomes $BLO_{pan} = 3.2 \text{ ms}$.

8 Program analysis

The system design contains all information required to perform timing analysis on the system, except the WCET’s on each distinct method. The tool automatically gives us interarrival-time and offset information for each task (but only for systems with a "simple" program flow [10]).

8.1 WCET estimation

In this case we have adopted a straight forward approach on finding the WCET’s for each function. For each method (m) the tool generates testcode that executes the method (m) and measures the execution time. All internal (in the method (m)) method calls (SYNC’s) and message passing (ASYNC’s) are replaced with call to a empty function. For data dependent operations the programmer must manually specify object state, method parameters and method call return values for the WCET measurement to be accurate. For this analysis to be correct the program code must remain unchanged and must be from the same compilation. So instead of recompiling the program we link the program differently when performing the WCET measurement. Effects of caches and pipelining are not an issue on the AVR platform.
8.2 Scheduler

The scheduler used is a non-preemptive DM scheduler, hence maximum blocking for a task is the maximum WCET of all other lower priority tasks in the system. Using a more advanced scheduler (such as a preemptive SRP scheduler) would reduce the blocking-time of sample and pan, but is not required in this case for the system to be schedulable.

8.3 The numbers

Job(j) is the first method called in an ASYNC call. Interference(i) is defined as the time a higher or same priority task might prevent (j) from starting execution. (Here we have to take into account the periods, higher priority task might have to be accounted more than once) Blocking(b) is the maximum time a lower priority task might prevent (j) from starting execution. (This is MAX (job i = all jobs, dl(i)>dl(j))) For a job to be schedulable the time i+b+e of a job j must be less than the deadline of j.

The first step in performing timing analysis is to check the total utilization (this is a required condition). In this case the utilization can be calculated as:

\[
\frac{(708 + 1139)}{52248} + \frac{(105 + 322 + 12)}{1600000} = 0.036 < 1, \text{ so this condition holds with a good margin.}
\]

Checking utilization is not sufficient for a system with shared resources or when deadline \(!=\text{ interarivaltime}\). A sufficient condition is to check that each job is schedulable, this can be achieved using standard scheduling theory. The table (4) shows that this condition holds \((m < 1 \text{ for all jobs})\).

8.4 Future work on analysis

Implementing other more advanced scheduling tests (e.g. offset, memory analysis) is feasible since the tool gives us all the necessary information. (WCET, time offset information between tasks) and by using a late version of GCC \((\geq 4.6)\) we will also get stack usage per function. Moreover, the tool also facilitates using SRP as scheduler, and can generate preemption levels and perform analysis [11]. Using SRP fits nicely with our model since locking is inherently hierarchical by using the SYNC primitive.

9 Conclusion

This paper shows how to design the software for an autonomous car using an integrated development environment (IDE). The IDE enforces a static component and communication structure, reflecting the concurrent behavior of the system. System models are constructed by composing components into hierarchical system models based on the system’s interaction with its environment. In particular, the system models are constructed graphically which supports understanding of complex models. Further, the IDE facilitates static analysis and generation of memory- and cpu-efficient executables.
<table>
<thead>
<tr>
<th>job(j)</th>
<th>deadline(d)</th>
<th>period(o)</th>
<th>interference(i)</th>
<th>blocking(b)</th>
<th>wcet(e)</th>
<th>i+b+e</th>
<th>margin(m)(i+b+e/d)</th>
<th>Schedulable (m &lt; 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample</td>
<td>2662</td>
<td>53248</td>
<td>0</td>
<td>1139</td>
<td>708</td>
<td>1847</td>
<td>0.6938392186</td>
<td>TRUE</td>
</tr>
<tr>
<td>pan</td>
<td>5325</td>
<td>53248</td>
<td>708</td>
<td>322</td>
<td>1139</td>
<td>2169</td>
<td>0.4073239437</td>
<td>TRUE</td>
</tr>
<tr>
<td>button</td>
<td>1600000</td>
<td>1600000</td>
<td>55744</td>
<td>0</td>
<td>105</td>
<td>55849</td>
<td>0.034905625</td>
<td>TRUE</td>
</tr>
<tr>
<td>buttonint</td>
<td>1600000</td>
<td>1600000</td>
<td>55527</td>
<td>0</td>
<td>322</td>
<td>55849</td>
<td>0.034905625</td>
<td>TRUE</td>
</tr>
<tr>
<td>enable</td>
<td>1600000</td>
<td>1600000</td>
<td>55837</td>
<td>0</td>
<td>12</td>
<td>55849</td>
<td>0.034905625</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Table 4: Timing analysis
The software for the autonomous car was constructed and integrated into the target platform. The result is a working prototype that operate according to its specification. That is, the car drives autonomously from the starting line to the beacon and stops in front of the beacon as intended.

