Zur Homepage der Dissertation

Carl von Ossietzky Universität Oldenburg
Fakultät 2, Department für Informatik

Dissertation

OOCOSIM - An Object-Oriented Co-design Method for Embedded HW/SW Systems

Frank Oppenheimer

zur Erlangung des Grades eines Doktors
der Naturwissenschaften

Gutachter/Supervisor: Prof. Dr.-Ing. Wolfgang Nebel
Zweitgutachter/Reviewer: Prof. Dr. Franz Rammig
Tag der/Date of Disputation: 9. Februar 2005

Product names described herein are trademarks of the respective companies

© 2003,2004,2005 by Frank Oppenheimer

To Rie and Mona

Acknowledgements

First of all I would like to thank my supervisor Prof. Dr. Wolfgang Nebel for creating an inspiring scientific environment. He always found the time for discussions when I asked for but also gave me the freedom to develop my own ideas. I would also like to thank Prof. Dr. Franz Rammig for taking the time to review this document.

Let me express my gratitude to my colleagues at OFFIS, and the University of Oldenburg. My special thanks goes to Dr. Guido Schumacher, Dongming Zhang, Thorsten Schubert, Andreas Schallenberg, and Michael Kersten for many discussions and their valuable comments. And all to the other people who shared their opinions and advice: thank you.

On a personal note I would like to say thank you to my family for their great support and for thrusting me during all these years. A special thank goes to Rie and Mona for giving me much strength by their love. I could not have done this without you.

Finally, my gratitude goes to the Carl v. Ossietzky University of Oldenburg and the DFG for giving me the opportunity to participate in the OOCOSIM project in which the foundations of the work at hand were conceived, and the Kuratorium OFFIS e.V. for allowing me to finish this thesis.

Contents

1 Introduction

With the constant technological progress of the miniaturisation of electronic components, embedded systems became a very important part of our daily live. Already today, embedded systems outnumber desktop and server computer systems by far. They are part of our car, where tens of microprocessors observe and control almost every subsystem of the vehicle. Embedded systems can be found in medical devices or in washing machines or in space probes. But not only their number, also their complexity and in particular their software complexity is increasing. While 4-bit and 8-bit microcontrollers were dominant in the past, recently 32-bit and even 64-bit processors executing the software at hundreds of megahertz clock speed are becoming a typical case.

The failure of such a system may have serious consequences. If for example an electronic brake assistant accelerates the car instead of stopping it or an airbag controller activates the airbag at the wrong time the consequences can be deadly.

While the design methodology for application software improved significantly after the software crisis, embedded systems are still developed based on individual design experience instead of a well defined methodology. This will become a serious problem in an industry facing a lack of qualified and experienced designers and the pressure to produce increasingly complex systems in ever shorter time.

The core contribution of this thesis is an object-oriented design flow for embedded hardware/software systems named OOCOSIM. It provides a notion to describe and stepwise refine a system into a set of communicating hardware and software objects. This high-level description is useful as it provides a design layer that is abstract enough to achieve a complete overview of a complex system but formal enough to reason about certain properties and identify design mistakes. The objects at this level define the entities that can be followed seamlessly through the entire design flow.

A particular challenge for the design of embedded systems is the communication across the hardware/software boundaries. To handle this problem, the work at hand introduces dedicated communication objects and integrates them into the graphical HRT-HOOD [BW95] notation. The unique property of this extended HRT-HOOD notation called HRT-HOOD+ is that is allows the object-oriented specification of embedded systems with specific emphasis on the hardware/software interface.

Since the communication objects are neither pure software nor hardware a new description language, named ComiX is introduced in the work at hand. The aim is, to describe and handle the hardware/software interface in a consistent way. The ComiX specification can be processed by an automatic code generator to achieve hardware and software source-code models or the documentation. The formal nature of ComiX allows defining a set of formal consistency rules for interface specifications. This consistency of ComiX specifications can be checked automatically, which makes interfaces more efficient and reliable.

Interface code generation highly depends on the target system. This thesis presents a template based method which enables the definition of arbitrary interface code generators. Thanks to the separation of interface specification, template specification, and the code generator tool, this method allows to efficiently adopt new target platforms or domain specific coding standards for interfaces.

Testing and debugging of embedded systems today takes a large and even increasing portion of the development effort. During testing two different kinds of errors in an embedded control system design may occur: functional errors and timing requirement violations. The first one is obvious. A violation of the specified functional behaviour is not acceptable. Many embedded systems are also subject to certain real-time constraints. If such a control system reacts functionally correct but not within the required deadlines it is regarded as erroneous.

The time-synchronous co-simulation as a core part of the OOCOSIM design flow allows checking both, the functional and timing behaviour of the specified system. Therefore, the objects are translated into an executable model for interface, hardware, and software which can be tested and evaluated in a so-called time-synchronous co-simulation. The co-simulation system developed in the course of this work enables interactive debugging of the system before the first physical prototype is built. It was necessary to develop the theoretical basis to combine the functional and timing behaviour of the hardware and the software model. The adoption of the discrete event model to a source-code software model allows for a time-synchronous co-simulation of a target independent system model.

OOCOSIM supports a design flow from an abstract object-oriented specification to the implementation. The refinement steps from the higher levels of abstraction to the lower ones are well defined and mostly automatic. The co-simulation of hardware and software in the design allows early errors detection and an extensive exploration of design alternatives. OOCOSIM can therefore reduce the design effort for complex embedded systems.

The rest of the thesis is organised as follows. The next chapter introduces some basic terms and notations, important for the understanding of the further work. Chapter 3 presents existing approaches and classifies them. The approaches are also compared with the methodology proposed in this work. Chapter 4 describes the characteristics of embedded system design and derives requirements for design methodology in general. The following chapters are dedicated to the different aspects of the OOCOSIM flow. Chapter 5 gives an overview of the OOCOSIM design flow and then explains how OOCOSIM fulfils the requirements (from a methodological point of view) identified in Chapter 4. The interface design problem and the newly defined languages ComiX and TempliX are the focus of Chapter 6. Chapter 7 explains the mechanisms to achieve a time-synchronous co-simulation of hardware and software in detail. Chapter 8 evaluates the presented approach (using embedded system benchmarks) with respect to the requirements named in Chapter 4. The last chapter concludes the work and discusses the limitation of the presented approach as well as possible future research directions.

2 Basic Terms and Notations

This chapter will introduce some notations and basic terms, which are essential for the understanding of the work at hand. Some terms may seem well known but they are used with varying meanings in different publications. Thus, the major intention is to disambiguate these terms in order to achieve a common understanding throughout this work. To avoid repetitions, terms like co-simulation and co-design are only briefly introduced as they are discussed in detail later in this thesis.

Furthermore, this chapter will introduce some methods and notations like HRT-HOOD or XML that are used or extended in this work. Those readers who already know them can easily skip these sections. Due to the spatial limitations of this work, it can impart only superficial knowledge about these concepts. For a deeper understanding, references to recommended literature are given where appropriate.

2.1 Embedded systems

Since most of this work is about design methods for embedded systems, this term is of central importance here. It is however non-trivial to find a generally agreed and unambiguous definition of embedded systems in general. Thus, the aim here is to achieve a common understanding of what – in the context of this work – an embedded system is and what is not.

While an embedded system is typically part of a bigger, physical system the term here means only the electronic components of the device. Embedded systems can be found almost everywhere in our daily live. From very small and rather simple electronic devices controlling consumer products like washing machines or toasters to large and complex embedded systems steering an Ariane 5 rocket or a deep-space probe. Although the applications are fundamentally different, the involved embedded systems share common aspects, which will be briefly described here[1].

Some typical embedded systems with some numbers about their components can be found in Table 2.1.

Domain (Example)	**Consumer (PDA)**	**Automotive (ESP)**	**Telecom (Switch)**
Processor	MC 68k	ARM 7	Multi-Pentium
Memory	16 KB	$>$ 1 MB	$>$ 2 MB
Hardware	$<$ 10 KGates	50 KGates	Multi-ASIC
RT-System	Simple Scheduler	Microkernel	Full RTOS
Software	100 KByte (C or Assembler)	50 KByte (C or Assembler)	2 MByte (C or C++)

Table 2.1: Examples for Embedded Systems

All embedded systems relevant for this work contain some software and some application specific hardware components. Embedded systems use dedicated hardware to fulfil certain (e.g. power or real-time) constrains. Furthermore, they need hardware blocks to interface with the environment,

[1] In Section 4.4 a more detailed discussion about the special characteristics will follow.

i.e. to drive the actuators according to a particular protocol or to read sensor values from the environment.

The CPU, memory and other off-the-shelf hardware components are necessary to execute the control software in embedded systems. However, since they are usually not designed for a particular embedded system, they are not regarded as hardware components in the design flow. Instead, these components are referred to as *software execution environment*.

Software in the embedded domain can be found in large varieties. In the simplest case, it is only a short, sequential program. In the other extreme, it may be multi-million lines of code program, containing several concurrent tasks observing multiple sensors and reacting on asynchronous events from the environment. In other cases, it can even be a combination of several almost independent processes running on a single processor or a distributed system running on a multi-core embedded system.

Real-Time requirements are imposed on many embedded systems. Depending on the criticality of a deadline-violation, real-time systems are either classified as hard or as soft real-time systems. In a *soft real-time system* rarely missing a deadline can be accepted whereas for *hard real-time systems* every deadline must be reached to preserve its correctness. For the management of real-time requirements in the software of such system a RTOS (Real-Time operating system) is typically part of its embedded system's architecture.

Depending on the embedded system's task, application specific hardware components build a smaller or bigger fraction of the system. Hardware is massively parallel by nature. While this property is enabling the superior performance of hardware, it is also a challenge for the designer and the co-design process. The custom digital hardware is implemented either in a FPGA (Field Programmable Gate Array) or in an ASIC (Application Specific Integrated Circuit) or combinations of both.

The decision process, which parts of an embedded system should be implemented in hardware and which in software is a called *system-partitioning* or *hardware/software partitioning*. The partitioning decisions are based on the estimated performance and cost on the one hand and the need for flexibility on the other hand. Generally, performance critical components are implemented in hardware while highly configurable or flexible part should be implemented in software. However, with upcoming architectures, i.e. reconfigurable hardware and dynamic computing, the boundary between hardware and software is blurring, hardware is getting softer [Vah03, SON04]. This brings new alternatives but make the design space exploration even more difficult. Still, hardware design has to cope with several limitations[2] in the design process. Thus, certainly the effort to implement functionality in hardware by far exceeds the software design effort.

No matter how the system's functionality is split among hardware and software; for the cooperation of the partitions communication is required. Through a hardware/software interface data, commands, and events must be transferred across the hardware/software boundary. The interface provides information about the environment measured through sensors or propagates commands by which it controls the actuators. If a computation is split into hardware and software components, it needs to transfer the intermediate results for further computation. As the interface is implemented partially in hardware and software, it is regarded here as an own category.

All embedded systems share the important property of being designed for a specific purpose. Very often they are embedded in a machine, a complex device (e.g. a car, a washing machine), or a production cell in a factory. The environment may be another digital system but in most cases will also contain non-electronic components. In contrast to personal computer systems, embedded systems quite often work autonomously. Some systems are designed to work in harsh environments or in safely critical applications. Thus, the modelling of the environment can not be omitted in the design process. Figure 2.1 sketches a general embedded system with its subsystems hardware, software, and interface as described above. The characteristics and the large variety of embedded systems make it difficult to define a single all-purpose design method. Therefore, the approach presented in this work focuses on the rather complex embedded systems with a significant and

[2]Software languages typically have a greater expressiveness and the turn-around times are significantly shorter.

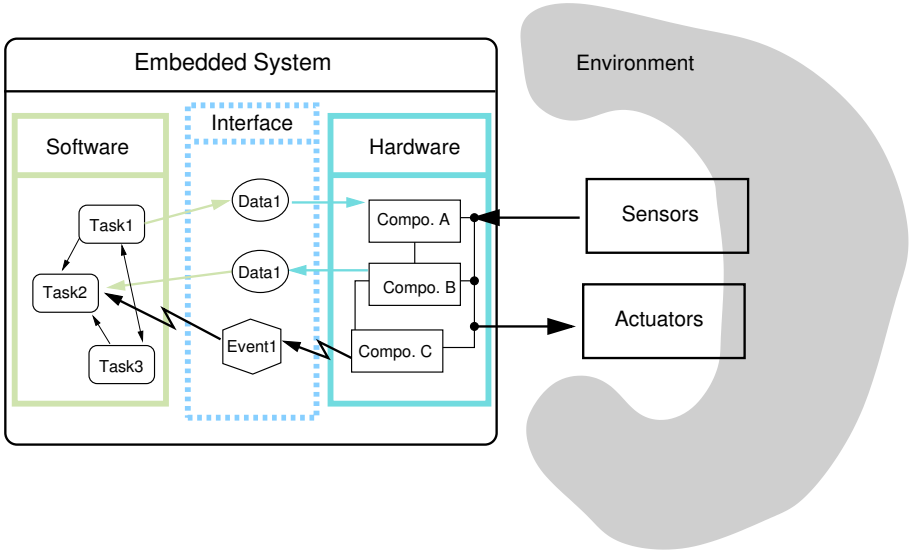

Figure 2.1: Embedded System

complex software component.

Application specific hardware will not be dominant but remains indispensable for performance and real-time critical parts of a system. Tasks, which face tough requirements on performance, power consumption, or reactivity, still must be implemented in application specific hardware.

The hardware/software interface, while not directly implementing any behaviour, is regarded as an important and critical component in the embedded system design process. The effective and correct design and implementation of the interface is a challenging task. This becomes evident in case of modifications or maintenance, where changes of hardware or software components very often entail changes in the interface.

2.2 Model of computation

A key aspect in co-simulation and co-design is the integration of different so-called *models of computation*. According to the *National Institute of Standards and Technology*, a model of computation is:

> *A formal, abstract definition of a computer. Using a model one can more easily analyse the intrinsic execution time or memory space of an algorithm while ignoring many implementation issues. There are many models of computation which differ in computing power (that is, some models can perform computations impossible for other models) and the cost of various operations. (from :http://www.nist.gov/dads/HTML/modelOfComputation.html)*

In the research area of this thesis, the term is often applied to characterise simulation models by the way they describe the behaviour of system or components under design. Prominent examples of computation models are discrete event model, synchronous model, and continuous time model. Since the computational model describes two key aspects of the simulation model, namely the temporal and functional abstraction mechanism, it is often be used synonym.

2.3 Co-design and co-simulation

The OOCOSIM design flow as presented in this work is a *hw/sw co-design* method based on *hw/sw co-simulation*. Therefore, almost every chapter is related to aspects of one or both of these terms.

Chapter 4 and Chapter 7 are devoted exclusively to discuss these thematic areas in relation to the approach presented. To clarify the terms for the following chapters they will be briefly introduced already here.

As mentioned in the previous section, embedded systems contain components from different domains, namely hardware, software, and interfaces. Due to their fundamental differences, the co-design paradigm can be clearly distinguished from conventional approaches.

Conventional design methods handle each domain separately while co-design methods are characterised by an integrated process addressing all relevant domains concurrently. In consequence, conventional design is closer to the sequential waterfall model applied in software design. The inherent risks and benefits of both paradigms are discussed later (Section 4.2, Section 4.3) in detail.

Co-simulation models are characterised by their ability to simulate heterogeneous system specifications. Such a specification may contain components from different domains modelled in different formalisms executed using different models of computation. For the integrated development process a co-simulation is often a necessary component[3] of a co-design method, where usually different specification languages are used in different domains.

The term co-simulation is also often used to describe the concerted simulation of different specification languages (e.g. VHDL, SystemC, and Verilog) describing components from the same domain (here hardware). While these co-simulation environments are particularly useful to integrate IP or legacy components into a single design, they provide no means to design an integrated hw/sw system.

To enable early detection of design flaws co-simulation environments usually provide techniques like debuggers and tracing functionalities to validate the embedded system with all its components interacting with a simulated environment.

With the above in mind, it becomes obvious that a co-simulation can only show the specified behaviour under a unified simulation model. Please note, that this does not mean to use a single simulation model for all sub-model. The unified simulation model may contain different sub-model unified by means of wrappers allowing the synchronised simulation of the system as a whole.

If the specification is the basis for synthesis of the implementation and the simulation model reflects all relevant properties, co-simulation provides a *virtual prototype* of the embedded system under development[4]. In summary, a co-simulation can be seen as an executable, heterogeneous specification of a system.

2.4 HRT-HOOD

The HRT-HOOD (**H**ard **R**eal-**T**ime **H**ierarchical **O**bject **O**riented **D**esign) [BW95] method is a specialisation of HOOD [Gro95] and was originally developed for the European Space Agency (ESA). It is a methodology for a structured design capture, based on a formal graphical and textual notation. The methodology provides strict guidelines for the refinement of objects, which finally guarantee certain properties of the design. In particular the runtime behaviour can easily be analysed by static methods like rate-monotonic analysis [KRP+93].

In contrast to other object-oriented notations which – like UML – use a class-based approach, HRT-HOOD follows the structural paradigm; that is, a system is represented by a set of communicating objects[5]. The absence of classes and thus inheritance in HRT-HOOD are logical consequences of the problems of combining inheritance and objects with an own thread of control.

[3] Necessary because a specification without the possibility for a validation seems useless.

[4] Some models reflect only a subset of system properties, e.g. they describe only the bus I/O behaviour of the system. While these models are useful within their specific context, they do not lead to an implementation. Hence, they are not regarded as prototypes.

[5] In [MDN+03] the reader can find a good comparison of UML, HOOD, and HRT-HOOD.

The objects in HRT-HOOD have two representations – a graphical and a textual representation – to achieve two different aims. The graphical representation provides an intuitive hierarchical view on the design. HRT-HOOD provides six different object-types. These are: *Environment, passive, active, protected, cyclic, and sporadic*. While the first three were taken from HOOD, the later three were newly introduced to reflect the real-time behaviour of objects. The hard real-time part of a system designed with the HRT-HOOD method contains no active objects because they are not fully analysable. Nevertheless, they are used during the refinement of the design or for non real-time activities. Objects are displayed by rounded rectangular boxes. In the left-upper corner one or two letters denote the objects type[6]. The *provided (method) interface* is enclosed in a box attached to the objects. Figure 2.2 shows the first decomposition of a HRT-HOOD design example[7].

The figure shows the top-level active object `Crane_System` and its decomposition into the main functional sub-objects. The arrows denote the *use-relation* of objects. For example, the cyclic `Job_Control` object uses the protected object `Sensors` and the active object `Actuators`. The arrows with circles denote the data-flow between objects. For example `Job_Control` receives the values of type `Alpha_T` and `PosCar_T` from the `Sensors` object. The flash-shaped arrows annotate *calling conventions* describing the blocking behaviour of a method invocation. Obviously, unbounded or unspecified blocking behaviour leads to non-analysable systems and therefore must be forbidden. However, limited blocking times can be allowed under certain circumstances. Thus, a formal treatment of calling conventions is necessary to enable the schedulability analysis of HRT-HOOD specifications.

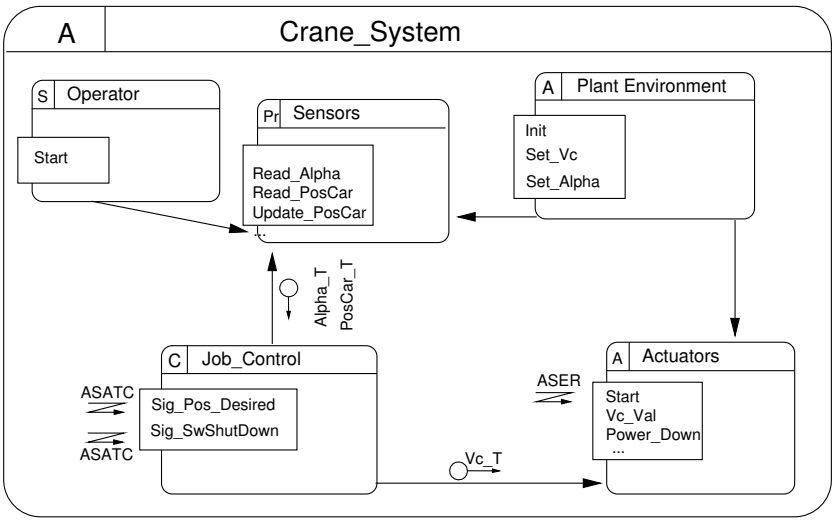

Figure 2.2: A top-level HRT-HOOD design

In contrast to the textual representation, the graphical representation does typically not contain all the details. The graphical representation helps mastering the complexity of system design by providing hierarchical views on different levels of abstraction. Symbolic representations display various types of relations (*use, call, include*) between objects.

The textual representation is useful to give a detailed description of each object and its relationship to other components. It defines all attributes required to specify exactly the system and to generate the (skeleton) source-code from it. The objects define the program elements like packages, tasks, procedures, and functions. The algorithmic behaviour of methods can be specified by natural language, pseudo-code, or source-code. Code-generators can integrate these building blocks into the skeleton.

The objects in HRT-HOOD may be nested under certain constrains[8]. The objects have a name, a type, a *provided interface* which enables other objects to use them and a *required interface* that indicates the components, required by this object. Objects typically have attributes to specify non-functional requirements. They for example specify the period or the importance of an object. The

[6] A stands for active, C for cyclic, S for sporadic, Pr for protected, and P for passive.
[7] The example is a top-level specification for benchmark introduced in Section 8.1.1.
[8] E.g. a periodic object may contain a passive object but not vice versa.

method's real-time behaviour within the objects is defined by attributes such as *worst case execution time* (wcet).

Commercial HRT-HOOD design tool like the Stood™ tool from tni-valiosys combine both representations into one design environment. The user can draw the system in the graphical representation and refine the textual representation on demand. The Stood tool is able to generate different mappings into target languages such as C++ and Ada95.

2.5 XML

Since XML plays an important role in the definition of ComiX (Section 6.3), this section will shortly introduce the XML meta language. The e**X**tensible **M**arkup **L**anguage, abbreviated XML [W3C] is a restricted subset of SGML (Standard Generalised Markup Language [ISO86]). The main goal for the World Wide Web consortium (W3C) was to provide an easy to use language, which is compatible with SGML. XML was designed to provide a standardised way to define simple description languages. Strictly speaking, XML in itself is thus not a language - it is rather a meta language or a notation.

2.5.1 Document type definition

The document type definition (DTD) defines, similar to what BNF is for context free languages, the syntax of an XML document. It defines the components allowed in a particular class of XML documents. Possible components in any XML documents are entities for symbolic values, elements as compounds of data and attributes to define structured data.

A great advantage of XML is the immediate availability of a universal parser API to traverse and analyse any given XML document. For most programming languages (such as Java, C, C++, Ada95, or Perl), XML parser implementations are available. An XML parser operates on an internal data structure called DOM (Document Object Model). Additionally, a *validating XML parser* can check, whether an XML document is consistent with a given DTD.

The following Listing 2.1 shows a DTD for a simple address book to illustrate the above-mentioned terms. It defines one entity `GENDERS` with the possible values `male` and `female`. The root element `AddressBook` contains a list of `Person` elements. Each `Person` element has a two subelements: `Address` and `Account` and contains three attributes: `name, age and gender`. The elements `Address` and `Account` are leaf elements with two attributes.

```
<!ENTITY % GENDERS   "male|female">
<!ELEMENT AddressBook (Person*)
<!ATTLIST AddressBook name ID #REQUIRED>
<!ELEMENT Person  (Address, Account)>
<!ATTLIST Person name ID #REQUIRED
                 age CDATA #IMPLIED
                 gender  (%GENDERS) "female" >

<!ELEMENT Address EMPTY>
<!ATTLIST Address street CDATA #REQUIRED
                  number CDATA #REQUIRED>

<!ELEMENT Account EMPTY>
<!ATTLIST Account bank CDATA #REQUIRED
                  number CDATA #REQUIRED>
```

Listing 2.1: A simple DTD for an address file.

2.5.2 XML notation

To explain the structure and the semantics of ComiX in Chapter 6, it is necessary to introduce some notations that will allow referring to components of an XML document. These notations will be illustrated by the following XML example in Listing 2.2.

```
<!DOCTYPE Address SYSTEM "address.dtd">
<AddressBook file="Customers">
  <Person name="John Doe"
                age="23"
                gender="male">
    <Address street="Baker street"
             number="12"/>
    <Account bank ="Bank of England"
             number="123456"/>
  </Person>

  <Person name="Mary Smith">
    <Address street="Pennylane"
             number="2..4"
             gender="female"/>
    <Account bank  ="Bank of Scottland"
             number="65432112"/>
  </Person>
</AddressBook>
```

Listing 2.2: Address book

XML documents are structured through elements defined in its DTD[9] which is given in the DOCTYPE declaration (line 1 in the example). Attributes can be defined (by the DTD) to be either `REQUIRED` or `IMPLIED`. While the former denotes that this attribute is mandatory, the latter means that it is optional. The `Person` element for example has an `IMPLIED gender` attribute and for the second `Person` (Mary Smith) this attribute has been omitted.

For reader interested in larger examples of DTDs are refered to Appendix A and Appendix B of this work.

2.6 Ada95

Since Ada95 is the software description language within the OOCOSIM method, this section will give a brief overview of the special characteristics and the motivation to choose Ada95 for the software. [Bar95] may help the reader to learn more about programming in Ada95 while the language reference manual (LRM) [TD97] defines the syntax and semantic of the Ada95 in a formal way.

Ada95 has a modern language concept providing genericity, a flexible library mechanism, an exception mechanism, and full object-orientation. One of the main concerns, when the language was developed for the DoD (Department of Defence) was reliability. Therefore, the syntax prefers readability against shortness. The compilers needed to prove their accordance to the LRM[10] to be regarded in safety critical projects. Many platform-depended issues (e.g. the run-time system) are encapsulated by language constructs, which enhances the portability of Ada95 across different

[9] The definition of a DTD is not mandatory for XML documents but for this thesis only XML documents defined by a DTD will be used.

[10] This was certified through an independent authority using a large set of conformance tests.

platforms. Thus, the language contains many concepts that are useful for embedded system software. This includes statement for the direct access to shared memory and interrupts for the communication with the hardware, a detailed and well-defined tasking concept to support concurrent programming, and a real-time annex to support real-time critical software.

Most other languages use direct operating system calls or target specific libraries to model concurrency, timing, and memory-mapped I/O. Among the few exceptions is Java [Coo98], which supports concurrent programming and is highly portable due to the concept of the so-called *Java abstract machine*. Concurrent programming is supported by rather simplified language concepts (so-called *threads*) compared to what most operating systems come with. While these language components could theoretical serve as a basis for more elaborated functionalities, the informally defined real-time behaviour of Java threads makes this rather difficult.

Finally, yet importantly, Ada95 is the preferred implementation language for HRT-HOOD, which is one cornerstone method for the OOCOSIM design approach. The object types defined in HRT-HOOD map easily to tasks, packages, and protected objects in Ada95[11].

2.7 VHDL

Verilog [Sag98] and VHDL [Ash95] (Very High Speed Integrated Circuit Hardware Description Language) are the hardware description languages dominating todays industrial practice in hardware design. Upcoming languages like SystemC [Swa01,GLMS02] and System Verilog [Acc03] offer promising advantages for system-level design but still lack tool support to gain the same level of acceptance. VHDL addresses hardware design at a wide range of abstraction levels as the behavioural level, the register transfer level down to the very low levels describing gate net lists. As a hardware description language, it supports hardware inherent concepts like parallelism, signals, processes, clocks, and timing very well. However, it lacks some important features to be a valuable system description language. While providing high-level concepts like genericity and packages, there is no abstraction mechanism for communication between design entities. All the communication is handled by signals. While this is sufficient at a rather low level of abstraction, it becomes a bottleneck for the design efficiency at system level. Furthermore, there is no support for object-orientation or dynamic creation of entities. A language like VHDL, being tailored only for hardware design, is semantically inadequate especially regarding concurrent software[12]. Besides these conceptual problems, there are also practical reasons why VHDL should not be used to model software. The performance of current VHDL simulators is sufficient to observe the fine-grained effects in hardware but far to slow for effective debugging of software. Moreover, there is no software IDE (integrated development environment) for VHDL, containing for example a software compiler translating VHDL into machine code.

2.7.1 Object-oriented VHDL

In the beginning of this thesis, the OOCOSIM approach was based on an object-oriented extension of VHDL to describe the hardware. In the following is described why this was abandoned in favour of VHDL.

In the past, there were three major attempts to add object-orientation to VHDL. These were: SUAVE[13] [AWM98a,AWM98b] by P. Ashenden, G. Schumacher with OO-VHDL [Sch99], and various participants in the REQUEST project with Objective VHDL [RPRN98c, RPRN98b, MNPRR97]. SUAVE had a very appealing syntax and language concepts like for example channels. Unfortunately,

[11] There also exists a mapping for HRT-HOOD onto C++ implemented for example by the Stood™tool of tnivaliosys [Dis00] but with severe limitations regarding the real-time behaviour.

[12] Processes in hardware are truly parallel, while tasks in software are concurrently using a single resource namely the processor.

[13] SUAVE is the abbreviation of a really long expanded name: Savant and University of Adelaide Very High Speed Integrated Circuit Description Language extension.

many advanced concepts of SUAVE are not synthesisable. The work on SUAVE stopped even before a simulator was available. OO-VHDL has a well-defined formal semantic but was found to be too difficult to use in industrial practice. Therefore, only the language definition but no tool support was ever developed.

Objective VHDL was designed to be synthesisable and there was a large EC-funded project developing the language and the supporting tools at the time when the work for this thesis started. Therefore Objective VHDL appeared to be the most promising approach. Unfortunately, Objective VHDL also failed to become an ISO-standard and the tool support was not sufficient. While there was a prototypic translator for Objective VHDL to VHDL [RPRN98a], it never reached a mature version. A native Objective VHDL simulation had never been attempted. Therefore, the validation of an Objective VHDL specification is rather difficult. Similar to the other two approaches, the development stopped and therefore the only option was to use an existing hardware description language. As the core aim for the co-simulation was to show the coupling of hardware and software in a time-synchronous way, VHDL could serve well as a proof of concept. Today, new tools like the ODETTE synthesizer [GO02] translating SystemC into VHDL allows using object-oriented concepts into embedded system design together with co-simulation.

2.8 ASIS

The *Ada Semantic Interface Specification (ASIS)* [Int99] provides an abstract programming interface (API) to analyse Ada environments[14].

An ASIS based application operates on the syntactic tree of the entire application including all dependent packages. ASIS supports the identification of particular components in the source-code of an application and to perform semantic transformations. Since ASIS operates on the syntactic tree according to the Ada95 LRM [TD97], ambiguous keywords like `delay` or `for` can be identified unambiguously.

To achieve the correct co-simulation behaviour in OOCOSIM some complex transformations of real-time and interface related Ada code segments are necessary. Thus, ASIS plays a significant role in the implementation. A great advantage of ASIS is its portability to describe a transformation tool independently of its platform. Even more important was the fact that ASIS is able to analyse a complete Ada environment. Some transformations are only possible with the knowledge about more than one compilation unit. An implementation with standard scanner and parser generators like Yacc [Joh79] and Lex [LS75], which was in fact the approach implemented first, would have been very difficult to implement, because these tools can only handle one file/context at a time.

While ASIS provides a very powerful and universal API for the analysis of Ada code, it is rather difficult to handle multiple, complex code transformations. Hence, the implementation in OOCOSIM uses an object-oriented framework for ASIS called OFRASIS. This framework, developed in Oldenburg, allows partitioning the transformation into less complex logical tasks. In each of the partitions, a particular class of Ada statements is identified and translated. The code transformations itself are explained in detail in Section 7.5.3.

2.9 Object-orientation

Since the term *object-oriented* is used in so many and so very different contexts such as programming languages, databases, and system analysis methods it seems necessary to define it here. Some definitions are rather unclear or confusing [BS02]. Others [Mey90] are to restrictive, as in these definitions object-orientation requires a large set of specific features. The description given here follows Jähnichen and Herrmann in [JH02] and Xing [XB03] to a large extend.

[14] An Ada environment is define by the set of components of an Ada application.

For this thesis object-orientation is a paradigm; that is, a general principle instead of a technique used in particular languages or methods. The paradigm can be characterised by a minimal set of indispensable concepts. The essential three concepts are:

Natural modelling of reality by objects: Object-orientation is useful because objects are natural abstractions of real-world things. As a consequence, in object-oriented methods real-world objects a represented by (model)objects that have at least an identity(name), a state, and a formally defined interface[15].

Sharing of common properties: This concept refers to the fact that objects typically have common features or resources and need to share them in an efficient way. Inheritance (sharing among classes) and aggregation/delegation (sharing among objects) are two ways to implement this concept in object-oriented methods.

Abstract data-types or encapsulation: This concept coined in the context of formal verification contains three mechanisms. *Encapsulation* guarantees that the internal state of objects is hidden to its external users and that it is only accessible through methods defined in an explicit method interface[16]. The *classification of objects by types* guarantees that all objects instantiated from a particular type have the same external interface. The third concept in abstract data-types is the *defined explicit semantic of types*. While rarely fully implemented for object-oriented programming languages[17], it is important for a well-defined system-level specification method.

Other concepts, like (multiple) inheritance, templates/genericity, and polymorphism are regarded here as closely related but not essential to the object-oriented paradigm. These concepts, if used carefully, can surely improve the expressiveness of a methodology. Especially for the specification of reusable so-called *IP-components*, inheritance, polymorphism, and templates can be useful. A new class of components could be achieved by simply deriving a new class from the base class or combining several classes by multiple-inheritance. Polymorphic interfaces can make objects very flexible in that they are able to handle different but similar data.

However, the semantic of inheritance for the typical objects found in an embedded system like active objects[18] in general is hard to define. An active object like a task or a process inherits methods/attributes from its parent-class and adds new methods or attributes to it. The difficulty is to define the own thread of control; that is, to integrate these new methods and attributes in its behaviour without completely redefining it. Consequently, in this thesis object-orientation is restricted to the basic features mentioned above.

[15] Interface denotes the means to access the object.
[16] Encapsulation was already a concept in non object-oriented languages like Modula 2 [Wir85].
[17] Some languages try to define the semantic of Acts like e.g. Eiffel by pre/post-conditions. However, Eiffel could never reach a high practical relevance in general and especially not in embedded system design.
[18] An active object here denotes an object with an own thread of control.

3 Related Works

For many years, hardware/software co-design has been well-recognised research area. The large amount of publications makes it virtually impossible to cover all existing approaches. Instead, this chapter tries to give an overview of how previous and actual methods in research and industry approached the challenges in embedded system co-design, hardware/software interface specification, and interface synthesis (see also Section 6.2).

Each section gives a short summary and presents the similarities and differences between the related work and the methods presented in this thesis.

The sections will present the most significant and characteristic approaches in co-design according to three categories namely *General frameworks, homogeneous*, and *heterogeneous* approaches.

General frameworks provide typically only a backplane or a synchronisation standard that allows domain specific tools or simulators to plug in. These frameworks are rather meta-methods as they allow building a particular design flow based on that framework. The frameworks presented here handle only the problem of co-simulation of a heterogeneous design that is using different models for the component-specifications.

Homogeneous approaches start from a single system description language, which consequently needs to capture all aspects of an embedded system. Since only one specification language is used, the simulation model is usually easy to achieve. However, depending on the expressive power of such system description languages, the mapping of language concepts onto hardware and software is difficult. Consequently, only small subsets of these languages are supported for the implementation of systems.

Heterogeneous approaches use specialised languages for each domain. The domain specific languages typically can be synthesised more easily than general-purpose specification languages. The challenge here lies in a co-simulation model that integrates and synchronises the simulation semantics of the domain model specifications. Similar to *Polis*[1], some heterogeneous approaches are based on generals frameworks.

3.1 General frameworks

The category *general frameworks* contains concepts that provide methods to co-simulate different design languages. These frameworks do not provide the language concepts or notations to describe the embedded system's components. They are rather meta models that allow to integrate arbitrary design languages into a co-design method.

Ptolemy

The most prominent framework for co-simulation systems is Ptolemy [EKL94, KL92, Le99]. It provides a general method to integrate different description models executed in separate simulation environments into one co-simulation system. The full Ptolemy framework is far too complex to be covered here. Thus, this section can only give a superficial overview of the terminology and the underlying principles of this powerful modelling environment.

The Ptolemy kernel pre-determines the structure of all components in the co-simulation system by means of a class hierarchy. The basic class is a *Block*. The Blocks interfaces are called *PortHoles* exchanging *Particles* (messages) for communication through *Geodiscs* (channels). The Block class

[1] Polis was implemented using the *Ptolemy* framework

defines methods for its initialisation (initialise) and execution (go) or the access of its PortHoles. All other classes are derived from these basic classes to ensure that they all provide at least this basic interface.

Stars representing basic functional elements are derived from Block. Collections of Stars describing a subsystem are called *Galaxy*. It also contains a class *Runnable* to describe the execution behaviour; that is, the joint behaviour of Stars, of the Galaxy. A *Universe* describes the entire simulation system. *Domains* are used to combine Galaxies using different computational models. *Wormholes* connect domains with each other. *EventHorizons* synchronise the domains, i.e. coordinate the local schedulers and convert particles. Since the structure and the interfaces of the components in a Ptolemy system are predefined, the kernel can coordinate the behaviour of the different sub-models.

Ptolemy provides a collection of pre-defined domains like SDF (Synchronous Dataflow), DE (Discrete Event) or FSM (Finite State Machine) but allows defining additional domains. The strength of Ptolemy lies in its flexibility and its expressive power. With this framework, arbitrary models of computation can be connected provided a corresponding domain is defined. The disadvantages are the high effort required to integrate a new model into the approach and especially the informal synthesis path to an implementation.

COSMOS(SOLAR) Co-Design environment

The abstraction for system specifications accepted by the *COSMOS* [CHM+99] design environment are communicating processes represented in an internal format called *SOLAR*. Its current implementation accepts only SDL (see below) as a front-end, but the COSMOS environment was designed to support multi-language inputs unified by the SOLAR intermediate format.

The behaviour of processes in SOLAR is described by so-called *Extended Finite State-machines* (EFSM). The processes communicate via remote procedure calls (RPC) defined in a target specific library. The general target platform is a distributed multi-processor real-time system composed of programmable processors and dedicated hardware components communicating through a network.

For co-simulation and analysis, the system-model is translated into SOLAR. For co-synthesis, the SOLAR model is translated into C (for software) or VHDL (for hardware). The co-simulation provides two levels. A functional high-level co-simulation and a real-time model, where the C code is executed in an RT-level model of the target processor. COSMOS takes a very general approach and aims to address complex systems. The mapping onto the target platform is based on libraries, providing implementations for the elements of SOLAR[2]. Hence, the efficiency and applicability of COSMOS depends to a large extend on the libraries.

3.2 Co-Design based on a homogeneous system specification

The homogeneous design flow approaches as shown in Figure 3.1 use a single specification language to describe hardware and software. There are obvious advantages of such an approach. All components of the system can be described using a single formalism, which makes simulation, and partitioning much easier. The disadvantage however is, that very different domains like hardware and software must be covered by a single language and a single simulation model. Some semantic concepts, which are inherent to hardware, like parallelism or signals, are meaningless in software while typical software concepts like dynamic memory allocation, recursive calls, or pointers are not applicable to hardware.

Many approaches try to extend existing languages for a particular domain by concepts from another. However, none of these approaches provides an integrated methodology that allows applying all modelling concepts to arbitrary components. Due to the semantic differences of hardware and software up to now, there is no simulation and synthesis semantic that allows the efficient modelling in a truly common language.

[2]For example for the mapping of RPC onto hardware and software.

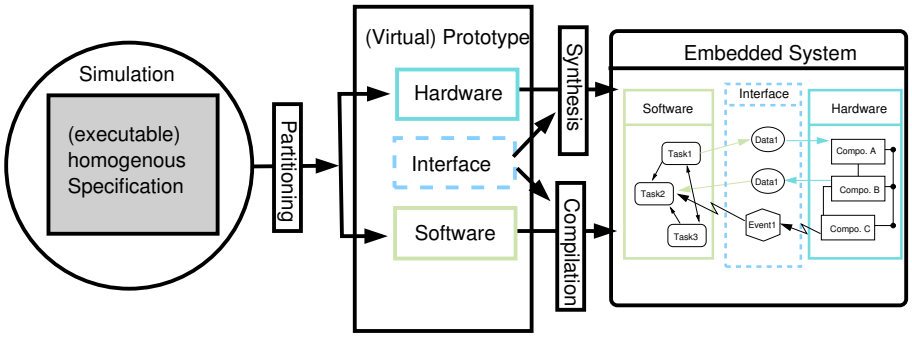

Figure 3.1: Design flow with homogeneous specifications

C-like languages

The *Cosyma* (COSYnthesis for eMbedded Architectures) approach from the Technical University of Braunschweig [BE97] is based on a C-like language called C^x. It augments C with process-oriented statements to cover the specification of hardware inherent concepts like parallelism and timing. The specific target architecture contains a single RISC-CPU[3], fast RAM, and an application specific co-processor or accelerator, all connected through a bus. The hw/sw-communication is modelled by C-functions using abstract channels. The channels are later either removed by optimisation or mapped onto physical channels, i.e. shared memory. An additional so-called *constraint and user directives file* can be used to control the synthesis process.

Cosyma's simulation model is based on a RT-level model of the target processor. The simulation mainly aims at helping the hw/sw partitioning into C-code for the software and HDL code for the hardware. The partitioning strategy starts from a full software system and optimises the performance by moving certain parts into hardware. Furthermore, Cosyma automatically generates the scheduler for the application.

Vulcan [GM93] developed in Stanford is based on a language called *HardwareC*. It is similar to Cosyma in that it also uses an extension of C and targets the same architecture. The partitioning strategy here starts from a complete ASIC implementation and tries to reduce the cost by moving functional blocks into software. Both, Vulcan and Cosyma, assume a single-threaded model of computation, i.e. if the hardware is active the software on the CPU waits in an idle loop and vice versa.

Synchronous languages

The LYCOS (LYngby CO-Synthesis) system [MGK97] mainly targets the problem of hardware/software partitioning, i.e. 'to find a feasible partition, that is, a partition which fulfils the requirements' which includes the architecture (CPU, interface, and ASIC). To obtain this goal, all blocks[4] are represented independently of their later implementation using a model of computation called *Quenya*. The LYCOS development system provides tools to translate C and VHDL specifications into Quenya[5], which serves as an universal representation upon which serial algorithms operate. Quenya represents a design by a network of asynchronously communicating CDFGs (Control/Data Flow Graphs). Finally, after analysis, partitioning, and optimisation, code generators produce assembler-code for the target processor and VHDL code for the custom hardware to obtain the implementation.

[3]The TU Braunschweig provides a SPARC processor with the method
[4]Also called *basic scheduling block* in LYCOS.
[5]Lycos could also be characterised as a heterogeneous approach, because C and VHDL can be used as input language. However, since all major design activities are based on Quenya, Lycos' core is homogeneous.

SDL

The SDL (Specification and Description Language) [EHS97] aims mainly at formal specification and definition of systems dominated by communication protocols. Typical examples for such systems are devices for the DECT or ISDN protocol standard. SDL comes in two representations namely SDL-PR (textual) and SDL-GR (graphical) describing the structure and behaviour by extended finite state machines. It is based on a synchronous semantic. SDL has proven its efficiency in its domain (communication/protocol dominated systems), but it seems not to be suitable for the class of real-time hardware/software systems targeted in this thesis.

Programming languages in embedded system design

For reactive systems many languages and especially real-time languages [Sto92] have been proposed. The following list is not exhaustive and instead names only the most prominent with their main characteristics:

- Esterel [MEI04]: An imperative, well-typed, and parallel real-time language based on the so-called *synchronous assumption*, i.e. only explicitly specified time is consumed. All other transitions occur instantaneous, i.e. without over-head. The communications between subsystems is realised by broadcasting. The advantage of these simplifying assumptions is that it eases the formal verification of the design.
- Statecharts [HN96]: Statemachine-based model enriched by hierarchy, parallelism and communication. Transitions are triggered by events and conditions.
- Software programming languages: Almost all programming languages have been tried with more or less success for system level specification. On one hand they are usually lacking of concepts for parallelism, event handing and timing. On the other hand, they provide concepts like dynamic memory allocation, pointers, or recursive functions that are hardly implementable in hardware. Thus, the majority of languages were used for functional specification and verification only.
- Language extensions: Several attempts were made to extend programming languages for the specification of hardware by adding parallelism and communication to these languages. In general, the extended languages do no longer fulfil their defined standard and thus commercial tools are no longer applicable. One prominent exception here is SystemC because it is implemented as a C++ library (see also Section 3.2).
- SpecC [GDPG01]: Uses a combination of finite state machines to describe the control flow and programming language segments annotated to the states for the data-flow related specifications. It also supports the synthesis based on a generic target architecture, which is very similar to an application specific processor.

Matlab™ [Hof98] provides an imperative programming language that is integrated in a commercial product of Mathworks Inc.. While Matlab is very powerful for the design of algorithms, it lacks of the ability to describe hardware at a lower level of abstraction. Simulink™which is part of the same tool environment than Matlab is popular in the industry as a graphical entry language for the so-called *golden model*[6]. If the components in a Simulink model are taken from the *Realtime Workshop* (Simulink library by MathWorks) it is possible to synthesise the design automatically from this model. However, the difficulties when additional user defined components are required, reduces the applicability to general design problems. In MASCOT [BJ00, BJ99] two approaches are described in which SDL and Matlab are combined to a system-level co-design method.

[6]The golden model describes the ideal system to be build. It is serves as a reference implementation of the embedded system, i.e. the final implementation must resemble the input/output behaviour of the golden model as exact as possible.

Hardware description languages (HDLs)

Hardware description languages like VHDL or Verilog [Sag98] provide good concepts for describing parallelism and hardware specific issues like ports and registers. They cover different levels of abstraction like behavioural level, register transfer level or at the very low-end gate/transistor level.

However, for the design of software they do not address important key concepts like abstract communication, data-types, object-oriented design, or dynamic behaviour. Furthermore, HDL simulators are optimised for hardware debugging. Their performance in a medium complex hardware design is sufficient in a hardware adequate time resolution[7]. These simulators are not able to execute large amount of software for system validation in a co-design environment. Finally, the user interface of these simulators is designed for hardware simulation and monitoring.

Software can be modelled either by executing the software using a processor model within the hardware simulation or using the so-called programming language interface (PLI) of the simulator. The latter allows executing arbitrary C-code on the simulation host machine. This co-simulation scenario has a superior performance but requires a user defined synchronisation mechanism.

C++ based approaches

SystemC [OSC01a, OSC01b, Swa01] is a system description language based on C++. Consequently, it contains the full set of object-oriented language concepts inherited from C++.

Additional to C++ it contains language constructs for typical hardware concepts like concurrency, signals, and modules. It is implemented using the class-library mechanism for its extensions. Therefore, every C++ compiler is able to process SystemC descriptions. To run a simulation, simply means to execute the compiled specification directly or within a C++ debugger.

Until today; that is, with the actual version 2.0, hardware/software co-design is addressed at a very abstract level. The software part of the system can be modelled within so-called *modules* using the full C++ language, but semantically it is handled like any hardware component. In this respect, modelling of software in SystemC is similar to using the foreign model interface in VHDL. Consequently, the means to model the software real-time behaviour and the hardware/software interfaces are hardware oriented. In particular, SystemC 2.0 offers no language concepts to model concurrency for software containing multiple tasks. The Open SystemC Initiative (OSCI) has already identified this deficit and plans to integrate abstractions for concurrency for the next major release SystemC 3.0. A more elaborated description of hardware/software co-verification with SystemC can be found in [SG00].

Until today, only a small subset [Syn01a, Syn01b] of the full SystemC 2.0 language can be synthesised automatically. The expressive power of this subset is very similar to VHDL or Verilog at register-transfer or behavioural level. Since higher-level concepts in SystemC (like channels or the master-slave library) lack universal applicable refinement strategies, the usage of SystemC as system design language are rather limited.

To overcome some limitations of the SystemC, an object-oriented extension to the synthesisable subset of SystemC named *ODETTE System Synthesis Subset (OSSS)* [GO01, GO02, GO03a, GO03b] has been developed. It adds hardware specific variants of class instantiation, inheritance, polymorphism and so-called *shared objects* to the synthesisable subset. This however, does not address the core problems in hardware/software co-design.

System Verilog

Recently, a new system level design language called *System Verilog* was promoted by Synopsys Inc.. It tries to overcome some basic limitations of Verilog by adding higher-level concepts already familiar from VHDL like enumeration types and user defined data types in general. Furthermore, abstract

[7] According to [HSK00] the typical performance of a hardware simulation of an embedded processor executed about 5 instructions per second. Thus, 1 millisecond on a 50 MIPS embedded CPU would take about 2.8 hours.

concepts like communication channels enhance the expressive power of Verilog into the system-level design domain. System Verilog also integrates concepts for verification into the language, thus providing a complete language for design. Another advantage is the wide industrial acceptance of its predecessor Verilog especially in the USA. However, until today there is very little tool support for System Verilog[8]. Thus, it is difficult to judge the suitability of the language in the domain of co-design. The core strength of Verilog was its simplicity and the efficient synthesis tool-chain. System Verilog adds many new features, which might make it difficult to keep its strengths.

UML

The UML (Unified Modelling Language) [OMG01] is probably the most prominent graphical object-oriented modelling language. Many of its dialects [Sel98, Neu00, MDN+03, GE03, KBNO03, Kab02] have been proposed for the design of embedded system. However, none of these have directly addressed the modelling of application specific hardware or the hardware/software interface. Hence, there is no seamless refinement flow from UML into a concrete implementation in hardware and software.

Due to its high expressiveness, UML is applicable for the requirement capture or for modelling complex, software-dominated systems. For the class of embedded systems addressed in this thesis, it seems to be not adequate.

Recap

Many of the homogeneous approaches presented above are well suited to express designs in a very specific application domain, dominated by either hardware or software. Some provide a formal semantic, which enables analysis or formal verification of system properties. The common strength of these approaches – the principle ability to express the whole design within a single formalism – is also the source of its weak point when it comes to implementation. Some concepts are extremely abstract[9] as with Esterel or Statecharts, which makes efficient mapping to hardware and/or software difficult.

Other approaches are restricted to an execution architecture that is easily and efficiently to implement either in hardware (HDLs, FSM and state-based models) or in software (programming languages). When applied to a system containing hardware and software, these approaches show severe deficits.

Another problem is how to model communication interfaces between the hardware and software components. The interfaces and the synchronisation mechanisms for hardware and software are very different but in a truly homogenous approach, they should be modelled by a common concept. Consequently, only very abstract concepts covering both domains are applicable. However, such abstractions like for example abstract channels are rather difficult to implement in hardware and software. Due to these difficulties, hardware/software interface design in this work is regarded a different domain, which requires dedicated methodological support. Some of the existing approaches in this context are presented in Section 6.2.

3.3 Co-Design methods based on a heterogeneous system specification

The obvious alternative for the design of heterogeneous systems is to use a heterogeneous specification; that is, using different description languages to specify the subsystems adequately. Figure 3.2 shows a generic design flow based on heterogeneous specifications.

[8] Many EDA tool vendors have announced support for System Verilog. In the first releases, they mainly target at the verification capabilities of the language.

[9] Abstract here in particular means to use concepts with no immediate representation in hardware or software.

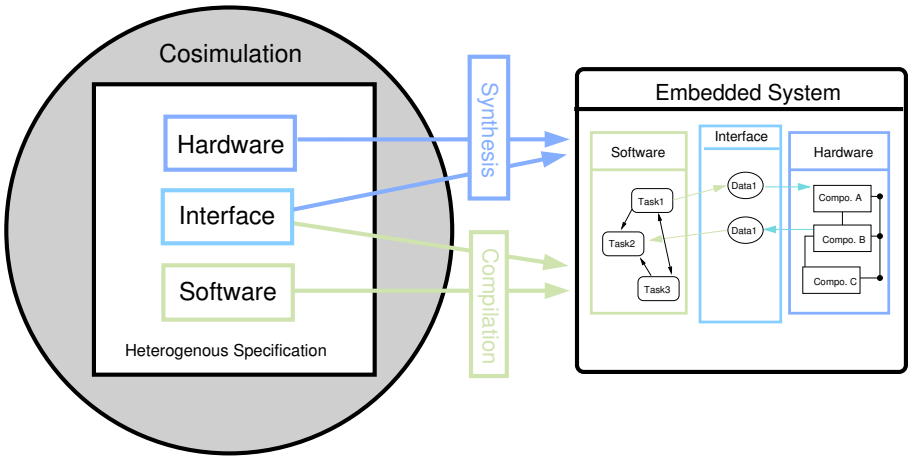

Figure 3.2: Design flow based on a heterogeneous specification

Within these approaches the designer can choose the appropriate concept or model of computation to describe and implement the various system components. The methodological challenge is to merge these modelling concepts into one unified co-design method. The co-design method needs to provide a notation to define and implement the communication interface and a mechanism for the synchronisation between distinct domains such as hardware and software. The OOCOSIM method falls into this category and a large part of this work is indeed dedicated to this unification of models and concepts.

Polis and Pia

Pia and Polis [BCG+97] are both co-simulation environments built on the PTOLEMY (Section 3.1) framework. A system in Polis is specified by a set of so-called *co-design finite state machines* (CFSMs), which can communicate with each other. The CFSMs are mapped to Stars within the discrete event domain of PTOLEMY. For the refinement of the real-time behaviour, instruction set simulators (ISS) can be used. Wrappers adopt the particular interface of each ISS to a standard interface of a PTOLEMY Star. By this mechanism different ISS can be coupled into the simulation through the same kind of Stars. To enhance the simulation performance, Polis provides a mechanism to avoid re-evaluation of previously calculated timing information. After the first calculation a key and the timing information can be stored. If a calculation with the same key occurs again the timing information is re-used.

Pia describes the system as a set of physical components using the Pia language. The components have interfaces, which can receive or send events through ports. Wires describe the connections between ports. The structure is similar to a hardware block diagram or to a hardware description language. Software components can be described in C-code. The hardware specified in Pia and the software in C-code is compiled by the Pia-preprocessor into the Pia-domain (an extension of the discrete event domain) of Ptolemy. The software code can be executed by a model of the target processor, described as a Pia hardware component or by host code execution[10] to speed-up the simulation. In Pia performance enhancement can also be achieved by abstraction of communication models between hardware and software or by optimistic scheduling.

[10]Source-code is compiled for the host machine and executed on it at a much higher speed.

Coware N2C™

Initially the CoWare tool-suite [RVBM96] was developed at IMEC, Belgium and later became the commercial product CoWare N2C [CoW00].

A CoWare design is composed of blocks. The behaviour of blocks can be described in C (untimed), *RTC* (C enriched by register transfer level constructs to describe hardware), VHDL, or Verilog using so-called *Host language encapsulation*. Blocks communicate via ports and can be decomposed hierarchically to achieve the desired structure. The CoWare design-flow typically starts with an untimed, functional C-model, which then can be stepwise refined to a cycle-accurate VHDL or Verilog model.

The software components (blocks) can be mapped onto a target processor and the hardware/software interface to memory mapped I/O. The co-simulation can be achieved using a cycle-accurate instruction set simulator.

CoWare N2C is a commercial framework including tools for the hardware/software partitioning, the hardware/software interface generation and the simulation. The execution of VHDL or Verilog is delegated to commercial HDL-simulators such as ModelSim™(Mentor Graphics Inc.) or Leapfrog™(Cadence Inc.).

Seamless CVE™

Mentor Graphics offers a commercial co-simulation tool called *Seamless CVE* [KN99]. The software can be described in C and the hardware in VHDL. The Seamless co-simulation kernel allows executing the software in an instruction set simulator or untimed as native compiled code. The hardware as well as the system bus is simulated in an HDL-simulator. The simulation kernel guarantees a consistent view on the shared memory for hardware and software. Executing each bus access in the VHDL simulation would slow down the simulation to unacceptable speeds[11]. Instead, Seamless allows determining which memory accesses are passed to the VHDL model and which are handled efficiently in local memory. The optimisation technique sketched above can speed up the simulation significantly, but requires a lot of non-trivial user input.

The simulation concept for the compiled code has no notion of real-time and is therefore only useful for pure functional verification, while the real-time capable simulation requires a cycle-accurate instruction set simulator for the target processor.

PLI based co-simulation

Most HDL simulators provide a so-called *programming language interface (PLI)*, which allows to execute functions written in C as part of the hardware model. The PLI mechanism provides an alternative way to describe hardware blocks. This is in particular useful when complex algorithmic blocks must be executed.

The co-simulation models based on this mechanism typically execute the software model in C by simply calling these functions without any notion of time. Thus, these co-simulations are useful only for functional verification. The real-time behaviour of the target software can not be efficiently modelled with this mechanism. Many more sophisticated co-simulation methods and tools (e.g. Seamless and CoWare N2C) rely on this mechanism to couple a HDL simulation with the software simulation environment. The co-simulation approach presented in Chapter 7 also uses extensively this basic mechanism to implement synchronisation and communication between hardware and software model.

Recap

Heterogeneous approaches, while making co-simulation and partitioning difficult, offer the great advantage of providing tailored primitives for each modelling domain. Thus, heterogeneous approaches

[11] In a full simulation every instruction fetch in the software simulation requires a bus access in the VHDL model.

not only model the components in an adequate way. Furthermore, they support the possibility to automatically implement the system's components based on their co-design model. Only with automatic implementation support a seamless design flow can be maintained.

The methods presented above all have particular strengths and weaknesses. HDL-based and ISS-based techniques support analysis at a very detailed and exact level but lack sufficient simulation speed to verify the class of embedded systems targeted in this work. The application of other approaches, like HCE or programming languages, provides abstractions which certainly improve the simulation performance but cannot cover important aspects such as the real-time behaviour of the co-design system. Moreover, some abstraction mechanisms (for example synchronous approaches or Ptolemy) omit important aspects (real-time or hardware/software interfaces), that need to be defined for an automatic mapping onto efficient implementations.

The method introduced in this thesis will provide a mechanism to model systems in a rather abstract way without neglecting important aspects such as the real-time behaviour or the hardware/software interface.

4 Design of Embedded Systems

The intention of this chapter is to describe and discuss the design process and the principles applied in embedded system development. After the introduction in Section 4.1, Section 4.2 and Section 4.3 describe and compare two general approaches to embedded system design. While Section 2.1 describes embedded systems from a more physical point of view, Section 4.4 characterises the class of embedded systems relevant for this work by their specific properties[1]. Deduced from these properties, the requirements on an appropriate design method are discussed in Section 4.5.

In particular, the list of requirements defined here provides a basis for the subsequent introduction (Chapter 5), detailed description (Chapter 6, Chapter 7), and evaluation (Chapter 8) of the OOCOSIM design methodology.

4.1 Introduction

The creative process of embedded system design begins with the *conceptual idea* of an electronic device to be build, i.e. it starts with needs, which forms only a vague specification of the intended system. *Requirement capture* translates these needs into an abstract but not necessary vague document. Requirements should not be too narrow for not becoming system specifications.

The choice of the formalism or notation is often directed by the biggest challenge in the design process. If software complexity is the most important issue, graphical notations like UML or HRT-HOOD can be quite useful. Due to their object-oriented and hierarchical character, the design of complex systems can be mastered with these methods.

If the algorithmic design consumes most of the design effort, often a tool like Matlab and/or Simulink [Hof98] is more appropriate. These tools provide expressive graphical notations and powerful abstractions for numerical and signal processing problems.

The transition point between requirement document and *system specification* is sometimes hard to define. However, at the point where the architecture of the embedded system is defined, the specification stage is definitely reached. Thus, *partitioning the system* into hardware and software is typically at the beginning of this phase. A system specification is a precise and unambiguous description of the system and its architecture but must not fully determine the implementation.

For a *seamless design-flow* (Section 4.5.1), the system specification should begin where the requirement capture ends. Since system specification involves many individual design decisions that are hard to formalise, this transition can hardly be automatic. Nevertheless, it should be supported in the methodology by providing similar abstractions in the early and the later phases of the design.

After a succession of refinements, the final system specification must lead to the direct input for building the implementation, i.e. the notation for the system specification must be processable[2] for automatic synthesis tools or software compilers. It should however not over-specify the implementation in a way that changing the implementation platform (e.g. the embedded CPU or the ASIC process technology) entails major changes to the specification.

The specification is the first level, where exact reasoning about the intended system becomes possible. Therefore, it should reflect all the important aspects of the system and enable at least the validation or verification of the systems functional behaviour.

[1] Due to the great variety of embedded systems, it seems unavoidable to constrain a design method to a certain class of systems.

[2] 'Processable' here does not mean that the specification notation must be the immediate input for the synthesis tool or compiler. It only requires an automatic path from the final system specification into the implementation flow.

Since embedded control systems consist of application specific hardware, software, and interfaces the system specification must support modelling at least these (sub-)domains of design. The system design may also contain analogue hardware components and sometimes the environment is specified separately, which would require a fourth and fifth sub-domain.

Finally, the specification must be translated into an immediately implementable description. At least here software must be specified in a programming language and complied for the target processor using a cross-compiler[3]. The hardware can be synthesised based on a description using a hardware description language (HDL) like VHDL, SystemC, or Verilog. Unfortunately, the hardware synthesis process based on existing tools is far more complex than the compilation of software. Only a strictly limited subset of the whole HDL is supported [Gro01]. Many of the more abstract language constructs and data-types must be refined manually to a synthesisable representation, e.g. bit-vectors. Furthermore, most synthesis tools require a certain coding style to be obeyed. Finally, the designer must be aware of the target technology and provide additional information to achieve an acceptable result from the synthesis tools. Consequently, the co-design flow must not only result in a HDL representation for the hardware but in a practically synthesisable description.

In summary the development process itself can be characterised as stepwise refinement[4] with each step adding more and more details and restrictions to the systems description. In conjunction with the refinement process, the level of abstraction is decreasing and the design space is shrinking. Finally, the implementation representing exactly one design alternative is left, covering all requirements. Figure 4.1 shows the general successive refinement steps relating the preciseness and the effort for changes in each design phase.

Late changes induce high costs: During refinement the systems description becomes more and more precise. Therefore, the possibilities for reasoning and validation are increasing. Unfortunately, the effort for changes in the later stages is also increasing. Obviously, in the first design phase *needs* can be added or changed easily. New or modified requirements must be kept consistent with the rest but this should not impose significant effort[5]. Changing the final implementation or a prototype, where all details are fixed even small changes or corrections will result in high costs for redesign. Therefore, changes should ideally be limited to the earlier design phases.

Costs for the redesign result from three major parts. The first part results from the need to build a new physical system or prototype. If the hardware is implemented in an application specific chip which needs to be changed the implied cost are enormous[6]. According to Gartner Inc. [TS04], about half of the ASIC (49 %) and FPGA designs (48 %) needed more than two iterations to solve timing problems. More than 10 % needed five or more iterations. Prototypes based on FPGAs (Field Programmable Gate Arrays) can help to reduce these physical device costs but since the prototype brings an additional phase into the design flow it may also incur additional design costs and delay to the release date.

The second part of the costs is related to the increased design effort. In the implementation, even minor changes often lead to major redesign steps, especially in a highly optimized system. For example, changed or added functionality must be integrated into the timing analysis of software or hardware and often the interface is affected. In the worst case, the changes lead to a full redesign of the system starting with a new specification. While the execution speed of the implemented design makes testing, i.e. the identification of errors fast and easy, it is often difficult to find the source of the erroneous behaviour. In embedded systems, the software cannot be traced as easy as in a personal computer system. Since typically only input and output are observable, the analysis of hardware is even more difficult.

[3] A cross-compiler generates machine code for processor different from the host machine it is executed on.
[4] Please note that the term 'stepwise refinement' here has a wider scope than the one Wirth defined in [Wir71].
[5] This is of course only true, when these new needs or requirement are imposed before the later design phases.
[6] The production of only the new mask for a 0.18 micron process can cost up to a million dollar and the price is increasing for every new technology.

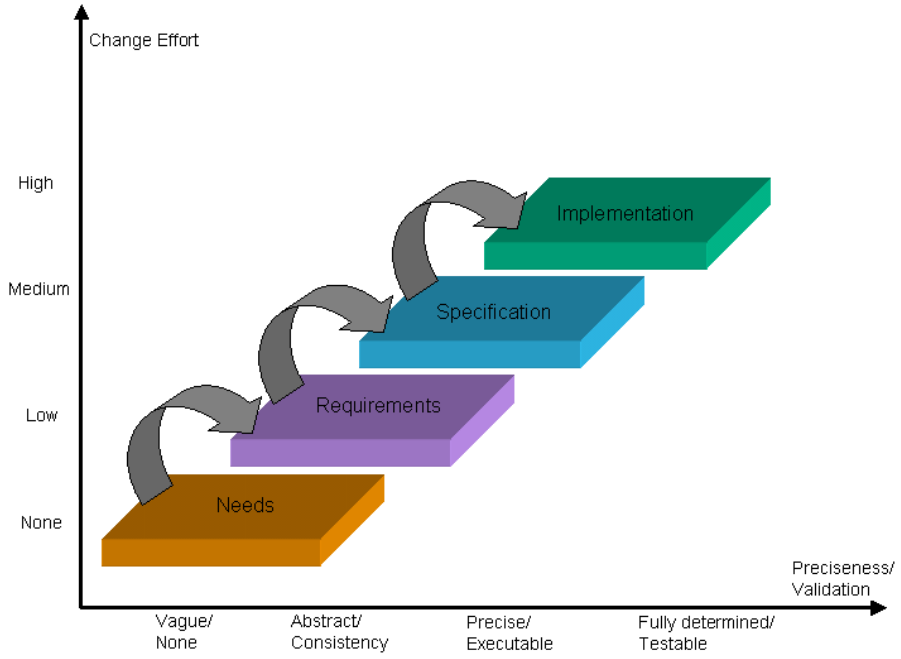

Figure 4.1: Refinement process for embedded systems

The large effort of late changes to the design does not only cause direct cost for designer effort and a new physical device. The third source for costs is due to the delay itself. Tight time-to-market constrains are imposed upon many products. Very often, only the first product on the market generates profits. Moreover, a delay may result in the rescheduling of production facilities and marketing efforts. These effects can easily lead to the complete failure of a project.

Therefore, the last stage where design mistakes or adverse decisions can be discovered and corrected with reasonable effort is the specification phase. In the specification, the system can be analysed and tested in depth because the behaviour of hardware, interface, and software can be observed relatively easy. Errors can be corrected by changing the implementation specification. This is the key reason why this work stresses the importance of an *executable and precise specification*, which allows the analysis of functional and non-functional behaviour of the system to be built. Consequently, methods for a precise specification and its validation therefore make the centre of the work at hand.

Two classes of design flows can be distinguished for embedded design. The traditional methodology is often used in current industrial practice but the modern co-design approach is gaining more and more interest and acceptance.

The approaches as described in the following two sections represent the idealised variants of their class. In the current practice, combinations of both are often applied. The description will neglect the initial phases (needs and requirements capture) prior to the initial system specification because they differ only after hardware and software are identified.

4.2 Classical embedded system design

The classical approach as depicted in Figure 4.2 uses a sequential approach where each sub-system (hardware, software, and interfaces) is developed in a separate design environment. The development cycle often begins with building a hardware prototype because embedded software usually

can be sufficiently tested only in interaction with the corresponding hardware. The application specific hardware is modelled using a hardware description language, simulated and tested at different abstraction levels and finally synthesised into an ASIC implementation.

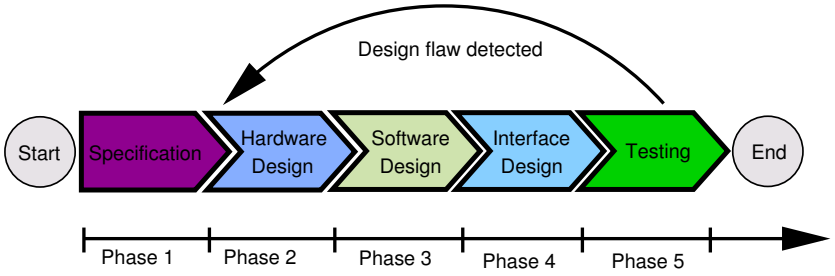

Figure 4.2: The classical design flow

Special prototyping boards carrying an FPGA (Field Programmable Gate Array) chip, optionally a CPU, memory and some I/O ports can reduce the effort to develop the execution platform for the embedded system. A large enough FPGA can implement the application specific hardware and even the CPU as a so-called *soft core*[7] to execute the software. Unfortunately, building such prototypes is still rather time-consuming and the result is rather inflexible since every change in the hardware model requires synthesising a new FPGA image.

Based on the execution platform, the target specific part of the software development process and the interface implementation can begin. Testing of software is done using the hardware prototype. When both parts are finished, the integration of hardware, software, and development of the interface in-between into the first prototype can take place.

A major drawback for this design flow is that the implementation and test of embedded software must be postponed until the hardware prototype reaches a semi-final status in the development process. The interface design, while being complex and error prone, takes place at the integration phase in the very end. The real-time behaviour and the hardware/software interaction of the later system cannot be analysed before it is actually running. If a design flaw is detected here, it might result in revisions of some design decisions, i.e. in building a new prototype.

Since time-to-market is an increasingly important issue for the success of a product based on an embedded system, the delay introduced by the classical approach often cannot be accepted. Moreover, design flaws or real-time problems can be detected only in late design phases. The real-time behaviour of the prototyping board and the final system is usually not the same as FPGA implementations of the processor or the ASIC in the finally implementation are functional equivalent at best. Thus, expensive changes or even redesigns introducing additional delays might be necessary.

The absence of a co-simulation environment is the main source for another serious problem namely the debugging. Prototype systems resemble the final system very well but they lack of appropriate debugging facilities. In general, it is impossible to stop the system in a certain situation and analyse the internal state of it in a comfortable way. It is in particular impossible to link the erroneous behaviour directly with the implementation specification; that is, the source-code. This makes debugging a difficult and time-consuming task.

In order to overcome the difficulties, the classical design flow is often combined with a strict version of the so-called *platform based design*. A rather static platform defining a processor, the bus system, fixed interfaces, and very limited resources for ASIC components is chosen as a basis for the embedded system. While this approach is suitable for software-dominated systems, it often leads to inefficient solutions for highly integrated hardware/software systems or so-called *systems on chips* (SoC). Especially in the latter case, tailored solutions become possible because all components of the system can be created from scratch.

[7] A soft core is a synthesisable computational component described in a HDL. Typically, the provider delivers the HDL code with additional synthesis scripts to ease the task of technology mapping, i.e. implementation for a particular technology process.

4.3 The Co-design approach

Most of the inefficiencies, delays, and extra costs identified for the conventional embedded system design can be avoided by the co-design approach depicted in Figure 4.3. All components of the embedded system are developed in an integrated process.

As in the traditional approach, the design begins with a initial specification. It may be based on a graphical notation or given as a so-called *golden model* in C++ or Matlab/Simulink. After the initial system specification is partitioned into hardware, software, and the interface, the development, refinement, and the validation can begin in parallel.

To be able to start productive software development, system validation must be possible while a working physical hardware prototype is not yet available. Therefore, it is necessary to provide a realistic executable model of the embedded system, a so-called *virtual prototype*; that is, a co-simulation.

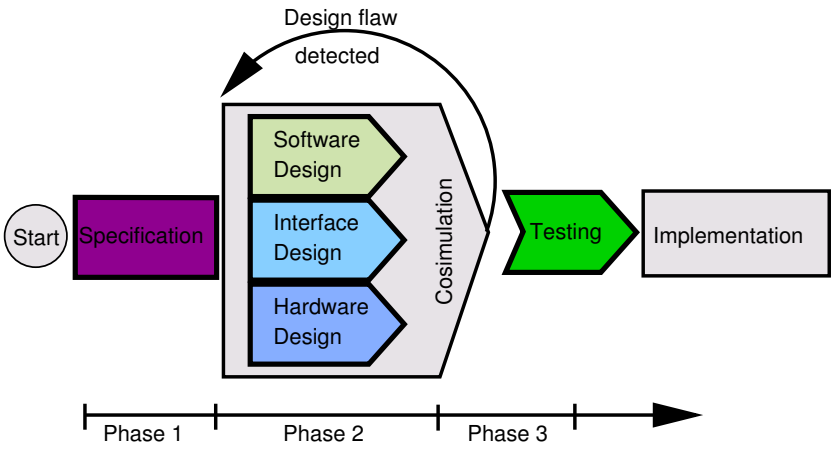

Figure 4.3: Parallel design in Co-design

Within the co-simulation (Chapter 7), the hardware and software components and their interaction through the interface can be tested and design alternatives can be evaluated. The virtual prototype is very flexible, as it exists only as a (source-code-) model. Source-level simulations help to relate errors and specification, which eases debugging significantly. The costs for a new version of a virtual prototype are negligible as it requires only setting up a new instance of the co-simulation model[8]. Obviously, there are also some difficulties and disadvantages related with virtual prototypes. The major disadvantage is the low performance of virtual prototypes. Since hardware, interface, and software are only simulated and the slowest component dominates the performance, it is usually orders of magnitude lower than of the implemented system. The other major back-draw of simulated systems in general is the accuracy of the model, i.e. the level of detail in which the important characteristics are represented in the virtual prototype. There is a clear trade-off between accuracy and simulation speed. The more accurate a simulation is, the slower it typically is. Hence, the challenge is to choose the right balance between accuracy and performance in a co-simulation environment. Due to great variety of systems and their requirements, there is no general optimal solution in this trade-off process. Instead, each class of embedded systems can be represented by a certain type of virtual prototype.

From the ideal executable model, the first implementation can be achieved through synthesis and compilation. Since real-time behaviour and the hardware/software interface was integrated early in the design process the implementation is already well tested which results in lower effort due to redesign and debugging.

4.4 Characteristics of embedded system

There are two major motivations for this section. One is to make clear that embedded systems fundamentally differ from other computational devices like desktop or mainframe computers. A suitable

[8] Depending on the mechanism chosen to implement the co-simulation, it is often required to modify or compile the source-code representation to prepare it for the co-simulation .

design method must address these characteristics to handle the special challenges closely connected with embedded systems design. The other motivation is to narrow the scope of the OOCOSIM method proposed in this work to a particular class of embedded systems.

There are many ways to characterise embedded systems. Giving many examples of representative embedded systems could be one approach but examples can easily be misinterpreted or be misleading. Another approach would be to describe them as the sum of their typical components. Due to the great diversity of embedded system, this would lead either to a detailed but virtual endless list or to a more general one, not being sufficiently distinguishing.

Hence, to determine the class of embedded systems that is in the scope of this thesis a characterisation from many different viewpoints is given in the following.

Specific task: As the term embedded already suggests, these systems are typically parts of larger systems. They are embedded into an environment to perform a specialised task. This gives the opportunity to optimise the system for this particular task. Special hardware components can be used to meet critical deadlines or the processors speed could be decreased to reduce the power consumption as long as the system is able to perform its task correctly. In contrast, a general-purpose computer is the execution platform for many and diverse applications, where the design target is being fast in the general case.

Tight non-functional constrains: Embedded systems are quite often hidden inside a complex mass product like an automobile or a cellular telephone. Due to the high volumes, the price for such a system often becomes critical for the profit of the selling company. For others, size or power-consumption (think of a pacemaker) can be the key factor for success. Therefore, the search for the optimal solution regarding many different constrains is of great importance in embedded system design. While the explicit modelling of such requirements is not part of this work, the method presented will allow exploring the design space. Different design decisions can be evaluated leading to a solution that fulfils the requirements imposed by the physical environment.

Applications with real-time constraints: Applications fall into one of the following categories. *Interactive Systems* that respond to external stimuli (often of a human user) as soon as they are ready. Most desktop applications fall into this category. *Transformational or data-streaming systems* process continuously certain blocks of input data into blocks of output data. Typical examples are telecommunication base stations or Ethernet cards in desktop PCs. The throughput of such a component is the dominant design goal. Many embedded systems are *reactive systems*, which means they react continuously on events from an external environment. Often several sources of events with different rates and priorities or criticalities exist. Such embedded system must be able to handle events or data properly; that is, correctly and in time.

Standard application designers often care for speed, whereas embedded system designer care for timing constraints. Moreover, typically embedded systems needs to be just fast enough to meet the real-time constrains, whereas the desktop computer should be as fast as possible for common desktop applications. If the correctness of embedded systems depends on meeting all deadlines, they are called *hard real-time systems*. An airbag controller for example, obviously is a hard real-time system; if it fires too late (or too early), the result is obviously catastrophic. This leads to the next characteristic.

Safety critical: Many electronic systems like airbags, ESP or ABS in modern automobiles or anti collision radars in planes are great safety improvements as long as they work correctly but may also cause an accident if an error occurs. Therefore robustness, gracefully degradation[9] and proven

[9] Gracefully degradation means that a failure of a component leaves the system in a reduced but safe situation. For example, a broken sensor in an airbag controller cannot cause the ignition of it.

correctness are important features. As stated above, correctness often means not only functional correctness but also keeping all critical real-time constrains.

Minimal user interaction: Most embedded systems operate almost autonomously. A human user may set the operational mode in the beginning but from then on the embedded system works in response to its physical environment.

Another important factor is the absence of a human observer. If the embedded system does not work correctly usually, there is no person to identify and act accordingly. A manual restart or a manual shutdown is possible in most standard applications but it is often impossible for embedded systems. The embedded system therefore often runs self-diagnosis tasks in parallel with the core functional tasks to identify erroneous system behaviour.

Heterogeneous components: As stated above embedded systems consist of hardware components, software components, and an interface for the internal communication. Systems strongly dominated by hardware or software, which could easily be modelled by hardware or software design flows are not in the scope of this work. The strength of the OOCOSIM approach is the ability to model, simulate, and implement heterogeneous systems.

The computational model and communication primitives in hardware and software models are fundamentally different. In digital hardware, truly parallel components communicate via electrical wires. Thus, the hardware is often modelled as a discrete event system where events on clocks or other signals trigger computations.

Software is essentially sequential; that is, there is a single thread of control in the software execution. Concurrent software components can be modelled by active or passive objects communicating via messages or shared objects. Typically, there are several active objects (task) in the application. A scheduler determines the order of execution; that is, the access to the processor resource according to a well-defined scheduling policy. This modelling style seems not appropriate for pure hardware systems since in general there are no such unique resources as the processor is for software. Modern system description languages such as SystemC (Section 3.2) try to come to a more homogeneous specification of hardware and software. However, since the fundamental differences remain, the approaches still are dominated by a single domain. SystemC for example is more an HDL than being appropriate for complex software models.

The interface is partially hardware and software. Obviously, the interface must be accessible for certain components in hardware and in software. Consequently, neither hardware nor software description languages can be appropriate to model the interface. An independent approach is required to handle interfaces in the design process. A detailed discussion and a proposed solution for this problem is presented in Chapter 6.

Complex reactive behaviour: Embedded control systems typically contain numerous sensors observing the environment permanently. Events occurring in the environment and registered by sensors need to trigger actuators within certain deadlines to achieve the intended correct real-time behaviour. Eventually asynchronous events can occur, which require immediately actions in the software, thus pre-empting the running activity. Parallel tasks might observe the validity of sensor values to detect broken components.

In many consumer products, more and more of their functional features are implemented by embedded systems. In the past, each functional component was implemented by a different controller. Nowadays, there is a need to integrate multiple tasks into one controller. While this simplifies the physical architecture (number of devices and their communication structure) in the product, it increases the complexity of the embedded software.

Hardware/software interface: Data and control flow between hardware and software is inherent to embedded systems. Data from the environment collected by sensors and internal state of the hard-

ware both are stored in special registers located in hardware. These data must be computed in the software part of the embedded application. The resulting control commands for the hardware must be propagated through the interface. To bridge the gap between hardware and software, the interface provides a bidirectional connection carrying data and control to the appropriate components.

Some events in the environment or in the hardware part require an immediate reaction of software part. These events are typically propagated by special communication channels implemented by interrupts. These asynchronous,[10] unidirectional channels are also regarded as part of the hardware/software interface.

4.5 Requirements for design methods

The characteristics of embedded systems mentioned above impose certain requirements on design methods that often differ from the design of software or hardware in general. These requirements ask for a different design methodology.

Some characteristics like the importance of real-time behaviour lead directly to special requirements for a design method of embedded control systems. Others, like for example reactiveness, have influence on various aspects of the method. The following will name the properties of a design method capable to develop the above-characterised systems. The OOCOSIM method described in the following chapter provides a design flow and a set of tools that aims to fulfil these requirements. Thus, this chapter also serves as a motivation and property-oriented description of the OOCOSIM method.

In Chapter 8 of this work, benchmarks are presented and discussed to proof that OOCOSIM in fact provides an approach, maintaining the hereafter presented requirements.

4.5.1 Seamless refinement

As described above embedded system design is a refinement process. During this refinement process, different formalisms can be employed. The process e.g. may start using a graphical notation like UML or a block diagram, which then must be translated into an executable specification to enable simulation and validation of the specification.

In the next phases, the initial simulation model will be refined into source-code for the compilation and synthesis of software, hardware, and interface components. To achieve a seamless design flow, these refinement steps must maintain two properties. First, the translation of one representation into another must be either automatic or following a constructive refinement strategy. Second, specified characteristics and properties should be propagated traceable into the lower level of abstraction.

While the first property avoids extra manual work and potential errors, the latter is particular crucial because the design flow is not always running straight. Eventually a requirement cannot be met at the actual level of abstraction and the designer has to return to an earlier design phase to correct a wrong design decision.

4.5.2 Executable heterogeneous specification

The heterogeneity of hardware and software components makes it necessary to use different languages to design them. These languages not only have a different syntax, they are often based on different computational models especially regarding their timing and communication model. To make testing of the complete embedded system possible already at specification level, a co-design method needs a co-simulation model that unifies the different models for hardware and software. In particular, it needs to synchronise the timing behaviour of hardware and software and provide a communication interface between them.

[10] These channels are called asynchronous because their information transfer does not follow the normal synchronisation schedule of ordinary data channels.

For the design of hardware, specialised languages like VHDL or Verilog define the de-facto standard in the industrial practice. Almost all hardware designers use these languages at different levels of abstraction to develop and test their hardware components. These languages provide an adequate syntax to describe hardware and a clear semantic for simulation and synthesis, which is supported by commercial tools.

For embedded software, a different paradigm is necessary. Certain software programming languages like Ada95, C/C++ or Java [Gio98] can be used to specify embedded software[11]. While C and Java are widely used for software development in general, they lack of some important properties necessary for complex embedded control systems. Java has no predictable real-time behaviour and no low-level programming interface to directly access to memory or interrupts. C or C++ has no notion of concurrency or time. These deficits can be reduced by extensions but at the price of portability.

Testing the embedded system model in a co-simulation is not enough to guarantee its safety in a critical application. It is an early and very important component in the verification process to achieve a reliable system. It should be accompanied by formal methods and operational tests of the physical system when it is build.

The need to integrate different people into a design project has already been mentioned above. Since people do not often share a 'common language', it requires understandable communication formalisms to bridge their semantic gaps. Each project partner and the customer may have an individual idea of how the system supposes to behave in particular situations. A simulation model showing the dynamic behaviour of the system can be understood by all project partners as long as it shows the relevant properties. An executable model can thus also be seen as a valuable part of the documentation. The meaning of user requirements can be demonstrated or confirmed best in an executable model.

4.5.3 Exploring the design space

The intended behaviour of embedded systems can be achieved by many different implementations. Many parameters need to be determined before the implementation can take place. Consequently, the design process can be seen as the determination of the best position in a multi-dimensional space - the so-called *design space*. Dimensions of this space can be for example memory consumption, CPU-speed, or ASIC-area. Every implementation is then a point in this design space as depicted in Figure 4.4. Other dimensions are for example the ASIC respectively FPGA-technology or even physical parameters like the size of the embedded system or the type of a cooling device.

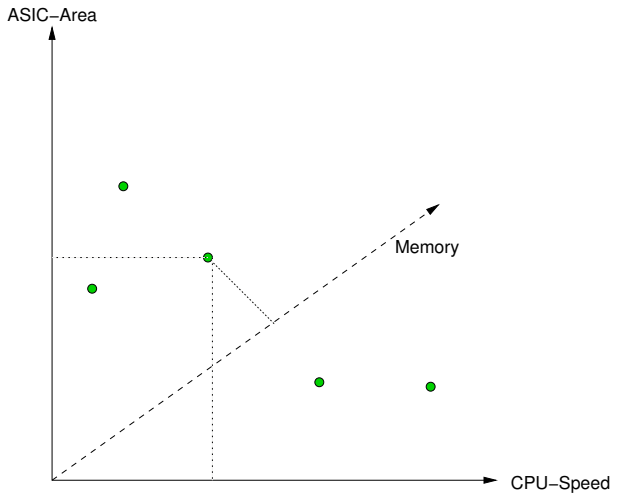

Figure 4.4: The design space

At the beginning of the design process, the design space of possible solutions is very large. By stepwise refinement, more and more alternatives are eliminated and thus the available design space is reduced.

The embedded system designer wants to explore the design space in its multiple dimensions to find the best alternative. To allow the exploration and the possible revision of choices, the ideal method should be flexible enough to change every aspect of the system specification without corrupting the

[11] For many small systems even assembler is still in use. Since this work focuses on complex system, this approach appears not suitable.

consistency of the design.

Hardware/Software partitioning

A key decision for each of the systems design is the distribution of functionality between hardware and software. The process to determine this central aspect of the architecture is called *hardware/-software partitioning*. The general partitioning problem is very difficult. It has influence on many characteristics of the system, like for example the timing behaviour or the power consumption. Partitioning also has influence on non-technical factors such as the flexibility of the design, the overall design effort, and the system price.

A common rule over the thumb tells the designer to implement performance critical parts in hardware while control dominated parts or components handling larger amount of data should better go into software. Unfortunately, this simple rule is much too coarse to guide the partitioning sufficiently because the partitioning has several side effects on other system properties.

The software components typically share the one processor, which has limited computation power. Since the computation power mainly depends on the type of CPU, it does not scale steadily but in discrete steps. As long as the computing power of a particular CPU is sufficient, the system cost remains constant[12]. Only if the partitioning requires more than the available processing power sometimes not only the CPU but also whole the implementation architecture must be changed because the faster CPU requires a different operating system, bus architecture or peripherals.

For the design flow, this means that the exact implementation architecture should be determined as late as possible. Since the real-time behaviour should be modelled and checked early in the design flow, (Section 4.5.5) the methods need to provide suitable abstractions for real-time.

4.5.4 Sufficient simulation performance

Relative (simulation) performance in this work is defined as the ratio between the model time and the simulation time[13]. Since a timed co-simulation requires a synchronisation of all components, the simulation performance is often dominated by the slowest component in the simulation environment.

The amount of model time that must be simulated depends on the granularity at which the system will be observed. At system level, where the focus lies on interaction of coarse-grained components, longer periods need to be simulated. In some cases, as for example the portal crane described in Chapter 8, the controlled process may last minutes or even hours. In other application domains, the embedded system fulfils its task within milliseconds. However, for the majority of applications the simulated period interesting at system level ranges from seconds to a few minutes.

Unfortunately high simulation speed cannot be achieved in a very detailed simulation. For the early design steps like partitioning and design space exploration it is often sufficient to simulate a rather abstract model of the system, while in later phases the detailed structure and behaviour of each system component is of great importance.