This paper also show how the system’s timing properties can be analyzed in the IDE using standard scheduling theorems and machine assisted WCET measurements. Analyzing the timing properties for this use case confirm that they will always hold.

To conclude, this paper demonstrates the applicability of the software design methodology in developing real-time software for resource-constrained systems using the IDE.

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 AESOP project.

References


Towards executing TinyOS models under hard real-time constraints

Authors:
Per Lindgren, Johan Eriksson, Henrik Mäkitaavola, and Jimmie Wiklander

Reformatted version of paper submitted to:
Design, Automation and Test in Europe (DATE), 2012.
Towards executing TinyOS models under hard real-time constraints

Per Lindgren, Johan Eriksson, Henrik Mäkitäavola, and Jimmie Wiklander

Abstract

TinyOS (TOS) has (arguably) become the de-facto standard for developing soft real-time applications on lightweight platforms such as wireless sensor nodes. TOS deploys a simplistic execution model, ensuring race free operation through atomic sections and non-preemptive execution of tasks. While current TOS run-time systems offer resource efficient execution, no explicit support for real-time operation is offered which makes programming of real-time applications using TOS a delicate matter.

In this paper we investigate how TOS models can be executed under hard real-time constraints. To this end, we develop an alternative execution model for TOS programs that exploits additional concurrency while maintaining race free execution.

Moreover, we propose a mapping from TOS models to Concurrent Reactive Components (CRC), and show that the resulting CRC model is able to express the aforementioned execution model. Furthermore, we show how timing requirements can be added to allow TOS models to be executed under hard real-time constraints. Finally, we discuss implications of the CRC model to implementation efficiency and offline resource and performance analysis.

1 Introduction

TinyOS (TOS), with its heritage to the Berkeley Smart Dust Project and the programming of “motes” [1,2], has over the last decade become a de-facto standard operating system for lightweight wireless sensor nodes. Its popularity can be explained from the conveniently simplistic event driven programming model, and its resource efficient execution model with tool support for commercially available motes [3]. However, it has been noticed that achieving real-time performance using TinyOS is difficult, mainly due to concurrency limitations enforced from by the TOS programming model and current run-time system implementations, see e.g. [4]. In this paper, we develop an alternative execution model allowing additional concurrency from TOS programs to be exploited while preserving race free execution. Furthermore, we propose a mapping from the TOS programming model to the notions of Concurrent Reactive Components (CRC) [5], and discuss how timing requirements can be incorporated. Additionally, we discuss the implication to real-time performance, resource requirements and static analysis.

With the target of simplicity, minimizing run-time overhead and memory footprints, the TinyOS v2.x [6] specification support a two layer execution model: non preemptive

169
tasks (offering deferred execution), and events (which are executed immediately and is allowed to preempt the currently executing task or event). Both tasks and events are run-to-end. This simplistic execution model offers efficient single stack execution as preemptions is strictly hierarchical. However, race conditions may occur in cases when a command preempt another event or task that access the same (shared) state variable(s). In order to ensure race free execution the programmer has to manually protect access to shared state variables through defining critical sections (using the `atomic` keyword).