Sufficient simulation performance is a critical requirement especially for validation and architecture exploration at system level. The ideal simulation system is able to support many different levels of abstraction resulting in appropriate simulation speeds. This would allow checking the systems behaviour for a significant period of real-time in early phases of the design process. With ongoing refinement, simulation allows observing detailed behaviour, for example the handshaking protocol of two components in a smaller period.

4.5.5 Early integration of real-time behaviour

Since the real-time behaviour is critical for a large class of embedded systems, the modelling of real-time behaviour must already begin at specification level.

[12] Please note that this assumption takes only the price for the CPU into account. Other factors, such as the required memory and the dynamic power consumption are neglected here.

[13] *Model time* denotes the time passed for the embedded system in the simulated environment, whereas *simulation time* denotes the time required to run the simulation.

Naturally, at this level only application-driven requirements can be specified. Throughout the design flow with more and more details about the implementation architecture being specified, these requirements can be refined accordingly. With the definition of the implementation platform, analytic estimations of worst-case execution times and latencies are possible. With these data available, the real-time requirements can be checked against the real-time behaviour of the implementation.

Disobeying the real-time requirements in the early phases of design will typically lead to implementations, which are either inefficient or even not implementable under the given (cost) constraints. Therefore, embedded systems must be designed based on a sound method that allows to reason about their behaviour under realistic circumstances including their real-time properties [Ame01].

4.5.6 Modelling hardware/software interfaces

A design methodology suitable for embedded systems shall put a special emphasis on those aspects of the targeted systems that are difficult to design. Special abstractions should be provided to overcome the deficits of for example the implementation languages.

Hardware/software interfaces are crucial for the correctness of embedded systems. They belong neither fully to hardware nor to software. Consequently, implementation languages for hardware and software are not suitable as they lack general abstractions to describe interfaces. Therefore, the methodology must support these components by providing additional abstractions. Such abstraction must not only describe interfaces in a comfortable way but also need a direct mapping into an implementation to maintain a seamless design flow.

4.5.7 Mastering complexity

Complexity is an important property to decide about the suitability of a design method. Some systems have a very simple structure and the application is rather simple. For such designs, it would be inappropriate to use a high-level co-design method addressing complex systems.

For complex systems however, an unsuitable method could lead to the total failure of the design project. The work at hand addresses the latter class of systems. The development of complex embedded systems often requires many designers to work together in different teams. Such complexity in the application and the team structure requires a well-organised process. Therefore, it is indispensable to divide the system into manageable portions for sub-teams or even single designers. These portions must have well-defined interfaces with a clearly specified behaviour while the internal implementation is hidden from the outside. Obviously, this matches perfectly with the object-oriented paradigm.

With rising complexity, low-level abstractions become more and more inappropriate to design reliable, complex embedded systems. A lesson that can be learned from the conventional software design is that higher abstraction, encapsulation, and separation of concerns help mastering complexity. For a seamless design flow it is however also required that the methodology allows low-level behaviour to be specified.

Consequently, co-design methodologies use modern object-oriented methods and provide appropriate abstractions for the essential concepts necessary in embedded system design. The method must in particular support abstractions for the real-time behaviour and the hardware/software communication throughout the full design process.

The design principle *separation of concern* here means that the modelling of functionality, hardware/software communication, and real-time behaviour should be separated. Obviously, these aspects in conjunction define the behaviour of the embedded system but their modelling and validation should be separated where possible.

Only if these prerequisites are fulfilled the efficient reuse of design components becomes possible. Reuse is probably a powerful concept to reduce the design effort. However, reuse is only effective if the components behaviour and interface is fully defined and understandable without looking into the implementation.

Complex systems are often developed in an incremental approach, i.e. the system specification starts with a core description that is refined to a certain level. Later the core is augmented by additional functional components. In the course of design-space exploration, some design decisions may be revised. This often requires returning to an earlier phase of the design flow. Consequently, the method should be robust to changes; that is, local changes should not affect the overall stability and consistency of the system description.

4.6 Recap

At first, this chapter introduced the co-design approach in contrast to the classical sequential design flow. It became obvious that co-design – while not being easy to implement – provides many advantages over the classical design flow. The parallel design of hardware, software, and the interface in-between can reduce the overall design time by early detection of design mistakes and errors. Due to the holistic view on the embedded system, new design alternatives can easier be found and evaluated.

The characterisation of embedded systems and the deduced requirements on design methods lead to the conclusion that a unified method appropriate for arbitrary embedded systems is very unlikely. Consequently, the work at hand will concentrate on a methodology for the class of reactive embedded systems containing significant hardware components and a multitasking software part.

5 OOCOSIM Design Method

In the previous chapter, a characterisation of embedded systems and a list of requirements for appropriate design methods were given. This and the following two chapters will describe the OOCOSIM design method for embedded systems [OSN99, OS99] aiming to fulfil these requirements.

After some general remarks on object-oriented design in the context of embedded systems, an overview of the design process of the OOCOSIM method (Figure 5.2) will be given. Furthermore, the specification and partitioning based on HRT-HOOD+ will be described. The following chapters describe the design phases after the partitioning more in depth. Examples will illustrate the application of certain aspects of OOCOSIM where appropriate.

5.1 Object-orientation in embedded system design

Object-orientation as a design principle was first mentioned by Dahl [DMN67] in 1967. Since then, due to significant work in research and industry to support this approach with languages and tools, it became the most popular design approach in software development today.

While it cannot solve all problems in software engineering [BS02], it has proven to be of great advantage in the development of large and complex software systems. In Section 2.9 a brief introduction of the object-oriented design paradigm has been given.

As described in Section 4.4, embedded systems differ in many aspects form standard software applications. These differences require a (slightly) different interpretation of the object-oriented paradigm when applied to this domain. The following will describe how the object-oriented approach can be applied to embedded system design and show up the similarities and differences to standard object-oriented design techniques.

An embedded system can be seen as a set of collaborating components or entities as abstractly depicted in Figure 5.1. Each component contains some data and contributes certain internal and external services to fulfil the overall task. Thus, these components fit very well with the abstract concept of objects in general. Components in embedded systems are usually static, i.e. the objects are not created during runtime. For hardware components, this is obvious and for the software, it is at least true for real-time systems[1]. Thus, in HRT-HOOD and consequently in OOCOSIM, objects rather than classes are regarded as first-level design entities.

For their collaboration, components need to communicate; that is, they need to exchange data or to synchronise their behaviour. Object-orientation provides therefore the concept of interfaces[2]. Restricted by the interface, an object allows only a well-defined set of services to be used outside the object. The functional behaviour of the services or methods is specified and visible only within the object. This allows hiding the implementation to the external system.

The exchange of data and the synchronisation can be modelled by sharing of data objects or by method invocations. At this rather abstract level, components in embedded systems resemble exactly the general concept of objects.

Similar to large software systems, complexity is a major challenge for embedded system design. Object-orientation masters complexity mainly by dividing the design hierarchically into simpler, collaborating parts.

[1] The dynamic creation of objects at runtime would make real-time analysis significantly more difficult.

[2] The term interface refers to the general concept of interfaces defining the outside view of objects rather than the specific interface in programming languages such as Java.

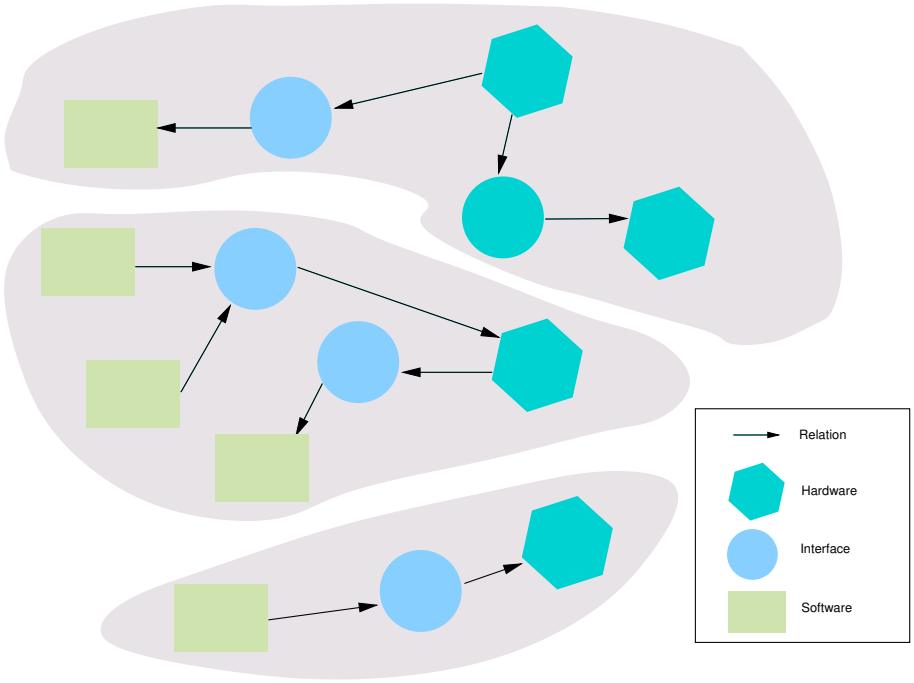

Figure 5.1: Collaborating objects

The OOCOSIM approach re-uses many concepts from HRT-HOOD. Therefore, it differs (again) from other object-oriented approaches like for example UML where the design is based on a class hierarchy. In contrast to that, in OOCOSIM the objects topology and their collaboration describe the structure of the system. A concept similar to a class is present in the definition of the objects by the so-called *provided interface*, which defines the signature of an object.

Scheduling and synchronisation of concurrent tasks are important factors in the design process for reactive embedded systems. Therefore, OOCOSIM provides means to model and validate these properties throughout the entire flow. Objects in OOCOSIM carry attributes that describe their non-functional properties. These attributes are propagated from the specification into the co-simulation behaviour and into the implementation units.

Object-oriented modelling of embedded systems in OOCOSIM is based on the idea that objects represent components of the design. The objects tie together all relevant properties of the components. It is important to note, that OOCOSIM provides the **means** to describe and analyse an embedded system rather than **automates** the designer's decisions. It provides notations like HRT-HOOD+ and ComiX to model the systems architecture or hardware/software interface. Tools, like the co-simulation framework or DESHICO interface designer environment enable the transformation or analysis of the design. However, it remains the designer's duty to create the system model and to decide between design alternatives, such as for example different hardware/software partitionings.

5.2 Overview on the general flow

In the scope of this thesis, the design flow starts after the initial specification of the system to be built. It may be written in a natural language or given as an executable, functional model in C++. This specification is typically not well structured and rather informal. Hence, it can be taken (at best) as a functional reference, often called *golden model*, for the final implementation.

In the first step, the informal specification of the intended system must be translated into a struc-

tured system-level specification that enables reasoning about the complete system. In OOCOSIM, HRT-HOOD+ provides means to model the system as a collection of well-defined objects. The process of refinement allows to successively dividing the system into smaller sub-objects bringing the specification closer to its implementation. Partitioning; that is, the allocation of objects to hardware and software, is supported in OOCOSIM by specialised objects reflecting the characteristics of hardware, software, and the hardware/software interface.

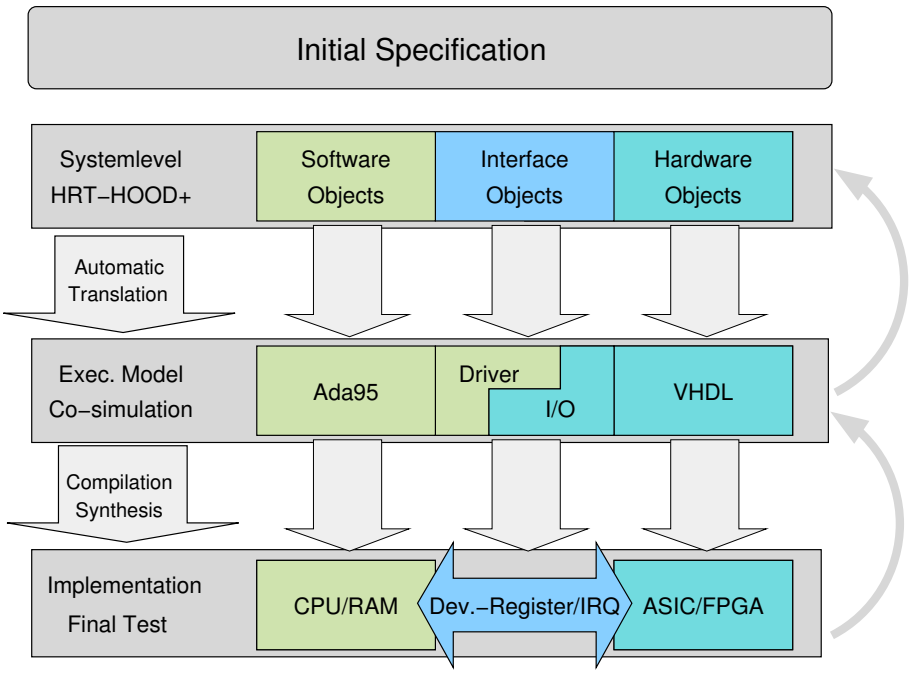

Figure 5.2: OOCOSIM design flow

With mostly automatic tools the objects of this level are translate into executable models. Hardware and software objects are translated into corresponding Ada95 and VHDL descriptions while the interfacing objects must be handled differently. Since they comprise a driver component (software) and an I/O device-part, interfacing objects must be split into software and hardware parts for the further design. Hardware, software, and the communication via the interface can then be validated including their real-time constraints in the co-simulation framework that is part of OOCOSIM. Eventually some details of the hardware and software model will be worked out and tested at source-code level before compilation and synthesis lead to the implementation of the embedded system.

Due to the encapsulation of design components within the object-oriented development process of OOCOSIM, partial changes to the design can be handled efficiently. If for example new components are added at system-level, changes can be propagated easily into later phases. The analysis of real-time requirements and the validation of the extended systems behaviour must reflect these new components. However, due to the hierarchical character of the specification, the scope that is affected by changes can often be limited to a certain subsystem. Since the transformations between the levels are mostly automatic and properties are propagated automatically, the consistency between the different levels can be achieved easily.

5.3 Specification in HRT-HOOD+

Complex embedded systems very often contain numerous distinct components making it very difficult to design the system based on an unstructured description. Therefore, it is essential for any system-level method to provide means to decompose the system into simpler sub-system and to define the relationship between the components.

As the name **Hierarchical** Object-Oriented Design (HOOD) already suggests, refinement (in this method) can be achieved by hierarchical decomposition of objects. This principle holds also for the newly introduced objects types (see below). Furthermore, similar to the original method it provides guidelines on how objects can be refined. These guidelines refer to legal decompositions as well as to the way they can interact with each other. For some objects, the definition of certain attributes is mandatory for a valid design while it is optional for others. Attributes often must obey certain constraints with respect to other attributes defined in the system. The ultimate goal of these guidelines is to preserve the consistency of the design.

The following will describe the HRT-HOOD+ method in a top-down manner. First, the top-level object types will be introduced and then the refinement process using lower-level objects will be explained.

5.3.1 The Root Object and its environment

The design process in HRT-HOOD+ starts with exactly one ROOT OBJECT (denoted with an R in the graphical symbol as depicted in Figure 5.3) that represents the embedded system plus a number of *Environment Objects* reflecting the environment. Environment Objects are already defined for HRT-HOOD for a very similar modelling aspect. The ROOT OBJECT capsules the entire embedded system

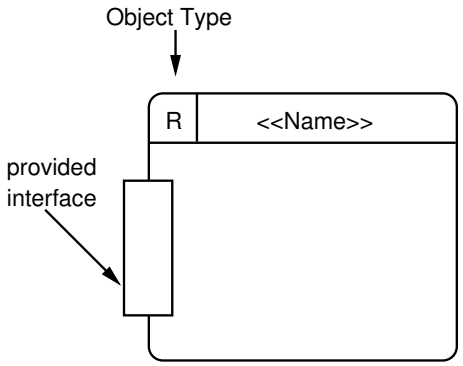

Figure 5.3: ROOT OBJECT symbol

to be modelled from its environment. It serves as a container object and provides no functionality except those defined in its sub-objects. The ROOT OBJECT can have a *provided interface* which is part of the system; that is, it must be implemented to allow the integration of the embedded system in its environment.

The *required interface* of the ROOT OBJECT denotes the services or the interface the environment must provide for the embedded system. Please note that the environment may contain other electronic devices that need to communicate with the system under design. Furthermore, the required interface can be used to define requirements on the simulated environment of the embedded system. It typically includes access functions to external sensors or actuators. Figure 5.4 depicts a simple initial system containing the ROOT OBJECT and one Environment Object.

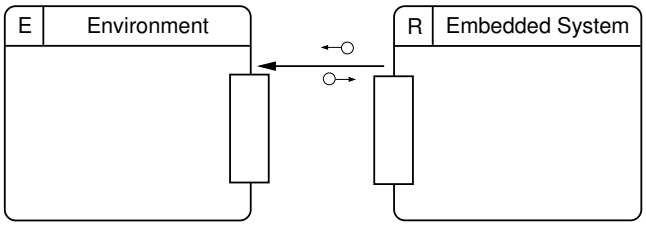

Figure 5.4: Simple initial system

The ROOT OBJECT defines the following attributes:

- *Name*: The name of the embedded system. All objects in a HRT-HOOD+ must define this attribute. Hence, it will not be explicitly mentioned in future. It provides an unambiguous reference to each object that is for example required to invoke methods of this object.
- $Priority_{max}, Priority_{min}$: These attributes define the range of scheduling priorities, that is the highest and the lowest priority, available to this system. This attribute applies only to the software part of the system. Either it can be defined by obligations of the application or by resources, the target platform can provide. In case the software execution processor is already determined in advance, it is defined as a constraint from the processor.

5.3.2 System Objects

Often the system comprises strongly related partitions of objects performing a particular sub-functionality. These sub-functionalities are typically implemented by a highly coherent set of implementation units. Such a set of units from now on will be called *logical partition* of the system. Logical partitions are based on a functional view of the system and thus should not be confused with the architectural partitions between hardware and software. If multiple designers or even multiple teams are contributing to the development of large systems, logical partitions can serve well as separated design tasks for the participants.

OOCOSIM introduces so-called SYSTEM OBJECTS[3] in HRT-HOOD+ to provide abstractions for the above-mentioned logical partitions. They, similar to ROOT OBJECTS[4], serve only as container objects, as they provide only an interface to the functionality defined by their sub-objects.

As a first step of refinement, the ROOT OBJECT is decomposed hierarchically into SYSTEM OBJECTS[5]. The introduction of SYSTEM OBJECTS in HRT-HOOD+ enables, besides functional decomposition, an incremental development process for complex systems. The design may start with the core functionality containing only essential subsystems. Then refinement will eventually lead to a stable core model. The system can then be augmented by adding new SYSTEM OBJECTS. In an object-oriented method like OOCOSIM, at first sight inheritance appears to be the natural mechanism to augment the systems functionality. A possibility would be to derive the new system by class inheritance from the ROOT OBJECT or the SYSTEM OBJECTS. However, due to the in general unsolved problem of inheritance anomalies [SN98] it is (in OOCOSIM) not allowed to apply inheritance to active objects. Instead, composition through aggregation is supported.

SYSTEM OBJECTS have, like all HRT-HOOD+ objects, a provided and a required interface to define the services needed and requirements for external SYSTEM OBJECTS. In the graphical notation similar

[3] Those readers familiar with HOOD should note that SYSTEM OBJECTS form an abstraction of the HOOD object types Active and Passive.

[4] In fact, ROOT OBJECTS can also be seen as a specialisation of SYSTEM OBJECTS. The main difference is that there is (by definition) exactly one ROOT OBJECT at the top-level of the specification. Consequently, the priority attributes underlie no inherent restrictions.

[5] For simple systems the ROOT OBJECT could also immediately be divided into specialised objects belonging to one of the implementation domains hardware, software, or interface objects.

to all predefined HRT-HOOD object types, related SYSTEM OBJECTS are connected with an edge. The edges may be annotated with data-flow information defining its type and its direction.

To support the analysis of real-time properties, each SYSTEM OBJECT must define its *priority range*. The priority range defines the possible priorities software objects can take in a decomposition of an object. The following rule formalises that the priority (range) of each inner object may take only priorities within the priority range of the decomposed object:

Rule 5.1 (Valid Priority Ranges) *Let* $S = s_1, .., s_n, n \in \mathbb{N}^+$ *be a finite set of objects, a refinement of the object* r*. Let* $P(o) = [p_{min}, .., p_{max}]$ *denote the priority interval of an object* o*:*
S is valid $\Rightarrow \forall s_i,\ 1 \leq i \leq n : P(s_i) \sqsubseteq P(r)$

For a valid HRT-HOOD+ specification it is furthermore required that the priority interval of all SYSTEM OBJECTS are disjoint. More formally:

Rule 5.2 (Disjoint Priority Intervals) *Let* S *and* P *as defined in Rule 5.1:*
S is valid $\Rightarrow \forall s_i, s_j$ *with* $1 \leq i, j\ \leq n,\ i \neq j : P(s_i) \sqcap P(s_j) = []$.

Please note that since priority intervals are complete, this rule imposes an order on the set of SYSTEM OBJECTS. This order reflects the importance of the functional partitions within an embedded system. Hence, the design should start with the most important functional partition and define the respective SYSTEM OBJECT and its priority interval starting with the maximal available priority. Then the designer may add successively SYSTEM OBJECT with descending importance.

In order to avoid side effects (unintended coupling) SYSTEM OBJECTS must obey certain constraints:

- SYSTEM OBJECTS must not share any objects expect for COMMUNICATION OBJECTS resulting from refinements of data flows between SYSTEM OBJECTS.
- Communication between SYSTEM OBJECTS may only use *Asynchronous Execution Request (ASR)*, hence only non-blocking method calls. Otherwise, the tight coupling and synchronisation between SYSTEM OBJECTS would make the design and in particular the real-time analysis very difficult[6].

The restrictions on the sharing of objects, disjoint priority ranges, and on the synchronisation prevent strong interference between SYSTEM OBJECTS.

Obviously, SYSTEM OBJECTS can be further decomposed hierarchically into lower level SYSTEM OBJECTS. However, as SYSTEM OBJECTS should represent considerable large subsystems it seems advisable to stop this refinement process very early.

5.3.3 Refinement of System Objects

SYSTEM OBJECTS are typically refined into combinations of PASSIVE OBJECTS and ACTIVE OBJECTS as sketched in Figure 5.5. This refinement allows defining a finer grained systems topology and serves as a first specialisation of objects. PASSIVE OBJECTS specify components, which provide only services that is they contain no thread of control. Note that ACTIVE OBJECTS may be refined into sets of arbitrary object types while PASSIVE OBJECTS may only contain PASSIVE OBJECTS. Moreover, ACTIVE OBJECTS may use PASSIVE OBJECTS but not vis versa.

5.3.4 Hardware/software partitioning

The classification of components as being active and passive makes specification about their architectural distribution over the implementation platform. The allocation of the objects to hardware

[6]The calculation of a blocking time in such a tightly coupled system would require the analysis of the entire system. Every change made in one sub-system would invalidate the static analysis of the complete system.

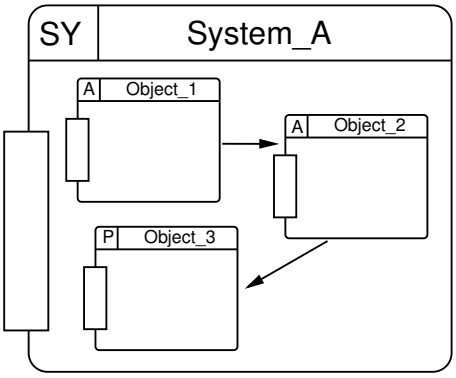

Figure 5.5: System Object containing objects.

and software - the so-called *partitioning* is thus the next logical step. To provide means to model the communication between the partitions, COMMUNICATION OBJECTS must be introduced. In a HRT-HOOD+ design these objects results from the refinement of data-flow and event-flow relations between functional objects assigned to different partitions, i.e. some allocated to hardware and some to software.

The HRT-HOOD+ method contains all object types from the HRT-HOOD method briefly described in Section 2.4. Since most of the terminal HRT-HOOD object types (Cyclic, Sporadic, and Protected Objects) are applicable only in software systems, HRT-HOOD+ adds two domains or classes of object types - namely the *communication domain* and the *hardware domain*. These new object types will be discussed in Section 5.3.6 and Section 5.3.10. For now it should be sufficient to understand that these objects allow for the specification of software, hardware, and interface components in the HRT-HOOD+ model.

To achieve a seamless design flow it is important to provide an implementation for the terminal objects in a design. The mapping of the different object-types to an implementable specification (also called implementation units) is discussed in the following sections.

5.3.5 HRT-HOOD software objects

HRT-HOOD provides an excellent method for the design of hard real-time software. Burns and Welling describe in [BW95] this method and its application in detail. With further refinement, software objects are specialised into terminal objects, such as Cyclic, Sporadic, Passive, and Protected Objects.

The challenge for the OOCOSIM methodology was not to enrich the software design method of HRT-HOOD but to find a sound modelling technique for a proper coupling with the hardware domain required for hardware/software co-design. The COMMUNICATION OBJECTS described in the following sections provide this coupling.

5.3.6 Communication Objects

Loose coupling of highly cohesive components is a desired characteristic of an embedded system and is indeed a prime claim for object-orientation. Modelling the interface as integrated parts of hardware and software (in contrast to separate entities) would break the capsule of hardware and software objects.

A COMMUNICATION OBJECT hides all implementation details of an interface component to the outside. It provides only a method interface to the other objects and has no required interface. Inside it carries all attributes belonging to this communication facility. These attributes determine the characteristics if the object like the type, the layout, or the address in memory. The major

difference between COMMUNICATION OBJECTS and other design objects is related to their twofold implementation. Since communication objects must be accessible by hardware and software their implementation requires two implementations - one in software and the other in hardware.

In order to achieve a formally analysable specification in the original HRT-HOOD method objects do not exchange data by means of rendezvous[7]. Instead, PROTECTED OBJECTS and PASSIVE OBJECTS provide means to access shared data. However, since these object types are defined only in the software domain, they cannot be accessed by hardware. Consequently, the modelling of hardware/software communication in OOCOSIM requires an extension to the HRT-HOOD method. To model shared memory access for hardware and software, specific COMMUNICATION OBJECTS, namely MEMORY OBJECTS and ASYNCHRONOUS MEMORY OBJECTS, are added.

Asynchronous external events, typically implemented by means of interrupts, are modelled in HRT-HOOD by Sporadic Objects in conjunction with ASER_BY_IT (Asynchronous Execution Request By Interrupt). These objects contain exactly one method representing the interrupt handler in software activated by the external event. Sporadic Objects in HRT-HOOD defining the minimum arrival time of the interrupt reflect only the software part of the system. HRT-HOOD+ defines two new specific object types to handle interrupt driven communication, combining the hardware and the software related part.

The COMMUNICATION OBJECTS are the major improvement HRT-HOOD+ brings in comparison to HRT-HOOD. They literally bridge the gap between the hardware and the software partition. They provide an abstract method interface to the physical layer between hardware and software. The COMMUNICATION OBJECTS allows exchanging arbitrary typed data and provides a limited asynchronous event mechanism between hardware and software. HRT-HOOD+ provides three concrete classes of communication objects derived from a common abstract[8] class called COMMUNICATION OBJECTS as depicted in Figure 5.6. The three concrete classes are the following.

- MEMORY OBJECTS model the data-flow between hardware and software based on shared memory. They provide the core mechanism to communicate across the hardware/software boundary.

- ASYNCHRONOUS SIGNALS model the flow of one asynchronous event created in hardware and handled in software. In the implementation model, ASYNCHRONOUS SIGNALS translate into hardware interrupts and attached interrupt service routines in software. They replace the above-mentioned Sporadic Object with an ASER_BY_IT execution constraint.

- ASYNCHRONOUS MEMORY OBJECTS also model asynchronous events created in hardware. In contrast to ASYNCHRONOUS SIGNALS these objects carry a data field to transfer data with the event. This kind of modelling is often referred to as *interrupt driven I/O*. It allows transferring data from hardware to software and enabling the hardware to signal that new data is available.

For each class a graphical and a textual view are defined. The graphical view aims at representing the components of the system under design in comprehensible way at system level. The following subsections will describe each object type including its attributes visible for this design view.

The textual view aims at describing the communication entities visible to a single object. It contains all attributes (including the ones from the graphical view) required to fully define the properties of each communication object. The textual view is based on the ComiX language that will be introduced in Section 6.3.

The first step to model communication in a HRT-HOOD+ design is to identify the information flow between the partitions; that is, between objects allocated to hardware and objects allocated to software. The COMMUNICATION OBJECTS provide the most abstract representation of an information

[7] A rendezvous in software describes a synchronised communication; that is, the communication between caller and callee can proceed only when both are ready. As this implies a potentially unbound blocking behaviour it is forbidden in hard real-time systems.

[8] Abstract here denotes the fact that it is not possible to derive an implementation unit from such an object. It can however be used to model abstractly the information flow.

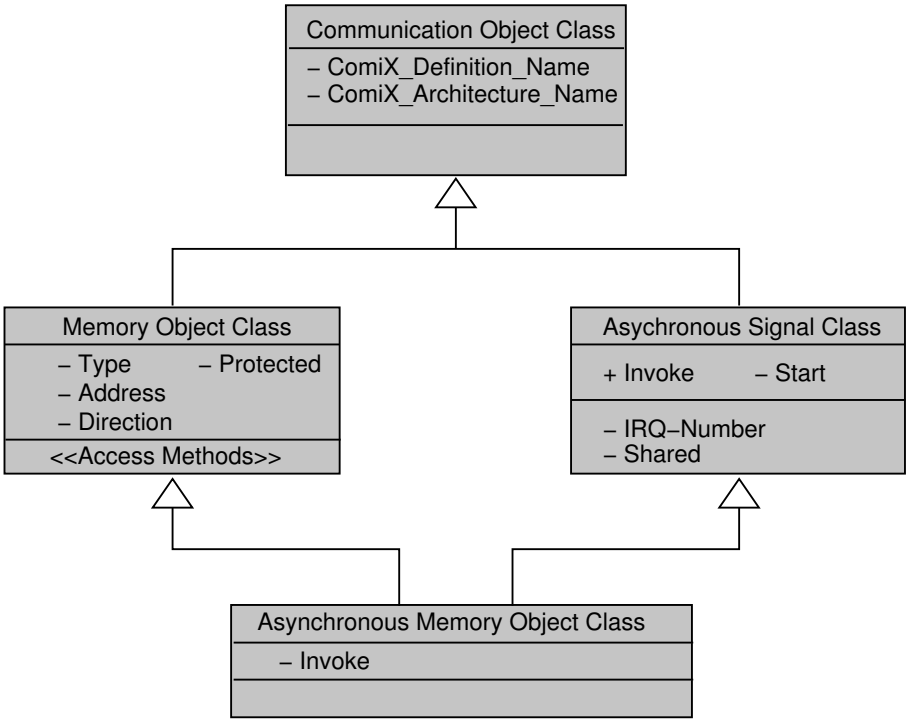

Figure 5.6: Communication class hierarchy in UML like notation.

flow as they are the root class for all interface objects within a HRT-HOOD+ specification. It puts no constraints on its implementation or any data being transferred.

Each communication object must define the attribute ComiX_Architecture and ComiX_Definition. These attribute link the objects to the object independent layers of the ComiX specification. These layers define for examples data types or the available communication resources. Each ComiX specification must be identified by a unique name. The attribute ComiX_Definition serves as a reference to this name. Every object is furthermore part of a so-called *architecture*. The attribute ComiX_Architecture is a reference to this architecture that must be defined in the respective ComiX_Definition.

COMMUNICATION OBJECTS must be refined into concrete objects to enable an efficient generation of implementation units for the hardware/software interface.

5.3.7 Memory Objects

In situations where hardware and software exchange data with no constraints on the synchronisation, MEMORY OBJECTS should be applied. This is a common case in designs where for example the software needs to poll sensor values or it sends commands to the hardware.

For MEMORY OBJECTS, the following attributes are defined:

- *Type*: The data type of the information being exchanged. This refers to a type name defined in the ComiX specification identified by the ComiX _Definition attribute. The type attribute is mandatory.

- *Direction*: This attribute defines the direction of the data flow. It can be immediately derived from the data-flow denoted in the HRT-HOOD+ model.

- *Protected*: To prevent unintended overwriting of data, mutual exclusive write access might be useful for MEMORY OBJECTS. This attribute defines whether this is required for the objects implementation.

MEMORY OBJECTS, as depicted in Figure 5.7, provide a method interface to access the data structure allocated in the memory mapped I/O area. The data structure can be of a simple type or be an aggregation of components (record). The methods allow accessing each component and the data structure as a whole. The implementation units in hardware and software will provide, depending on the *Direction* attribute, the methods as defined in Table 5.1.

Direction	Hardware Interface	Software Interface
HW to SW	Read & Write methods	Read methods
SW to HW	Read methods	Read & Write methods
Bidirectional	Read & Write methods	Read & Write methods

Table 5.1: Provided method interface of MEMORY OBJECTS

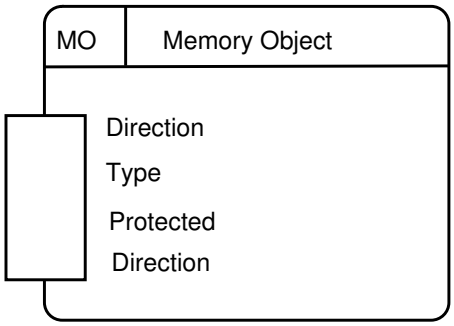

Figure 5.7: Memory Object (graphical)

In the implementation units, the attribute *Protected* determines whether mutual exclusive write access has to be guaranteed to this MEMORY OBJECTS. In the software implementation unit, this can be achieved by wrapping the data structure into an Ada95 Protected Object. In hardware, a semaphore can be used to provide a similar functionality.

In general, it is not recommended to use the Bidirectional mode. The complexity, introduced by multiple, unsynchronised write accesses in hardware and software, often leads to erroneous behaviour. In particular, it is not allowed to enable the Protected attribute together with the Bidirectional mode as it is not possible to implement this in an efficient way.

Many attributes are noted only in the textual representation based on ComiX. In particular, attributes like the size, the access mode (atomic/volatile), or physical address at where the MEMORY OBJECT will be allocated in the memory provide only implementation details. Hence, they are not included in the graphical notation.

5.3.8 Asynchronous Signals

The ASYNCHRONOUS SIGNAL as depicted in Figure 5.8 encapsulates the activation of a physical interrupt and its handling through software. It defines only the *Invoke* method in its provided method interface. With this method, the hardware can activate the interrupt. The software is notified by the handler method *Start* which is not visible to other objects[9]. The software aspect is defined exactly like in the original HRT-HOOD method as a Sporadic Object that is enclosed in the ASYNCHRONOUS SIGNAL. The ASYNCHRONOUS SIGNAL object can be also seen as a hardware/software pattern, where the outer ASYNCHRONOUS SIGNAL reflects the hardware aspect and the inner sporadic object represents the software aspect.

[9]In the implementation this method is typically registered in the operating system as an interrupt handler

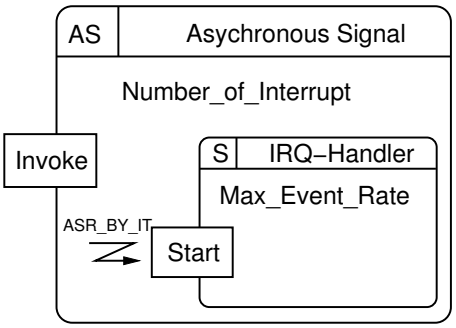

Figure 5.8: ASYNCHRONOUS SIGNAL symbol (refined)

5.3.9 Asynchronous Memory Objects

ASYNCHRONOUS MEMORY OBJECTS provide the functionality of asynchronous events combined with the access to a data structure in the memory-mapped I/O area. Thus, they combine the functionality of MEMORY OBJECTS and ASYNCHRONOUS SIGNALS.

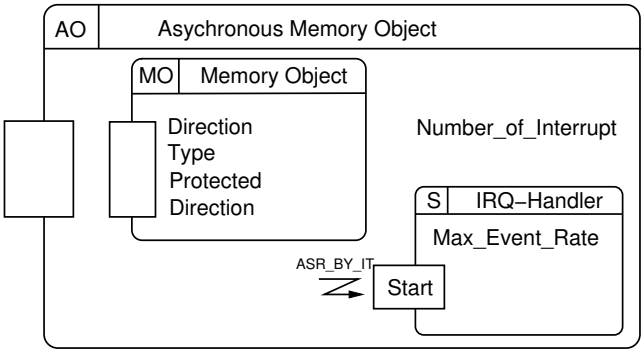

Figure 5.9: Asynchronous Memory Object (refined)

ASYNCHRONOUS SIGNALS inherit all attributes from the MEMORY OBJECT class and the ASYNCHRONOUS SIGNAL class. They as well inherit the provided interface of the MEMORY OBJECT but hide the invoke message because the invoke method is implicitly started after each hardware access.

In an ASYNCHRONOUS SIGNAL every access of the hardware to this object also activates the associated interrupt. A handler method defined in the Sporadic Object can be used (depending on the data flow direction) to either put a new data packet into the memory object or receive the newly generated data from the hardware. In a typical producer/consumer situation where data packets are produced in either hardware or software, ASYNCHRONOUS MEMORY OBJECTS can handle the synchronisation. For example the software, as the producer, will generate a new data packet and store it into the data-structure. After each hardware access, the interrupt handler will trigger the generation of the next data-packet.

5.3.10 Hardware objects

In HRT-HOOD, hardware components are modelled by environment objects. This will remain mostly unchanged in HRT-HOOD+ for the following reasons. One important objective of HRT-HOOD was to

model the synchronisation between the software objects in an analysable way. There is no need to do this for hardware objects since they run in parallel in contrast to concurrency of the software objects.

The other reason is the lack of an adequate implementation language for hardware objects. Synthesisable object-oriented HDLs like OSSS (ODETTE System Synthesis Subset, Section 3.2) have not yet reached a mature state. Therefore, hardware components cannot be specified in an object-oriented language, while maintaining an automatic synthesis, which is required for a seamless design flow. Since state-of-the-art HDLs do not support a method interface for entities/architectures hardware components can be modelled only in a black box view. The only way to integrate them into the system design is to show their relation to the software components through the COMMUNICATION OBJECTS.

Thus, there is only one type of hardware objects containing its name and an informal description of its purpose. The use relations and the data-flow to and from the COMMUNICATION OBJECTS are nevertheless useful to get a clearer picture of the system architecture.

5.4 Example

This section intends to illustrate the HRT-HOOD+ objects by a simple example. The particular example was selected because it is easy to understand while maintaining many important issues. In order to keep it short, it represents a ROOT OBJECT with only one SYSTEM OBJECT. The example will be reused through this thesis to illustrate other aspects of the method.

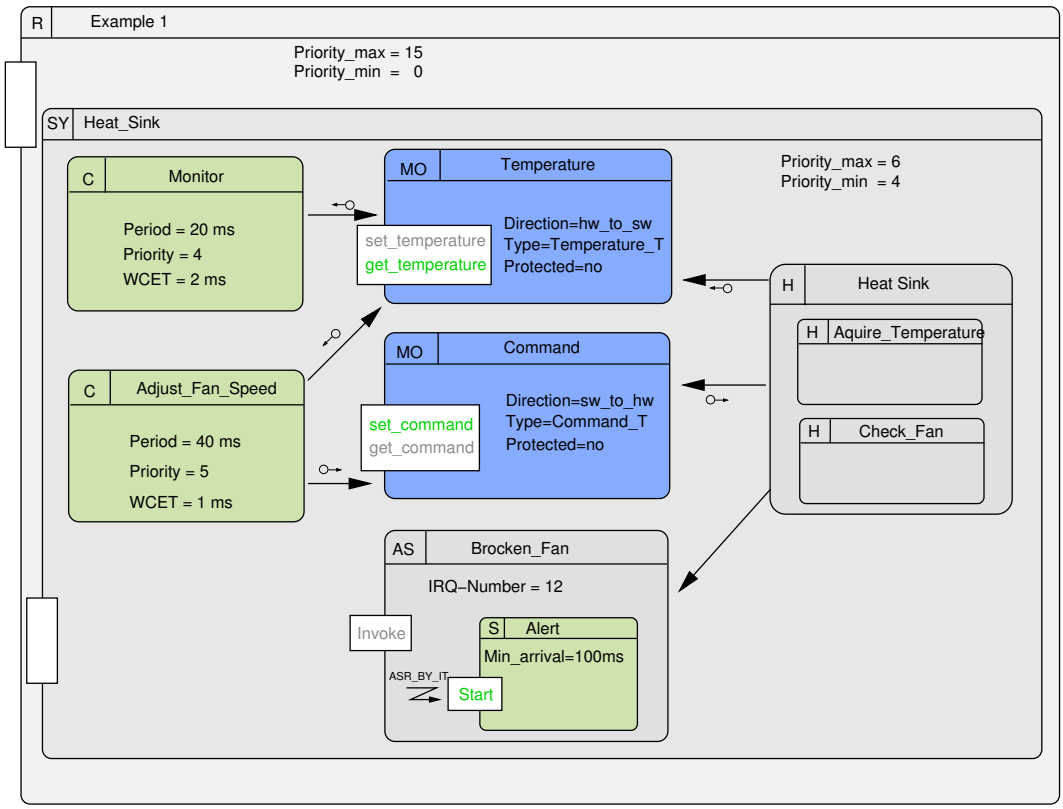

Figure 5.10: An embedded heat sink controller

Figure 5.10 shows the simplified graphical representation of the HRT-HOOD+ design of an embedded controller for a heat sink. The system consists of two software objects, three COMMUNICATION OBJECTS, and one hardware object. The hardware will for example detect the fan failure, read the temperature sensor, and convert the analogue value according to the sensors characteristic curve into a valid digital temperature value.

The arrows show the use relationship and the arrows with a small circle on one end the direction of data-flow between objects. The purpose of the software system is to adjust the speed of the fan on the heat sink to keep the device in a desired temperature range and to show the temperature through a display. Therefore, the Cyclic Objects `Adjust_Fan_Speed` and `Monitor` need access to the temperature of the heat sink provided through the MEMORY OBJECT `Temperature`. The command to decrease or increase the fan speed is transferred to the hardware through the MEMORY OBJECT `Command`.

If a failure occurs in the fan hardware the ASYNCHRONOUS SIGNAL `Broken_Fan` will propagate this event asynchronously from the hardware to invoke the handler provided by the Sporadic Object `Alert`.

Each COMMUNICATION OBJECT defines its method interface according to the type of data to be accessed. The data-flow determines the visibility of methods to other hardware and software objects (Table 5.1). For example, the `set_command` method in is only visible to the software while the `set_temperature` method is only available in hardware[10].

The attributes mentioned above are filled with exemplary values obeying the rule defined in the previous section.

5.5 Executable specification and mapping to implementation

As stated in Section 4.5.1 it is very important for a design method to provide means for an early validation and the implementation of the specified system. Since hardware objects are not specified formally in HRT-HOOD+, it is not possible to map them automatically to an executable specification, say in VHDL.

For all other objects, which means the software objects and the newly introduced COMMUNICATION OBJECTS, mappings to their respective simulation and implementation models are defined in OOCOSIM. The models for simulation and implementation differ only in certain minor aspects related to the specification of the real-time behaviour and to physical resources such as direct memory access and interrupts (Section 7.5.3).

5.6 Recap

First, a brief overview of the OOCOSIM methodology as a whole was given. Then the graphical modelling notation HRT-HOOD+ was introduced. It provides means to decompose complex systems into simpler sub-systems through the introduction of SYSTEM OBJECTS. It furthermore augments HRT-HOOD by the class of COMMUNICATION OBJECTS. These object types allow modelling the communication between hardware and software in a comfortable and analysable way.

Even though the work at hand suggests an object-oriented design-flow for entire embedded control systems the design of hardware is not a central issue in this work. Instead, two other domains will be addressed here: the design of real-time software and the design of the hardware/software interface. As stated in previous sections it makes no sense to develop hardware and software independently while it seems inappropriate to use a single design language for the entire system. To overcome this dilemma OOCOSIM suggests a temporal coupling mechanism for real-time software and the hardware while supporting the functional coupling via COMMUNICATION OBJECTS.

[10] In the figure, software methods are coloured green while hardware functions are coloured grey.

6 Hardware/Software Interface Design

Most embedded systems contain some components implemented in hardware and others implemented in software, which jointly perform an application specific task. While being divided into many separate components a close coupling between them is essential to achieve the intended behaviour. Software components easily communicate by shared variables or by method calls. Hardware components communicate through signals or busses. These communication mechanisms are somehow incompatible because software has no ports and hardware has no runtime system to handle method calls.

The hardware/software interface plays the important role of bridging this communication gap in embedded systems. Through such an interface, sensor-data acquired by a hardware component is transmitted to the software for further calculations. Vice versa, the software can send commands to actuators connected through with the hardware. The exchange of data and control commands is often implemented by so-called *device registers* [Por00]. These registers are stored in shared memory areas reserved for this purpose.

Some events occurring in the hardware or the environment of the embedded system require an immediate activity in the software. At the physical layer, these events are transmitted through interrupts activated in hardware and handled in software by a so-called *interrupt service routine*. Interrupts alone can carry only the plain information that an event has occurred. In combination with a device register, any kind of data can be transmitted with the occurrence of the event. The central position of the hardware/software interface is depicted in Figure 6.1.

The design activity of modelling such interfaces is (in general) time consuming, difficult and error prone. Many different types of information must be exchanged between hardware and software. Some device registers represent the value of a particular sensor, for example the temperature (fixed point real) or pressure (integer) of a tank. Others contain the control commands to be send to an actuator like the discrete mode of an engine (`Off, Standby, Slow, Half, Full`).

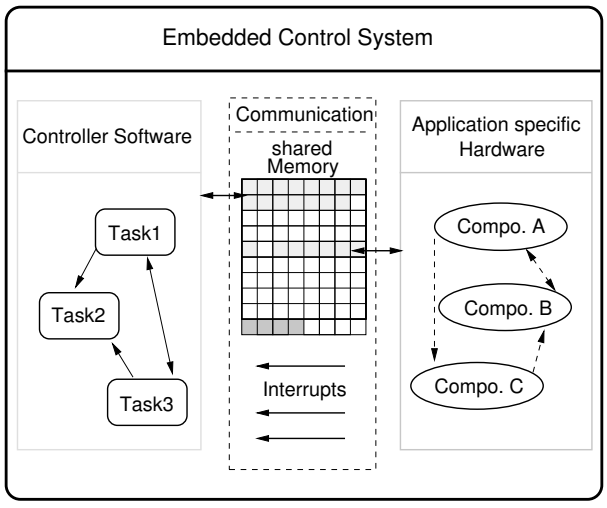

Figure 6.1: HW/SW interface in an embedded system

Typically, only limited resources are available to implement the interface. Hence, memory efficiency requirements do not permit to use large standard data types for the information exchange but require 'bit knitting'.

Since components in hardware and software need access to the information stored in the interface, the location and representation of data types or commands in the shared memory must be the same for both communicating partners.

The (usually virtual) address of a normal data structure in software and the representation in memory can be determined automatically by the compiler. In hardware, a synthesis tool can deter-

mine the representation. Depending on whether the data structure is located in a random access memory or in a register, the allocation is determined by either the designer or the synthesis tool.

An isolated approach like this is no solution for the hardware/software interface. First, because the coherency of the layout in both domains could not be achieved and second, because in many cases the hardware components contain predefined device registers that must be addressed by specific software drivers. Thus, the allocation, layout, and representation of data types in memory must be determined unambiguously either by the designer or a tool, covering the complete interface design process.

For embedded systems, there is no dominant platform like for PC's and there is hardly one approaching because the requirements for embedded systems are so manifold. To match these requirements many hardware platform already exist and more are emerging. Therefore, a universal concept for the interface design should not rely on primitives taken from a target specific communication library. The porting effort and the time gap between the appearance of a new architecture (variation) and the port for the library makes this approach unattractive. The alternative chosen in this work is to define a target-independent description language for interface objects.

There is no ultimate solution for the synthesis of hardware/software interfaces. Every application domain might need a slightly different synthesis for the interface. In some applications, large amount of data must be transferred between hardware and software. The generation should try to minimise the access times or the space requirements. Other applications need mutual exclusive access to the data structures to avoid interference. Therefore, the interface synthesis should be flexible and customisable for the designer. Consequently, this work introduces a flexible mechanism for the generation of the hardware/software interface.

Unfortunately, there is no synthesisable language available for both, hardware and software. Thus, interfaces need to be synthesised from different description languages. Consequently, the interface components need to be created twice - once for hardware and once for software. Every detail of the representation must be determined twice - exactly consistently in different languages. Problems arising in the manual approach are the large effort for the designer and the danger of difficult to find mistakes.

Some programming languages provide features to determine the representation of data in the memory. Ada95 for example does this with so-called *representation clauses* (Listing 6.1). They allow determining the size, address, layout, and other attributes for data types and data structures.

```
for Temperature_T use
  record
    Temperature_High at 0 range 6..0;
    Temperature_Low at 1 range 7..0;
  end record;
for Temperature_T'Size use 15;
Temperature : Temperature_T;
pragma VOLATILE(Temperature);
for Temperature'Address use To_Address(16#1A2#);
```

Listing (Ada95) 6.1: Representation clauses.

In HDLs like VHDL, device registers are modelled by signal vectors. Theses signals can be assigned to bit vectors and their layout can be controlled.

Since the hardware/software communication is obviously a key component for the embedded system, it should be documented very well. This is in particular important since it is literally an interface between hardware- and software designers. Keeping this documentation up to date through all the changes during the design process and maintenance of the system can be challenging. Hence, a mechanism to generate the documentation automatically from the specification is highly desirable.

The approach presented here uses a unified specification of the interface regardless of its implementation in hardware or software. It provides an abstraction-layer that allows to specify the

interface with COMMUNICATION OBJECTS (Section 5.3). The interface description language ComiX [OZN01b, OZN01a] provides a processable and human readable representation of these objects. It furthermore provides the so-called *textual representation* of COMMUNICATION OBJECTS.

Based on this representation, automatic tools can generate the implementation in hard- and software as well as the documentation. In line with the code generation (Section 6.5), consistency checks improve the reliability of the specification.

All aspects of the interface specification and implementation are implemented in a graphical design tool called DESHICO (Section 6.6).

6.1 Specification of hardware/software interfaces

This section starts with a description of the traditional design technique for the hardware/software interface to identify its shortcomings and a list of required features for an interface definition language like ComiX. Then the layered structure of the language is presented.

6.1.1 Mainstream interface design

Traditional interface design is based on so-called *memory map tables* denoting the mapping of shared data-structures to the memory mapped I/O area. Usually these tables look like in Figure 6.2[1].

These maps describe in particular the memory address of control information and the access mode. Since the mapping is usually given for a single memory-word (for example a byte), it is not immediately possible to define larger data-structures bigger than this. In Table 6.2 the data structure `Temperature` used two successive bytes. The work around used in this description is to use two components `Temperature_High` (higher 7 bit) and `Temperature_Low` (lower 8 bits). This however requires additional effort when accessing the data structure as a whole.

Address	Access	7	6	5	4	3	2	1	0
16#1A0#	ReadWrite		Ax	Vx	Recv	Ack			
16#1A1#	ReadOnly					TMode			
16#1A2#	ReadWrite		Temperature_High						
16#1A3#	ReadWrite	Temperature_Low							

Figure 6.2: Example of a memory map table

The memory map serves as specification used by hardware and software designers. The software designer typically encapsulates the memory mapping in functions to access shared variables. The hardware designer defines similar functions in a VHDL package. This methodology provides at least a higher maintainability of the code since changes must be applied only to two modules. The coding and decoding of typed data from and into the shared registers must be done manually[2].

The drawbacks of this approach are manifold. The effort to achieve an implementation is huge since the specification must be mapped manually to hardware and to software. The coding and decoding of data types like larger integer or enumeration types into bit-vectors has to be done manually in general. Since two different languages (for example C and VHDL) are involved, inconsistent encodings can cause errors in the application, which are hard to detect.

[1] The address is given in hexadecimal numbers. The grey boxes indicate unused bits; the identifiers in white boxes refer to the components names of the mapped data structures.

[2] In the above example: *Temperature* = 256 * *Temperature_High* + *Temperature_Low*

6.2 Interface design in research

Even though the problem of hardware/software interface design often is regarded as difficult and time consuming little research has been done so far in this area. The consistent modelling and implementation of interfaces requires a formal notation to describe the interface components plus a mechanism to translate this notation into an implementable representation, e.g. source-code in C or VHDL, which can be fed into the further design flow.

Hovater presents in [HMB00] an ASIS (Section 2.8, [Int99]) based tool to generate documentation for device registers described by representation clauses written in Ada. The approach presented in his paper analyses the code rather than to create it from an abstract specification.

Standards like XDR [SUN87] or ASN.1 [Dub00] provide detailed formal methods to specify data types but with these methods the designer cannot specify the particular layout of device registers. While XDR and ASN.1 are able to specify communication using various protocols, the method presented in this work focuses on a communication architecture based on memory mapped I/O and interrupts.

In contrast, the approach presented here starts with an abstract interface specification. It generates the target code for hardware and software as well as the documentation for the specification automatically. None of the approaches mentioned above provide an appropriate method to specify a hardware/software interface at this level of detail and generate a synthesisable description for its hardware and software components.

Many approaches like [LPN98, VSV99] use communication primitives taken from a target specific I/O library to map abstract communication channels to an implementation. These works intend to generate automatically the interface from the description of the communicating components. The problem with these approaches is that these libraries must be implemented for each processor and operating system. Furthermore, they do not provide a methodology to describe the low-level mapping and layout of the communication on the memory.

The approach described in [OJ00] separates the platform specific characteristics of the processor and the operating system in libraries specified in a language called *ProGram*. Based on this information and the interface specification for the software device drivers are generated. This approach however completely disregards the hardware side of the interface.

The *SYNPHONY*™ tool that is part of the CoWare N2C™(Section 3.3) tool suite uses the methodology described in [VSV99] to generate hardware/software interfaces based on processor interrupts to guarantee real-time requirements on the communication. The tool depends on the fact that the entire system including hardware and software is captured and generated with the CoWare N2C environment. It is therefore unable to support interfaces to hard IP components. Moreover, it again depends on library implementations for specific processors and operating systems supported by the tool suite. In [HCL+99] Hessel et al. stress the importance of the ability to connect heterogeneous models in co-design. The work presents an abstraction concept for communication based on channels. The models may call methods like `send()` or `receive()` to model communication. The method then selects the appropriate protocol by choosing the required interface connectors from a library. This library needs to be provided by the user for the different languages used in the design. It needs to contain the connectors and a cost function that is the base for the protocol selection. Therefore [HCL+99] rather presents a meta method for interface synthesis than the synthesis itself. Furthermore, the method addresses communication at a rather abstract level while the approach presented in this work allows the detailed specification of each interface object.

With the *Devil* [MRC+00] approach, Merillon et al. present an interface description language (IDL) to describe a functional model of the hardware implementation of an interface. The Devil compiler generates C code from this hardware abstraction to provide a set of functions to access the interface. The concept of the Devil approach has been proven by the implementation of various device drivers for the Linux kernel. The approach can improve the development of software drivers if the hardware is given. However, Devil is not able to generate hardware models or documentation from the specification. Similar to the approach presented in Chapter 6 the Devil compiler can check

the specification for consistency. In contrast to the work at hand, the code generation is limited to the Devil compiler. There seems to be no concept to improve or change the code generation by the user.

The thesis of Lehmann [Leh02] describes a method for the driver synthesis to access so-called *communication channels* connecting software and hardware. The approach concentrates on the generation of software drivers, which are able to access hardware architectures containing multiple components to implement the data transfer. The hardware structure is first analysed and then described in an abstract communication graph. The method is able to (at least partially[3]) handle even complex communication infrastructures as they can be found in modern personal computers. In contrast to the approach to the work at hand, the driver synthesis is based on an existing hardware architecture.

There is one commercial product called VCI compiler [VCI00] targeting directly the same problem domain. The tool is web-based and uses an interface specification called VCI Data Model to generate VHDL and C-header files for the implementation of the hardware/software interface. It supports data types like boolean, enumeration, integer, and their aggregation. The tool generates the address allocation and the layout automatically, which reduces the effort for the designer but also the flexibility since there is no way to control the result of the code generation.

6.2.1 Existing vs. OOCOSIM approach

In contrast to the above, in the following a concept will be presented, which allows (but not requires) to specify and implement an interface in a very detailed way. It is thus a descriptive and a constructive method. The concept enables the generation of code for arbitrary target platforms. The generation scheme as well as consistency checks can be adopted by the designer to the specific application domain or the preferred modelling language.

None of the existing concepts can offer the expressive power and the flexibility in the code generation offered by the approach described in this work. The existing approaches are either limited to the specification like XDR or ASN.1 or allow the synthesis only within the limits defined by a predefined communication library. The method at hand instead provides an extensible notation as well as a mechanism to define translation rules to generate arbitrary implementations or checks for the interface specifications.

The approach presented here regards interface objects as distinct objects and allows the designer to customise them to their need. These objects are mapped by predefined translation rules to language primitives in Ada95 for the software components and VHDL for the hardware blocks. The specification and implementation method is the seamless continuation of the graphical notation for COMMUNICATION OBJECTS of HRT-HOOD+ , introduced in Section 5.3.6.

6.2.2 Requirements for ComiX

The main requirement for ComiX is to provide a universal intermediate format to describe hardware/software interfaces. Thus, it is necessary to define an abstract format that depends neither on a particular input tool at the front end nor on a particular target language at the back-end.

An IDL like ComiX must allow the designer to specify the detailed layout of the intended communication interface. This is especially necessary if parts of the interface are predefined by e.g. an off-the-shelf component[4].

On the other extreme if no off-the-shelf components are involved, it should also be possible to neglect details in the description, which can be automatically determined by an 'intelligent' code generator. In both cases, the consistency of the interface can be guaranteed because the code generator chooses the required values from a single specification for both sides of the communication.

[3] In his work, Lehmann states that not all components of driver synthesis can be handled automatically.

[4] The reader may think of an off-the-shelf hardware device providing a device register in a particular layout. In such situations, the designer needs to describe the interface rather than design a new one.

As embedded systems become larger and more complex, the interface modelling technique should be able to split the description into several specification modules possibly organised hierarchically. The designer might want to reuse some interface components defined in previous designs. Thus, the IDL should provide an easy-to-use library mechanism.

Flexibility is an important issue. With ongoing research, it might be necessary to extend the IDL to cope with new requirements. By the time of this thesis version 1.1 of ComiX is in use but later versions may emerge from further development. For this aspect, it was not sufficient to define the IDL itself in an extendable manner, since every modification to the language would require a new code generator. Hence, it was also necessary to introduce a modular and template based concept for the code generation as described in Section 6.5.

Hardware/software interfaces eventually become very large. Specifications sometime occupying 50 pages and more written down in memory maps are a challenge to the design methodology. Therefore, the IDL should provide a well-defined structure allowing for changes bound to a certain scope in the specification and consequently reduce the specification's complexity.

6.3 The ComiX language

In this section an IDL named ComiX (Communication Interfaces in XML) fulfilling the requirements mentioned above will be introduced. Like most XML-based languages, ComiX is specified by a document type definition (DTD)[5]. The DTD for ComiX can be found in Appendix A. Using XML as a basis for the language makes ComiX flexible and extensible as the XML notation guarantees the immediate existence of a parser for every XML compliant language.

A ComiX interface description consists of four layers depicted in Figure 6.3. These layers allow

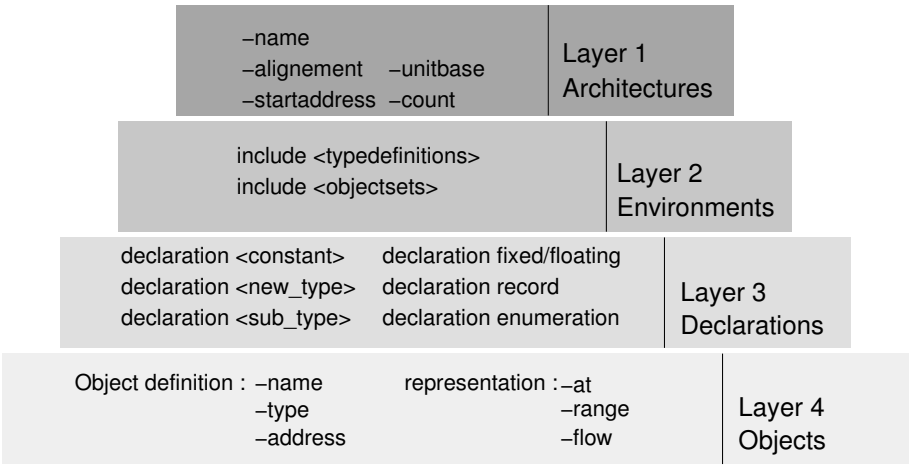

Figure 6.3: Structure of a ComiX description

specifying and refining the interface in a stepwise manner. The design will probably begin with a description of the overall architecture (Layer 1), that is the available resources in terms of memory and interrupts, of the interface. Then it might be considered to reuse some predefined[6] components

[5] A DTD is like a grammar for a XML-document. It defines the allowed tags and attributes for the language. See also Section 2.5.1.

[6] Predefined here means interface specifications defined in earlier designs in contrast to predefined by the language.

of the interface (Layer 2). New data types will be declared in Layer 3 to be used in the definition of COMMUNICATION OBJECTS in Layer 4.

More formally, the ComiX structure is defined by the following.

Definition 6.1 (ComiX Layer Structure) $\mathcal{C} = \langle \mathcal{L}_1{}^\star, \mathcal{L}_2{}^\star, \mathcal{L}_3{}^\star, \mathcal{L}_4{}^\star \rangle$, *where:*

- $\mathcal{L}_1$ *denotes an Architecture Layer (Section 6.3.2),*
- $\mathcal{L}_2$ *denotes an Environment Layer (Section 6.3.3),*
- $\mathcal{L}_3$ *denotes a Declaration Layer (Section 6.3.4),*
- $\mathcal{L}_4$ *denotes an Object Layer (Section 6.3.5).*

Even though this design flow appears to be natural, the designer may choose a different flow or jump back and forth between the layers. Since these layers build upon each other a complete interface specification requires the definition of Layer 1, Layer 3[7], and Layer 4.

The ComiX DTD contains some so-called *optional elements*. These elements describe parts of the interface that exists in some but not all specifications. For example, in many interface architectures no interrupt is reserved. Hence, the *reservedinterrupt* element is an optional sub-element of Layer 1 which allows to omit this part in the specification.

For each layer specific mandatory information must be provided. In Layer 1 for example, the start address and the size of the memory mapped I/O area must be defined to allow subsequent consistency checks (Section 6.4). These properties are called *required attributes*. Other attributes, which can be determined by the designer or calculated by a tool, are called *implied attributes*. The following will describe the different layers and their elements and attributes providing a detailed view on the interface specification methodology in OOCOSIM.

6.3.1 ComiX Root Element

The root element of each interface specification in OOCOSIM is the ComiX element containing the four layers mentioned above plus some descriptive data about the interface specification like the author and the projects' name.

Further important is the *name* attribute, defining a unique identifier for the ComiX document. This name is in particular useful if this document will be referenced in Layer 2 (Section 6.3.3) of other ComiX for reuse. It also provides the link between the graphical design entry in HRT-HOOD+ and the textual ComiX specification.

The *Comix_version* attribute specifies which DTD of ComiX is used in this document[8]. For a processing tool, like for example the code generator in DESHICO, this information is probably important to decide whether the document uses an appropriate format.

6.3.2 Architecture Layer 1

The first layer describes the overall physical architecture of the interface in the architecture element. It describes the properties of the memory mapped I/O area and the interrupts provided by the processor. Figure 6.4 depicts graphically the scope of Layer 1 and shows exemplary values for some attributes. This layer separates the target specific physical characteristics from the rest of the specification. If throughout the design process, the target architecture must be changed this layer must be adapted. The consistency checks as described in (Section 6.4) will then identify implied changes in the specification to achieve a consistent ComiX specification.

Formally, Layer 1 is defined by the following:

[7] Layer 2 is optional but can also replace other layers, as it allows to reuse specifications from previous specifications.
[8] The version described in this thesis is version 1.1 but there might be successive versions of ComiX in the future.

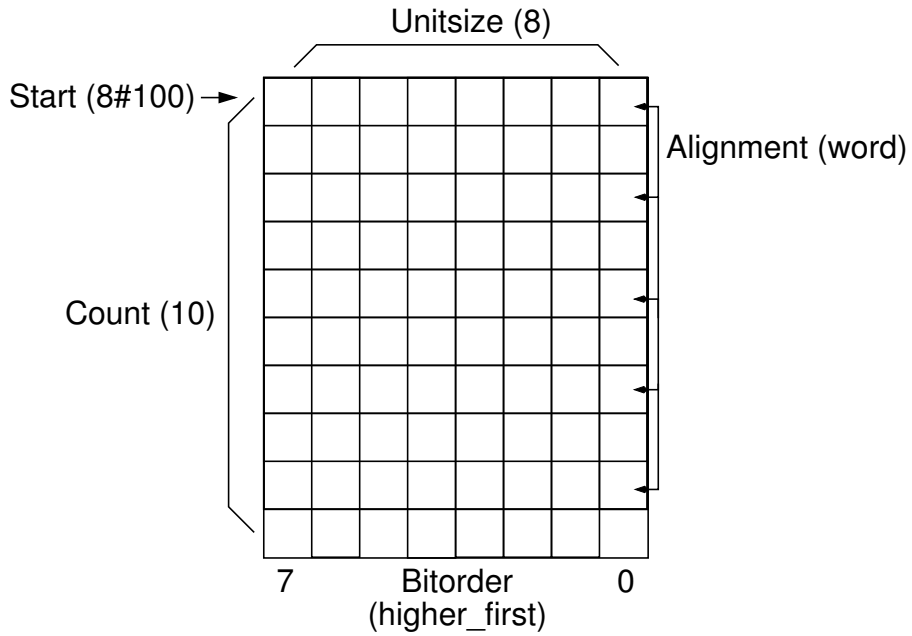

Figure 6.4: Architecture Layer

Definition 6.2 (Layer 1) $\mathcal{L}_1 = \langle M, a, u, se, he, f, l, \bar{\mathfrak{I}} \rangle$, *where:*

- *The set M of memory blocks m, which define the available memory areas in the communication architecture*[9]*. A memory block m is defined as the triple $m = \langle s, c, \bar{\mathfrak{M}} \rangle$ with:*
  - $s \in \mathbb{N}_0 \cup \{\nu\}$ *: The Startaddress s defines the first valid address of the memory mapped I/O area. To indicate that no memory mapped I/O area is available s takes the value ν (undefined). Note, that in this case, no* MEMORY OBJECTS *or* ASYNCHRONOUS MEMORY OBJECTS *can be defined; that is, the interface is restricted to physical interrupts.*
  - $c \in \mathbb{N}_+ \cup \{\nu\}$ *: The Count c defines the number of storage units available in the memory mapped I/O areas. Consequently, the last storage unit is at address $s + c - 1$. Again, ν represents the undefined value in case that no memory mapped I/O area is available.*
  - $\bar{\mathfrak{M}} \subseteq \{s, .., s + c - 1\}$ *: Denotes the set of reserved memory addresses. In the memory-mapped I/O area, some addresses might not be usable for this communication interface. Through $\bar{\mathfrak{M}}$ it is possible to exclude some memory addresses from possible use in this specification (in particular allocating* MEMORY OBJECTS *to these addresses). Every element reservedaddress in $\bar{\mathfrak{M}}$ has only one attribute address that excludes a single storage unit from further use.*
- $a \in \{1, 2, 4, 8, \nu\}$ *: For many target architectures the memory access is restricted to so-called aligned addresses. In this case, read or write operations can accesses only addresses at word (even addresses) or longword (multiples of 4) or even longlongword boundaries. Addressalignment denotes this requirement on legal addresses for* COMMUNICATION OBJECTS[10]*. Formally, the set of aligned addresses can be defined as follows:*

$$\mathfrak{A}_{aligned}(a) = \{v \in \mathbb{N}_0 \mid \exists i \in \mathbb{N} : v = i \cdot a\} \tag{6.1}$$

[9] Please note that the available memory in the communication architecture is not necessary continuous. Instead, it can be distributed in several blocks within the overall address space.

[10] Please note that address here refers only the address of the first storage unit of an access.

- $u \in \{1, 4, 8, 16, 32, 64, \nu\}$:*Storageunit determines the size of a storage unit in number of bits, in the memory mapped I/O. The symbolic value of ν denotes that the storage unit size is undefined.*

- $se, he \in \{big, small\}$ *: Softwareendianness and hardwareendianness describes the position of the most significant bit in a storage unit. It may take the values big and small for big-endian and small-endian. The endianness of the hardware components may differ from the software. Hence both (se and he) are required to be defined.*

- $f, l \in \mathbb{N}_0 \cup \{\nu\}, f \leq l$ *: Firstinterrupt and Lastinterrupt determine the interrupts available in the communication architecture. Only interrupts numbers in the interval between f and l can be used in Layer 4.*

- $\bar{\mathfrak{I}} \subseteq \{f, .., l\}$ *: Denotes the set of reserved interrupts. Similar to the set of reserved addresses $\bar{\mathfrak{M}}$ this set excludes certain interrupts from further use. This is in particular useful if the communication architecture is shared by multiple applications*[11]*. The element reservedinterrupts contains only the interrupt attribute defining the interrupt number.*

Each component in $\mathcal{L}_1$ corresponds to an attribute or an element in the Layer 1 part of a ComiX *document. The name of the attribute can be found in the above definition of $\mathcal{L}_1$.*

The definition of $\mathcal{L}_1$ mainly aims at declaring the resources available, that is the accessible memory and interrupts in a communication architecture. Only resources declared in a Layer 1 specification can be used to implement the COMMUNICATION OBJECTS defined in the Object Layer (Section 6.3.5) of the specification.

More formally, $\hat{\mathfrak{M}}$ denotes the set of storage units in the memory mapped I/O area:

$$\hat{\mathfrak{M}} = \{i | i \geq s \wedge i < s + c\} \tag{6.2}$$

Then $\mathfrak{M}$, the set of storage units available for COMMUNICATION OBJECTS, is defined as follows:

$$\mathfrak{M} = \hat{\mathfrak{M}} \setminus \bar{\mathfrak{M}} \tag{6.3}$$

Furthermore, $\hat{\mathfrak{I}}$ denotes the set of interrupts in a communication architecture:

$$\hat{\mathfrak{I}} = \{i | i \geq f \wedge i \leq l\} \tag{6.4}$$

Then $\mathfrak{I}$, the set of accessible interrupts, is defined as follows

$$\mathfrak{I} = \hat{\mathfrak{I}} \setminus \bar{\mathfrak{I}} \tag{6.5}$$

The resources $\mathfrak{R}_{\mathcal{L}_1}$ of a communication architecture specified by a ComiX Layer 1 is defined by the 2-Tuple $\langle \mathfrak{M}, \mathfrak{I} \rangle$

Definition 6.3 (Legal addresses) *Let $\mathcal{L}_1$ be a Layer 1 in a* ComiX *specification. Then the set of legal address in this architecture is defined as follows:*

$$\mathfrak{L}_A = \{i \in \mathfrak{M} \mid \exists n \in \mathbb{N}_0 : i = n \cdot a\} \tag{6.6}$$

Simple communication interfaces will contain only one Architecture Layer. To accommodate also complex communication interfaces it is also possible to define multiple architecture describing different memories for the interface. In case of multiple Layer 1 specifications, the resources are defined as the unification of the resources in each Layer 1 specification:

[11] For example, some interrupts may be already in use for the systems functionality.

Definition 6.4 (Resources in ComiX) *Let $\mathcal{L}_{1_0}, .., \mathcal{L}_{1_k}$, $k \in \mathbb{N}$ be Layer 1 specifications of an* ComiX *interface specification C, let $\langle \mathfrak{M}_i, \mathfrak{I}_i \rangle, i \leq k, i \in \mathbb{N}$ the respective resources, then the resources of this interface are:*

$$\mathfrak{R}_C = \bigcup_{i=0}^{k} \mathfrak{R}_{\mathcal{L}_i} = \bigcup_{i=0}^{k} \mathfrak{M}_i \times \bigcup_{i=0}^{k} \mathfrak{I}_i \tag{6.7}$$

This layer may also be included (Section 6.3.3) from a former design. Thus, the Architecture Layer may be omitted if there is at least one defined by the Environment Layer.

6.3.3 Environment Layer

The *Environment Layer* (also called Layer 2) allows to include library components such as communication architectures, type definitions, or predefined communication objects. Hence, this layer enables the reuse of former work as well as the decomposition of complex designs into multiple specifications.

Since the reuse mechanism is simply based on inclusion of parts of former specifications, there is no need for a specific formal treatment of Layer 2. Instead, the inclusion can be treated as textual insertion of the respective parts into the ComiX specification under design[12].

The ComiX language element `include` provides the means to specify the part of an existing specification to be reused in the specification under design. The `include` element therefore contains the following attributes:

interfacename: Defines the name of the ComiX specification from which the specified part is to be included.

part: The part attribute determines, which kind of component or which set of components will by added to the ComiX specification under design. The predefined values this attribute can take and their semantic are defined in the following list.

- **All:** means layers 1 to 4 will be added to the specification under design. Using this value the designer can for example split the interface specification into disjoint partitions each defined in a separate document. A master ComiX document includes the partitions into the final interface.
- **AllArchitectures:** Adds the complete Layer 1 of the referred ComiX document. This can be in particular useful when many designs use the same target architecture. A specific target platform usually provides the same interrupts and the same memory mapped I/O area.
- **AllDeclarations:** Adds Layer 3 from a former design to the actual one. This is useful when a collection of types is used again. A ComiX document can use for example the definitions from a standardised type coding like XDR [SUN87], ASN.1 [Dub00] or an in-house standard without defining them explicitly in the specification.
- **AllObjects:** Includes all COMMUNICATION OBJECTS. This typically only makes sense if also all declarations were included. Using all objects can be in particular valuable when a preceding design needs to be enhanced by new interface components while all the old objects are kept.
- **Declaration:** Adds a single declaration of a type to Layer 3. If the transitive attribute is set to `yes` for a subtype, the base-type definition is included and for a record type the transitive set of component types will be added to Layer 3.
- **Asynchronoussignalset:** Includes a named set of ASYNCHRONOUS SIGNAL elements will be added to Layer 4.

[12] The mechanism is very similar to the # `include` preprocessor statement in the C programming language. The included part in textually inserted before the compiler processes the program.

Asynchronousobject: Includes the named ACTIVE OBJECT into Layer 4.

Memoryobjectset: Adds a set of MEMORY OBJECTS into Layer 4.

Memoryobject: Adds a single MEMORY OBJECT to Layer 4.

partname: Depending of the value of the part attribute it might be necessary to specify name of a particular component. This is required for all part attributes not starting with the prefix 'All'.

Since Layer 2 can include arbitrary elements from an existing ComiX specification, it has potential impact on all other layers. With careless use of the include element, inconsistent specifications can be created[13]. Since, the consistency checks (Section 6.4) apply to the full ComiX specification and therefore help identifying errors introduced through this layer the risk is limited[14].

6.3.4 Declaration Layer 3

This third layer in the ComiX language is enclosed by the XML element *declaration* and allows to define constants and to declare types. Constants are defined by their name and their value attribute. For the processing of the specification, the constants are textually replaced by their values. The types that can be defined in ComiX include new range types, subtypes, enumeration types, fixed-point types, floating-point types, and records to aggregate the preceding types. For each type definition, a ComiX element with a number of attributes defines the desired properties. In contrast to general programming languages, the type declaration for records in ComiX also contains the so-called *relative layout information (RTI)*. The RTI defines the bit wise allocation, more exactly the unit-number, and the bit-range of a component in memory. Together with the start-address of an object of this type, the layout in memory is then exactly defined. Since COMMUNICATION OBJECTS in ComiX can only be of record types there is no need to define the RTI for other types[15].

Every type must have the attribute *name* defined uniquely to allow an unambiguous definition of COMMUNICATION OBJECTS. The *size* of each type must be determined - either computed by a tool or by the designer. The size is in particular useful to enable consistency checks as described in Section 6.4. Other attributes are associated with certain types and will be explained in the following:

New range types: Range types are useful to represent arbitrary intervals of integer values. They are defined by the `rangetype` element with the specific attribute `range`. In the following Example 6.2, a range type named `Temperature_T` for the values from 0 to 300 is defined:

```
<rangetype name="Temperature_T"
    range="0 .. 300"
    size="9" />
```

Example (ComiX) 6.2: Rangetype in ComiX

Enumeration types are defined using the `enumeration` element and a number of `item` sub-elements. Since enumeration types are specified by an ordered sequence of items, the enumeration element itself has only the standard attributes `name` and `size`. The `item` sub-elements have the attributes `itemname` and `coding` to define the literal and its explicit representation in memory. Enumeration types are in particular useful to communicate system states and commands for hardware devices. The following ComiX code fragment in Example 6.3 will illustrate this:

[13] If the design would for example include a certain object set but does not include the required declarations.

[14] Since no new semantic element are introduced here, the consistency rules defined for the other layers are sufficient for Layer 2 as well.

[15] To create a COMMUNICATION OBJECTS of a simple type (for example an enumeration type) it is sufficient to define a simple record with exactly one component of this enumeration type.

```
<enumeration name="State_T"     size="2">
    <item name="Broken"         coding="2#00#" />
    <item name="OK"             coding="2#01#" />
    <item name="unknown"        coding="2#10#" />
</enumeration>
```

Example (ComiX) 6.3: Enumeration type.

Sub-types are used to specialise already defined types; that is, to specify a reduced range of values. The ComiX language element *subtype* carries the attribute *basetype* to define the base-type, which can be any defined type except a record type. The *range* attribute specifies the possible values of this type. In the Example 6.4, a smaller temperature interval is defined, using only 7 bits in memory.

```
<subtype name="Low_Temperature_T"
      basetype="Temperature_T"
      range="2 .. 100" size="7" />
```

Example (ComiX) 6.4: Subtype.

Real: These types decompose into two numeric categories or representations: floating-point and fixed-point. The only common[16] is the range attribute that defines the minimum and maximum values of the type.

The specific attributes `delta, small` and `digits` are defined by the sub-elements *fixed* and *float*. For fixed-point representation delta defines the accuracy of the type and small the base-unit for the type; that is, all values are represented by multiples of small.

For the floating-point representation, digits defines the precision of the number. The sub-element float also allows defining the detailed representation of a float type through the optional attributes `signed, mantissasize, exponentsize, exponentbias`. While many control dominated embedded system do not support floating-point types, they can be rather useful in the prototyping phase. COMMUNICATION OBJECTS using floating-point types can be replaced by fixed-point types in the final system, where preciseness and efficient implementation in hardware is important. Example 6.5 shows both categories of real types in ComiX.

```
<real name="Angle_Float_T"
    range="0.0 .. 360.0" size="32">
    <float digits="6"/>
</real>

<real name="Angle_Fixed_T"
    range="0.0 .. 360.0" size="16">
    <fixed delta="0.01" />
</real>
```

Example (ComiX) 6.5: Two different real types.

Records: The record element with only the standard attributes and a list of `component` sub-elements defines records (aggregates) in ComiX. Each component sub-element has the following attributes:

[16] Apart from name, comment, and size which are obligatory for all types.

name: Defines the name of the component.

type: Defines the type of the component.

private: When a MEMORY OBJECT of this record type is declared this attribute determines whether this component is visible through the objects interface. The default value is no.

unitnumber and bitrange: Define the RTI of this component in the memory. The `unitnumer` defines the offset in storage units for this component. The `bitrange` defines the interval of bits used by the component. The bitrange can cover more than one storage unit.

init: Defines the initial value for a component. The default value for this attribute is `unknown`.

The following Example 6.6 defines a record with two components namely `Normal_Temperature` and `State`. The component `Normal_Temperature` uses the first seven bits in the first storage unit and `State_T` the last two bits in the third storage unit. Please note that due to the spread allocation of its components the size of the record is 24 while the sum of its components is only 9 bits. The resource requirements for record types will be discussed in greater detail in Section 6.3.5.

```
<record name="Low_Temperature_Sensor_T" size="24">
    <component name="Temperature"
      type="Low_Temperature_T" private="no"
      init="Normal_Temperature"
      unitnumber="0" bitrange="0 .. 6" />
    <component name="State"
      type="State_T" private="no"
      init="unknown"
      unitnumber="2" bitrange="6 .. 7" />
</record>
```

Example (ComiX) 6.6: A record type.

The types defined in Layer 3 form the base for the definition of COMMUNICATION OBJECTS in Layer 4 defined in the following section.

6.3.5 Object Layer 4

The fourth layer in ComiX is called *Object Layer* and is enclosed in XML element *objects*. It defines the COMMUNICATION OBJECTS already introduced in Section 5.3.6. While in that earlier section the graphical representation and the integration with the whole design was the major focus now the detailed representation of the objects in the implementation platform is the main concern.

Many hardware/software communication interfaces contain large numbers of COMMUNICATION OBJECTS. Hence, ComiX allows decomposing the Object Layer 4 specification into multiple object sets. An object set may contain only objects belonging to one category of objects, namely MEMORY OBJECTS, ASYNCHRONOUS SIGNALS, ACTIVE OBJECTS. For a better structuring, the Object Layer is decomposed into three parts. Each part contains all object sets of one category of objects (see following definition).

Definition 6.5 (Object Layer 4) $\mathcal{L}_4 = \langle P_M, P_S, P_A \rangle$, *with:*

- $P_M = \{\mathfrak{O}_{M_i} | i \in \mathbb{N}_0, i \leq p\}$ *: The part*[17] *for the Memory Objects Sets* $\mathfrak{O}_{M_i}$.
- $P_S = \{\mathfrak{O}_{S_j} | j \in \mathbb{N}_0, j \leq q\}$ *: The part for the Asynchronous Signal Sets* $\mathfrak{O}_{S_i}$.
- $P_A = \{\mathfrak{O}_{A_k} | k \in \mathbb{N}_0, k \leq r\}$ *: The part for the Asynchronous Object Sets* $\mathfrak{O}_{A_i}$.

In the following paragraphs each type of object set is describe in detail.

[17] Mathematically these parts are sets of sets.

Memory Object Sets: Even rather simple embedded (reactive) systems typically exchange data and control information through many different MEMORY OBJECTS. Thus, the largest and usually most complex section of a ComiX specification contains the MEMORY OBJECTS. Therefore, ComiX allows to group memory objects in so-called MEMORY OBJECTS.

Each object set has a unique name and may contain one to arbitrary many MEMORY OBJECTS. In the ComiX language each Memory Object Set is represented by an XML element `memoryobjectset` with its sub-elements `memoryobject`.

Definition 6.6 (Memory Object Set) *A Memory Object Set $M \in P_M$ (Definition 6.5) is defined as follows: $M = \{m_u | u \in \mathbb{N}, 0 < u \leq x\}$ where each m_u specifies one* MEMORY OBJECT.

Memory Objects: While Memory Object Sets are only structural elements to support the overview in a complex specification, each MEMORY OBJECT determines all relevant information for a single memory mapped interface object. Hence, the `memoryobject` element contains the following attributes:

address: The location in the memory mapped I/O area as specified in Architecture Layer 1 (Section 6.3.2).

type: Defines the type of the object. For the definition of a MEMORY OBJECT, a record type is required since only these types allow specifying their detailed representation in memory. Since record types in ComiX may contain all other types as components this is no hard restriction.

dataflow: The direction in which the data is allowed to flow. Three possible values are allowed:

- **hs:** The hardware to software flow allows the hardware to write a MEMORY OBJECT and the software to read it.
- **sh:** The software to hardware flow allows the software to write a MEMORY OBJECT and the hardware to read it.
- **bi:** Allows both sides to read and write the information located in memory. As already discussed in Section 5.3.7, this can lead to several problems, but may be valuable in special situations.

accessmode: The accessmode attribute can have the values `atomic`[18] or `volatile`. Volatile in this context means, that the value for the objects component is written into the memory rather than into processor's cache. Atomic means that an access to any component of the MEMORY OBJECT must be executed in an atomic transaction.

protected: The attribute can take the values `yes` or `false`. If set to `yes`, similar to an Ada95 protected object, each access method in software is executed mutual exclusive. This option is usefull, if more than one task in software needs access to one MEMORY OBJECT.

size: For a MEMORY OBJECT the attribute size determines the maximum amount of bits, this object may use.

According to the description above a MEMORY OBJECT can be formally defined as follows:

Definition 6.7 (Memory Object) *A* MEMORY OBJECT *$m \in M$ (as in Definition 6.6) is defined as follows: $m = \langle ad, ty, df, am, pr, si \rangle$ where each component in the tuple corresponds positional to the attributes description above.*

The following Example 6.7 of a MEMORY OBJECT named `Engine_Temperature` will illustrate the attribute defined above:

[18] Note, that atomic implies volatile.

```
<memoryobject name="Engine_Temperature"
    address="8#100#"
    type="Low_Temperature_Sensor_T"
    dataflow="sh"
    accessmode="atomic"
    protected="no"
    size="36"
/>
```

Example (ComiX) 6.7: A MEMORY OBJECT.

Asynchronous Signal Sets: In reactive embedded systems quite often events must be modelled, which require the immediate reaction of the system. The parallelism and the very high performance in hardware allow detecting these events very fast. The immediate reaction of the software part however requires a specific mechanism, called *interrupt mechanism* to handle these events adequately. This mechanism is based on the interrupt signals almost every processor provides. The events, typically propagated through the operating system, are handled by a parameterless procedure called *handler* or *interrupt service routine (ISR)*[19].

In ComiX the modelling primitive that contains the hardware and the software part handling one event is called ASYNCHRONOUS SIGNAL. This type of COMMUNICATION OBJECTS defines the handler's name in the software part and the interrupt in the hardware part of the embedded system.

An *asynchronoussignalset* element contains a set of ASYNCHRONOUS SIGNALS. Each asynchronoussignalset has an attribute `setname`, which defines a common naming prefix for the access to all ASYNCHRONOUS SIGNALS within one set. An Asynchronous Signal Set $S \in P_S$ (as in Definition 6.5) is defined as follows:

Definition 6.8 (Asynchronous Signal Set) $S = \{s_v | v \in \mathbb{N}, 0 < v \leq y\}$, *where each* s_v *specifies one* ASYNCHRONOUS SIGNAL.

Each ASYNCHRONOUS SIGNAL is specified by the ComiX XML element *asynchronoussignal* with the following two attributes:

interrupt: The identification (typically a number) of the interrupt used for this event.

handler: This attribute defines the name of the interrupt handler to be registered in the operating system for this event or interrupt.

Definition 6.9 (Asynchronous Signal) *An* ASYNCHRONOUS SIGNAL $s \in S$*(as in Definition 6.8) is defined as follows:*
$s = \langle irq, hd \rangle$, *where irq denotes the interrupt and hd the handler's name.*

The following Example 6.8 of an Asynchronous Signal Sets containing two ASYNCHRONOUS SIGNALS is meant to illustrate the use of the above described ComiX elements.

```
<asynchronoussignalset setname="most_important_events">
    <asynchronoussignal handler="emergency_stop_pressed"
        interrupt="12"/>
    <asynchronoussignal handler="power_failure"
        interrupt="11"/>
</asynchronoussignalset>
```

Example (ComiX) 6.8: An Asynchronous Signal Set.

[19] Please note that handler is only the general name and not the name for a specific procedure.