Experiences from TinyOS v1.x have shown that errors in system operation often stem from such race conditions, and that such faults (bugs) are typically hard to track down and eliminate. This problem has been acknowledged and is (to some extent) addressed in TinyOS v2.x through compile time code analysis in nesC compiler [7]. The compiler is able to detect most potential race conditions and issue warnings to aid the programmer in developing race free code.

However, real-time performance is limited by the simplistic execution model and its implementation by current TOS run-time systems. Firstly, the use of atomic sections to protect state variables from race conditions, introduces system wide blocking limiting system responsiveness. Secondly, while allowing for user defined task schedulers (e.g., EDF, TinyOS v2.x TEP 106), tasks are still non-preemptive (on a global basis) and run-to-end. Thus, in order to accomplish adequate real-time response the programmer may be forced to split long running tasks into multiple shorter tasks (this in order to allow for new scheduling decisions). Splitting operations puts an additional burden onto the programmer, and the outcome heavily depends on finding suitable partitionings. This work is further complicated whenever states are shared in between multiple tasks and/or events, as the nesC compiler cannot detect races spanning such split task operations (there is not even a concept for this type of linked tasks).

As an alternative approach to improve concurrency and real-time performance TOSThreads (TinyOS v2.x TEP 134) has been introduced. TOSThreads adds a third execution context (low priority task, besides events and ordinary tasks), which runs at the lowest priority and may preempt each other but not events or ordinary tasks. The implementation of TOSThreads requires the allocation of a separate stack for the execution, either statically (where the thread component is wired at design time), or dynamically (during system execution). The first case, causes the well known memory overhead of stack pre-allocation, while the latter case cause run-time overhead and possible fragmentation of stack memory (due to malloc/free like memory allocation). Programming using TOSThreads goes outside the simplistic programming model of TOS. The programmer has no longer any explicit support from nesC to obtain race free code (the programmer has to manually protect states though mutex/semaphore constructs).

In this paper we take an alternative approach, with the outset of extracting potential concurrency directly from the TinyOS v2.x TEP 1 specification. We propose an alternative execution model for TOS, that exploits concurrency from TOS programs while maintaining race free operation. The TinyOS model, after concurrency extraction, is then mapped to the notions of Concurrent Reactive Components (CRC) [5]. The CRC model offers automatic state protection (is race free), while being inherently concurrent.
The execution model of CRC is reactive, where state updates and output to the environment is specified as reactions to events, either originating from the environment or internally emitted. Each event (or message) has a destination object, giving a context of execution (state) and a method (implementing the execution). Messages operating on different objects, are concurrent (thus, the model is inherently concurrent), while only a single message may operate on an object at any time (thus, state access are race free by nature). Messages can be either synchronous, or asynchronous (opening up for concurrency). In the latter case timing constraints (in terms of base and deadlines) can be explicitly set, while synchronous messages execute under the timing constraints of the sender.

A CRC based framework for real-time embedded software has recently been presented [8]. It offers a graphic representation of CRC models (the model offers hierarchical composition of components). Component methods in the CRC model is specified in a C based language. The CRC model can be synthesized into plain C-code that can be compiled and linked with a minimalistic run-time kernel to create a bare-metal executable for a given target platform. Timing constraints from the model is preserved throughout the synthesis, and used by the kernel during run-time to offer real-time system operation. Recently, a compile time analysis has been presented, deriving information for Stack Resource Policy (SRP [9]) schedulability test (deriving message inter-arrival times), and preemption levels (for run-time scheduling). This allows for design time performance analysis and memory/CPU efficient execution of hard real-time systems onto resource constrained platforms (such as sensor nodes) [10].

The paper is outlined as follows: In Section 2, we investigate the TinyOS execution model and discuss limitations of the TinyOS execution model and current implementations. Furthermore, we propose an alternative execution model for TOS programs. In Section 3, we briefly review the concepts of CRC. In Section 4, we propose a mapping of TOS programs to CRC and show that the execution model is compliant to the proposed execution model. Moreover, we discuss how the resulting CRC model can be further annotated with timing constraints. In section 5, we conclude the paper with additional remarks on ongoing and future work.