Asynchronous Memory Object Set: ASYNCHRONOUS MEMORY OBJECTS can be used to transfer data and control asynchronously between hardware and software. Each *Asynchronous Memory Object Set* may contain any number of ASYNCHRONOUS MEMORY OBJECTS plus an attribute *setname* to define a unique naming prefix for the contained objects. Hence, an Asynchronous Memory Object Set $A \in P_A$ (as in Definition 6.5) is defined as follows:

Definition 6.10 (Asynchronous Memory Object Set) $A = \{a_w | w \in \mathbb{N}, 0 < w \leq z\}$, *where each* a_w *specifies one* ASYNCHRONOUS MEMORY OBJECT.

An ASYNCHRONOUS MEMORY OBJECT is specified through the element `asynchronousmemoryobject` containing the union of the attributes defined for MEMORY OBJECTS and ASYNCHRONOUS SIGNALS except for the handler name. Formally, an ASYNCHRONOUS MEMORY OBJECT $a \in A$ is defined as follows:

Definition 6.11 (Asynchronous Memory Object) $a = \langle ad, ty, df, am, pr, si, irq \rangle$ *with the components of the tuple as defined in Definition 6.7 and 6.9.*

One application scenario of ASYNCHRONOUS MEMORY OBJECTS has already been described in Section 5.3.9. For other embedded applications, more asynchronous events are required than the number of physical interrupts the processor's physical architecture (Section 6.3.2) can provide. With ASYNCHRONOUS MEMORY OBJECTS, it is possible to handle any number of events using only one physical interrupt.

In this case, the event is encoded in the MEMORY OBJECT component of the ASYNCHRONOUS MEMORY OBJECT(as in Example 6.9). The interrupt activates asynchronously the master handler[20], which then reads the stored event to determine the correct handler routine for this event.

```
<asynchronousobjectset setname="multi_event_handlers">
    <asynchronousobject name="my_mouse"
          address="8#104#"
          type="Mouse_Event_T"
          dataflow="hs"
          accessmode="atomic"
          protected="no"
          size="8"
          interrupt="14"/>
</asynchronousobjectset>
```

Example (ComiX) 6.9: An Asynchronous Object Set.

Figure 6.5 concludes the description of the ComiX language elements with a graphical overview as a DTD tree. The next section will introduce consistency rules based on the formal definitions given in this section.

6.4 Consistent ComiX descriptions

Only *consistent* ComiX interface specification can be used to generate correct interfaces implementations. The consistency of a ComiX document cannot be determined by a single property. Instead, it needs a set of properties and rules ranging from simple syntactical correctness rules to complex resource-constraint rules to make a ComiX file a well-formed specification.

Obviously, a ComiX specification must be syntactically correct XML. This property can be checked by any XML parser. Moreover, every consistent ComiX specification must be a *well-formed*

[20]Please note that this master handler is not visible for the application. It serves only to handle the access to the MEMORY OBJECT.

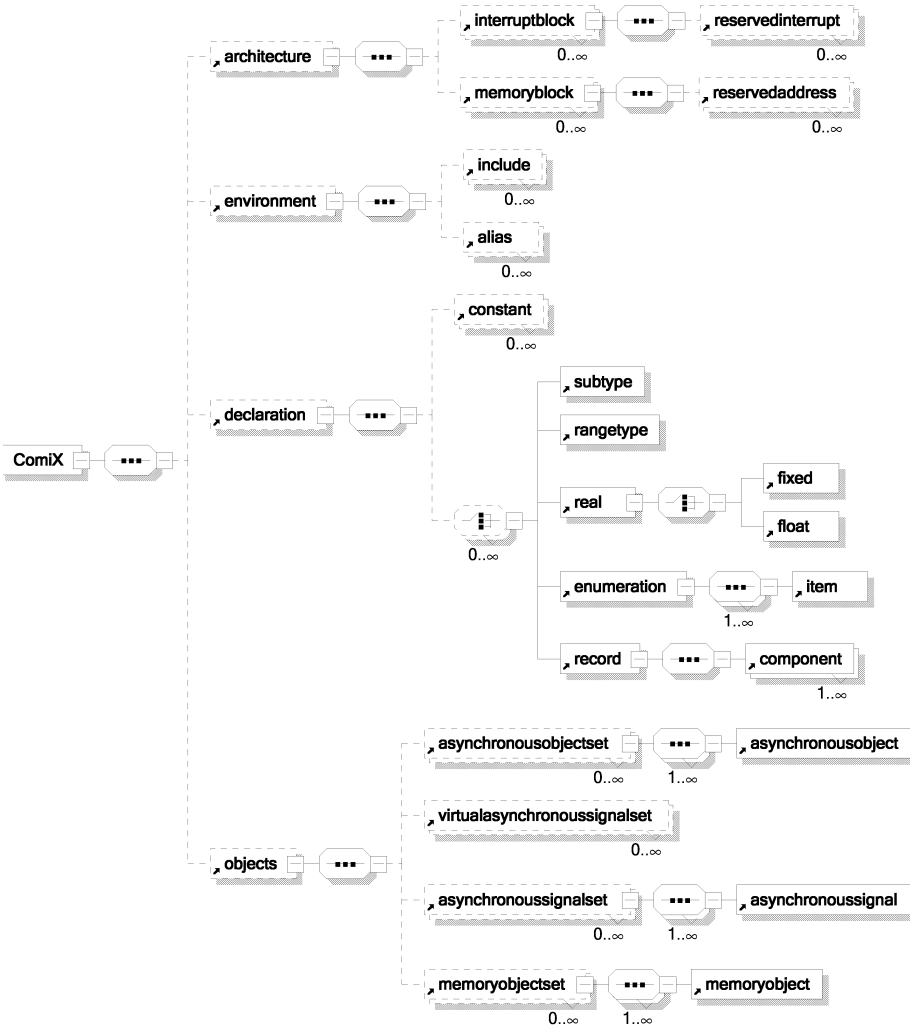

Figure 6.5: The ComiX DTD

ComiX document, that means it conforms to the grammar defined by the ComiX DTD (Appendix A). If a so-called *validating XML parser* is used, this property can be check with the parser as well[21].

Similar to most programming languages, syntactical correctness is not enough to guarantee a correct or consistent specification. Instead, it is required to check the semantic consistency of the specification to avoid contradictions within the specification. The following sections will define these consistency rules based on the formal definitions made in the previous sections to allow automated consistency checks based on the ComiX specification.

An alternative to consistency checks based on the ComiX specification would be to postpone this task to the compiler translating the target language code generated from the specification. The disadvantage of this approach would be, that error messages produced by the compiler are related to the target code and not to the ComiX specification. Thus, the designer would need to understand

[21] Note that many XML parsers ignore the DTD and thus accept every XML document. A validating XML parser like the JAXP parser used for the implementation of the code-generator in this thesis validates the document against its DTD.

the target code and the way it was generated. Since this is not desirable, an automatic checking mechanism implemented in the TEMPLIX language for the consistency of COMIX specification is part of the OOCOSIM method.

Some consistency rules apply only to one element or layer in COMIX. However, many rules apply to attributes taken from several elements located in different layers. For example, the size attribute of a type T (defined in Layer 3) must not be larger than the size attribute assigned to a MEMORY OBJECT (defined in Layer 4) of type T. Hence, the subsequent subsections organise consistency rules mainly by the attribute. Note that the short description given (in parenthesis) with each rule refer to a **violation of the rule**. To ease the formalisation of the following rules, a simplified notation (Definition 6.12) to refer to values of attributes in elements is used.

Definition 6.12 (Attribute Selection) *Let E be an element in $\mathcal{C}$ and α be an attribute of E then: $E \diamond \alpha$ refers to the value α of the element E.*

Most of the following rules are formalised based on the earlier given definitions of COMIX in Section 6.3. Additionally they are all illustrated by small (counter) examples with a COMIX code fragment violating the respective rule.

6.4.1 Size rules

The attribute `size`, which can be assigned to type elements or object elements defines the maximum amount of bits that the object or object-component of the type[22] in COMIX may occupy. Hence, the size attribute may but does not need be equal to the so-called *representation size* of an element[23]. The representation size for an element E, $\varrho(E)$, is equal to the number of bits the actual representation of E occupies in memory. For the types that can be defined in COMIX Layer 3, it is rather simple to compute the representation size[24]. For example, the representation size of enumeration types is determined by the size of the mandatory attribute `use`, which explicitly defines its representation or for a `rangetype` it is: $\varrho(rangetype) = \lceil log_2(max(rangetype) - min(rangetype) + 1) \rceil$.

Rule 6.1 (Representation size to small) *Let E be a COMIX element in a COMIX specification $\mathcal{C}$:*
$\mathcal{C}$ is consistent $\Rightarrow \forall E \in \mathcal{C} : \varrho(E) \leq E \diamond size$

```
<rangetype name   = "Temperature_T"
           range  = "-40 .. 120"
           size   = 7
           comment = "Temperature of the heat sink" />
```

Example (ComiX) 6.10: Invalid size attribute.

In Example 6.10 the given size of 7 bits is too small because the representation of the range -40 .. 120 (161 values) requires at least 8 bits. The following size rules are all related to record types and their components.

Rule 6.2 (Components greater than record) *Let T_R be a record type element defined in Layer 3 and $C_1, ..., C_n$ be its components with component types $T_{C_1}, ..., T_{C_n}$:*
$\mathcal{C}$ is consistent $\Rightarrow \forall T_R \in \mathcal{C} : T_R \diamond size \geq \sum T_{C_i} \diamond size$

[22] Strictly speaking, not types but objects instantiated of a type occupy memory. For simplicity-reasons in the following speaking of the size of a type will refer to the size a data-structure of that type.
[23] This is in particular useful in earlier phases of the interface design where object types often are changed or extended.
[24] Assuming that the compiler or synthesiser chooses the canonical representation of the data structure.

Rule 6.3 (Bitrange to small) *The size of component types must equal to the size of their bitrange representation for every component in a record. Let C be an arbitrary component of a record and T_C be the element defining the type of C:*
$\mathcal{C}$ is consistent $\Rightarrow \forall C : T_C \diamond size = \rho(C \diamond bitrange)$ *where ρ denotes the size of the bitrange.*

In the following example Example 6.11 the bitrange attributes allows only 5 bits for the component `Temperature_T`. However, the size attribute of that type (Example 6.10) requires 7 bits.

```
<component name="Temperature_Value"
    type="Temperature_T"
    unitnumber="0" init="0" bitrange="0 .. 4" />
```

Example (ComiX) 6.11: Bitrange too small violation.

Rule 6.4 (Overlapping components) *Components in a record type may not overlap. Let T_R be a record type element defined in Layer 3 and $C_1, ..., C_n$ be its components. Let S denote the size of a storage unit. Then:*

$\mathcal{C}$ is consistent $\Rightarrow \forall C_i, C_j$ *with* $i \neq j$:

$$\begin{bmatrix} C_i \diamond unitnumber \cdot S + \\ min(C_i \diamond bitrange) \end{bmatrix} \leq \begin{bmatrix} C_j \diamond unitnumber \cdot S + \\ min(C_j \diamond bitrange) \end{bmatrix}$$

$$\Rightarrow$$

$$\begin{bmatrix} C_i \diamond unitnumber \cdot S + \\ max(C_i \diamond bitrange) \end{bmatrix} < \begin{bmatrix} C_j \diamond unitnumber \cdot S + \\ min(C_j \diamond bitrange) \end{bmatrix},$$

where min(b) / max(b) denote the first and last bit of a bitrange b.
The idea behind this formula is rather simple: if the first bit of a component A is allocated before the first one of another component B, than also A's last bit must be allocated before the first bit of B.

Rule 6.5 (Layout size violation) *The components of a record type must be allocated within the given record allocation area*[25]*. Let T_R, $C_1, ..., C_n$, $T_{C_1}, ..., T_{C_n}$, and S be as defined in Rule 6.4:*
$\mathcal{C}$ is consistent $\Rightarrow \forall C_i : 0 \leq C_i \diamond unitnumber \cdot S + min(C_i \diamond bitrange) \wedge$
$C_i \diamond unitnumber \cdot S + max(C_i \diamond bitrange) < T_R \diamond size$

In the following ComiX fragment, the record allocation area of 16 bits would be large enough for the two components if allocated denser. But since the `Sensor_State` component is allocated at the end of the third storageunit, the record requires at least 24 bit to fulfil Rule 6.5.

```
<record name="Temperature_Sensor_RT" size="16">
    <component name="Temperature_Value"
        type="Temperature_T"
        unitnumber="0" init="0" bitrange="0 .. 7" />
    <component name="Sensor_State"
        type="State_T" private="no"
        unitnumber="2" init="0" bitrange="6 .. 7" />
</record>
```

Example (ComiX) 6.12: Layout size violation.

Please note that Rule 6.5 and Rule 6.4 imply Rule 6.2.

[25] The attributes unitnumber and bitrange in the component elements define the allocation layout in memory. The size of the record element defines the interval in which this allocation is legal - the so-called record allocation area.

6.4.2 Resource conflicts

Each COMMUNICATION OBJECT declared in Layer 4 allocates resources provided by the communication architecture. To simplify the following definition let us assume that the COMIX specification $\mathcal{C}$ contains only one $\mathcal{L}_1$[26]. The resources allocated by a COMMUNICATION OBJECT may be used exclusively and must be declared to be available in the Architecture Layer $\mathcal{L}_1$.

Interrupt in asynchronous signals and objects

Rule 6.6 (Interrupt not available) *The architecture must provide the interrupt for* ASYNCHRONOUS SIGNALS *and* ASYNCHRONOUS MEMORY OBJECTS*:*
Let S denote an ASYNCHRONOUS SIGNAL*, let A denote an* ASYNCHRONOUS MEMORY OBJECT *in Layer 4 then:*

$\mathcal{C}$ *is consistent* $\Rightarrow \forall S, A : S \diamond interrupt \in \mathfrak{I}, A \diamond interrupt \in \mathfrak{I}$,
with $\mathfrak{I}$ as defined in Equation 6.5.

Rule 6.7 (Multiple use of interrupts) *Every interrupt is associated to at most one asynchronous communication object:*
Let S, T denote an ASYNCHRONOUS SIGNAL *or an* ASYNCHRONOUS MEMORY OBJECTS *defined in Layer 4 then:*

$\mathcal{C}$ *is consistent* $\Rightarrow \forall S \neq T : S \diamond interrupt \neq T \diamond interrupt$

The following example Example 6.13 violates Rule 6.6 because interrupt 6 is not within the declared interval between 0 and 3. Moreover, it violates Rule 6.7 because interrupt 6 is used twice.

```
<interruptblock
    firstinterrupt="0"
    lastinterrupt="3" />
...
<asynchronoussignal name="Alert"
  interrupt="6" />
<asynchronoussignal name="InvalidSensorData"
  interrupt="6" />
```

Example (ComiX) 6.13: ASYNCHRONOUS SIGNALS violating two consistency rules.

Address and size of objects

Every MEMORY OBJECT and ASYNCHRONOUS MEMORY OBJECT occupies a number of storage units in the memory mapped I/O area. The exact location and amount is defined by its address and its size (Definition 6.13). Since both attributes (size and address) are used with the same static semantic in ASYNCHRONOUS MEMORY OBJECTS and MEMORY OBJECTS the following rules will not distinguish between the two object types and refer to them as *Storage Objects*.

Definition 6.13 (Storage units of Storage Objects) *The set of storage units $\mathfrak{S}_O$ allocated for a Storage Object O is defined as follows:* $\mathfrak{S}_O = \{s \in \mathbb{N} \mid s \geq O \diamond address \wedge s \leq O \diamond address + \lceil \frac{O \diamond size}{\mathcal{L}_1 \diamond storageunit} \rceil \}$

Rule 6.8 (Illegal address) *The address of every object must be legal; that is, at least being aligned and within the memory mapped I/O area.*

$\mathcal{C}$ *is consistent* $\Rightarrow \forall O \in \mathcal{L}_4 : O \diamond address \in \mathfrak{L}_A$,
with $\mathfrak{L}_A$ as defined in Equation 6.6.

[26] The following rules could easily be extended for multiple Layer 1 specification however at the cost of readability.

Rule 6.9 (Storage Objects in architecture resources) *Every memory object O must allocate only storage units that Architecture Layer 1 defines as available resources.*

$\mathcal{C}$ *is consistent* $\Rightarrow \forall O \in \mathcal{L}_4 : \mathfrak{S}_O \subseteq \mathfrak{M}$[27]

Rule 6.10 (Objects overlap) *Storage objects may not allocate overlapping storage units.*

$\mathcal{C}$ *is consistent* $\Rightarrow \forall O_1, O_2 \in \mathcal{L}_4 : \mathfrak{S}_{O_1} \cap \mathfrak{S}_{O_2} = \emptyset$

For the following example let us assume that `Temperature_Sensor_RT` is defined as in Example 6.12 but with a correct size value of 24.

```
 ...
 <memoryblock startaddress="8#104#"
   count="8" storageunit="byte"
   addressalignment="longword"
   bitorder="low"
   softwareendianness="big"
   hardwareendianness="big"
  <reservedaddress address="8#104#" />
 </memoryblock>
...
<memoryobject name="Engine1"
   type="Temperature_Sensor_RT" size="24"
   dataflow="sh" accessmode="atomic" protected="no"
   address="8#103#" />
<memoryobject name="Engine2"
  type="Temperature_Sensor_RT" size="24"
  dataflow="sh" accessmode="atomic" protected="no"
  address="8#104#" />
```

Example (ComiX) 6.14: Overlapping objects.

The example Example 6.14 illustrates several violations of the above rules:

- Rule 6.10: This example shows two MEMORY OBJECTS allocated at different addresses where the allocated storage units overlap. Since both objects allocate three storage units, which leads to a violation of Rule 6.10 (overlap at address 8#104#).
- Rule 6.8: This rule is violated by both MEMORY OBJECTS. `Engine1` is not within the architecture address space between 8#104# and 8#114# and the address of `Engine2` is not aligned on `longword`.
- Rule 6.9: Finally the address for `Engine2` is already reserved. Hence, this storage element is not in the set $\mathfrak{M}$ (as in Equation 6.3).

Rule 6.11 (Type size greater than object size) *The type size may not be greater than the object's size. Let O denote an Storage Object and T_O the respective type:*

$\mathcal{C}$ *is consistent* $\Rightarrow \forall O \in \mathcal{L}_4 : O \diamond size \geq T_O \diamond size$

The ratio to allow the object's size to be greater than the type's size is, that types may increase or decrease in size during the refinement process. With a stricter rule, demanding the equivalence of the size, every change in a type declaration would result in changes to all objects of this type.

[27] Please note that Rule 6.9 implies Rule 6.8. The rules however make sense to classify the consistency violation more exactly hence giving the designer a more exact indication for the mistake made.

6.4.3 Type rules

Rule 6.12 (Enumeration order) *Enumeration types must be defined in a way that their item representations are given in ascending order. Let T denote an enumeration type with items elements $I_1, .., I_n$ with $n \in \mathbb{N}, n > 1$.*

$$\mathcal{C} \text{ is consistent} \Rightarrow \forall j \in \mathbb{N}, j < n : I_j \diamond coding < I_{j+1} \diamond coding$$

The ratio for this rule is that the order of items in the enumeration items also defines the order-relation for values of the respective type.

The following ComiX segment (Example 6.15) shows the definition of the enumeration type `State_T` violating this rule, because the coding of the `BROKEN` item is larger than that of the other items.

```
<enumeration name="State_T" size="2">
  <item name="BROKEN"    coding="2"/>
  <item name="OK"        coding="0"/>
  <item name="UNKNOWN"   coding="1"/>
</enumeration>
```

Example (ComiX) 6.15: Order violation for enumeration types

Rule 6.13 (Well defined base type) *For each subtype element the basetype attribute must refer to a ComiX type. The range and the size must be smaller or equal than those of the base type. Let S denote a subtype of a rangetype T in a ComiX specification $\mathcal{C}$:*

$$\mathcal{C} \text{ is consistent} \Rightarrow S \diamond size \leq T \diamond size \wedge S \diamond range \sqsubseteq T \diamond range$$

6.4.4 Completeness of this rule set

The rules mentioned given in this section do not cover the rules intrinsically guaranteed by the DTD of ComiX. Not only syntactic correctness is guaranteed by the DTD. The attributes of elements in XML can be restricted to guarantee certain consistency properties. The `name` attribute of several elements is restricted to be an `ID`. This restriction guarantees already that the value of name is a unique identifier hence avoids identifier clashes. From the scientific point of view, however the formalisation of the missing rules is simple, provided the underlying specification is sufficiently formal and sound.

The formalisation in combination with the consistency rules offers the potential for a large degree of automation in the task of specifying a hardware/software interface. For many attributes in such a specification, consistent values can be computed automatically. As one example, the address of a MEMORY OBJECT can be determined by a tool, as long as the available resources are known. Since the access function to the interface is generated automatically (Section 6.5) from a single specification it can be guaranteed all access functions use the same physical address.

6.5 Automated code generation and consistency checks

An efficient process for the design and implementation of hardware/software interfaces should be maintained by the automation of consistency checks based on formal rules as described in the previous section and the automatic generation of code for the implementation of the interface. The code generation takes its complete information from a single specification - namely the ComiX document - and maps the COMMUNICATION OBJECTS to their respective implementations.

The naive approach to implement the required tool support would have been to write specialised code generators for the target languages (for example Ada95, VHDL and LaTeX [Kop92]). Such

an approach is depicted in Figure 6.6. The code generation would already be based on a single specification in ComiX. However, three individual tools produce target code in Ada95 (Generator A), VHDL (Generator B), and for the documentation of the interface LaTeX(Generator C). This approach has the following disadvantages:

- The effort to implement these generators increases with each new target language. Even if the implementation would be based on an existing generator, the support of a new target language would require to understand the entire tool, adopt its source-code and finally to recompile it.
- Large portions of the generator are responsible for parsing and editing the specification, generating output and other common functions necessary for all the generators. These components must be maintained for all separate generators. A library, which would support these common tasks and would allow re-using these components, could ease this problem to some degree.
- With isolated tools it would be rather difficult to keep mappings for hardware and software consistent. The tool maintainers for example have to make sure that changing a mapping for the software drivers is reflected in an adequate change on the hardware side.
- Finally, such an approach would not allow the designer to modify certain code generation mappings individually for a specific domain or application.

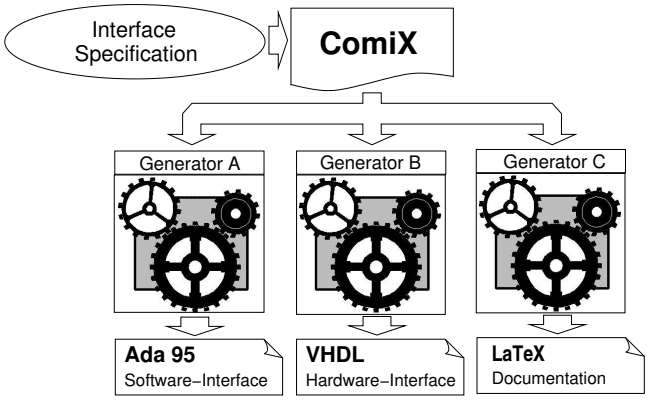

Figure 6.6: Individual code generators

Therefore, in the work at hand, the code generation as well as the consistency checks are based on hierarchical templates described in a language called TempliX(Templates in XML). Literally, a template set defines an algorithm or target code mapping to be applied on a ComiX document. From the programmers point of view, the ComiX document can be regarded as the input-data and the templates as the algorithm that is applied on the data. This approach results in a tool architecture as depicted in Figure 6.7. It shows the TempliX interpreter, or TempliX Abstract Machine (TAM) as it will be called in the following, which computes different target languages by executing different template set on the data extracted from a ComiX specifications[28].

6.5.1 TempliX language definition

Simplicity and orthogonality was the major design guideline for the definition of TempliX. Therefore, TempliX contains only a few and rather application specific language elements plus a mech-

[28] In this intuitive view, the TempliX abstract machine is an execution unit for TempliX.

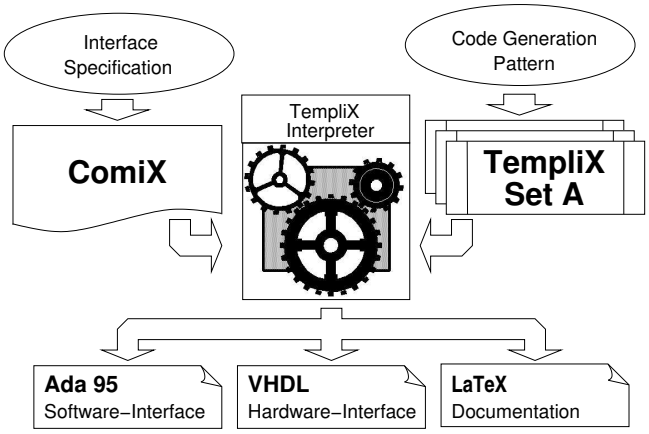

Figure 6.7: Programmable Generators

anism to extend the language with user defined library functions written in Java. TempliX has proven to be suitable to implement the code generation for three target languages and most of the consistency checks[29] from the previous section.

TempliX contains the following groups of language elements:

File creation: In most cases, the result of a template or a template set is a file containing the target language. Files are created by using the `template` element. The attribute `name` is defining the name of the template. `filename` and `fileext` define the name and extension of the output file. When the TAM reaches the end tag of the template element, the file is closed. There is always at most one open file in TempliX .

Output: For the code generation and to give feedback to the user, the following language elements write to the output file:

- `newline:` Writes a carriage return to the open file.
- `print, println:` Write the value of their attribute name to the file. `println` is equivalent to a `print` followed by `newline`.
- `indentfore, indentback:` To ease the task of creating pretty printed code, these elements add or subtract an indent level to the actual output position.
- `debug, error:` Write the value of their attribute name to the error or debug channel.

Control structures and tree traversal: TempliX provides only a minimum set of control structures. In contrast to general purpose programming languages (GPPL), it contains mainly tree-traversing language elements:

- Conditional Branching: With the elements `if`, `elseif`, `else`, and `donothing`[30] arbitrary branching control flows can be described.
- Sub-template branching: With the `&`*filename* operator the control flow can branch into a sub-template described in *filename*.
- Recursive Branching: The element `section` allows to define blocks. With the `callsection` element recursive control flows to the start of a section can be specified.

[29] Some rules where identified after closing the implementation phase of the tool.
[30] The `donothing` element is necessary, because the `else` elements in mandatory for every branching in TempliX .

- `find`: The find element is a very flexible language element. It allows searching the first element or attribute that matches a given expression. The search may start from the current position in the tree or from the root element. The attribute `depth` determines, whether the query is traversing through all sub-elements or is applied only at the given element.
- Loops: In contrast to GPPL, TempliX does not contain control loops with a loop condition. The only possible iteration is to apply all statement enclosed in a `forsibilings` element to a set of siblings[31] in the ComiX specification.
- Descending the tree: The `child` element descends after the start-tag from a parent to the leftmost child node. The end-tag reverses this step; that is, the control flow is at the point before the child element has been reached.

Data structures, Expressions The `assign` element allows to define local variables and assign values to these variables. Similar to XML TempliX contains no typing concept. The operators `%` and `&` allow accessing the values of variables or attributes.

Java language interface With these limited language elements alone some more sophisticated operations are difficult or even impossible. Therefore, TempliX provides a Java language interface. The element `check` allows to call a method defined in a Java class and to pass arbitrary parameters to this method. While this concept is very powerful, it is very rarely required for normal code generation where the basic TempliX elements are sufficient.

The following TempliX code example Example 6.16 tries to give the reader an impression on the way TempliX works in general. This toy example generates a file *Hello.txt*. It extracts the attribute `author` from the ComiX specification. Then it writes the sentence `Hello` *author*! to the open file. If the `author` attribute is left empty, it writes `Hello World!` instead. Then it branches into the sub-template described in `next_template`. Finally it closes the file indicated by the `</template>` statement.

```
<template name="World" filename="Hello" fileext="txt">
  <find expression="2" of="element"
   from="root" depth="false">
    <assign variable="who" value="World"/>
    <find expression="author" of="attribute"
      from="current" depth="false">
      <if expression="author" of="attribute" equal="">
          <donothing/>
        <else>
          <assign variable="who" value="?author"/>
        </else>
      </if>
    <print text="Hello %who!"/>
    <newline/>
    &next_template;
  </find>
</template>
```

Example (TempliX) 6.16: Hello world.

The next example illustrates the application of TempliX in the context of code generation. Example 6.17 shows a code fragment that generates the type definition for an enumeration type in Ada95. Let us assume the code generation has reached exactly the point where this code fragment is

[31] Siblings in this context denote all child elements of a single parent element.

applied to a ComiX enumeration type. It first prints the Ada95 keyword `type` followed by the name of the type and the keyword `is`. The enumeration items are defined by sub-elements in the ComiX document. Hence, `child` moves the TempliX scope to the children of the type definition. The application of the `forsibilings` loop construct leads to a comma separated list of the enumeration items' names in round brackets is printed.

```
...
<print text="type ?name is ("/>
<child>
  <forsibilings>
    <lastnode>
      <print text="?name"/>
    <else>
      <print text="?name, "/>
    </lastnode>
  </forsibilings>
</child>
<print text=");"/>

<print text="\textbf{for} ?name!'Size
             \textbf{use} ?size;"/>

...
```

Example (TempliX) 6.17: Enumeration type code template

6.5.2 Hierarchical Template Sets

The TempliX code to generate the target language code for a complete ComiX specification eventually becomes too complex for a single TempliX file. The key concept for the decomposition in TempliX is called hierarchical template set. A template set starts with a top-level template building the root of hierarchical tree of templates. For the top-level template, the so-called *entry point* is the root of the ComiX definition called `Comix`. With processing the TempliX code in the template, the ComiX tree is being traversed. The point to which the processing has advanced in the ComiX tree is called the *working point*. From any template, so-called *sub-templates* can be called. The entry point for a called sub-template is the working point of the caller. When a template is finished (including all called sub-templates), the working point always returns to the entry point of this particular template.

With hierarchical TempliX sets, it is even possible to call different template sets for each target language that needs to be generated from a single master root template.

6.6 Implementation aspects

The Java implementation of a tool called DESHICO for the specification of hardware/software interfaces with ComiX and the related code-generation with TempliX has been done in a diploma thesis [Zha01] supervised by Prof. Dr.-Ing. W. Nebel and the author of this thesis.

The screen-shot depicted in Figure 6.8 is meant to give the reader an impression of the graphical user interface build around the ComiX language and the code generation based on TempliX.

The DESHICO tool implements the consistency checks from Section 6.4 as well as a code generation for Ada95 (software), VHDL (hardware), and LaTeX(documentation). The following section will elaborate on the general mappings principles applied for the code generation.

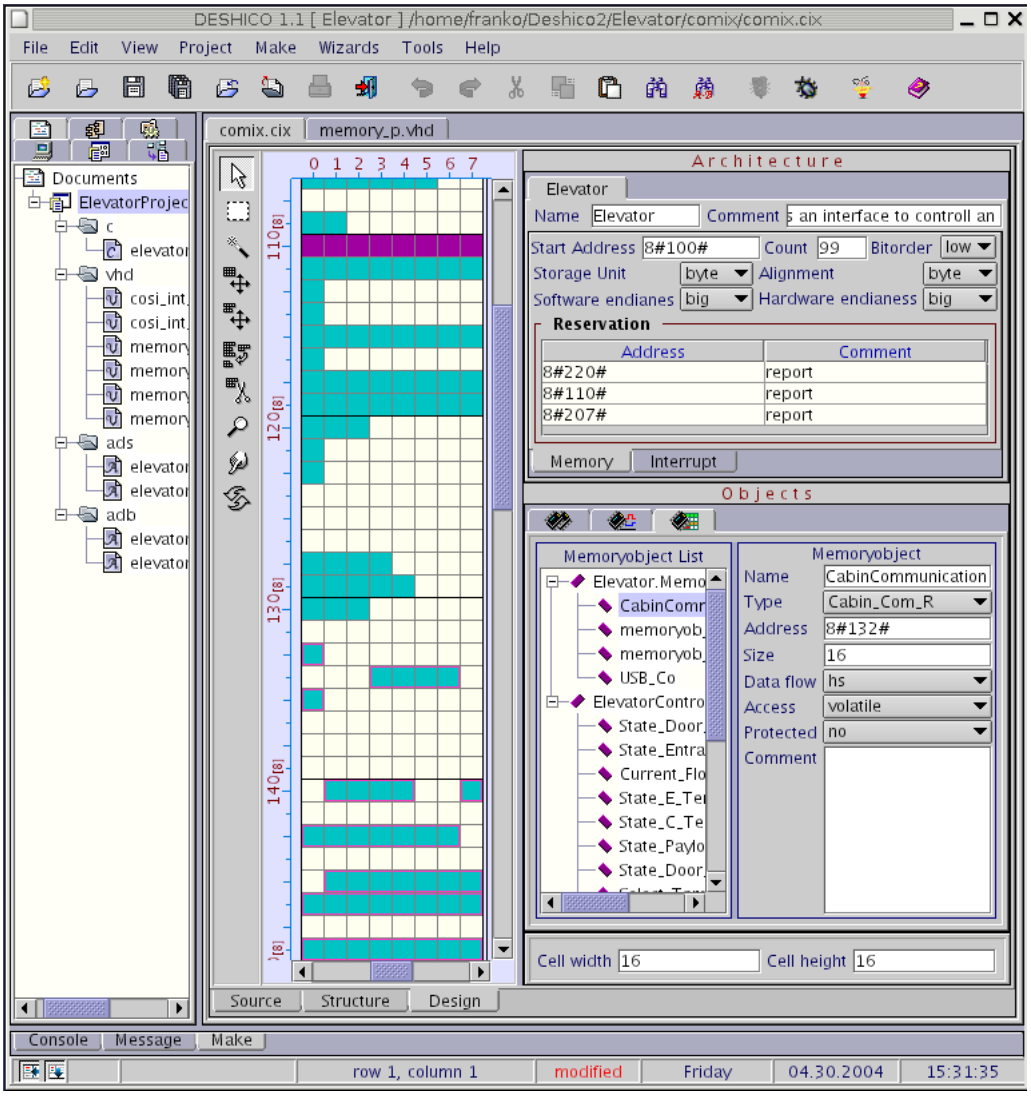

Figure 6.8: A screen-shot from the Deshico tool

6.7 Code-generation for Ada95 and VHDL

For a seamless design methodology, it is essential to support automatic transformations from an abstract level to the next lower level. This requires translation schemes from ComiX specifications into the target languages for documentation, hardware, and software. The concrete templates for these mappings contain about 4400 lines of TempliX code. Therefore, in this work only the general idea behind the mapping can be introduced.

The abstract goal of the code generation is the same for hardware and software, which is to provide a method interface that allows accessing the communication objects. The methods for each object are defined in a code-package with the objects' name. The set of methods for each communication object depends on its type:

- memory objects and asynchronous memory objects: For each component in the record

type a `set_componentname` and a `get_componentname` method is defined. Depending on the data-flow direction, only set or get method are made public (Table 5.1).

- ASYNCHRONOUS SIGNALS: The hardware side contains only the method `invoke` to activate the signal. The software side contains the event handler, which is hidden in the object.

Object sets are each translated into a package that contains the packages for all objects in the object set. The following will briefly describe the target language specific mappings for COMMUNICATION OBJECTS in software, and hardware.

6.7.1 Mapping of ComiX to Ada95

Ada95 contains already a complete set of language primitives to describe memory mapped I/O and interrupt based communication. Hence, most of the attributes found in the specification for individual COMMUNICATION OBJECT can be naturally mapped onto Ada95 language-primitives.

Let m be a MEMORY OBJECT[32] of record type R with two components $c1, c2$ of type $t1, t2$, then the code generation must perform tree tasks:

Declare component types t1, t2: The transformation of component types in ComiX into Ada95 is rather strait-forward. Every required[33] type element (Layer 3) is translated into a type declaration. The `coding` attributes in the enumeration items is translated into a representation clause for the Ada95 enumeration type.

```
<rangetype name="t2"
  range="0 .. 127"
  size="7" />
<enumeration name="t1"
  size="1">
  <item name="ZERO"
    coding="0" />
  <item name="ONE"
    coding="1" />
</enumeration>
```

ComiX 6.18: Range and enumeration type.

⇒

```
type t1 is (ZERO, ONE);
...
for t1 use
  (ZERO => 0, ONE => 1);
...
type t2 is new Integer
  range 0 .. 127;
```

Ada95 6.19: Declaration in software.

Declare record type R: For the record type R, a specific type definition is generated using the Ada95 representation clause for record types as shown in the following example.

[32] Due to the similarities in principle with the other object types this thesis describes only the mapping for MEMORY OBJECT .

[33] Note that the ComiX specification may contain type elements that are not used by any COMMUNICATION OBJECT. For these types, code does not need to be generated.

```
<record
   name="R"
   size="7">
 <component name="c1"
    type="t1" init="ZERO"
    unitnumber="0"
    bitrange="7 .. 7" />

  <component name="c2"
     type="t2" init="0"
     unitnumber="0"
     bitrange="0 .. 6"/>
</record>
```

ComiX 6.20: Record type R.

⇒

```
type R is
record
   c1 : t1 := ZERO;
   c2 : t2 := 0;
end record;

for R use
record
   c1 at 0 range 7 .. 7;
   c2 at 0 range 0 .. 6;
end record;
```

Ada95 6.21: Generated for R.

Create data-structure in memory: Having defined the types for the data-structure, it can now be allocated in the memory-mapped I/O area. This again is achieved by simply using the representation clauses in Ada95. The ComiX attributes `address` and `size` can be mapped one-to-one onto the respective Ada95 attributes.

```
<memoryobject name="M"
 type="R"   size="8"
 dataflow="sh"
 address="8#110#" />
```

ComiX 6.22: Memory object M.

⇒

```
for R'size use 8;
M: R;
for M'Address
   use 8#110#;
...
function set_c1(v:t1) is
    M.c1 := t1;
 ...
```

Ada95 6.23: Generated for M.

6.7.2 Mapping of ComiX to VHDL

Since VHDL lacks a concept like the representation clauses in Ada95, code generation is here a more difficult. Where in Ada95 the compiler is responsible for the encoding data into the appropriate bit-pattern, for VHDL this task must be taken over by the code generation.

However, large sections of the generated code look similar. The following code fragment for example is the result of the code generation in DESHICO for the component type declarations $t1, t2$ taken from Example 6.18. The attribute ENUM_ENCODING in this fragment is a pragma to force the synthesis process to use the same encoding as specified in the ComiX type.

```
attribute ENUM_ENCODING : string;
type t1 is (ZERO, ONE);
attribute ENUM_ENCODING of t1 : type is "0 1";

subtype t2 is (0 to 127);
```

Listing (VHDL) 6.24: Enumencoding for synthesis.

Since further discussion of the more complicated code generation mechanisms would not provide a significantly deeper insight in the methodology, they are neglected here.

6.8 Recap

This chapter presented the following key concepts:

1. A formal and hence analysable interface specification language called ComiX.
2. The notion of a consistent ComiX specification has been defined by a set of consistency rules, introduced in Section 6.4.
3. A language TempliX to support the flexible implementation of code-generation and consistency checks.
4. Finally, the transformation of a ComiX specification into hardware and software has been sketched in the last section.

With these concepts at hand, oocosim is capable to specify, analyse, and implement hardware/-software interfaces in a very efficient and reliable way.

7 Co-simulation

This chapter describes the techniques used to co-simulate the functional and temporal behaviour of software/hardware systems in OOCOSIM. The arguments for the co-simulation of hardware and software, which is providing an executable specification of the system, are manifold. The dynamic behavioural model of the hardware/software system in interaction with its environment model provides the necessary insight to allow for an in-depth analysis of complex designs.

'Software alone does not hurt anybody - it needs assistance from some hardware device'. In particular, the collaboration of different components in the system can be analysed only in a co-simulation. If for example, the hardware delivers slightly wrong sensor values, which is not critical as such, but the resulting software action may cause a chain-reaction leading to a disaster. This was the case in the Ariane 5 accident. A type-conversion overflow was caused by a sensor value that was out of its specified range. The overflow exception led to the shutdown of the primary controller because a hardware failure was assumed. Since the redundant second controller was an exact replication of the first it had to shutdown as well. The rocket was then out of control and thus self-destruction was initiated.

Specifications are not always complete and adequate. Hence, even if one can formally verify that an embedded system fulfils a specification, this cannot guarantee save behaviour. It is often necessary to observe the dynamic behaviour under as realistic circumstances as possible in a virtual prototype.[1]

A simulation is always based on a model of the system to be built and its real environment. There are two general problems, associated with the validation based on such a model.

One is that tests are inherently incomplete. For the type of systems addressed here, we can assume that complete and exhaustive testing is virtually impossible. Apart form the numerous execution paths the software might take, the real-time behaviour and complex stimulus generated by the environment makes complete testing an almost infeasible task. Well-defined testing strategies, starting with the first executable model and ending with the operational test of the implemented system can help to guarantee that at least most requirements are checked.

The other problem lies in the difference between the simulation model and the real system. These differences may result from the reduced simulation model. In particular, the environment is often difficult to simulate, because it typically includes analogue components[2] that are discretised for the simulation. A further source for differences is the execution or target platform for the embedded system in contrast to the simulation system executing on a workstation. While the functional behaviour of the software can be defined almost target independently through the programming languages, the real-time behaviour of the software is usually not addressed adequately. Since for reactive systems, the real-time behaviour is often essential for their correctness (Section 4.5.5), one big challenge for the hardware/software co-simulation lies in the correct simulation and in particular the synchronisation of temporal aspects of the combined hardware/software system. This thesis presents an approach, which enables the co-simulation of a real-time system based on a target independent source-code level model.

Finally, the analysis of hardware/software interfaces needs to be addressed by a co-simulation method. The interface components of the embedded system must be modelled carefully since errors in the interface are often the source of severe malfunctions of the system. The previous chapter provides means to achieve a consistent interface specification. Since this specification cannot guarantee the

[1] In the crane controller case study with OOCOSIM such an incomplete/erroneous specification could be identified by a detailed analysis of the simulated behaviour (Section 8.1.1).

[2] Real-world objects unfortunately very rarely show a discrete behaviour.

functional and temporal correct use of the interface within the application, the dynamic behaviour and usage of interface components must be validated in the co-simulation.

For the design-flow introduced in Chapter 5, co-simulation plays an important role. Since the high-level specification in HRT-HOOD+ is not executable as such, the co-simulation serves as the executable specification of the embedded system. Thus, co-simulation allows testing an embedded system early in the design process, fulfilling the requirement stated in Section 4.5.2. Moreover, the approach proposed here allows the immediate implementation of the co-simulation model. Therefore, it can be characterised as executable **and** implementable specification. To support a seamless design flow the co-simulation model must be the basis for the implementation - in other words the code-base for simulation and synthesis must be the same. OOCOSIM supports this requirement (Section 4.5.1) by providing a translation tool that is transforming the implementation model automatically into the co-simulation model (Section 7.5.3).

As mentioned above, a seamless design flow requires a simulation model containing the implementable specifications of hardware, software, and interfaces. The large majority of hardware designs in industrial practice, base on a specification in a hardware description language like VHDL or Verilog. Such an approach requires the integration of the software model and the definition of the synchronisation between the hardware and the software model. Consequently, the classification of co-simulation approaches in Section 7.2 within the narrower scope of this thesis concentrates on these two aspects.

Since for this thesis mainly the co-simulation of software described in Ada95 and digital hardware described in VHDL is relevant, a unified simulation semantic for theses languages will be discussed in Section 7.3. The main challenge here is to achieve simulation behaviour of a source-code model that matches the functional and temporal behaviour of the embedded system's implementation.