2 TinyOS Concurrency and Execution Model

It is well known that concurrency may be utilized to improve system schedulability and responsiveness, however (equally well known) is that concurrent systems may be subject to race conditions (which in turn causes unexpected and/or malfunctional behavior). In the case of lightweight platform, it is also of importance that the execution model chosen can be efficiently implemented. These aspects were taken into consideration when designing the TinyOS execution model, the nesC programming language and component model, and the TinyOS run-time system installments.

TinyOS 2.x undertakes a simplistic execution model, (inherited from its predecessor, first version TinyOS 1.x), encompassing non preemptive run-to-completion tasks and preemptive interrupt handlers that are dispatched asynchronously by the underlying
hardware. This allows the scheduling of external events to be efficiently implemented directly by the hardware.

2.1 TinyOS Reactive Concurrency

Tasks are an explicit entity in the language; a program submits a task to the scheduler for execution with the post operator. The scheduler can execute tasks in any order, but must obey the run-to-completion rule.

Because tasks are not preemptive and run-to-completion, they are atomic with respect to each other. However, tasks are not atomic with respect to interrupt handlers or to commands and events indirectly invoked from interrupts. To facilitate the detection of race conditions, the TinyOS model distinguish synchronous and asynchronous code [3,11]:

- **Synchronous Code (SC):** code (functions, commands, events, tasks) that is only reachable from tasks.
- **Asynchronous Code (AC):** code that is reachable from at least one interrupt handler.

In general, any update to shared state that is reachable from AC is a potential data race. To reinstate atomicity in such cases, the programmer has two options: convert all of the conflicting code to tasks (SC only), or use atomic sections to update the shared state. Thus, nesC (version 1.3 [11]) should enforce:

**Race-Free Invariant 1:** If a variable \( x \) is written in AC, then all accesses to \( x \) must occur in atomic sections.

**Race-Free Invariant 2:** If a variable \( x \) is read in AC, then all writes to \( x \) must occur in atomic sections.

Furthermore, the nesC compiler should ensure that atomic sections are strictly hierarchical.

2.2 Implementation of TinyOS

In order to deal with race conditions (that frequently occurred in v1.x) TinyOS v2.x comes with a component model and compiler (nesC), aiding the programmer to develop race free code by enforcing scoping rules and issuing warnings on potential data race hazards. The current nesC [11] compiler distribution does a fair job at statically detecting races. However, in case of complicated patterns spanning multiple atomic sections, race detection might fail. Moreover, the current distribution may issue false positives when it comes to identifying potential races. For this presentation, we assume TinyOS models according to the race free invariants.

The standard TOS scheduler follows a FIFO policy to implement the TinyOS v2.x task semantics. Atomic sections are by current run-time system implementations enforced through globally disabling interrupts.
2.3 Extensions

The basic TinyOS execution model (non-preemptive FIFO scheduling of tasks) imposes a significant limitation to scheduling of real-time applications where tasks are subject to different timing constraints. By making the task scheduler as a separate entity (component), other scheduling policies have been introduced (e.g., EDF, TinyOS v2.x TEP 106). However, in order to comply to the execution model of TinyOS, tasks cannot be executed preemptively (since this would break the race free invariant).

Thus, independent of the scheduling policy applied the fundamental problem of non-preemptive execution remains. It is stressed in the case where a low priority task can block a higher priority task from execution. In order to limit the blocking times, the programmer has to take full responsibility of partitioning longer running tasks into several short running tasks. The race free invariant allows arbitrary access to shared state variables for SC (task) code, thus arbitrary splitting could cause race conditions, hence splitting must be done in such a way that race conditions do not occur.