In order to implement the embedded system, the software parts of the models written in Ada95 will be cross-compiled for the target processor, while the hardware parts written in VHDL will be synthesised for the ASIC or FPGA technology chosen. The key for a useful co-simulation is that each individual component as well as the interaction between components in the co-simulation behaves semantically equivalent to the implementation. It is important to note that semantically equivalence includes functional and real-time behaviour at the desired level of abstraction.

In Section 7.5, implementation aspects of the co-simulation are discussed. Finally, Section 7.6 recapitulates the chapter.

7.1 Classification criteria

For the comparison of different software models and synchronisation mechanisms, the following aspects will be evaluated in the following sections:

The representation of time: This aspect describes the way in which real-time is represented in the software model - in other word the discrete steps in which time advances in the software model. For the classification of software models and the synchronisation between the sub-models, the following levels of granularity will be distinguished:

- Non: That means time is not specified in the software model and it is therefore pure functional. The synchronisation of the software model with the inherently timed hardware model can only be based on function calls between the sub-models.
- RTOS: The software model uses a coarse-grain timing model. It resembles the timing behaviour of the software as specified by concepts from the underlying real-time operating system (RTOS). Section 7.4.2 will define this in detail for the software model in OOCOSIM.
- Instruction: A step in the software model corresponds to the execution of one instruction in the assembler code of the compiled software.

- Cycle: In this timing model, a step in the software model tyically corresponds to one clock-cycle of the processor. It is however possible, to define a different simulation cycle. In such cases, the cycle length is usually chosen as an integer multiples of the processor clock.
- DE (discrete event): A step in the software model corresponds to a discrete event in the processor model executing the software. Therefore, depending on the processor model, it can be arbitrarily small.

Synchronisation of heterogeneous models: In a real-world embedded system[3], hardware and software share a common time and have access to identical interface resources often implemented by shared memory and interrupts.

In a homogenous model, as for example a system model completely specified in an HDL, the synchronisation is trivial, since all parts of the model naturally execute using a single model of computation. Consequently, they (as the real-world embedded system) share a common model of time and use the same model of interface resources.

Unfortunately, without further effort this is not true for heterogeneous models, that means in a co-simulation containing separate hardware and software models[4]. The software model would advance only depending on the performance of the workstation it executes upon while the hardware simulation would depend on the performance of the hardware simulator. Obviously, their unrelated progress would not be as intended. Therefore, it is necessary to synchronise the temporal behaviour of hardware simulation and software execution using so-called *synchronisation events*. Furthermore, the synchronisation mechanism is responsible for the consistent behaviour of the interface model. Hence, the sub-models exchange the state of the interface model typically at the synchronisation events.

Relative performance: The relative performance P of a model describes the ratio between the simulated *model-time* and the *simulation-time*.

More formally:

$$P = \frac{t_{model}}{t_{host}} \tag{7.1}$$

The model-time is determined by the performance of the hardware model, the software model, the communication, and the synchronisation[5]:

$$t_{model} = t_{soft} + t_{hard} + t_{comm} + t_{sync} \tag{7.2}$$

Note that the above Equation 7.2) assumes a heterogeneous simulation model running on a single CPU. In a homogenous simulation model running on a single CPU, t_{comm} and t_{sync} can be neglected.

$$t_{model} = max(t_{M_1}, ..., t_{M_n}) + t_{comm} + t_{sync} \tag{7.3}$$

Obviously, the relative performance of an approach depends on the computation power of the host machine used for the simulation. Hence, comparisons between different approaches are only valid if the same simulation host is used for both approaches. Due to higher communication latencies, distributed simulation models often show a significant higher synchronization time than local models. Consequently, distributed simulation models should be applied, where the synchronization is loose enough.

[3] The physical embedded system as the result of the development process is called *real-world embedded system* in contrast to the simulated one.

[4] Since the hardware model is executed in an HDL simulator, while the software is executed in a separate process, two independent processes are necessary to execute the respective models.

[5] Communication refers here to the effort required to process the interface data before it is transferred to the other model. Synchronisation refers to the effort incurred by the data transfer and the blocking and unblocking of the models.

Model accuracy: Here means, how exact the simulation reflects the behaviour of the implementation. In general, improved accuracy can only be achieved at the cost of performance, as it requires a more detailed model. Therefore, it is essential for the software model to reflect all and only the relevant aspects. Since the relevant aspects as well as the level of detail differ between applications or design phases, it is impossible to define the 'always right level of accuracy'. Instead, the co-simulation environment should be flexible enough to allow different levels of detail in the model.

Model availability: An important aspect for a universally valid methodology. If the co-simulation approach restricts the potential target platforms, for example by proprietary processor models, then these restrictions must be acceptable for the targeted application domain.

7.2 Execution models for the software part

Co-simulation is a well-investigated research topic. Several approaches have been proposed to enable co-simulation in different application domains and at various levels of abstraction. Chapter 3 has already given a general overview of the related works in this field.

Many of these approaches use an HDL-simulator for the application specific hardware and combine/synchronize them with software execution models. For these approaches, three subtypes can be classified by the processor model or the software execution model:

Full processor hardware model: For this approach, the full system model together with the hardware model[6], is executed in a HDL-simulator. The system model therefore contains a processor model of the target platform, which is used to execute the software model.

The abstraction level for this type of processor models can be very low, describing the processors behaviour at register-transfer level, or rather abstract, representing only the functionality and the estimated real-time behaviour. Provided the processor model is correct and detailed enough, the cycle-accurate behaviour of the system can be simulated. Even internal registers of the processor, intermediate computational results, or the I/O behaviour at the ports can be observed at bit-accurate level.

The major disadvantages of this approach are the poor performance of such a simulation model and the need to have available specific hardware models for the potential target platforms. Since detailed processor models are not freely available this results in a low model availability[7]. To achieve a fully time-synchronous behaviour of the system model, the processor specification must at least reflect the functional behaviour at a cycle-accurate abstraction level[8]. Since modern embedded processors often run at hundreds of megahertz, a simulation of the detailed behaviour (including for example pipelining and caches) is extremely time consuming.

The poor performance (Table 7.1) makes it almost impossible to simulate the embedded application as a whole. Only a few seconds typically take hours of simulation-time. For example, an RT-level VHDL model of a 80c32 at only 20 MHz needs about 1050 sec of simulation-time[9] to simulate about 1 sec of software model-time. Note, that an 80c32 microcontroller has a simple architecture compared to modern embedded cores.

The poor performance is contradictory to the requirement stated in Section 4.5.4. More abstract model can ease the performance problem at the cost of lower model accuracy. With higher abstraction, the timing behaviour of the processor model can no longer be (at cycle-level) synchronous

[6] Please remember that the term hardware model in this thesis refers only to the application specific (in particular non-processor) hardware.

[7] Processor manufactures are very reluctant to publish HDL-models of their cores, because with such models it is rather simple for competitors to analyse and copy key ideas of a CPU.

[8] Otherwise the synchronisation with the cycle-oriented hardware model is problematic.

[9] Executed on a SPARC Solaris 5.8 workstation with 500 MHz.

with the hardware. The more abstract the HDL-models becomes, the closer they get (in terms of performance and accuracy) to instruction set simulators.

Software model on an instruction set simulator (ISS): To overcome some problems with the previous approach, highly optimised simulators for particular processors can be used to execute the software model. These simulators called *instruction set simulators* typically model the behaviour of the processors at cycle-level or instruction-level accuracy. The advantage of such a more abstract model is a significant higher performance than HDL based processor models can offer. Recent publications like [NBS+02, RBMD03, RMD03] report a simulation performance of 8 to 12 MIPS (million instructions per second) for their retargetable ISS of an ARM7 and SPARC V7 processors at instruction level accuracy. While this is much faster than typical HDL based models, it is still about a hundred times slower than the host machine[10].

In such a co-simulation, the hardware simulator still simulates the application specific hardware, but the software model is executed using the ISS for a specific target processor. Since the ISS process is running independently from the HDL simulation, the hardware model and software model need to be synchronised. The co-simulation system therefore defines synchronisation points for both models. Usually these synchronisation points are chosen as multiples of the system-clock or instruction executions. Both sub-models execute, until the synchronisation point is reached. In a distributed simulation, hardware and software can even execute in parallel. Then the hardware/software interface is updated before the next simulation-cycle starts.

Depending on the framework, the hardware/software interface model can be modelled using either abstract communication channels or low-level models. The CoWare N2C™, as a prominent commercial example, allows defining a memory map for the interface and Mentor CVS™allows mixing channels at different abstraction levels.

While this approach significantly speeds up the system simulation, the performance of ISS is still far lower than modern embedded processors. Moreover, many co-design frameworks support only a small set of embedded processors. With such restrictions, it is difficult to achieve a target independent design flow, required to enable full design space exploration (Section 4.5.3).

Recent developments, like the LISA language [RWT], enable the automatic generation of instruction set simulators based on an abstract processor specification. These approaches, if widely adopted, can probably help mastering problem of diversity of embedded processors.

Host code execution (HCE): In this approach, also known as *compiled code execution*, the software source-code model is compiled for the simulation host machine. Naturally, this delivers the best performance among the here presented approaches. Since the host computer is usually faster than the target platform, the performance in general will be higher than the performance of the target system.

Within a co-simulation system, the software model technically can be synchronized with the hardware model through a call-mechanism like remote procedure calls (RPC), which means at functional level.

There are several disadvantages associated with this approach:

- The processor internal state is not visible, as the HCE model does not contain the target processor model.
- The temporal behaviour is different. This includes not only the performance but also the scheduling behaviour of the underlying operating system.
- Hardware/software interfaces cannot be modelled adequately because the host machine does not support the same interface facilities like memory mapped I/O or interrupts as the target platform.

[10] Noll et al. [NBS+02] reach about 8 MIPS on a Athlon 1.2 GHz. Reshadi, Mishra, and Dutt [RBMD03] reach up to 12 MIPS on a Pentium III 1.0 GHz for common benchmarks.

Consequently, such models can be used only for early functional tests. With a HCE model as such, nothing meaningful can be said about the hardware/software interaction or performance of the integrated system. It is however possible to augment the HCE model with mechanism to simulate the operation system or the real-time behaviour as shown in the co-simulation approach presented in this thesis.

Bus functional model (BFM): While the previous approach only models the functional behaviour, the so-called *bus functional model* concentrates on the evaluation of the correct bus and the I/O behaviour.

> ***Bus Functional Models of Processors and Buses*** *Bus-functional models of processors, controllers, and buses are used for early hardware debug. The models simulate the bus cycles of the device or bus, providing a complete model of the pins, cycles and on-chip-functions – everything except the instruction core. (from :Synopsys Inc., http://www.synopsys.com/products/lm/swmodel_ds.html)*

The functional behaviour is modelled here in the most abstract way. The BFM usually describes a timed sequence of signal transitions on the bus-interface of the processor - typically specified in a command language or a C-subset. The benefit of co-simulation systems using a BFM is the possibility of a detailed validation of the interaction between the processor and its environment through busses. Since such a model is rather a (meta-)testbench-like substitute of a full software model, this approach is not in the central scope of this thesis.

Quantitative comparison: With the categories mentioned above, it is possible to classify processor models. Since the classification is rather coarse, every class contains a large variation of concrete models. These models differ in many aspects. In particular, the performance of models shows a significant variation, as for example the complexity of the processor architecture and the efficiency of the model implementation have impact on its speed. Due to the large variations, it is difficult to compare the classes of approaches on a quantitative scale. Table 7.1 provides therefore only a rough overview with respect to their timing model, their performance class, and the scope in which they are typically applied.

Attribute/ SW-Model	**Timing**					**Perfor.**	**Scope/Size**
	Non	RTOS	Instr.	Cycle	DE		
HCE	✓					++	Functional, large
BFM				✓	✓	−	Bus I/O, small or medium
ISS			✓	✓		◦	Functional, medium
HDL				✓	✓	−−	RT-Level, small
OOCOSIM		✓				+	Functional, large

Table 7.1: Comparison of processor models in co-simulation (part 1).

Table 7.2 supplements the previous table with a comparison with respect to accuracy, availability,

and the interface resource model.

Attribute	Accuracy	Availabilty	Interface ressource model
HCE	−−	++	abstr. values in RPC
BFM	(+)	−−	low-level physical
ISS	+	−	(abstr.) memory, interrupt signals
HDL	++	−−	low-level physical
OOCOSIM	+	++	memory array, async. interrupt signals

Table 7.2: Comparison of processor models in co-simulation (part 2).

Recap Regarding these aspects, the above approaches all have their benefits and drawback. The host-code execution is fast but not accurate enough to allow for an analysis of the temporal and the interface behaviour. This is contradictory to the requirements stated in Section 4.5.5 and 4.5.6. HDL based CPU models provide a very high model-accuracy, but their poor performance does not allow to validate real complex hardware/software systems. This is a violation of the requirement stated in Section 4.5.4. Co-simulation based on ISS models offer a better performance but still are often to slow for complex embedded systems. Moreover, fast models are (at best) available for mainstream embedded processors. Bus-functional models allow analysing the physical interface of processors and busses very well but due to their restricted applicability do not represent a vital alternative for a complete functional validation.

7.3 Co-simulation in OOCOSIM

The overall goal of the co-simulation in OOCOSIM is to provide a so-called *time-synchronous executable model* of a heterogeneous system model. Time-synchronous in this context means that the synchronisation mechanism in the co-simulation is based on real-time synchronisation events in hardware and software in contrast to a purely functional synchronisation.

More concretely, the system model in OOCOSIM contains a software model written in Ada95 using its real-time annex[11] and a hardware model written in VHDL. These models communicate via an interface model representing an abstract model of memory mapped I/O and interrupts.

The COMMUNICATION OBJECTS as described in Section 5.3.6 implemented in a real-world embedded system must have access to physical interrupts and objects in the physical memory. On the simulation host system these resources may not be existent and if, access to these could not be granted[12]. Hence, it is necessary to provide a mechanism, which simulates the behaviour of the communication architecture (Section 6.3.2) - in other word an interface model.

From these facts is obvious that the interface models for simulation and for implementation cannot be identical. The models for real-world hardware and software, mapped via compilation and synthesis onto the target system, will be called *implementation models*.

For the co-simulation a few modifications at source-code level regarding operations affecting synchronisation and the hardware/software interface behaviour are necessary. The modified model will

[11] The real-time annex is standardised with the Ada95 language.

[12] The target system may have more or other interrupts than the host system. Moreover, direct access to physical memory resource would usually lead to a segmentation fault.

be called *co-simulation model*. Naturally, the goal is to guarantee that the co-simulation model shows functionally[13] and temporally the same behaviour as the implementation model.

The real-time behaviour of the software implementation model in general depends on the software target architecture; that is, it depends on the performance of the processor, the memory system, and others. To solve this problem, an implementation independent timing model for the software-simulation will be defined based on the Ada95 real-time annex. This standardised annex contains statements that specify the real-time behaviour of the source-code model independently from the target architecture. The real-time annex provides similar concepts to an RTOS (scheduling, task suspension), which are than mapped by the Ada95 compiler onto the target runtime system.

Before presenting the co-simulation in detail, a brief overview or the 'big picture' will be given in the following section. Section 7.5 introduces the techniques implementing the co-simulation system in OOCOSIM.

7.3.1 Overview

The co-simulation in OOCOSIM advances in so-called *co-simulation cycles (CSC)* as depicted in Figure 7.1[14]. Every co-simulation cycle contains a *Hardware Execution Phase (HEP)* and a *Software Execution Phase (SEP)*. While the co-simulation advances in one phase, the other phase is blocked. Furthermore, an *Initial Synchronisation Phase (ISP)* is needed, which is executed before the simulation enters the first HEP or SEP to achieve a clearly defined initial state for the co-simulation. The

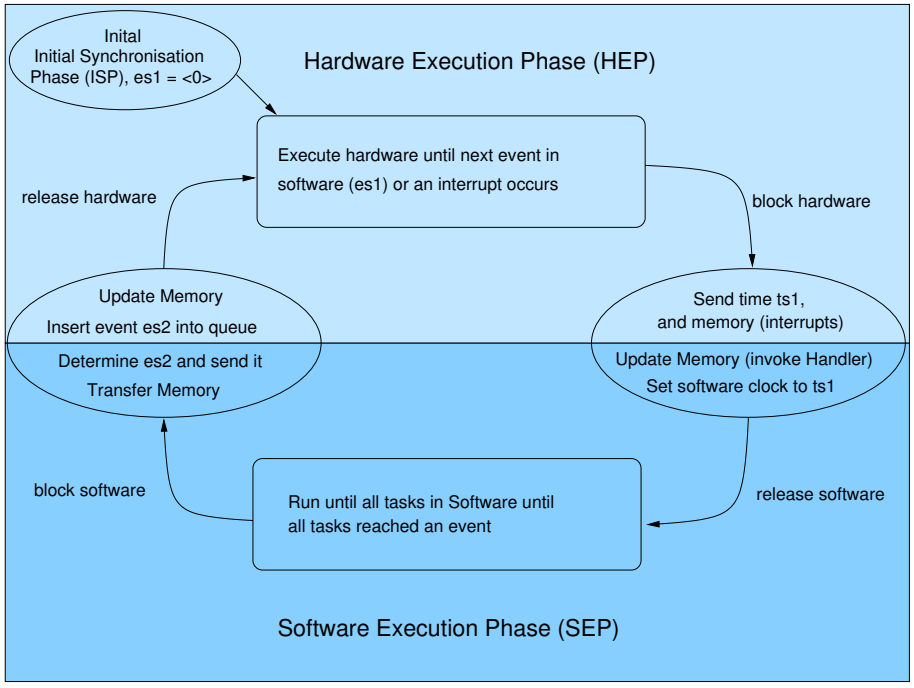

Figure 7.1: The co-simulation cycle

co-simulation execution semantic depends to a large extend on the notion of *events*. While they will

[13] This includes the interface behaviour.
[14] Please note that the figure shows only the ISP and the first simulation cycle of the co-simulation.

be introduced formally in the following subsections for this overview, it will be sufficient to regard events as 'defined points in model-time'.

Initial Synchronisation Phase (ISP)

Before the co-simulation enters the first simulation-cycle, exactly one ISP sets the co-simulation model into a defined state. In particular, it sets the simulation model-time to zero and the interface becomes initialised. In the software model all task are initialised and then started. The co-simulation system determines the earliest software synchronisation event e_{S_1}.

Then hardware simulation model receives this first synchronisation event e_{S_1}. The execution of the software will be blocked until the first HEP is finished. Now the co-simulation can enter the first simulation cycle with the first HEP.

Hardware Execution Phase (HEP)

The hardware simulation model advances until the earliest software synchronisation event; that is, until the model-time reaches the first software event or an interrupt occurs[15]. If e_{S_i} is reached, the state of the memory interface and the actual model-time (t_{S_i}) are transferred to the software model. Then the simulation enters the SEP. By updating the model-time in the software model at least one task becomes active[16]

If an interrupt has occurred the actual model-time (t_{S_a}) will be transferred and the asynchronous event will be signalled to the software model. This will release the associated interrupt service routine in the SEP to handle the event.

Software Execution Phase (SEP)

Since there are two ways possible, to enter the SEP (synchronisation event or asynchronous event) there are consequently two ways to execute the SEP:

1. If the SEP was released by an interrupt, the model-time is set to e_{S_a}. The handler associated with the handler is executed. The handler may possibly release some previously block tasks, which can generate new synchronisation events.

2. If the SEP was unblocked by a synchronisation event, after updating the memory interface, the software model will be unblocked by setting its clock to e_{S_i}, that is the time of the actual synchronisation event. Now the software model executes until all tasks are blocked again by reaching synchronisation events or other blocking calls, for example calls to protected objects.

The earliest synchronisation event $e_{S_{next}}$ will be determined by the simulation system and transferred to the hardware model together with the state of the memory interface. $e_{S_{next}}$ is the earliest synchronisation event after either e_{S_a} or e_{S_i}. Now this SEP ends by blocking the software model and transfers the control to the hardware model. Now the next cycle can start.

7.4 Temporal synchronisation

The above overview already introduced the basic concept of the synchronisation of the hardware and the software model in the co-simulation integrated in the OOCOSIM design flow. This section will define in detail the underlying timing model that is required to adopt the discrete event simulation concept to the software model execution.

First, the notion of time in a VHDL model will be described. Second, a similar model for the software, based on the real-time annex of Ada95, will be defined. Finally, a mechanism to integrate these timing models into one consistent timing model for the co-simulation will be introduced.

[15] An interrupt marks an asynchronous event.
[16] This is guaranteed by definition of the software events.

7.4.1 Time in the VHDL model

The hardware simulation model is taken as is for the co-simulation. Hence, it will be introduced here only briefly. For a detailed description, the reader might be referred to [Ash95].

A VHDL model can be seen as a collection of processes, which are sensitive to events on certain connecting signals. Whenever such an event occurs, it will stimulate every sensitive process. The stimulated process will then execute and eventually schedule value changes to signals later in the simulation. Such a scheduled assignment to a signal in the future is called *transaction* (Definition 7.1). If the signal value changes through the transaction, it is called an *event* (Definition 7.1).

During the initialisation of the VHDL model at model-time zero, each process becomes active and executes its sequential statements, which may schedule transactions for the future. When all processes have reached a wait statement (or the end of the process) and scheduled their transactions and events, the initialisation phase has finished and the sequence of *simulation cycles* can begin.

At the beginning of each simulation cycle, the time advances until the first transaction that is scheduled. Then all events scheduled for the actual simulation-time are executed, that is the values are assigned to the signals. If the transaction was an event, it may activate processes, which are sensitive on the modified signals. The execution of sequential statements in the activated processes will typically schedule new transactions. When all processes have suspended at a wait statement the simulation cycle is over. The simulation ends when no more transactions are scheduled at the end of a simulation cycle.

In such a simulation model, time advances non-contiguous - in so-called *discrete steps*. Hence, it is called a *discrete event model*.

Definition 7.1 (Transactions and Events) *A* **transaction** t *is a triple* $t = \langle s, v, t \rangle$ *comprising a signal* s*, a scheduled value* v*, and a* **time stamp** t*. A transaction is called an* **event***, if the actual value of* v *is different from the actual value of* s*.*

Definition 7.2 (Temporal order of events) *Let* $e_i = \langle s_i, v_i, t_i \rangle$ *and* $e_j = \langle s_j, v_j, t_j \rangle$ *be events.* e_i *is called* **earlier** *than* e_j*, if* $t_i < t_j$*:*

$$e_i < e_j \Leftrightarrow t_i < t_j \tag{7.4}$$

A sequence $S = e_1, ..., e_n, n \in \mathbb{N}$ *of events is called* **temporally ordered** *if:*

$$\forall i, j \leq n, i < j : e_i < e_j \vee t_i \leq t_j \tag{7.5}$$

Definition 7.3 (Event queue) *The central component in the implementation of a discrete event simulation is the so-called* **event queue***. It keeps the temporally ordered sequence of all scheduled events.*

For the execution of a discrete event simulation, the events in the event queue are executed in the given order. Please note that every event can trigger new events, which are then inserted into the event queue.

Definition 7.4 (Execution of an event) *Let* Q *be an event queue as defined in Definition 7.3. Let* $e_0 = \langle s_0, v_0, t_0 \rangle$ *be the first event in the queue and* t_m *with* $t_m < t_0$ *be the actual model-time. Then the execution of* e_0 *consists of three steps:*

1. $t_m \Leftarrow t_0$*: Assign* t_0 *to* t_m*. Since* e_0 *is the earliest pending event, there is no point in time that needs to be reached in the simulation between* t_m *and* t_0*.*

2. $s_0 \Leftarrow v_0$*: Assign* v_0 *to* s_0*. Now, that the scheduled time for the assignment has come, the assignment can take place*[17]*.*

[17] This step is omitted if the event is a software event.

3. $Q' \Leftarrow Q \setminus e_0$: *Delete* e_0 *from the event queue* Q. *The event is executed and can therefore be removed.*

If there is more than one event at a point in time in the queue, only the first event needs to set the time. The order of executions for events with the same time stamp is then non-deterministic.

As described above during the simulation new events created by active processes need to be inserted into the event queue. The following defines an insertion operation, which maintains the temporal-order properties of the event queue.

Definition 7.5 (Insert an event in Q) *Let* $Q = |e_0, ..., e_n|$ *be an event queue as defined in Definition 7.3 at model-time* t_m *and* $e_j = \langle s_j, v_j, t_j \rangle$ *be a new event with* $t_i \geq t_m$. *Then Q', the event queue after insertion of* e_j *is:*

$$Q' = \begin{cases} |e_j, e_0, ..., e_n| : t_j < t_0 \\ |e_0, ..., e_n, e_j| : t_j \geq t_n \\ |e_0, ..e_i, e_j, ..., e_n| : t_0 \leq t_j < t_n, \nexists e_k : e_k > e_i \wedge e_k < e_j \end{cases},$$

7.4.2 Time in the software model

Software languages in contrast to hardware languages typically have no inherent notion of time. However, to achieve a time-synchronous execution, a concept of time for the entire co-simulation system is essential.

The real-time annex of Ada95 provides a set of statements that describe and enforce the real-time behaviour of the specified software to a certain extent[18]. Three of these statements allow defining events in the software model and forming the basis for the co-simulation concept of time in this thesis.

- The function `clock` returns the current time and allows relating the behaviour to the system's real-time clock.
- The `delay until t` statement suspends the execution of the invoking task until time t is reached. For example `delay until 10.0 ms` suspends the invoking task until `clock` reaches the value 10.0 ms. The *expiration time* of a statement `delay until t` is defined by its parameter (here t).
- For the clock-relative `delay t` statement the expiration time is given based on to the value of `clock` at the time the statement is invoked. For example `delay 10.0 ms` at the actual simulation-time c suspends the invoking task until `clock` reaches the value `c + 10.0 ms`. The expiration time of a statement `delay t` is therefore `t+clock()`.

The concept of time in the software model defined in this thesis advances similar to the hardware concept of time based on a discrete event model. Due to the fundamental differences between hardware and software - in particular software lacks the concepts of parallelism and signals - events in software are defined differently. Therefore a *software event* can simply be defined by its expiration time, which is the time it is scheduled for execution. The other two components in the event-triple (Definition 7.1) can be omitted or remain undefined.

There are two sources for events in the software model:

1. The `delay` and `delay until` statements. Every invocation of such a statement D creates a synchronisation event e with a time stamp equal to the expiration time of D.
2. The interrupts invoked by the hardware model. The time of such an asynchronous event is the invocation-time from the hardware model.

[18] Naturally, due to limited processor resources not all the specified real-time behaviour can be achieved.

Definition 7.6 (Software Event Queue) *To define the execution of the software model according to the discrete event model we need to agree on the following notations. Let denote:*

t_0 *the model-time at the* **start of the simulation**,

$\Delta(t)$ *the* **set of synchronisation events at model-time t**.

then:

Initial Queue *:* $Q_S(t_0) = ||$

Software Event Queue *:* $Q_S(t) = |e_1, ..., e_n|$, *where* $e_1, ...e_n$[19] *denotes the temporally ordered sequence of events (Definition 7.2) in* $\Delta(t)$.

Step in Simulation *: A step in simulation in this software model describes the* **transition from** $Q(t_1)$ **to** $Q(t_2), t_2 > t_1$, *in other words, the co-simulation system steps from time* t_1 *to time* t_2. *It is defined by the execution of all events with a time stamp equal to* t_2 *(Definition 7.4). With the execution of an event in the software model, the expiration time of the earliest event in* Δ *has been reached. The respective tasks now can proceed until it creates an event through the invocation of its next delay/delay until statement or it is blocked.*

7.4.3 Coupling hardware and software models of time

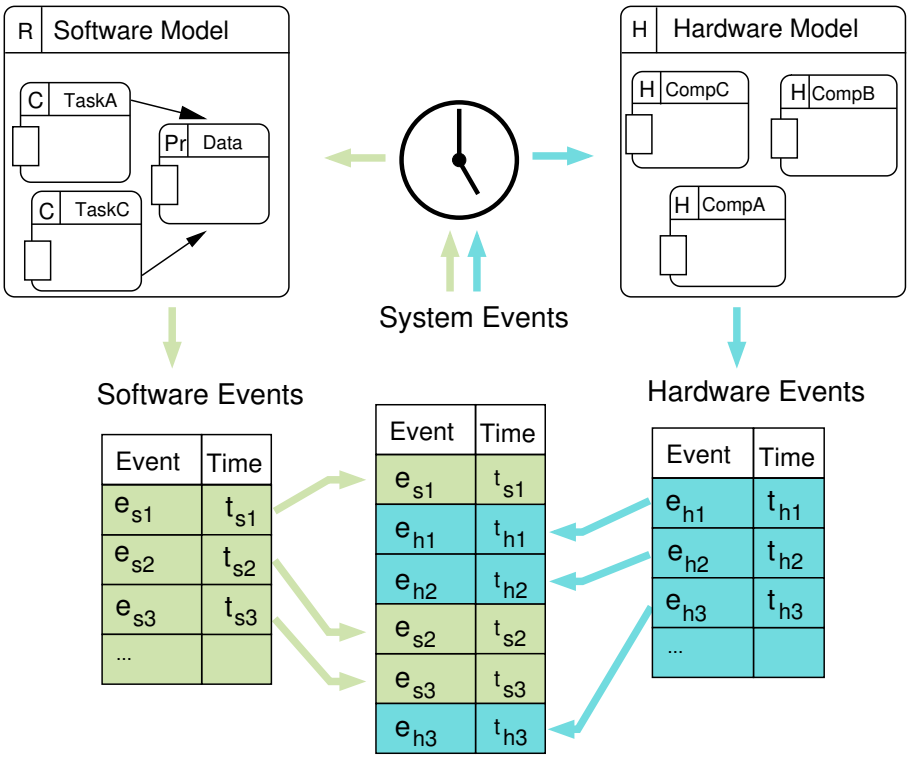

Figure 7.2: Unifying the event queues

[19]Please note that n can take different values between 0 and the number of tasks + 1 during the simulation.

In order to achieve time-synchronous co-simulation behaviour the models of time in hardware and software must be coupled into one *unified time model*. For that purposed the co-simulation is based on a conceptual *unified co-simulation event queue* that merges the events created in hardware and software in the correct temporal order as depicted in Figure 7.2.

Definition 7.7 (Unified co-simulation Event Queue) *Let $Q_S = |e_{s1}, ..., e_{sn}|$ denote the software event queue (Definition 7.6) and $Q_H = |e_{h1}, ..., e_{hm}|$ denote the hardware event queue. Then the* **Unified co-simulation Event Queue** *Q_C is defined as the result of the merger ($\uplus$) of Q_S and Q_H:*

$$Q_C = Q_S \uplus Q_H = |e_1, ..., e_{n+m}|, \ with: \tag{7.6}$$

$$\begin{aligned}
&\forall i \in 1, .., n \ \exists j \in 1, .., n+m : e_{si} = e_j, \\
&\forall k \in 1, .., m \ \exists l \in 1, .., n+m : e_{hk} = e_l, \\
&\forall r, s \in 1, .., n+m, r \neq s : r < s \Leftrightarrow e_r \leq e_s
\end{aligned}$$

7.4.4 Handling asynchronous events

As mentioned above, the second source for events in the software model and therefore in the co-simulation model are the interrupts resulting in asynchronous events. An asynchronous event can only be created while the hardware model is active but must be handled by the software immediately at the activation time of the event.

Hence, an asynchronous event immediately blocks the further execution of the hardware model and initiates an (asynchronous) simulation cycle (Section 7.3.1). The software model gains control and the actual **state of the hardware model** is transferred. This state contains the state of the interface memory, the actual time, and the asynchronous event that occurred. The co-simulation scheduler dispatches the asynchronous event (interrupt) to the designated method representing the handler for the co-simulation. The execution of the handler may cause side effects on other tasks in the software model. A blocked task can be released and then executes until it creates a new software event or it is blocked again.

From the formal point of view, an asynchronous event is equivalent to the external creation of a software event. Hence, the event is inserted into the software event queue and the execution proceeds as described in Definition 7.4.

7.4.5 Synchronising the memory-mapped I/O area

In the target system, the interface is implemented by a memory-mapped I/O area and a number of interrupts. The main goal of the interface synchronisation is to approximate the behaviour of such a physical interface within the co-simulation.

Since the simulation models for hardware and software are implemented as loosely coupled processes, they both have their own representation of the interface model. They read and write independently to and from this interface model and propagate their changes at each software event; that is, at transitions between hardware and software model.

As described in Section 7.3.1, the co-simulation model executes the hardware and the software model in mutual exclusive phases. At the transition between the phases, the state of the memory model is transferred. All transformations of the memory state within one phase are concealed from the other. Consequently, the design should avoid multiple write accesses to a single memory location within one simulation cycle. The easiest way to achieve this property is to restrict the access to a COMMUNICATION OBJECT to a single write access from a single active component[20] in the design. The HRT-HOOD+ model as described in Section 5.3 helps to identify such write-conflicts in the graphical model, but the designer is responsible to guarantee this property.

[20] A task or process in hardware or software.

Please note, that this is not only useful due to the restrictions of the co-simulation model. It also keeps the model robust with respect to changes in its real-time behaviour. If for example, the content of a MEMORY OBJECT is written multiple times within one invocation of a task, a minor change in the performance of the target system can have severe impact on the functional behaviour.

7.5 Implementation of the co-simulation

To prove (the co-simulation) concept, a simulation environment has been implemented. Only the key aspects of this implementation will be described here, as most of the implementation is rather straightforward.

In this implementation, the hardware model is described in VHDL and simulated by the ModelSim™ VHDL simulator. The software model described in Ada95 is generated in two steps. First, some source-code modifications related to the interface model and real-time statements in the software model are required (Section 7.5.3). These modifications are applied automatically by means of a pre-compiler implemented in the course of this work. Second, the modified source-code is compiled using a standard Ada95 compiler. The resulting executable then contains the co-simulation software model.

Since hardware and software model are executed in separate processes, they need to synchronise their behaviour; that is, execute in mutual exclusive phases and exchange information about the memory mapped I/O area, interrupts, and the actual model-time. The communication itself is implemented by standard UNIX inter process communication (IPC, Section 7.5.4).

7.5.1 Unified co-simulation event queue

The hardware simulation model is based on the event queue of the ModelSim simulator. To synchronise the hardware and the software model as described above it is necessary to insert events from the software model into the simulator's internal event queue.

In the actual implementation the so-called *Foreign Language Interface (FLI)* of the ModelSim simulator is used for this purpose.

> *FLI routines are C programming language functions that provide procedural access to information within Model Technology's HDL simulator, vsim. A user-written application can use these functions to traverse the hierarchy of an HDL design, get information about and set the values of VHDL objects in the design, get information about a simulation, and control (to some extent) a simulation run. The header file* `mti.h` *externs all of the FLI functions and types that can be used by an FLI application. (from :ModelSim Foreign Language Interface Reference Manual)*

The top-level architecture of the VHDL model needs to be extended by a pre-defined component called `cosim_interface`, which contains a so-called *foreign process*. This foreign process is activated at each software event. It immediately blocks the hardware model, and is responsible for the communication and synchronisation with the software model via IPC.

7.5.2 The software co-simulation scheduler

A time-synchronous co-simulation can only be achieved if the progress of the software simulation is under control of the co-simulation system.

This goal was be achieved by replacing the access to the real-time part of the runtime system of Ada95 in the software model. The runtime system constitutes an abstraction layer between the application and the underlying operating system. In particular, this runtime system controls the real-time clock and the suspension and activation of tasks caused by `delay` and `delay until` statements.

Obviously, the value of clock delivered by the host operating system and the derived behaviour of the real-time statements is not appropriate for the co-simulation.

In the actual implementation of the co-simulation, a *protected object* replaces the original runtime system. It provides the application with suitable replacements for the original clock, delay, and delay until statements. The implementation make extensive use of some advanced features like *guards, re-queueing, queueing_policy* of Ada95 protected objects to achieve the desired behaviour. The implementation of the core software co-simulation scheduler is rather small[21], which makes it quite easy to verify the scheduler for an experienced Ada95 programmer.

7.5.3 Code transformations for the co-simulation - Automatic pre-compiler

As mentioned above the correct behaviour of the software model in the co-simulation is based on the replacement of the real-time related statements, access to memory mapped I/O and to the interrupts in the implementation software model. Since the manual replacement of all such statements would be tedious and error-prone, in the course of this work a tool has been implemented to automate this work.

Transformation of real-time related Ada95 statements

With respect to the real-time behaviour, the pre-compiler needs to perform the following modifications in the software model:

1. Extend the implementation software model by the `SimuTime` package, containing the co-simulation scheduler and the replacements for the real-time statements. This could be achieved by adding the following source-code fragment to every Ada95 package in the implementation-model that needs access to any real-time statement:

```
-- add the SimuTime package
with SimuTime;

-- allow unqualified access to functions in SimuTime
use SimuTime;
```

2. Replace `clock` statement: Since the original `clock` function would deliver the wall-clock rather than the co-simulation clock, it is replaced by `SimuClock` from the `SimuTime` package.

```
-- original statement
-- start_time := clock;

-- replacement for co-simulation
start_time :=  SimuClock;
```

3. Replace `delay` and `delay until` statements: Similar to the `clock` function, the suspending statements need to be replaced by their counterparts for the co-simulation.

[21] Less than a hundred lines of Ada95.

```
-- original statement
-- delay until clock + offset;

-- replacement for co-simulation
SimuDelayUntil(SimuClock + Period);
...
-- original statement
-- delay 10 ms;

-- replacement for co-simulation
SimuDelayUntil 10 ms;
```

Note that real-time statements can also be part of expressions. Hence, the implementation of the pre-compiler was more complex than it might seem at first sight.

Transformations related to memory-mapped I/O access and interrupts

The communication between hardware and software in embedded systems is often implemented using interrupts and direct mapped shared memory. Obviously, the co-simulation cannot provide access to the required physical resources. Hence, all physical accesses to these resources must be replaced by co-simulation counterparts.

1. Replace declaration of direct memory-mapped data-structures: To avoid the access of non-virtual memory addresses the pre-compiler replaces the representation clause by its co-simulation counterpart. Please note that the data-structure itself (`P_IF`) remains unchanged. Only the mapping onto the physical address is replaced.

```
-- declaration of data-structure
P_IF : My_T;

-- original statement
--  for P_IF'Address use P_IF_Address;

-- replacement for co-simulation
P_IF_Handle : DevRegHandle(
  To_Integer(P_IF'Address), P_IF'Size,
  StorageUnitSizeOfRegister(P_IF'Size),
  WantedAddress => P_IF_Address
);
```

The `P_IF_Handle` is declared as a so-called *limited type* pointing to the original interface data-structure `P_IF`. This handle allows the co-simulation to track every access to `P_IF` and map it onto the simulation model memory.

2. Replace the binding of interrupts to ISR in the implementation model. The co-simulation scheduler evaluates the asynchronous event received from the hardware model and calls the dispatcher method in the co-simulation `Interrupts` package. This dispatcher resembles exactly the dispatcher in the operating system by calling the method (ISR) associated with the asynchronous event.

7.5.4 Mutual exclusion of hardware and software model - interprocess communication

As described in Section 7.3.1, a co-simulation cycle consists of two phases, namely the HEP and SEP, which are executed mutual exclusively. At the transition from either phase to the other, the simulation model-time and the actual state of the interface memory is encoded into a compact data structure[22] and transferred between the models. Both aspects, the mutual exclusion as well as the data transfer is implemented by means of IPC [Ste98, Ste99] using a single UNIX socket.

Since a blocking protocol has been chosen for the socket, each communication participant (here each model) is blocked whenever it is awaiting a data-packet being send by the other participant. Consequently, the hardware (software) model is blocked at the end of the HEP (SEP) by awaiting the next data-packet from the software (hardware) model.

7.6 Recap

In this chapter, the following aspects of OOCOSIM have been presented:

- A concept of time for the software model based on the real-time annex of Ada95.
- The merging of the concept of time for the hardware and the software model into a unified co-simulation model.
- The key ideas behind the implementation of the co-simulation in OOCOSIM.

This concludes the description of the OOCOSIM methodology, which starts with a graphical specification in HRT-HOOD+, proceeds with the interface design based on COMIX and finishes with an executable and implementable model in hardware and software.

[22] To improve the efficiency of the co-simulation , the time stamp and the state of the interface are both coded into one data-packet.

8 Evaluation of the OOCOSIM Method

The aim of this chapter is to discuss and evaluate the OOCOSIM design flow as presented in Chapter 5-7. The evaluation will follow the criteria as defined in Section 4.5.

Two of these criteria, namely *performance* and *modelling of hardware/software interfaces* can be underpinned by quantitative data. Some criteria are not measurable in principle like for example the seamlessness of the design flow or the adequacy of abstractions. For such criteria, the following sections will discuss the means that OOCOSIM provides to support them.

The first section will briefly introduce the benchmarks used in this evaluation chapter. The following sections will focus on different requirements on design methods and discuss them with respect to the approaches in OOCOSIM.

8.1 Benchmarks for OOCOSIM

A major motivation for benchmarks in general is to validate the effectiveness of an approach and compare the results with other methods. Practical use of a method also promises to identify potential improvements. This section presents two embedded system benchmarks that focus on different methodological aspects. The *crane controller benchmark* is used mainly for the evaluation of HRT-HOOD+ and the co-simulation. The hardware/software interface in this application is fairly simple. Due to the importance of this aspect, a second benchmark is used to evaluate the COMIX-based interface design approach in a larger context. The *elevator system* contains a large number of communication objects to exchange data and control commands between the software controller and various sensors/actuators located in hardware.

8.1.1 Crane controller benchmark

This benchmark was created in the apron of the DATE'99 (Design Automation and Test in Europe) conference, where a panel was held to discuss the pros and cons of different design languages. In the run-up of the conference, the competing languages, namely Ada95, SpecC, Java, and VHDL-AMS, were applied to this common benchmark.

A few month later at the FDL'99 (Forum on Design Languages), a second panel about the same topic took place. This time other language settings (Simulink, OCCAM, and OOCOSIM [OSN00b, OSN00a]) were applied to the same benchmark. Each participating research group received the specification [MN99] containing the functional specification, timing requirements, and some test cases. The main tasks to be fulfilled in the design contest were:

- Model the physical crane in its environment and the embedded system controlling it.
- Apply a set of test scenarios to the model to validate the model.
- Present the results and discuss them in the panel.

The comparison documented in [GMNV00] gives an overview on the different strengths and weaknesses of the approaches. All models (Ada95, OCCAM, VHDL-AMS, Matlab/Simulink, and Java) are executable[1], but the design aspects represented in these model are very different. The Java model

[1] There was no working model and simulator for VHDL-AMS available by the end of the comparison. This however is no general property of the language.

provides a graphical real-time animation to demonstrate the dynamic behaviour of the crane. The VHDL-AMS design concentrates on modelling the continuous aspects of the benchmarks. OCCAM and the OOCOSIM represented the parallelism of system components explicitly by language primitives. The key difference between OOCOSIM and the other methods presented is, that the OOCOSIM approach bases on the implementation model of the system, which is already partitioned between hardware (Objective VHDL) and software (Ada95). It contains, in contrast to the others, a model of the physical interface. This leads to a more detailed model enabling an in-depth analysis of the system's functional and timing behaviour also with respect to implementation decisions. On the downside, this costs significantly more design and simulation effort as documented in the comparison[2]. Due to the different aspects in the design studies, a comparison of the result reported in [GMNV00] with the quantitative figures about the OOCOSIM method (Section 8.6) is difficult. Instead, the comparison of the co-simulation model with a homogenous model that provides similar accuracy in functional and timing behaviour will be presented in Section 8.6.3.

Basic functionality

The benchmark describes a so-called *portal crane* as depicted in Figure 8.1.

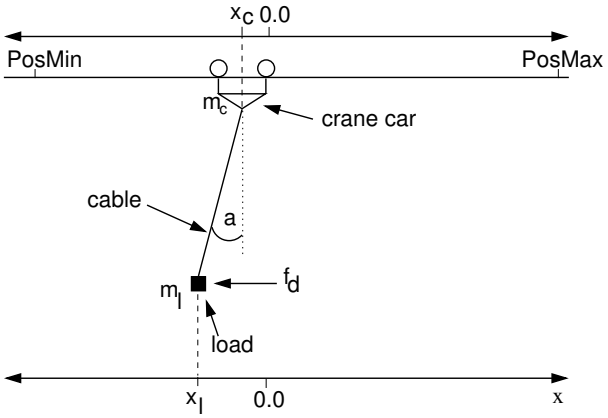

Figure 8.1: The crane system. Source: [MN99]

The crane car moves along a track, carrying a load m_l attached to the car with a flexible cable of fixed length.

Six sensors (PosCar, SwPosCarMin, SwPosCarMax, Alpha, SwShutDown, PosDesired)[3] provide data about the physical system in which the embedded system controls the crane. Three actuators can be applied to the crane car: The driving voltage V_c accelerates the car via a motor, the brake stops it, and emergency-stop stops and applies a circuit breaker immediately in case of a system failure.

When the car accelerates, the car-cable-load system begins to swing. The dynamic physical system is given as a fourth-order linear system. The benchmark specification describes the control algorithm and certain real-time requirements for the controller. In particular, the formula and parameters to compute the driving voltage V_C as well as the control flow for all mode-changes (e.g. operational mode to emergency stop) are described in detail.

First, the crane moves slowly to the left and right limit of the track to check the function of its safety sensors ($SwPosCarMin$ and $SwPosCarMax$). After successfully passing this safety check, the system enters the *operational mode*. In operational mode, the *job-control task* of the embedded control system awaits a desired position (via PosDesired sensor) to be set and moves the load to that position. The acceleration of the car can be controlled by the voltage V_c within a range of $\pm$ 40V applied to the motor of the car and the brake to stop the car.

In parallel with the job-control, a *diagnosis task* applies periodical sensor plausibility checks. If for example the Alpha sensor fails, the diagnosis task will detect this and the crane enters a specified *emergency mode*[4].

[2]It should be noted that by the time of the benchmark, especially the co-simulation environment in OOCOSIM was very prototypic. With the actual co-simulation tools, the simulation performance is about a factor 20 better (Section 8.6).

[3]The 'Sw' in the sensor names indicates that the sensor is a switch. 'PosDesired' is the target position of the load. It is (in accordance with the specification) modelled as a sensor.

[4]In the emergency mode, the control algorithm calculates V_c and controls the break only based on the PosCar sensor.

Also part of the benchmark is a suite with three different test scenarios. These scenarios contain the sensor checks and regular jobs to drive the load to different target positions. One scenario includes a disturbance (e.g. wind) modelled by an external force f_d applied on the load; others contain the failure of certain sensors leading to emergency mode or an emergency stop.

The benchmark can be described by the following characteristics:

Detailed physical model: As mentioned above, the physical system (car, load, and disturbance) has been defined by a fourth-order linear system. The benchmarks specification recommends to use the *Runge-Kutta* algorithm [RK24] to solve the differential equations. Hence, modelling the environment surrounding the embedded controller took large portions of the design effort.

High accuracy requirements: The parameter matrices (A and B in [MN99]) applied in the control algorithm contain values in the range between 10^{-8} to 10^{4}. Since these matrices are iteratively multiplied, numerical accuracy requirements are high. Hence, this benchmark was also a good opportunity to check whether the design languages (Ada95 and VHDL coupled by the co-simulation) are able to provide an adequate level of accuracy in terms of numerical precision.

Hard real-time constraints: Several mode changes are specified according to functional and real-time conditions. If, for example, $Alpha > AlphaMax$ for more than 50 ms within a period of 100 ms, the diagnosis task initiates the emergency-mode[5]. Naturally, such a specification can only be checked with a co-design method, which regards real-time in its model representations.

Long model-times: The crane is a rather large physical system. With 10 m distance between PosCarMin and PosCarMax, it takes about 40 minutes model-time to perform each of the given test scenarios in the benchmark. Most of the time (36 minutes) is consumed by the sensor check preceding the full operational mode[6].

Limited parallelism: The benchmark specification describes only two explicit parallel activities for the controller: the job-control and the observer. However, the actual model for the embedded controller in this experiment contains also active objects for the power-up, the voltage-control (in hardware), and the polling of interface values (in software). Other parallel components in the model were the environment, the operator, and the monitoring tasks.

The HRT-HOOD+ model

The top-level specification of the codesign model of the crane system as depicted in Figure 8.2 contains six first-level objects that resemble the basic structure.

- The `Operator` and the `Physical_Plant` are objects, which obviously do not need to be implemented by the embedded system. Nevertheless, they are required for the testbench in the simulation model of the portal crane.
- The `Job_Control` object, here assigned to the software partition, implements the main control process for the crane.
- The `Voltage_Control`, here implemented in hardware, calculates the driving voltage and detects the breaking condition. Due to strict real-time constraints, this object was allocated to the hardware partition.
- Two abstract COMMUNICATION OBJECTS, namely `Sensor_Values` and `Actuator_Values`, model the communication between the hardware and software objects.

[5] The specification moreover defines a minimum arrival time for sensor events of 2 ms. This allows a discrete integration of sensor value times with a period of 2 ms.

[6] The sensor check runs before the crane enters operational mode; that is, the brake can not be applied. Hence, only very small driving voltage (2 mV) can be applied to the motor to safely accelerate and stop the car.

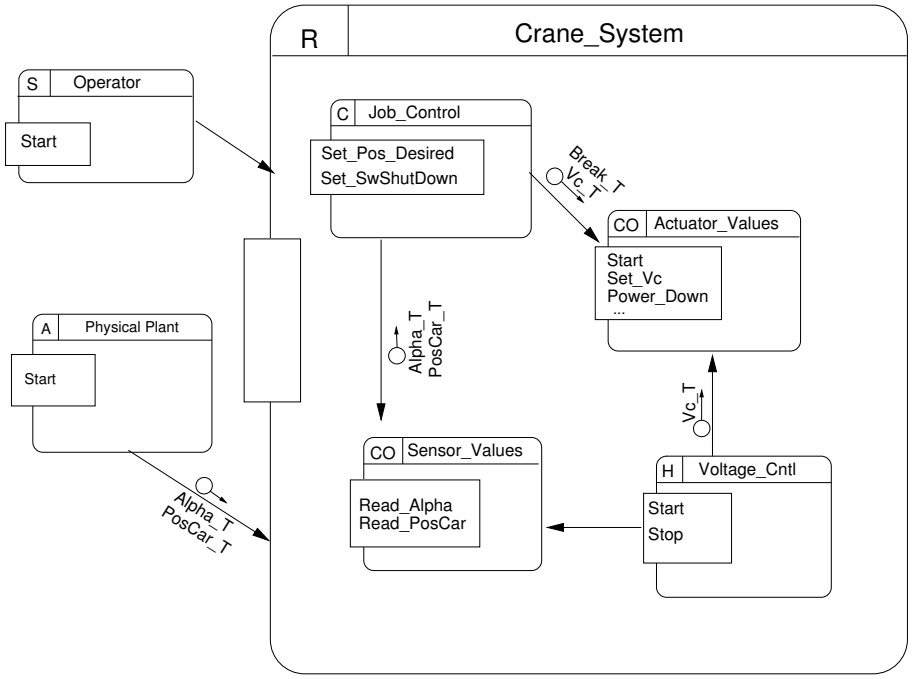

Figure 8.2: Initial HRT-HOOD+ model

Based on this top-level model, the hierarchical refinement added more details to the model. Soon the specification became too complex to be depicted in a single diagram. The hierarchical decomposition resulted in several sub-diagrams. Figure 8.3 shows such a sub-diagram with special emphasis on the actuators of the system. All other objects are depicted as simple boxes indicating only the type and the name of the object[7]. This refinement of certain components allows the designer to analyse the relations and data-flows between fine-granular objects – here some COMMUNICATION OBJECTS for the actuators – and the rest of the model.

The executable models

From the HRT-HOOD+ model two executable models have been derived: A pure Ada95 model and a co-simulation model containing a hardware sub-model in (Objective) VHDL[8] and a software sub-model in Ada95. Since both models are based on the same HRT-HOOD+ specification, they can easily be compared in terms of size, function, and performance (see Section 8.6).

Table 8.1 compares the two models according to their size in lines of code and size of the executable in kilobytes.

Full Ada95		Generic code	Co-sim. Ada95		Co-sim.VHDL
Code [LoC]	Exec. [kByte]	Code [LoC]	Code [LoC]	Exec. [kByte]	Code [LoC]
8035	559	2096	6847	1100	1477

Table 8.1: Comparison of model size.

[7] In HRT-HOOD these objects are called *Uncle Objects*.
[8] The hardware objects in HRT-HOOD+ first have been modelled in Objective VHDL and then automatically translated into VHDL.

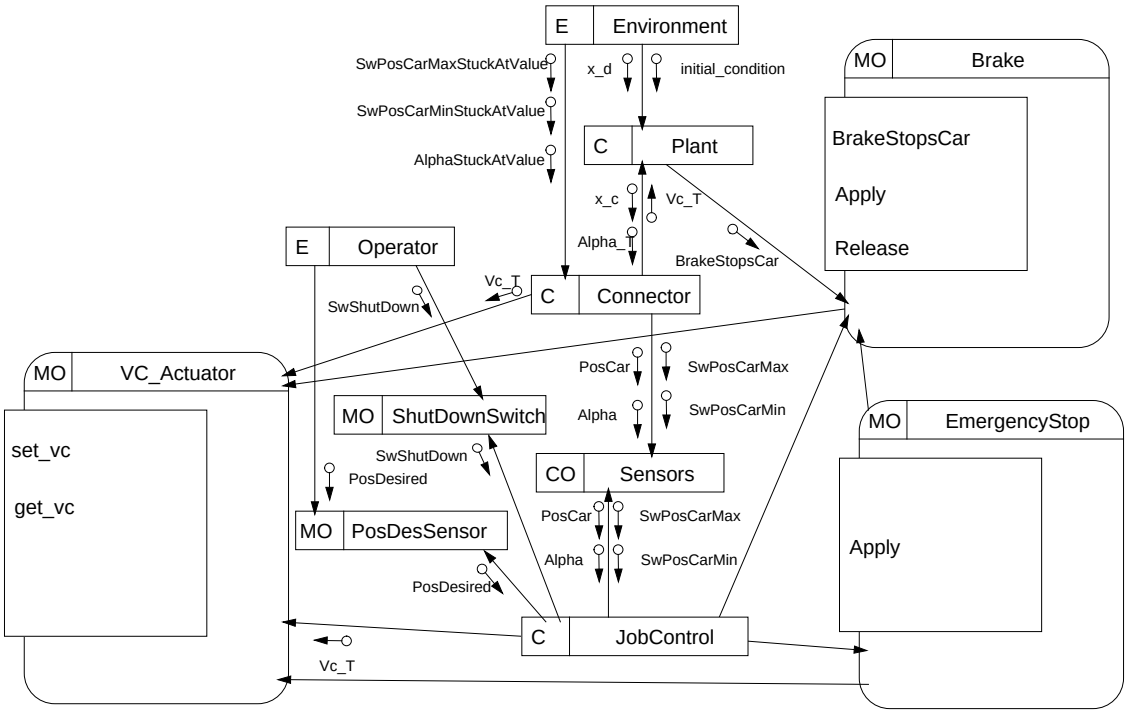

Figure 8.3: Actuators in detail

It can be observed that the co-simulation model is only slightly bigger than the pure software model. Both models contain a large portion (about 25%) generic code; that is, library code for the Runge-Kutta algorithm and matrix arithmetic. In the functional behaviour, only small numerical deviations (in the order of 10^{-6} m in the position of the load) have been observed. The deviations could be traced to the slightly different representations of real numbers in VHDL and Ada95[9].

8.1.2 Elevator system

The second benchmark models a passenger hoist, transporting people between different floors in a building. It is equipped with several comfort features (e.g. air condition) and safety mechanisms (fire sensor, sprinkler). The elevator system as depicted in Figure 8.4 has been modelled using OOCOSIM in a student research project [Zha99]. The model contains the following subsystems:

Cabin: The cabin carries the passengers between the floors. It contains the cabin panel (e.g. floor select buttons), sensors (e.g. smoke, temperature), and actuators (e.g. lights, sprinkler).

Elevator control (ECU): This central controller receives all data provided by the sensors and generates the commands for the actuators.

Motor: The motor receives commands from the ECU to move the cabin to the desired position and sends data about the motor temperature to the ECU.

Floor panels: On each floor of the building, there is one simple panel with a call button and the arrival signal.

[9] Ada95 allows user-defined floating point types, while VHDL supports only a standard IEEE-floating point type.

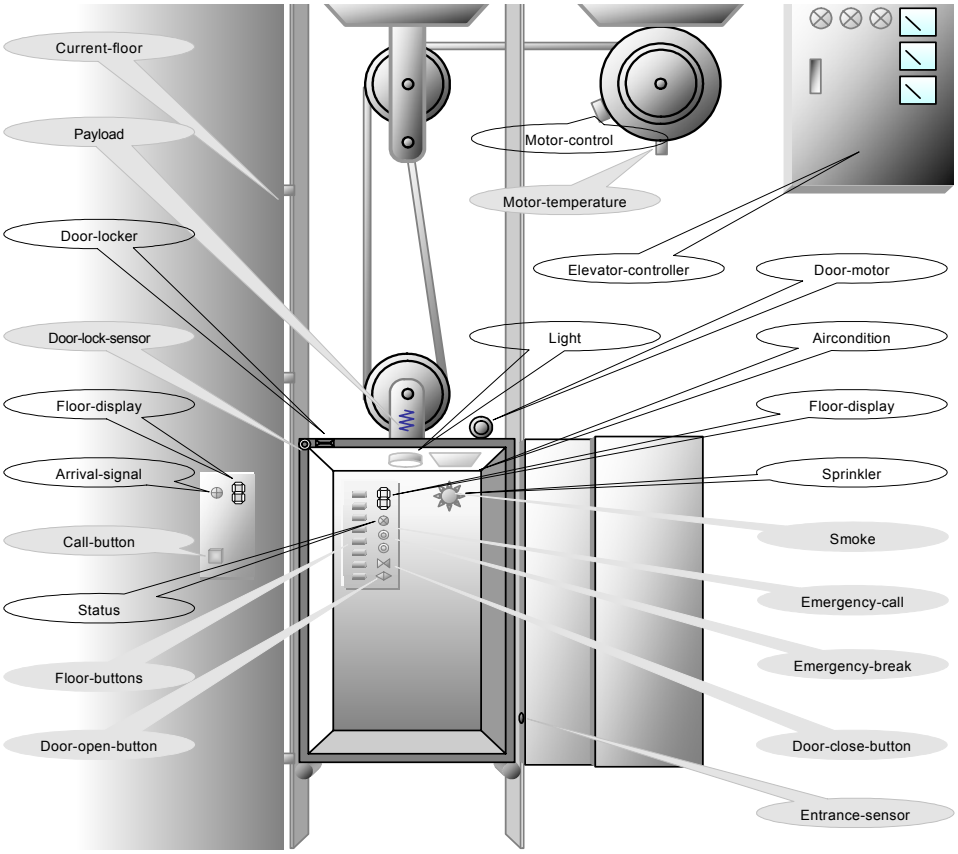

Figure 8.4: The elevator system. Source: [Zha99]

Many independent control processes (for example multiple user inputs, monitoring of safety sensors) result in a complex reactive behaviour, including several asynchronous signals for emergency sensors. The large number of sensors and actuators makes the elevator system case study especially useful for the evaluation of the interface modelling capabilities in OOCOSIM (Section 8.7).

8.2 Complexity handling

OOCOSIM provides several mechanisms to handle complex embedded systems. The graphical representation of HRT-HOOD+ supports the visualisation of complex relations between system components. The graphical overview helps the designers to communicate with each other or with externally involved persons, for example the customer. The core techniques to handle complexity in OOCOSIM are abstraction, decomposition, and decoupling of system components. The main ratio behind this approach is that primary the interaction between components, rather than the components as such, are the cause for system complexity. In the first design phase, SYSTEM OBJECTS allow decomposing the system into loosely coupled sub-systems also called logical partitions. These partitions can then be handled - maybe by different design teams - almost in isolation. In the crane benchmark, the design team decided to split the HRT-HOOD+ design into three sub-systems: The physical plant, the job-control and the voltage-control (Figure 8.2). These loosely coupled sub-systems have been refined

almost independently. This helped to split the design into simpler design tasks. Due to the clearly defined interfaces and the specified real-time requirements in the graphical model, the integration of the sub-systems was rather easy.

Abstraction allows to defer low-level decisions to a later design phase or to concentrate on a certain aspect of the design problem. Figure 8.3 depicts an example of this principle called *separation of concerns*, where only the role of the actuators in the crane system is concerned. Decoupling allows the component-wise refinement of a single component without corrupting the rest of the system. Decoupling is supported by two major principles of the object-oriented design paradigm: Well-defined component interfaces specifying the outside view of a component, and encapsulation[10] avoids the access to non-public attributes or methods of component.

OOCOSIM supports the modelling of hardware/software communication throughout the entire design flow. It augments the graphical modelling language HRT-HOOD by COMMUNICATION OBJECTS for the design capture and analysis of interface aspects. With this early explicit interface model the identification of communication bottlenecks and potential conflicts (e.g. multiple writers to a COMMUNICATION OBJECT) becomes possible. The detailed specification of the interface architecture and the implementation of the refined COMMUNICATION OBJECTS is supported by the languages ComiX and TempliX (see also Section 8.7).

An aspect of embedded system design, OOCOSIM does not fully cover, is the early phase of application specific hardware design. For this domain, HRT-HOOD+ does not provide adequate means to model and refine the concurrent behaviour of hardware objects. Due to this over-simplification in HRT-HOOD+, hardware cannot be maintained at the same level as software or hardware/software interface design.

The validation of complex embedded hardware/software systems requires efficient and sufficiently accurate models. OOCOSIM provides a co-simulation mechanism based on the source-code models which are automatically generated from the implementation model of the embedded system (Section 7.5.3). The performance and the modelling of real-time aspects is subject to Section 8.4 and Section 8.6.

8.3 Seamlessness

The seamlessness of a design flow refers to way the *transformations* between different representations can be handled. For a seamless design flow these transitions must be well-defined or – where possible – even be automated. Typically, such transitions include the transformation of one model notation into another. Note that not all transitions are refinements as some of them do not add details to the model. These transformations are depicted by horizontal arrows.

Figure 8.5 illustrates the different model representations[11] and transformations supported in the OOCOSIM design flow. The rounded boxes contain graphical representations and the straight boxes contain textual representations. Three different types of transformations are indicated by three types of edges:

Solid edges: Denote automatic transformations; that is, supported by a complete set of rules describing the transformation, which can be implemented in an automatic tool. Please note that most automatic transformations in OOCOSIM are performed by automatic tools (see also below).

Dashed edges: Denote transformations, which are guided by specific refinement guidelines. Theses transformations cannot be automated but the methodology provides guidelines how to find suitable transformations or refinements.

[10] Also known as information hiding.
[11] The figure contains only such models, which are covered by OOCOSIM.

Dotted edges: Denote manual transformations. In this case the designer needs to decide between different design-alternatives and manually implement them.

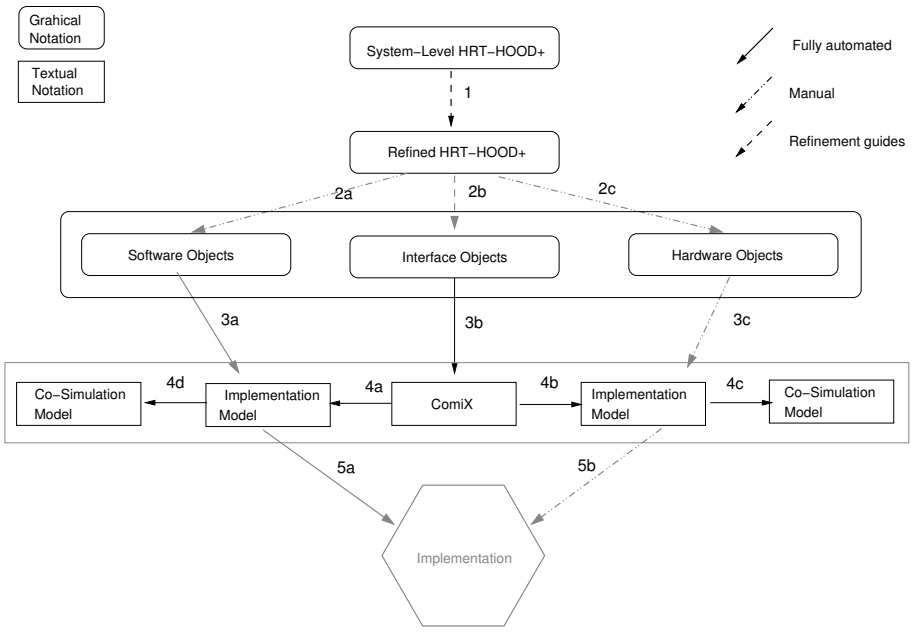

Figure 8.5: Model transformations in OOCOSIM

Furthermore, grey edges indicate pre-existing transformations in contrast to black edges, which indicate transformations defined in the course of this thesis. The following list comments on each of the edges in Figure 8.5.

1: The system-level model in HRT-HOOD+ describes the overall system architecture by using SYSTEM OBJECTS as described in Section 5.3.2. The first transformation leads to a refined HRT-HOOD+ model containing ACTIVE OBJECTS and PASSIVE OBJECTS. OOCOSIM defines some (simple) modelling guidelines for this first decomposition (Section 5.3.3).

2a, 2c: Transformations 2a and 2c refer to hardware/software partitioning of the system; that is, the allocation of objects to hardware or software and the specification of communication objects. This is a critical and difficult design decision. HRT-HOOD+ provides the notation to partition the model but does not assist the decision in itself. Hence, this transformation is regarded as manual.

2b: Provided transformations 2a and 2c have been done, the identification of COMMUNICATION OBJECTS can follow methodological guidelines. First, the data-flow between hardware and software objects is modelled by abstract COMMUNICATION OBJECTS, which are then refined into implementable, concrete COMMUNICATION OBJECTS as defined in Section 5.3.6.

3a: This transformation is inherited from the original HRT-HOOD method and therefore indicated by a grey edge. The code templates defined by Burns and Wellings in [BW95] are implemented in commercial tools like the graphical STOOD^{TM} tool of tni-valiosys Inc., which has been used in the crane benchmark. Hence, this transformation is regarded as automatic.

3b: The transformation of COMMUNICATION OBJECTS into their textual ComiX representation is fully determined. For each graphical COMMUNICATION OBJECTS, an equivalent textual ComiX representation is defined. Please note that the ComiX specification itself must be refined before it can be transformed into the implementation model using transformations 4a and 4b.

3c: As already mentioned in Section 5.3.10, hardware objects in HRT-HOOD+ can be modelled similar to environment objects known in the original HRT-HOOD method. Hence, they can be transformed only manually into the implementation models.

4a, 4b: These transformations are fully automated and implemented in the DESHICO tool as described in Section 6.5.

4c: For this transformation only a few lines of constant VHDL code must be added to the implementation model. Since this transformation is very simple, it is marked as automatic – even though an appropriate tool has not yet been implemented.

4d: This non-trivial transformation has been described in Section 7.5.3 and is implemented by an automatic tool.

5a, 5b: The transformation into the lower-level implementation models and finally the synthesis and compilation of hardware and software components is beyond the scope of this thesis. The result of 3a, 3b, 4a, and 4b can directly be transferred into the implementation. 5a and 5b are therefore depicted by (grey) solid and dotted arrows, respectively.

The crane benchmark design experiment has been carried out in 1999. By that the transformations 3a, 4c, and 4d were already supported by automatic tools. Especially the missing tool support for the interface generation resulted in significant design effort. The elevator benchmark has been carried out twice. As the first experiment started with an UML specification of the system, it was done only with support for the transformations 4c and 4d. The second experiment could benefit from (automated) support for transformations 3a, 3b, 4a, 4b, 4c, and 4d. The interface related transformations significantly reduced the design time.

The different aspects are for both benchmarks discussed in detail in Section 8.7.

8.4 Real-Time modelling

OOCOSIM supports the modelling and in particular the co-simulation of real-time aspects in an embedded hardware/software system at a high level of abstraction. Chapter 7 describes in detail the timing model of the co-simulation and the means available to specify the real-time behaviour of the system.

The modelling of timing aspects of the software model is based on the real-time attributes defined already in the original HRT-HOOD method and propagated by means of the real-time annex Ada95 into the source-code model. As mentioned above, hardware is not covered well in HRT-HOOD+. Timing can therefore only be specified in the hardware model at source-code level. The co-simulation in OOCOSIM maintains the validation of real-time behaviour in the heterogeneous model by the time-synchronous execution model as described in Section 7.3.

In the benchmarks, complex reactive behaviour including software parallelism and interrupts created in the hardware could be modelled and simulated adequately in the co-simulation. Several errors in the model and even some in the specification have been identified with the help of the co-simulation. It should however be noted that this co-simulation operates at a high level of abstraction. Hence, it can only support a coarse-grained timing model. Consequently, the co-simulation in OOCOSIM alone is not sufficient to verify the real-time behaviour especially of safety critical embedded systems. In such cases, a detailed analysis is mandatory. The role of the co-simulation is to evaluate quickly the behaviour of the model in a coarse-grained real-time model. Especially in

complex systems with long model times involved, the co-simulation can serve as a virtual prototype and help to identify design mistakes early in the design flow.

8.5 Hardware/Software partitioning

As already mentioned before, OOCOSIM does not automate partitioning; that is, the designer must decide, which component should be implemented in hardware or software. However, OOCOSIM methodologically supports partitioning decisions by the following means:

1. Decomposition of the design into objects. First, structural HRT-HOOD+ objects like SYSTEM OBJECTS help to divide the system into simpler sub-systems. Then subsequent hierarchical decompositions allow refining the design towards an executable and finally implementable model.

2. Specification of objects as hardware, software, and interface. This allows expressing and exploring different partitionings, while considering an explicit interface model.

3. Executing the model in a time-synchronous co-simulation. This allows analysing (manually) the suitability of the partitioning according to functional and timing aspects.

4. Abstraction, decoupling, and encapsulation of objects allow changing partitioning decisions without affecting the entire design.

8.6 Performance of the co-simulation

The co-simulation in OOCOSIM operates at a high level of abstraction[12] in order to maintain the high performance demands of complex embedded system validation. It follows a heterogeneous approach allowing the detailed and adequate modelling of hardware and software in dedicated languages. This approach incurs overhead for the synchronisation and communication between the hardware and software sub-models. This section will evaluate the relative performance (Equation 7.1) of the co-simulation; that is, the ratio between model time and simulation time. Furthermore, the impact of the synchronisation and communication overhead will be analysed. In OOCOSIM, the some operations required for the communication are an integral part of the processes simulating the hardware model and the software model. Hence, T_{comm} can not be measured separately. Consequently, it is necessary for the assessment of the performance to deviate from the *conceptual components* of simulation time as introduced with Equation 7.1. The computational effort for the co-simulation of a model (t_{model}) in OOCOSIM contains the *measurable components*, as depicted in Figure 8.6:

1. Ada95 software model execution, from now on called *software model execution effort* (SMEE),

2. VHDL hardware model execution, from now on called *hardware model execution effort* (HMEE),

3. *(Process) synchronisation effort* (SYNE); that is, the computational effort for the exchange of data via the interprocess communication mechanism and the blocking/unblocking of the simulation processes in the host's operating system. This part of the co-simulation mainly requires system calls to the operating system. Hence, it is almost equal to the system time reported by the UNIX `time` command[13].

[12] Compared to other timing-accurate approaches as described in Section 7.2.

[13] Other processes could have side effects on the measurements. Hence, during the measurements, it had to be guaranteed that only the co-simulation processes were executed on the simulation host.

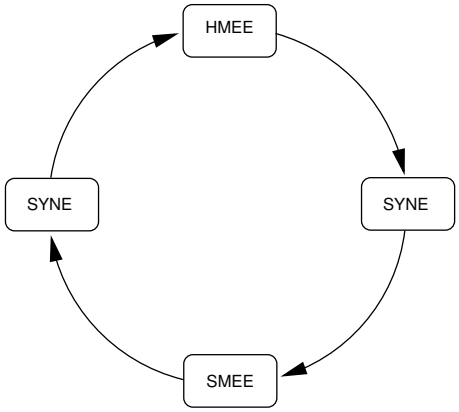

Figure 8.6: The computational effort in a co-simulation cycle.

The *model state encoding effort* (MSEE) refers to the computational effort needed to prepare the data for the transition between SEP and HEP (and vis versa). In the simulation, it is part of the hardware simulation and the software model execution. Hence, in the measurements reported by the `time` command, it can not be separated from SMEE and HMEE. The impact of the MSEE in the co-simulation will be discussed using models of different interface sizes (Section 8.6.4).

8.6.1 Assessment of performance

The assessment of the relative performance uses the computationally complex crane benchmark (Section 8.1.1). The measurements are all based on the first 600 sec = 10 minutes of model-time. The relative performance of the crane model is (almost) constant over the complete benchmark. The difference between the relative performance measured based on the full benchmark; that is 2700 seconds of model-time, and the 600 sec time frame used here, is below 1 %. Hence, this limited time frame is long enough for the evaluation.

For the crane benchmark (besides the co-simulation model), a functionally equivalent homogeneous Ada95 model exists. The comparison between these two models allows assessing the absolute performance of a heterogeneous model compared with a homogeneous model.

To discuss the impact of the synchronisation (SYNE) and communication (MSEE) effort in the co-simulation, an artificial benchmark is used. This benchmark contains only a minimal hardware and software model (Section 8.6.4). The only purpose of this model is to perform a defined number co-simulation cycles containing (almost) only SYNE and MSEE.

All co-simulation models have been created, simulated, and measured on the following host system:

SUNBlade 100 workstation (500 MHz Ultra SPARC IIe, 640 MB memory) equipped with the Solaris^{TM}8 operating system provides the host execution platform for the benchmarks;

GNAT3.13p Ada95 compiler to create the executable program representing the software model;

ModelSim 5.8 VHDL simulator to simulate the hardware model;

UNIX time command to measure the execution time of the processes. The time command in particular delivers the computation time spend for the user application and for calls to the operating system.

8.6.2 Relative performance and accuracy

Figure 8.7 and Figure 8.8 depict the simulation time (T_{host}, y-axis) of the co-simulation model for the crane in relation to the model time (T_{model}, x-axis). The black diamonds show the overall

simulation time at multiples of 100 seconds model time. The overall simulation time contains the time for the Ada95 model (SMEE) depicted by the green squares, for the hardware model (HMEE) depicted by blue squares, and the synchronisation time (SYNE) shown in red triangles. The dashed lines show the linear interpolations through the measuring points. For a better comparability the grey line shows the performance of a fictitious model with a relative performance of 1.

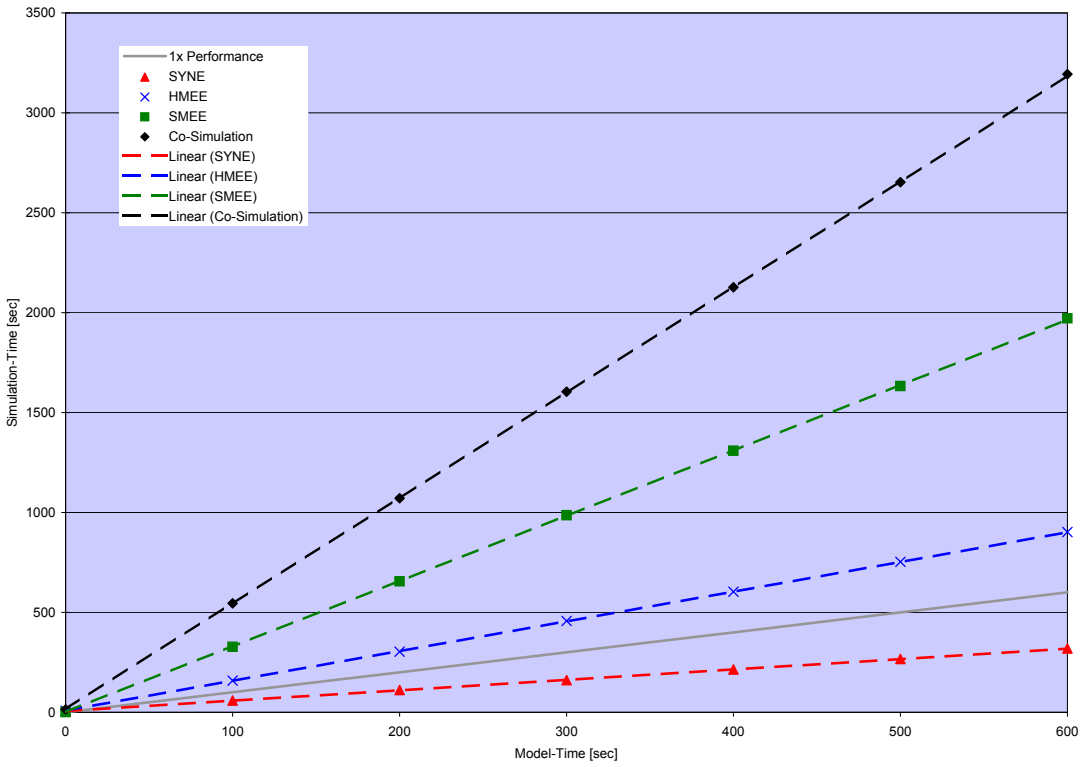

Figure 8.7: Relative performance of the co-simulation with 1 ms minimum cycle-time.

The crane benchmark specification proposes the Runge-Kutta method with a step size of 1 ms to solve the fourth-order linear system describing the plant. Variations of the model accuracy and the simulation performance are achieved by changing this step size from 1 ms to 10 ms. The variations in the model behaviour with different step sizes are acceptable for the first validations. The positions of the load in the model with 1 ms and 10 ms typically differ by less than 1 cm. Please see below for more details about the accuracy of the models. Since the Runge-Kutta requires quite complex computations, the impact on the relative performance is dominant. Moreover, the task calculating the Runge-Kutta algorithm has the shortest period in the co-simulation model. Hence, with increasing its period from 1 ms to 10 ms, the number of synchronisation events can be reduced by a factor of ten[14].

The comparison of the results depicted in the Figure 8.7 and Figure 8.8 underpins two properties of the co-simulation. First, the performance of the co-simulation model scales with the accuracy of the simulation. While the most accurate model delivers a relative performance of approximately 0.19, the

[14]Please note that the periods of all other tasks are integer multiples of 10 ms. The next greater period in the benchmark is 20 ms.

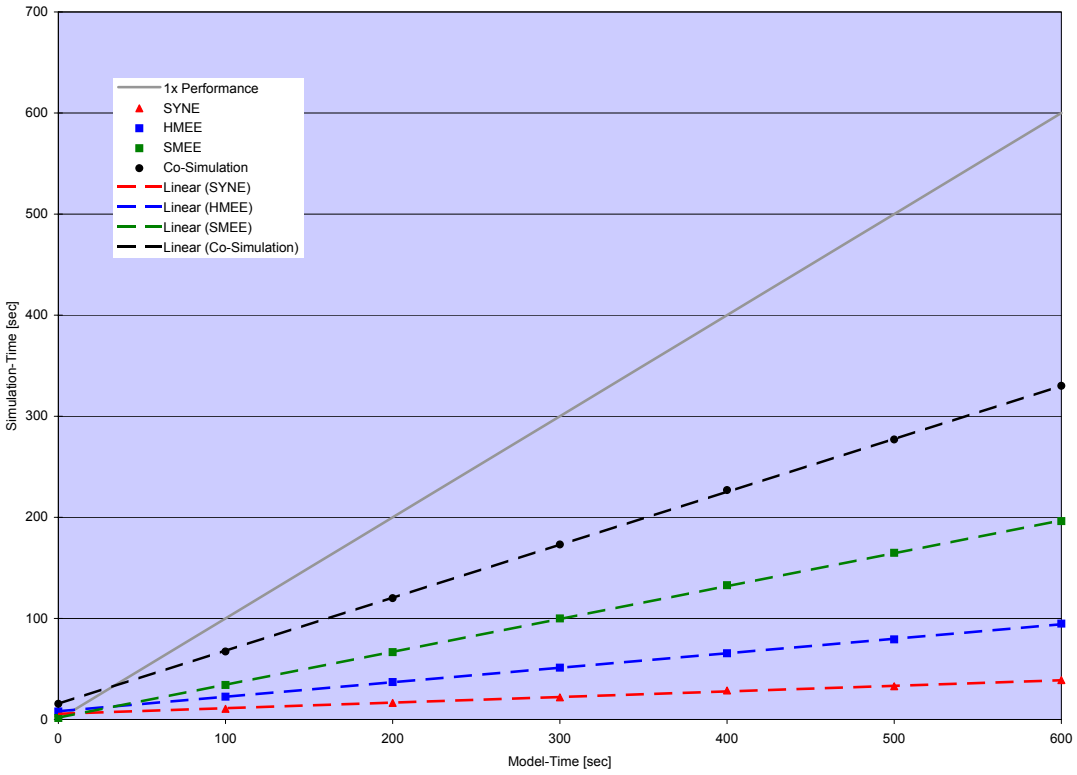

Figure 8.8: Relative performance of the co-simulation with 10 ms minimum cycle-time.

less accurate model provides a relative performance of nearly 1.9. Second, the high-level co-simulation concept in OOCOSIM demonstrates to be fast enough to simulate complex hardware/software system at a reasonable speed allowing for efficient model validation. With the same constrains, the co-simulation model is clearly slower than the homogenous model. Hence, the extra effort[15] for a co-simulation model can only justified, if the interaction of hardware and software components is relevant for the validation of the embedded system or a validation based on the implementation model is required.

Accuracy Figure 8.9 depicts the position of the load in crane simulations[16] with different step sizes or periods for the controller. The green line shows the target positions for the load (posdesired); that is, the position to which the controller should move the load. With the step size as specified in the benchmark (1ms, light-blue), all positions are reached exactly. With 10 ms step size (black), the controller needs a little more time and produces bigger overshoots, but reaches the desired positions. Note that in worst case, the difference between the models (for 1ms and 10 ms step size) reaches 73.4 cm. This difference results from the stronger and delayed overshot in the model with 10 ms step size. Whether this is acceptable or not depends on the focus of the validation.

For 11 ms step size (red), the two first target positions are nearly reached[17]; the remaining positions (-3.5, 3.5, 3.7) can not be reached at all. With 15 ms step size, even the first position can not be reached. The model behaviour for these step sizes is not sufficiently accurate for an analysis of the crane system.

[15] That is, the design time spend for modelling and simulation.
[16] It shows only the time frame between 2300 and 2640 sec model-time to provide a better resolution in the diagram.
[17] The load is near the desired position but swings cannot be eliminated.

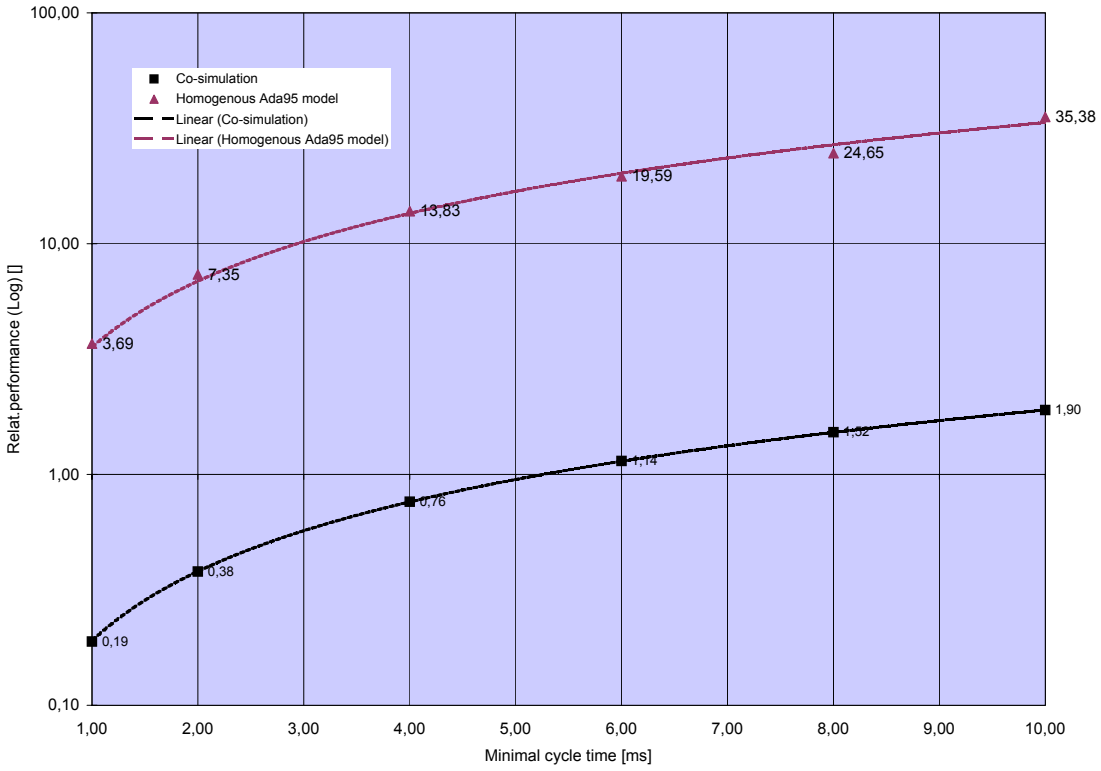

Figure 8.9: Behaviour of the crane with different step sizes.

These experiments illustrate the importance of accuracy and the coverage of timing aspects in the simulation model. It was possible to relax the constraints given by the benchmark in order to improve the performance of the model. With step sizes greater the 10 ms, the behaviour of the model changed drastically into an unstable system. Hence, increasing the step sizes is no general solution to the performance problem in general.

8.6.3 Comparison with homogeneous model

Since the relative performance relates only the simulation time and the model time, it provides no evidence for the absolute performance or the overhead of the co-simulation compared to a homogeneous version of the model. Figure 8.10 shows a comparison of the co-simulation model of the crane benchmark with the functionally equivalent homogeneous Ada95 simulation model. It shows the performance (y-axis, co-simulation in black squares, homogeneous model in purple triangles) using step sizes for the Runge-Kutta algorithm between 1 ms and 10 ms.