An alternative solution proposed is to introduce the concept of traditional threads. TEP 134 The TOSThreads Thread Library defines a thread extension to the TinyOS execution model. TOSThreads adds a third (pre-emptive) tier of scheduling (to the preemptive events and non preemptive tasks of TinyOS), TOSThreads run in the idle periods (when no events or tasks are being executed). As such TOSThreads does not improve the real-time performance of already made TinyOS code, and requires code to be rewritten (primarily by moving functionality of non-preemptive tasks to threads). Since TOSThreads adds a traditional thread model, the programmer has to manually undertake the tedious and error prone efforts of concurrent programming (using mutexes etc.) and the simplicity and elegance of the original TinyOS model is lost.

2.4 Atomic Sections

The TinyOS 2.x programming model states:

\textit{atomic} guarantee that the statement is executed “as-if” no other computation occurred simultaneously, and furthermore any values stored inside an atomic statement are visible inside all subsequent atomic statements

This definition offers extensive concurrency in theory (as atomic sections with non overlapping state variable accesses can run concurrently). However, the current nesC [11] implementation does not discriminate in between atomic sections and will generate C-code using a single macro to indicate the presence of atomic sections. As the nesC compiler does not supply any information on what specific section/state variables to protect, current run-time system implementations are prohibited to implement fine grain atomicity, and must therefore deploy atomic sections at the global level (typically by means of disabling all interrupts).

An exact analysis of overlapping state variable access in atomic sections would render the maximum potential concurrency. However, this would be fairly complex to implement. In this paper, we propose an approximation based on the TOS module/component
scoping rules. As state variables are only accessible within the module where defined, we can assign a monitor $m_a$ to each module, and let atomic sections operate under the module’s monitor $m_a$. In this way atomic sections defined in separate modules (operating on clearly disjunct states) to be executed concurrently, while prohibiting concurrency between atomic sections operating on potentially overlapping states.

2.5 Task Model

The TinyOS 2.x programming model stipulates tasks to run non-preemptively, i.e., a task may never preempt another ongoing computation. As mentioned this has the advantage of race free execution in between tasks and limits stack depth during run-time. The drawback is that concurrency is severely limited.

On closer inspection, race free execution can be achieved under the condition that tasks with overlapping state access are executed non-preemptively. As discussed above, while requiring complex implementation a full analysis of task state accesses would render the maximum potential concurrency. Alternatively, we propose an approximation where we assign a monitor $m_a$ to each module, and let the tasks defined in the module operate under the modules monitor $m_a$. Similarly to the discussion on atomic sections, this allows concurrency in between tasks to operate concurrently iff operating on clearly disjunct states. Notice however, in order to guarantee race free execution:

\begin{itemize}
  \item **Synchronous Code (SC):** code (functions, commands, events, tasks) that is only reachable from \textbf{internal} tasks.
  \item **Asynchronous Code (AC):** code that is reachable from at least one interrupt handler \textbf{or other module}.
\end{itemize}

The increased concurrency has the potential to improve system responsiveness to external events (limiting the negative impact of atomic sections) and to improve real-time execution of tasks. To this end, we need additional information to prioritize execution. Typically external events are captured through the interrupt hardware (usually with associated priorities). Such interrupt priorities are not part of the TOS programming model per se, but is used in practice to control precedence of interrupt handling. With respect to tasks, the basic/default FIFO scheduler does not incorporate priorities. However as discussed earlier, the TOS programming model allows customized schedulers to be deployed, e.g., EDF and monotonic schedulers, which put to use in TOS programs can give us information needed for prioritizing execution.

In Section 4 we will discuss how the above proposed execution model can be expressed in terms of CRC, and we will discuss how additional information for prioritizing execution can be exploited to improve real-time performance of TOS programs.

3 CRC Concurrency and Execution Model

Concurrent Reactive Components (CRC) builds on the concept of time bound reactions to external and internal events (messages). Each message $M = (bl, dl, o, m, arg)$ defines
3. CRC Concurrency and Execution Model

A permissible execution window \((bl, dl)\), destination object \(o\), method to be executed \(m\), and optionally parameters \(arg\). The execution window for \(M\) is defined as absolute points in time, where \(bl\) is the baseline and \(dl\) is the deadline. Messages can be either synchronous \((SYNC(M))\) where the caller is blocked until message execution returns), or asynchronous \((ASYNC(M))\), where the caller continues execution. In the first case, the timing information \((bl, dl)\) is always inherited from the caller, while in the latter case it may be either inherited or explicitly set (allowing the programmer to express timing requirements in a straightforward manner). Events that occur in the system environment is conceptually treated as asynchronous messages with their baseline set to the time of arrival, and their deadline set according to the requirements on response time for the event.