Two aspects in this comparison are evident. First, it shows that the homogeneous model is significantly (approximately by factor of 19) faster than the co-simulation model. There are two factors responsible for this effect:

1. The hardware model simulated in the HDL simulator is slower than its software counterpart in the co-simulation.

2. The synchronisation and communication overhead associated with the co-simulation.

While the first factor can not be substantiated directly (due to the lack of a complete and functionally equivalent VHDL model of the crane), the qualitative impact of the second factor will be discussed

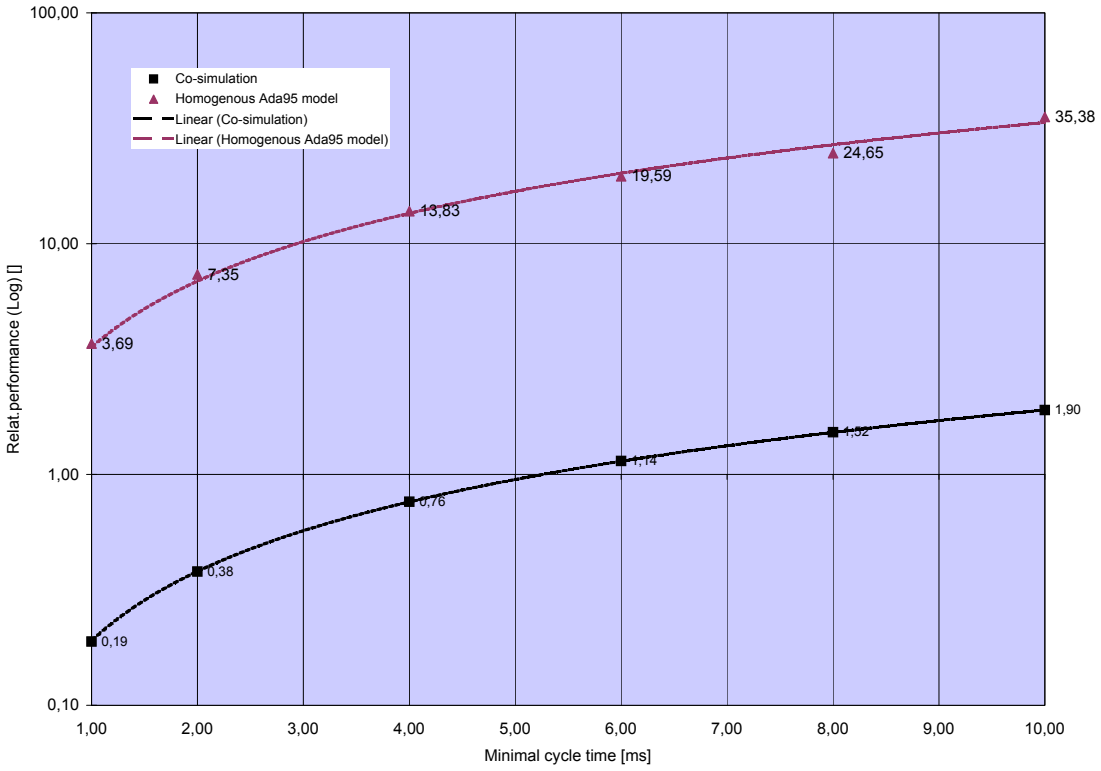

Figure 8.10: Relative performance of the homogeneous software model vs. co-simulation.

in the next section.

Second, the performance gap between the two simulation approaches remains almost constant, regardless of different levels of accuracy. That means, both models benefit to a similar extend from the relaxed accuracy requirements. With the least accurate model, both simulation approaches achieve a performance increased by a factor of 10 compared to the most accurate model. For the co-simulation this results in a maximum relative performance of 1.9; that is, the co-simulation is here even faster than the system in real-time.

8.6.4 Synchronisation and communication effort

The synchronisation and communication overhead can (without significant implementation effort) only be measured indirectly. For this purpose, a co-simulation model with only a trivial hardware and software model has been used. For the further discussion, this model will be called the *NULL-model*. The software model contains only the main procedure with an infinite loop containing one `delay until` statement and the interface model. The period of the loop is 1 ms. The hardware model contains a single process with a single `wait` statement. Due to this simplicity, in the NULL-model, T_{hard} and T_{soft} are negligible small[18].

The diagram in Figure 8.11 shows the components of the simulation effort (y-axis) for 600 sec model time with interface models of different sizes (x-axis). The green triangles represent the effort in the software model (SMEE); blue squares the represent hardware model (HMEE). Please note that both (SMEE and HMEE) contain almost only the effort for encoding and decoding the interface model to be transferred to the other model. The synchronisation effort (SYNE) is depicted in orange

[18] After removing the interface model, both models completed 1 million cycles in less than 300 ms CPU time.

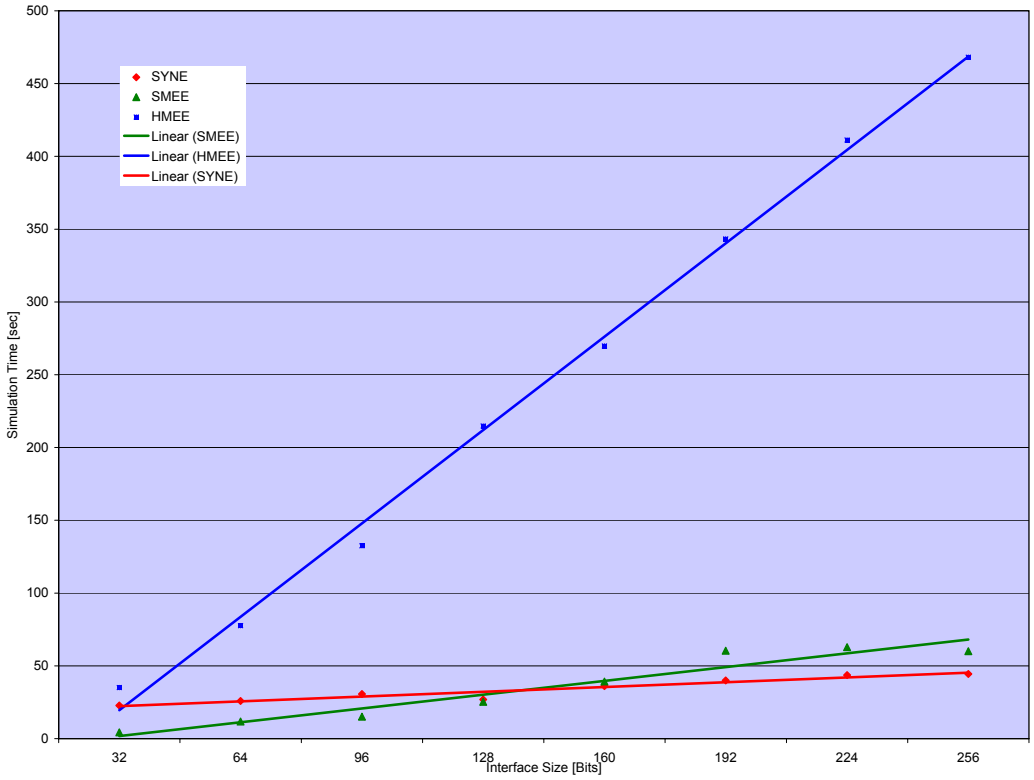

Figure 8.11: Simulation times for different interface sizes.

diamonds and corresponds to the effort spend for operation system functions responsible for process control and the exchange of data packages between hardware and software model.

From these results, the following characteristics can be observed:

1. The effort for synchronisation increases only slowly with the size of the interface; with an interface size of 64 bits it needs 26 seconds and 44 seconds for a 256 bit interface. Since the complete interface fits into one so-called *socket frame*,[19] only one data packet needs to be transfered between the models. Hence, this observation has been anticipated.

2. The effort for SMEE and HMEE increases almost linearly with the size of the interface.[20] While both models use the same C-library for encoding und decoding the data, the effort in the hardware model is higher than on the software side. This effect can be explained by the different mechanisms used to call the library routines. The software model uses an Ada95 language concept to call C-functions, while the ModelSim simulator uses the foreign model interface mechanism. It seems that the lower efficiency of the latter is responsible for the significant difference with respect to the total effort.

For large interfaces the communication overhead takes significant amount of the overall simulation time. Obviously, there are two ways to reduce the communication overhead for larger interfaces. One is to implement a more efficient encoding and decoding, especially in the hardware model. The other is to reduce the total number of co-simulation cycles in a simulation run. This however would

[19] A socket frame in an UNIX IPC contains 8192 bytes, which seems to be large enough for the vast majority of possible applications.

[20] For example, an interface of 64 bits needs 12 seconds on the software side and 78 second on the hardware side. With 256 bits it needs 60 sec. (SMEE) and 468 sec. (HMEE).

also lead to a less accurate model with respect to the synchronisation of the hardware and software model.

8.7 Hardware/Software interfaces

The hardware/software interface of both benchmarks (crane and elevator) has been modelled twice. Once by manually writing the models in Ada95/VHDL and a second time using the automated ComiX-based method as described in Chapter 7. For the ComiX-based approach, the deshico design tool (see also Section 6.6) has been used extensively.

The quantitative results for these experiments are shown in Table 8.2 for the manual approach and Table 8.3 for the ComiX-based approach, respectively. The last two rows of the tables contain the lines of code and the size of the code, including the comments. The numbers in brackets correspond to the comments only.

For the automatic interface generation, the designer only provides the specification in ComiX. The interface models for hardware, software, and the documentation are generated automatically from the specification.

In the manual design approach, hardware and software models must be written by the designer. Due to the high modelling effort, for elevator example only a simplified model with 12 instead of 21 communication objects has been used in the manual experiment.

Benchmark	Model Type	Language	Lines of Code	Code in Bytes
Crane Controller	Software	Ada95	438 (75)	13595 (3631)
	Hardware	VHDL	293 (81)	10843 (4121)
Elevator System (12 objects)	Software	Ada95	612 (202)	20348 (7695)
	Hardware	VHDL	408 (126)	15211 (6756)

Table 8.2: Model sizes for the manual interface design approach

From the quantitative figures given in the tables, the following conclusions can be drawn:

1. The ComiX models in the automatic approach are much smaller then the hardware or the software models. Moreover, the ComiX models were created with the assistance of the deshico tool. Hence, the effort to write the model was very low compared to the manual approach.

2. The models for hardware and software generated by the deshico tool are significantly larger than the manual ones. This is due to the fact, that (in general) human designers write more efficient code than automatic generators. Moreover, the ComiX-based approach automatically generates comments to improve the readability of the source-code model.

Please also note that there is documentation available for the ComiX-based model, while there is none for the manual approach. Since documenting often is a disliked job for designers, they tend to avoid the extra effort for writing documentation. It is however often necessary to have an exact description of the interface, in particular to maintain it for long times of operation.

The design effort in the ComiX-based approach was much lower than in the manual design experiment. For the crane controller, it took about 1.5 hours to specify the model in ComiX. For the elevator, it took about 1 person day to specify the systems interface. The time for the consistency checks or the generation of the code and the documentation was negligible. For the manual design approach, the development effort for the interface code cannot completely be separated from the overall design effort. Hence, it is difficult to determine the effort exactly. An estimation of the minimum design effort for the manual approach came to the result that it took about 6 person days for

Benchmark	Model type	Language	Lines of Code	Code in Bytes
Crane Controller	Specification	ComiX	64	3545
	Documentation	LaTeX	311 (10)	22510 (226)
	Software	Ada95	695 (261)	15835 (5942)
	Hardware	VHDL	541 (142)	18753 (5517)
Elevator System (21 objects)	Specification	ComiX	235	17792
	Documentation	LaTeX	984 (10)	49643 (192)
	Software	Ada95	2180 (723)	59527 (16904)
	Hardware	VHDL	1405 (263)	55998 (11746)

Table 8.3: Model sizes for the automatic interface generation (ComiX-based)

the elevator interface and about 4 person days for the crane interface. These times include significant effort needed for modifications in both models and debugging of errors caused by inconsistent hardware and software interface models.

While the consistency of the manually created models can only be checked by extensive tests, the models derived from the ComiX specification are consistent by construction. In particular in large and complex interfaces, this is a real improvement over the manual state-of-the-art approach.

Without methodological support, hardware/software interface design costs a lot of effort and is prone to errors. In both benchmarks, the manual design of the interface took large portions of the design effort. In the crane benchmark one problem was to find the right encoding of real numbers for some interface objects. Since this problem could have been solved by a library supporting the encoding, the associated effort is not included in the effort estimation given above. Still, the layout for the interface objects must be determined correctly in both models. The problems in the elevator interface were caused by the large number and complexity[21] of user defined data types and their efficient layout. The resources for this interface were quite limited. Hence, a very compact representation of the data types was required. The ComiX-based approach presented in this thesis improved the productivity significantly. Moreover, the consistency guaranteed by automatic synthesis helped to increase the safety of the design.

8.8 Recap

In this section the OOCOSIM design flow has been evaluated according to the requirement for design methods defined in Section 4.5. Due to their informal character, many aspects like e.g. seamlessness or managing of complexity could only be discussed rather than being measured. The quantitative requirements; that is, the performance and the productivity have been assessed in two benchmarks and the results have been related to state-of-the-art approaches where possible.

The performance of the co-simulation has shown its ability to deliver a performance, which allows to analyse the timing and functional behaviour of complex or at least non-trivial hardware/software systems. The relative performance in this particular benchmark application is significantly lower than that of the corresponding homogenous model. In contrast to the homogenous model, the co-simulation model allows describing and analysing the hardware and the hardware/software interface based on an implementable specification.

The interface modelling approach based on ComiX and the respective code generation mechanism have shown their benefits for the designer, first in terms of productivity and second in terms of reliability.

[21] Due to the large number of sensors (Figure 8.4) and the requirement to handle interrupts for the emergency situations, the interface design in this benchmark was quite difficult.

9 Conclusion

To conclude this thesis, in this chapter the main contributions of this work will be summarised. Some problems could not be solved within the scope of this work and therefore had to remain open. First ideas for possible solutions are presented.

The key achievement of the work at hand is a new co-design method named OOCOSIM for embedded hardware/software systems. The method starts with an abstract specification using the HRT-HOOD+ notation (Section 5.3). HRT-HOOD+ is an extension of the HRT-HOOD method, which is used for the design of hard real-time software programs. COMMUNICATION OBJECTS have been added to the basic method to create a formal and logical bridge between hardware and software.

The hardware part of the embedded system is encapsulated in abstract hardware objects in HRT-HOOD+. Unfortunately, no sound refinement for the hardware object of HRT-HOOD+ could be found. The reasons for this results mainly from the conceptual differences of hardware and software and the way they are described at language level[1].

With the appearance of OSSS [GO01, GO02], this could be changed. The method interface of so-called *shared objects* in OSSS serves well for an object-oriented design method like OOCOSIM and the fact that it is synthesisable maintains the seamless design flow advocated here. Further work could identify additional object types in the hardware domain based on OSSS and define a high-level abstraction of it similar to the HRT-HOOD+ objects. Maybe a shift from HRT-HOOD to UML could improve the industrial acceptance and provide a broader tool support.

Like for the basic HRT-HOOD method, it was necessary to provide code generation templates for the introduced COMMUNICATION OBJECTS. Since interfaces contain hardware and software components it was not possible to adopted any existing programming language for the textual description of the objects or the definition of the templates. The approach presented here, was to separate the specification of the COMMUNICATION OBJECTS from the definition of the templates. For the COMMUNICATION OBJECTS, a new language called ComiX (Section 6.3) based on XML has been introduced. The role of ComiX is central to this work, as it serves as textual representation of the graphical COMMUNICATION OBJECTS in HRT-HOOD+ and captures the properties of the interface in an implementation independent representation required for the automatic code generation into hardware and software. Thanks to its formal basis, a definition for consistent ComiX interface specifications (Section 6.4) could be found. Since this property can be checked automatically, it enables an efficient design of more reliable interfaces.

With the here introduced language TempliX, templates for a wide range of target languages can be defined. A tool processing ComiX and TempliX has been implemented to prove the applicability of this concept. With the also implemented templates, it is possible to generate synthesisable VHDL for the hardware, Ada95 for the software, and LaTeX for the documentation of a consistent ComiX specification.

The concept of validation is essential for the co-design of embedded systems. The work presents a co-simulation technique (Chapter 7) based on a discrete event simulation for the entire system model. This approach became possible with the definition of a discrete event timing-model for the software based on the real-time annex of Ada95. The translation of real-time specific and hardware/software interface related parts in the specification allows to concurrently simulate hardware and software in a target independent but time-synchronous way. This concept enables early design space exploration with the real-time behaviour already modelled and simulated. Due to the lack of an object-oriented hardware description language, hardware is not integrated well into the co-simulation

[1] The exact reasons are presented in Section 5.3.10.

presented here. With the introduction of object-orientation into hardware design (for example with so-called *transaction level modelling* in SystemC), these problems can possibly be solved in the future. With the current advances in the system description languages, another vital alternative for the validation might come up. If for example SystemC would include also software and interface aspects appropriately[2], the need for a heterogeneous co-simulation would possibly disappear.

All techniques and concept have been evaluated in modelling experiments using prototypic tools and were check for applicability. The implementation ideas for the interface synthesis tool, the translator, and the co-simulation environment are presented in this work. The benchmarks are described and the major insights from these are discussed in Chapter 8.

[2]The integration of language concepts to model at least the runtime behaviour of software has been announced for version 3.0 of SystemC. However, the release of this version has been postponed year after year.

A ComiX Syntax

```
<!--==========Entities Definitions==========-->
<!--  -->
<!ENTITY % LOGICAL "yes|no">
<!ENTITY % ENDIAN "big|small">
<!ENTITY % PART "all|architechture|memory|interrupts|
        alldeclarations|singledeclaration|allobjects|
        asynchronousobjectset|asynchronoussignalset|
        virtualasynchronoussignalset|memoryobjectset">
<!ENTITY % FLOW "bi|sh|hs">
<!ENTITY % UNIT "byte|word|longword|fourbyte|eightbyte|zero">
<!ENTITY % ACCESS "volatile|atomic">
<!ENTITY % BITORDER "low|high">

<!--==========ComiX Layer1-4==========-->
<!ELEMENT ComiX (architecture?, environment?,
        declaration?, objects?)>
<!ATTLIST ComiX
        name ID #REQUIRED
        templix IDREF #REQUIRED
        source CDATA #IMPLIED
        projectname CDATA #IMPLIED
        author CDATA #IMPLIED
        organisation CDATA #IMPLIED
        date CDATA #IMPLIED
        version CDATA #IMPLIED>

<!ELEMENT architecture (interruptblock*, memoryblock*)>
<!ATTLIST architecture
        name ID #REQUIRED
        storageunit (%UNIT;) "byte"
        addressalignment (%UNIT;) "byte"
        bitorder (%BITORDER;) "low"
        softwareendianess (%ENDIAN;) "big"
        hardwareendianess (%ENDIAN;) "big"
        comment CDATA #IMPLIED>

<!ELEMENT interruptblock  (reservedinterrupt*)>
<!ATTLIST interruptblock
        firstinterrupt CDATA #REQUIRED
        lastinterrupt CDATA #REQUIRED>

<!ELEMENT reservedinterrupt EMPTY>
<!ATTLIST reservedinterrupt
        interrupt CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT memoryblock (reservedaddress*)>
<!ATTLIST memoryblock
        startaddress CDATA #REQUIRED
        count CDATA #IMPLIED
        comment CDATA #IMPLIED>
```

```
<!ELEMENT reservedaddress EMPTY>
<!ATTLIST reservedaddress
        address CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT environment (include*, alias*)>

<!ELEMENT include EMPTY>
<!ATTLIST include
        name ID #REQUIRED
        part (%PART;) "all"
        partname NMTOKEN #IMPLIED
        path CDATA #IMPLIED
        comment CDATA #IMPLIED>

<!ELEMENT alias EMPTY>
<!ATTLIST alias
        name ID #REQUIRED
        value CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT declaration (constant*,
        (subtype|rangetype|real|enumeration|record)*)>
<!ATTLIST declaration strict (%LOGICAL;) "no">

<!ELEMENT constant EMPTY>
<!ATTLIST constant
        name ID #REQUIRED
        value CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT rangetype EMPTY>
<!ATTLIST rangetype
        name ID #REQUIRED
        range CDATA #REQUIRED
        size CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT subtype EMPTY>
<!ATTLIST subtype
        name ID #REQUIRED
        basetype NMTOKEN #REQUIRED
        range CDATA #REQUIRED
        size CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT real (fixed|float)>
<!ATTLIST real
        name ID #REQUIRED
        range CDATA #REQUIRED
        size CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT fixed EMPTY>
<!ATTLIST fixed
        delta CDATA #REQUIRED
        small CDATA #IMPLIED>
```

```
<!ELEMENT float EMPTY>
<!ATTLIST float
        digits CDATA #REQUIRED
        signed (%LOGICAL;) "yes"
        mantissasize CDATA #IMPLIED
        exponentsize CDATA #IMPLIED
        exponentbias CDATA #IMPLIED>

<!ELEMENT enumeration (item)+>
<!ATTLIST enumeration
        name ID #REQUIRED
        size CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT item EMPTY>
<!ATTLIST item
        name NMTOKEN #REQUIRED
        coding CDATA #REQUIRED>

<!ELEMENT record (component+)>
<!ATTLIST record
        name ID #REQUIRED
        size CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT component EMPTY>
<!ATTLIST component
        name NMTOKEN #REQUIRED
        type IDREF #REQUIRED
        private (%LOGICAL;) "no"
        unitnumber CDATA #REQUIRED
        init CDATA #IMPLIED
        bitrange CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT objects (asynchronousobjectset*,
        asynchronoussignalset*,
        memoryobjectset*)>
<!ATTLIST objects
        name ID #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT asynchronousobjectset (asynchronousobject)+>
<!ATTLIST asynchronousobjectset
        name ID #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT asynchronousobject EMPTY>
<!ATTLIST asynchronousobject
        name ID #REQUIRED
        packagename IDREF #REQUIRED
        handlername CDATA #REQUIRED
        type IDREF #REQUIRED
        size CDATA #REQUIRED
        dataflow (%FLOW;) "sh"
        accessmode (%ACCESS;) "atomic"
        protected (%LOGICAL;) "no"
        address CDATA #REQUIRED
        comment CDATA #IMPLIED>
```

```
<!ELEMENT asynchronoussignalset (asynchronoussignal)+>
<!ATTLIST asynchronoussignalset
        name ID #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT asynchronoussignal EMPTY>
<!ATTLIST asynchronoussignal
        name CDATA #REQUIRED
        interrupt CDATA #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT memoryobjectset (memoryobject)+>
<!ATTLIST memoryobjectset
        name ID #REQUIRED
        comment CDATA #IMPLIED>

<!ELEMENT memoryobject EMPTY>
<!ATTLIST memoryobject
        name ID #REQUIRED
        type IDREF #REQUIRED
        size CDATA #REQUIRED
        dataflow (%FLOW;) "sh"
        accessmode (%ACCESS;) "atomic"
        protected (%LOGICAL;) "no"
        address CDATA #REQUIRED
        comment CDATA #IMPLIED>
```

B Templix Syntax

```
<!ENTITY % CONTROLS  "(%CHECK;|%SIMPLE;|%SEGMENTS;|callsection)+">
<!ENTITY % ITEMS     "element|attribute|variable">
<!ENTITY % FILETYPE  "ads|adb|vhd|tex|log">
<!ENTITY % BOOL      "false|true">
<!ENTITY % FROM      "root|current">
<!ELEMENT TempliX        (%CONTROLS;)+>
<!ATTLIST TempliX        start NMTOKEN #REQUIRED
                         indentsize CDATA #REQUIRED>

<!ELEMENT template       (%CONTROLS;)>
<!ATTLIST template       name NMTOKEN #REQUIRED
                         fileext (%FILETYPE;) #REQUIRED
                         filename CDATA #REQUIRED>

<!ELEMENT forsiblings    (%CONTROLS;)>

<!ELEMENT section        (%CONTROLS;)>
<!ATTLIST section        name NMTOKEN #REQUIRED>

<!ELEMENT callsection    EMPTY>
<!ATTLIST callsection    name NMTOKEN #REQUIRED>

<!ELEMENT indentfore     EMPTY>
<!ELEMENT indentback     EMPTY>
<!ELEMENT error          EMPTY>
<!ATTLIST error          message CDATA #REQUIRED>
<!ELEMENT donothing      EMPTY>
<!ELEMENT print          EMPTY>
<!ATTLIST print          text CDATA #REQUIRED>
<!ELEMENT println        EMPTY>
<!ATTLIST println        text CDATA #REQUIRED
                         message CDATA #IMPLIED>
<!ELEMENT assign         EMPTY>
<!ATTLIST assign         variable CDATA #REQUIRED
                         value CDATA #REQUIRED>
<!ELEMENT newline        EMPTY>

<!ELEMENT child          (%CONTROLS;)>
<!ELEMENT find           (%CONTROLS;, else?)>
<!ATTLIST find           name CDATA #REQUIRED
                         of (%ITEMS;) "element"
                         from (%FROM;) "root"
                         depth (%BOOL;) "true">
<!ELEMENT if             (%CONTROLS;, (elseif | else)?)>
<!ATTLIST if             expression CDATA "this"
                         of (%ITEMS;) "element"
                         equal CDATA #REQUIRED>
<!ELEMENT elseif         (%CONTROLS;, (elseif | else)?)>
<!ATTLIST elseif         expression CDATA #REQUIRED
                         of (%ITEMS;) "element"
                         equal CDATA #REQUIRED>
<!ELEMENT else           (%CONTROLS;)>
<!ELEMENT firstnode      (%CONTROLS;, else?)>
<!ELEMENT lastnode       (%CONTROLS;, else?)>

<!ELEMENT check          EMPTY>
<!ATTLIST check          class NMTOKEN #REQUIRED
                         param CDATA #IMPLIED>
```

List of Figures

List of Tables

Bibliography

[Acc03] Accellera Organisation, Inc., http://www.eda.org/sv/SystemVerilog_3.1_final.pdf. *System Verilog 3.1 - Extensions to Verilog*, 2003.

[Ame01] Peter Amey. Logic versus magic in critical systems. In Alfred Strohmeier Dirk Craeynest, editor, *Proceedings 6th International Conference on Reliable Software Technologies - Ada-Europe 2001*, volume 2043, pages 49–67. Springer, 2001.

[Ash95] Peter J. Ashenden. *The Designers Guide to VHDL*. Morgan Kaufmann, New York, 1995.

[AWM98a] P. J. Ashenden, P. A. Wilsey, and D. E. Martin. SUAVE: Object-oriented and genericity extension to VHDL for high-level modeling. In *Proceedings of the Forum on Design Languages, FDL '98*, 1998.

[AWM98b] Peter J. Ashenden, Philip A. Wilsey, and Dale E. Martin. SUAVE: Extending VHDL to improve data modeling support. *IEEE Design & Test of Computers*, pages 34–44, 1998.

[Bar95] John Barnes. *Programming in Ada95*. Addison-Wesley, 1995.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems The Polis Approach*. Kluwer Academic Publishers, 1997.

[BE97] Th. Benner and R. Ernst. An approach to mixed systems co-synthesis. In *Procedings of the 5th International Workshop on Hardware/Software Codesign*, pages 45–52, 1997.

[BJ99] Per Bjuréus and Axel Jantsch. Heterogeneous system-level cosimulation with SDL and Matlab. In *Proceedings of the Forum on Design Languages, FDL '99*, pages 477–487, 1999. NJR.

[BJ00] Per Bjureus and Axel Jantsch. MASCOT: A specification and cosimulation method integrating data and control flow. In *Proceedings of Design Automation and Test in Europe, DATE'00*, 2000.

[BS02] Manfred Broy and Johannes Siedersleben. Objektorientierte programmierung und softwareentwicklung. *Informatik Spektrum*, 25(1):3–11, February 2002.

[BW95] Alan Burns and Andy Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier, 1995.

[CHM+99] P. Coste, F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romhdani, R. Suescuin, N. Zergainoh, and A.A. Jerraya. Multilanguage design of heterogenous systems. In *Proceedings of the CODES '99*, 1999.

[Coo98] Rick Cook. Java embeds itself in the control market, 1998.

[CoW00] CoWare Inc. *CoWare N2C User Manual - V2.2*, August 2000.

[Dis00] Pierre Dissaux. *Stood Users Manual V4.1.* TNI - Techniques Nouvelles de Informatique, January 2000.

[DMN67] O.-J. Dahl, B. Mhyrhaug, and K. Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Center, 1967.

[Dub00] Olivier Dubuisson. *ASN.1 - Communication between heterogeneous systems.* Morgan Kaufmann Publishers, 2000.

[EHS97] Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL - Formal Object-oriented Language for Communication Systems.* Prentice Hall, 1997.

[EKL94] B. Evans, A. Kamas, and E. Lee. Design and simulation of heterogeneous systems using Ptolemy. In *Proceedings of ARPA RASSP Conference*, pages 97–105, 1994.

[GDPG01] A. Gerstlauer, R. Dömer, J. Peng, and D. Gajski. *System Design - A Practical Guide with SpecC* . Kluwer Academic Publishers, Boston, 2001.

[GE03] Peter Green and Martyn Edwards. Platform modelling with UML and SystemC. In *Proceedings of the Forum on Design Languages 2003 (CDROM)*, page 11. ECSI, 2003.

[Gio98] Rinaldo Di Giorgio. Java in embedded systems. http://cloak.wpi.com:7378/javaworld/jw-09-1996/jw-09-javadev.html, 1998.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC.* Kluwer Academic Press, 2002.

[GM93] R.K. Gupta and G. De Micheli. Hardware/software cosynthesis for digital systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[GMNV00] Gulian Gorla, Eduard Moser, Wolfgang Nebel, and Eugenio Villar. System specification experiments on a common benchmark. *IEEE Design & Test of Computers*, 17(3):22–33, July-September 2000.

[GO01] Eike Grimpe and Frank Oppenheimer. Object oriented high level synthesis based on SystemC. In *Proceedings of the ICECS 2001*, 2001.

[GO02] Eike Grimpe and Frank Oppenheimer. *System on Chip Design Languages*, chapter Aspects of Object-Oriented Hardware Modelling with SystemC-Plus, pages 213–224. Kluwer Academic Publishers, 2002.

[GO03a] Eike Grimpe and Frank Oppenheimer. Extending the SystemC synthesis subset with object oriented features. In *Proceedings of the CODES/ISSS 2003, New Port Beach, California, USA*, 2003.

[GO03b] Eike Grimpe and Frank Oppenheimer. *SystemC Methodologies and Applications*, chapter Object-Oriented Hardware Sesign and Synthesis based on SystemC 2.0. Kluwer Academic Publishers, 2003.

[Gro95] HOOD Technical Group. *HOOD Reference Manual Release 4.* HOOD User's Group, 1995.

[Gro01] VSI Working Group. Ieee p1076.6/d2.01 - draft standard for VHDL register transfer level synthesis. unapproved draft. Technical report, IEEE, 2001.

[HCL+99] F. Hessel, P. Coste, P. LeMarrec, N. Zergainoh, JM. Daveau, and A.A. Jerraya. Communication interface synthesis for multilanguage specifications. In *Procedings of the Tenth Workshop on Rapid System Prototyping 1999*, pages 15–20, 1999.

[HMB00] S. Hovater, W. Marksteiner, and A. Butturini. Generation of interface design description using asis. In *Proceedings of Reliable Software Technologies Ada-Europe 2000*, number 1845 in LNCS, pages 138–148, 2000.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[Hof98] Josef Hoffmann. *MATLAB und SIMULINK : Beispielorientierte Einführung in die Simulation dynamischer Systeme*. Addison-Wesley, 1998.

[HSK00] David Harris, DeVerl Stokes, and Russel Klein. Executing an RTOS on simulated hardware using co-verification. In *Procedings of the ECS (Embedded System Conference) 2000*, 2000.

[Int99] International Standard ISO/IEC 1529. *Ada Semantic Interface Specification (ASIS)*, 1999.

[ISO86] ISO. ISO 8879:1986(e), standard generalized markup language (SGML). Technical report, International Organization for Standardization, 1986.

[JH02] Stefan Jähnichen and Stephan Herrmann. Was, bitte, bedeutet Objektorientierung? *Informatik Spektrum*, 25(8):266–276, August 2002.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[Kab02] Laila Kabous. *An Object Oriented Design Methodology for Hard Real Time Systems: The OOHARTS approach*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2002.

[KBNO03] Michael Kersten, Ramon Biniasch, Wolfgang Nebel, and Frank Oppenheimer. Erweiterung der UML um Zeitannotationen zur Analyse des Zeitverhaltens reaktiver Systeme. In Rolf Drechsler, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 11–20. Aachen: Shaker, 2003.

[KL92] A. Kalavade and E. Lee. Hardware/software co-design using Ptolemy – a case study. 1992.

[KN99] Russ Klein and Ross Nelson. Seamless cve(tm) hardware/software co-verification technology. Technical report, Mentor Graphics Corporation, 1999.

[Kop92] Helmut Kopka. *LateX: Eine Einführung*. Addison-Wesley, 1992.

[KRP+93] M. Klein, T.A. Ralya, B. Pollak, R. Obenza, and M.G. Harbour. *A Practioner's Handbook of Real-Time Analysis: A Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[Le99] Edward A. Lee and et.al. *Overview of the PTOLEMY Project*. Department of Electrical Engineering and Computer Science University of California, 1999.

[Leh02] Thomas Lehmann. *Towards Device Driver Synthesis*. PhD thesis, Department of Mathematics and Computer Science of the University of Paderborn, 2002.

[LPN98] Karsten Lüth, Thomas Peikenkamp, and Jürgen Niehaus. Hw/sw cosynthesis using statecharts and symbolic timing diagrams. In *Proceedings of the 9th IEEE International Workshop on Rapid System Prototyping*. IEEE Computer Society, 1998.

[LS75] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical Report No. 39, Murray Hill, N.J., 1975.

[MDN+03] Silvia Mazzini, Massimo D'Alessandroy, Marco Di Natale, Andrea Domenici, Giuseppe Lipari, and Tullio Vardanega. Hrt-uml: Taking hrt-hood onto uml. In *Proceedings of the Ada Europe 2003, LNCS 2655*, pages 406–416. Springer-Verlag Berlin Heidelberg, 2003.

[MEI04] MEIJE Team : Concurrency, Synchronisation, Reactivity, http://www-sop.inria.fr/meije/esterel/esterel-eng.html. *The ESTEREL Language*, 2004.

[Mey90] Bertrand Meyer. *Objektorientierte Softwareentwicklung*. Hanser, Prentice-Hall, 1990.

[MGK97] Jan Madsen, Jesper Grode, and Peter V. Knudsen. Hardware/software partitioning using the Lycos system. In J. Staunstrup and W. Wolf, editors, *Hardware/Software Codesign - Principles and Practice*, pages 283–305. Kluwer Academics Publishers, 1997. Good definition of partitioning.

[MN99] E. Moser and W. Nebel. Case study: System model of crane and embedded control. In *Proceedings of Design Automation and Test in Europe, DATE'99*, pages 721–723. IEEE Computer Society, 1999.

[MNPRR97] Serge Maginot, Wolfgang Nebel, Wolfgang Putzke-Röming, and Martin Radetzki. Final Objective VHDL language definition. Technical report, ESPRIT Project 20616 : Request, 1997.

[MRC+00] Fabrice Merillon, Laurent Reveillere, Charles Consel, Renaud Marlet, and Gilles Muller. Devil : An IDL for hardware programming. Technical report, INRIA Rennes, France, http://www.inria.fr/rrrt/rr-3977.html, July 2000.

[NBS+02] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th conference on Design automation*, pages 22–27, http://doi.acm.org/10.1145/513918.513927, 2002. ACM Press.

[Neu00] Dr. Horst A. Neumann. Interaction of active objects via shared protected objects: UML design and ada realisation. In *Ada Europe Conference 2000, Potsdam*, 2000. Not in the Proceedings.

[OJ00] Mattias O'Nils and Axel Jantsch. Operating system sensitive device driver synthesis from implementation independent protocol specification. In *Proceedings of the DATE Conference and Exibition 1999*, 2000.

[OMG01] OMG, http://www.omg.org. *OMG Unified Modeling Language Specification (Version1.4)*, Sep 2001.

[OS99] Frank Oppenheimer and Guido Schumacher. OOCOSIM - objektorientierte Spezifikation und Simulation eingebetteter Realzeitsysteme. In Hubert B. Keller, editor, *Workshop Objektorientierung und sichere Software mit Ada*, pages 1–11. Institut für Angewandte Informatik, FZI Karlsruhe, 1999.

[OSC01a] OSCI (Open SystemC Initiative), http://www.SystemC.org. *Functional Specification for SystemC2.0 - Final Version*, 2001.

[OSC01b] OSCI (Open SystemC Initiative), http://www.SystemC.org. *SystemC2.0 Users Guide*, 2001.

[OSN99] Frank Oppenheimer, Guido Schumacher, and Wolfgang Nebel. OOCOSIM - eine objektorientierte methode zur Spezifikation und Simulation eingebetteter Systeme in Realzeitumgebungen. *it+ig Schwerpunktthema: Entwurfsmethoden für eingebettete Systeme*, 41(2):27–31, 1999.

[OSN00a] Frank Oppenheimer, Guido Schumacher, and Wolfgang Nebel. Modellierung und Simulation eines Portalkrans mit der OOCOSIM-Methode. In *Proceeding of the AES2000, Paderborn*, pages 123–129, 2000.

[OSN00b] Frank Oppenheimer, Guido Schumacher, and Wolfgang Nebel. Section: Portal crane cosimulation in Ada95/Objective VHDL in system specification experiments on a common benchmark. *IEEE Design & Test of Computers*, 17(3):26–27, July-September 2000.

[OZN01a] Frank Oppenheimer, Dongming Zhang, and Wolfgang Nebel. COHSID: ComiX HW/SW Interface Designer. In Jürgen Ruf and Carsten Schulz-Key, editors, *Demonstrations at University Booth at the DATE Conference 2001*, page 10. Wilhelm-Schickard-Institute for Computer Science, 2001.

[OZN01b] Frank Oppenheimer, Dongming Zhang, and Wolfgang Nebel. Modelling Communication Interfaces with ComiX. In Alfred Strohmeier Dirk Craeynest, editor, *Proceedings 6th International Conference on Reliable Software Technologies - Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2001.

[Por00] Brett Porter. Ada 95 suits embedded programming. *Embedded Developers Journal*, pages 24–29, May 2000.

[RBMD03] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, and Nikil Dutt. An efficient retargetable framework for instruction-set simulation. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign & system synthesis*, pages 13–18, 2003.

[RK24] C. Runge and H. König. *Vorlesung über numerisches Rechnen.* Julius Springer, Berlin, 1924.

[RMD03] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation. In *Proceedings of the 40th conference on Design automation*, pages 758–763, http://doi.acm.org/10.1145/775832.776026, 2003. ACM Press.

[RPRN98a] M. Radetzki, W. Putzke-Röming, and W. Nebel. Übersetzung von Objektorientiertem VHDL nach Standard VHDL. In *9. GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, 1998.

[RPRN98b] Martin Radetzki, Wolfram Putzke-Römig, and Wolfgang Nebel. A unified approach to object-oriented VHDL. *Journal of Information Science and Engineering*, 14:523–545, 1998.

[RPRN98c] Martin Radetzki, Wolfram Putzke-Röming, and Wolfgang Nebel. Objective VHDL: Tools and application. In *Proceedings of the Forum on Design Languages, FDL'98*, 1998.

[RVBM96] Karl Van Rompaey, Diederik Verkest, Ivo Bolsens, and Hugo De Man. CoWare - a design environment for heterogeneous hardware/software systems. In *Proceedings of the EURODAC 1996*, 1996.

[RWT] RWTH Aachen - Lehrstuhl fur integrierte Systeme der Signalverarbeitung, http://servus.ert.rwth-aachen.de/lisa/. *LISA - Language for Processor Design.*

[Sag98] Vivek Sagdeo. *The complete Verilog book.* Kluwer Academic Press, 1998.

[Sch99] Guido Schumacher. *Object-oriented hardware specification and design with a language extension to VHDL.* PhD thesis, Carl v. Ozzietzky University of Oldenburg, 1999.

[Sel98] Bran Selic. Using UML for modeling complex real-time systems. In A. Bestavros F. Mueller, editor, *Languages, Compilers, and Tools for Embedded Systems - ACM SIGPLAN Workshop LCTES'98*, number 1474 in LNCS, pages 250–260. Springer, 1998.

[SG00] Luc Séméria and Abhijit Ghosh. Methodology for hardware/software co-verification in C/C++. Technical report, OSCI, http://www.systemC.org/papers/05b_4.pdf, 2000.

[SN98] Guido Schumacher and Wolfgang Nebel. How to avoid the inheritance anomaly in ada. In L. Asplund, editor, *Reliable Software Technologies - Ada-Europe'98, Uppsala, Schweden*, pages 53–64, 1998.

[SON04] Andreas Schallenberg, Frank Oppenheimer, and Wolfgang Nebel. Designing for dynamic partial reconfigurable fpgas with systemc and osss. In *Proceedings of the Forum on Design Languages 2004*, 2004.

[Ste98] Richard W. Stevens. *UNIX network programming - 1. Networking APIs : sockets and XTI.* Prentice Hall, 1998.

[Ste99] Richard W. Stevens. *UNIX network programming - Interprocesses communications.* Prentice Hall, 1999.

[Sto92] Alexander D. Stoyenko. The evolution and state-of-the-art of real-time languages. *The Journal of Systems and Software*, pages 61–84, April 1992.

[SUN87] SUN Microsystems, Inc., http://jandfield.com/rfcs/rfc1014.html. *XDR : External Data Representation standard, RFC 1014*, 1987.

[Swa01] Stuart Swan. An introduction to system-level modelling in SystemC 2.0. Technical report, Cadence, http://www.systemc.org/papers/SystemC_WP20.pdf, 2001.

[Syn01a] Synopsys, Inc. *CoCentric SystemC Compiler Behavioral Modelling Guide*, 2001.

[Syn01b] Synopsys, Inc. *CoCentric SystemC Compiler RTL User and Modelling Guide*, 2001.

[TD97] S. Tucker Taft and Robert A. Duff. *Ada95 Reference Manual.* Number 1246 in LNCS. Springer, 1997.

[TS04] Sharon Tan and Gary Smith. Conservative times, conservative design in EDA. Technical report, Gartner Inc., Feb. 2004.

[Vah03] Frank Vahid. The softening of hardware. *IEEE Computer*, pages 27–34, April 2003.

[VCI00] VCI Inc., http://www.EASICS.com/wwwtools/vcic/vcic_whitepaper/vcic_whitepaper.html. *Hardware-software interface design*, 2000.

[VSV99] Steven Vercauteren, Jan Van Der Steen, and Diederik Verkest. Combining software synthesis and hardware/software interface generation to meet hard real-time constrains. In Dominique Borrione and Rolf Ernst, editors, *Proceeding of Design, Automation and Test in Europe DATE 1999*, pages 556–561. IEEE Computer Society, 1999.

[W3C] W3C, http://www.w3.org/TR/REC-xml. *Extensible Markup Language (XML) 1.0.*

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communication of the ACM*, 14(4):221–227, 1971.

[Wir85] Niklaus Wirth. *Programming in Modula 2.* Springer, 1985.

[XB03] Cong-Cong Xing and Bourmediene Belkhouche. On pseudo object-oriented programming considered harmful. *Communication of the ACM*, pages 115–117, October 2003. Discussion of what is essential in OO design style. In particular it states that classes are non-essential for OO.

[Zha99] Dongming Zhang. Modellierung eines eingebetteten Echtzeitsteuerungssystems am Beispiel eines Personenaufzugs. Technical report, Carl von Ossietzky University of Oldenburg, 1999.

[Zha01] Dongming Zhang. Kommunikationsmodellierung für HW/SW-Systeme. Master's thesis, Carl von Ossietzky University of Oldenburg, 2001.

Curriculum Vitae

1967	Born in Delmenhorst
1973 - 1977	Grundschule Adelheide in Delmenhorst Elemenatry school
1977 - 1979	Orientierungsstufe Schulzentrum Süd in Delmenhorst
1979 - 1984	Realschule Königsberger Strae in Delmenhorst
1984 - 1988	Fachgynasium Technik at the BBS II in Delmenhorst
1988 - 1989	Military training
1989 - 1997	Study of 'Allgemeine Informatik' General computer science at the Carl von Ossietzky University Oldenburg with Diplom in Informatik (MSc in Computer Science)
1997	Work as freelancer and consultant for computer applications
01/1998 - 09/2001	Research assistant at the Carl von Ossietzky University Oldenburg
10/2001- date	Group manager System Design Methodology at Kuratorium OFFIS e.V.

Decalration of original work

I herewith declare to have written this thesis only based on the sources listed and without the help of others. I have not submitted or prepared the submission of this or any other doctoral thesis at the Carl von Ossietkzy University Oldenburg or any other university.

Hiermit erkläre ich, diese Arbeit ohne fremde Hilfe und nur unter Verwendung der angegebenen Quellen verfasst zu haben. Ich habe bis dato weder an der Carl von Ossietzky Universität Oldenburg noch an einer anderen Universität die Eröffnung eines Promotionsverfahrens beantragt oder anderweitig eine Promotion vorbereitet.

_____

(Doktorand)