Asynchronous messages introduces concurrency in the model, and hence race conditions may occur. To this end, the CRC model enforces complete state encapsulation, (i.e., state is only accessible to the methods of the object), and each object can execute at most one message at any instance. In effect, object state is protected from race conditions by treating the object as a monitor, where the execution of a message \(M = (bl, dl, O, m, arg)\) amounts to \(lock(O) \rightarrow execute(m(arg)) \rightarrow unlock(O)\). For correct operation the message should be executed within the permissible execution window \((bl, dl)\).

3.1 Implementation of CRC

For lightweight systems, RAM memory is often a limiting factor. Hence, for the execution of CRC based systems it is instrumental that memory overhead can be kept to a minimum. Additionally, it is important that properties (e.g., schedulability and memory requirements) can be predicted at design time.

To this end, a tool-suit for CRC based embedded real-time software has been devised [8]. The tool suit encompasses an IDE for component based design, tools for static analysis (extracting information from the model to derive static object instance structure and to obtain communication pattern information.)

For the execution on ultra-light weight platforms (e.g., the 8-bit AVR-5 series MCUs), the tool derives pre-emption levels and (interrupt) priorities for Stack Resource Policy (SRP [9]) Deadline Monotonic (DM [12]) scheduling. SRP ensures deadlock free execution, offers a single stack to be shared in between preemptive tasks and gives a limit to the number of context switches (at most 2 per task in the system). SRP together with DM based scheduling, can be efficiently implemented onto by commonplace interrupt hardware (as long as the number of hardware interrupt levels and interrupt sources is sufficient, the scheduling will be done directly by the interrupt hardware) [10]. The derived information can be exploited for offline resource analysis (e.g., memory, schedulability etc.), and is used by the tool to generate C-code, that can (in a similar fashion to TinyOS) be compiled for a selected target platform. The generated code will also incorporate the run-time system primitives (e.g., for locking/unlocking objects, postponing events for future execution, etc) thus allow for aggressive “single program” code
optimization similar to that of nesC.

4 TinyOS to CRC mapping

We propose an alternative execution model for TinyOS v2.x, that improves concurrency while maintaining race free operation of TOS programs fulfilling the race free invariants as discussed in Sections 2.4 and 2.5. In this Section, we outline a mapping from TOS programings onto the notions of CRC.

4.1 Interrupts and Entry Points

Both TinyOS and CRC models are reactive, driven by events occurring in the environment. In the case of TinyOS, an (asynchronous) event handler is associated with a hardware interrupt. Additionally, for the implementation attribute can be set to indicate whether this interrupt should be treated as an atomic section (@atomic_hwevent) disabling interrupts globally, or allowing preemption (@hwevent).

In the CRC model undertaken in the REKO IDE, a special environment component construct is used to define interaction with the system environment. Each component in the model has a provided and required interface. In the case of environment components (used to interface the underlying hardware), the required interface ports corresponds to the entry points for the interrupt handlers. Such entry points can be connected to the provided ports of target CRC.

If entry points are to be treated atomically w.r.t to each other, this can be encoded directly in the CRC model by connecting them to ports that are connected to methods of a shared target object. In this way the CRC model provides means to express the exact amount of desired concurrency in between interrupt handlers.

4.2 Configurations and Modules vs. Components

TinyOS components are specified in terms of configurations, that at the lowest level is implemented by modules. In the configuration interfaces are wired to sub component instances. Each module defines a name space, all variables defined in the body are accessible (in scope) to tasks, commands and events defined in the module. There is no sharing of variables in between modules, unless explicitly passed through a call/signal. Commands that may execute preemptively is marked with the async keyword, and checked by the nesC compiler for consistency w.r.t., to interface type and wiring (as specified by the configuration/instantiation). However, no escape analysis is performed, thus when passing pointers (references) to data, the responsibility of avoiding race conditions is left with the programmer. The assumed use is that the receiver is delegated access to the data.

In the CRC model a component is defined in terms of a CRC (which defines the required and provided interface ports of the component). Parts of the component is connected either to sub components (CRCs), or to methods of CRCs. Each component
defines a name space, and methods and state is accessible only within the component. A CRC can be seen as a means to allow for hierarchical decomposition, similarly to TinyOS configurations.

4.3 Atomic sections and tasks

In our proposed execution model, each module is associated with a monitor for atomic sections ($m_a$ in Section 2.4) and a monitor for tasks ($m_t$ in Section 2.4). Hence, for each TOS module we create a CRC $c$ holding two inner CRCs ($o_a$ and $o_t$), where state variables occurring in atomic sections are defined in $o_a$ and remaining state variables are defined in $o_t$. $c$ is used to give a unified interface to $o_a$ and $o_t$ in accordance to the TOS module interface.

Tasks of the TOS model translates to methods of $o_t$ (post $t$ corresponds to an asynchronous message $\text{ASYNC}(..., o_t, t, ...)$ executing $t$ under the object/monitor $o_t$). Commands and events are functions (possibly) bound to the interface of $c$ (according to the TOS module interface) and will be executed directly by the caller/signaller. Each atomic section corresponds to a method $m$ in $o_a$. Entering $m$ (from a command/event or task) corresponds to a synchronous message $\text{SYNC}(..., ..., o_a, m, ...)$ executing $m$ under the monitor $o_a$. In the case where entering from an internal task, state variables in $o_t$ must be accessible to $m$. This can (conceptually) be achieved through passing $o_t$ as an argument and update $o_t$ as a result of executing $m$, i.e., $o_t = \text{SYNC}(..., ..., o_a, m, o_t, ...)$. Passing and updating large state constructs would impose performance penalty, however, the hierarchical locking allows direct access to $o_t$ inside the monitor $o_a$ (even if $o_t$ is outside the syntactic scope of $o_a$). Thus:

```c
taskA() {
    code # executes under monitor o_t
    o_t_s = \text{SYNC}(..., ..., o_a, m, o_t_s, ...)
    # m executes under monitors o_t and o_a
}
```

translates into

```c
taskA {
    code # executes under monitor o_t
    LOCK(o_a)
    m # m executes under monitors o_t and o_a
    UNLOCK(o_a)
}
```

Notice, this rewritten model should not be exposed to the programmer, as explicit $\text{LOCK}$ access is not allowed in the CRC model (but rather an implementation technique applied during code synthesis).
4.4 Evaluation

As previously mentioned, in order to fully exploit the proposed execution model, information to prioritize execution (make scheduling decisions) can be added to the CRC model. As discussed, the CRC programming model undertakes the notion of deadlines which during run-time can be utilized for scheduling decisions.

4.5 Timing information

Firstly, when it comes to external events (interrupts), the TOS programmer already today makes an active decision into programming the interrupt hardware by selecting interrupt priorities. This information can be directly applied in the REKO tool, into setting the priority (in terms of relative deadline) for the corresponding entry point. (Notice, in case of implementing hard real time systems, interrupt priorities are initially derived from timing constraints.)

Secondly, when it comes to task priorities, the post constructs is after the mapping to CRC translated into \texttt{ASYNC}(bl, dl, ...). The default values of bl and dl is derived from the poster. The bl is directly inherited from of the poster’s bl. dl is set to infinity for tasks posted by tasks (lowest priority), while dl is inherited for post operations initiated by external events. Thus, the default behavior reflects the urgency of the sender onto the receiving task. However, that is not necessary the correct assumption on desired timing behavior and the programmer is free to adjust the deadlines accordingly. (Notice 1. In cases when the TOS program used alternative schedulers, e.g., EDF, timing information might be directly derived. Notice 2. By setting the baseline offset for a post, the deferred operation is delayed, which makes the use of explicit TOS Timers superfluous.)

4.6 Implementation

The concept of deadlines in the programming model for CRC opens up for EDF based scheduling. However, for lightweight platforms, hardware assisted scheduling using static interrupt priorities offers an attractive alternative with its efficient and low complexity implementation. To this end, static priorities can be directly derived from deadlines in the CRC model and utilized by the run-time system in cooperation with the interrupt hardware to perform scheduling under SRP-DM [10].

The CRC model can be seen as a specification for operation under hard real-time constraints. Given worst case execution times and additional information on minimum interarrival times for external events, and important class of systems can be analyzed for schedulability [10]. However, it is worth to mention, that also systems with soft real-time requirements would benefit from the proposed execution model. In such case, we may choose to prioritize execution from e.g., relative importance and tailor the scheduler accordingly. In any case, the proposed execution model offers advantages of reduced blocking times (the atomic sections are enforced per module not globally) and increased flexibility for scheduling decisions (preemption is allowed in between tasks of different modules).
5. Conclusions

However, increasing the concurrency of the system also lead to increasing the depth of preemptions. In the TOS model, the maximum depth of preemptions are the number of interrupt handlers plus one task (under the assumption that interrupts are non-reentrant). The number of potential preemptions in the proposed execution model is only limited by the number of modules in the system (since we allow each module to execute one task). For resource constrained platforms, stack memory may be a limiting factor. Under SRP-DM, we can elegantly trade-off concurrency to stack depth, i.e., enforcing two tasks to share a preemption level makes them execute under mutual exclusion and cannot occupy stack space at the same time. Thus, merging priority levels reduces stack memory requirement. Exploiting this tradeoff is a set topic for future research.

4.7 Limitations

In this paper we have focused on the programming and execution models of TOS, leaving out details on additional constructs (like parametrized interfaces, output repeaters etc.) and syntactic issues all needed to be addressed in order to detail a complete mapping.

Furthermore we make the assumption that atomic sections are used in TOS programs according to the language specification, not the language/run-time system implementation. That is, if the program exploits, requires or relies on the fact that atomic sections are implemented through globally disabling interrupts, the TOS program will after translation have a different and possibly unintended and possibly erroneous behavior.

With respect to tasks, the proposed execution models opens up for concurrency. In cases where tasks rely on non-preemptive execution (for example when accessing a shared external resource) the program will express a different behavior. This can be solved by letting tasks that share access to an external resource belong to the same module (i.e., execute under the same monitor). In some cases this will require refactoring of the original TOS program, but on the other hand reveal dependencies on external resources.

Notice also that that the new condition for AC/SC code is not enforced by the current nesC compiler.

5 Conclusions

TinyOS (TOS) has arguably become the de-facto standard for programming lightweight wireless sensor nodes. However, its simplistic execution model and run-time system implementations poses restrictions to exploiting concurrency of TOS programs, which in turn prohibit applicability to systems operating under real-time constraints.

In this paper we have investigated the TOS programming and execution models and we propose an alternative execution model to efficiently exploit concurrency while preserving race free execution. We have introduced a mapping from TinyOS models to Concurrent Reactive Components (CRC), revealing concurrency by means of asynchronous messages and enforcing atomicity through the object structure.

We have discussed how execution of CRC models can be efficiently implemented onto lightweight platforms through resource management under the stack resource policy.
using deadline monotonic scheduling. This allows resource efficient and deadlock free scheduling exploiting commonplace interrupt hardware. Additionally, we show how timing constraints can be added to allow systems to operate under hard real time constraints. Finally we have discussed implications of the execution model and CRC mapping to implementation efficiency and offline resource/performance analyses.

Current research involves further investigating the trade off in between schedulability and resource utilization. A topic of interest for future research is to develop an automatic translation mechanism from TOS models to CRC in order to evaluate achieved real-time performance gains.

References


