Specification, Synthesis and Validation of Hardware/Software Interfaces

Mattias O'Nils

Stockholm 1999

Electronic system Design, Department of Electronics
Electrum 229, Isafjordsgatan 22-26
S-164 40 Kista, Sweden

Submitted to the School of Electrical Engineering, Royal Institute of technology, in partial fulfillment of the requirements for the degree of Technical Doctor.

Abstract

Design based on intellectual property (IP) is emerging to close the gap between steadily increasing capacity, in terms of transistors on integrated devices, and design productivity, in terms of the number of transistors designed in a given period. However, the integration of several IP blocks into a single system on one chip makes the specification and implementation of interfaces (for example, bus interfaces and device drivers) a dominant design problem for embedded systems. There is therefore a need for effective ways of modelling, refining, and implementing communication within embedded systems.

This thesis presents one approach to hardware/software interface synthesis that ranges from the specification to the implementation and validation of hardware/software interface protocols. The information required for hardware/software interface synthesis is separated into three parts: the protocol specification, information related to the operating system, and information related to the processor. From these inputs a synthesis tool generates (a) device driver functions, (b) a combination of device driver functions and a DMA controller, or (c) simulation models, depending on what the designer decides. The clean separation of information facilitates (1) efficient design space exploration with combinations of different processors, operating systems and protocols, and (2) efficient maintenance of a large number of different versions and variants of hardware/software interfaces. The three-phase validation approach is based on standard simulation methods and facilitates simulation of the interfaces at several steps during development. We keep all the simulation models consistent with both the specification and the implementation by generating the models using the same technique that is used for synthesis. Validation in several phases is justified (1) by the faster simulation of early phases (up to four times faster than late phases), and (2) by allowing both hardware designers and software developers to work in their familiar tool environments as long as possible.

Protocols are specified as a grammar, which is fully independent of architecture and implementation. After the initial selection of implementation alternatives, the methods presented are fully automated. Using real-life examples we demonstrate the effectiveness of the simulation models and show that the quality of the generated code is close to handwritten quality in terms of performance, area and code size.

Acknowledgements

I would like to begin by expressing my gratitude to Professor Hannu Tenhunen for accepting me as a PhD student at the Electronic System Design Laboratory. I would also like to send huge thanks to Dr Axel Jantsch for agreeing to be my supervisor and for his little jokes, which helped me overcome the setbacks caused by rejected papers and other obstacles. In my opinion Axel's supervision has been a textbook example of how a supervisor should work.

I am very grateful to Professor Ahmed A. Jerraya for agreeing to be the opponent of this thesis.

I would also like to thank the people at the ESD Lab at the Royal Institute of Technology, especially Henrik Olson for making my stays in Kista more fun, and Johnny Öberg for his help during this work, together with the people at ITE, Mid Sweden University, for being good colleagues, specially Hans-Erik Nilsson and Bengt Oelmann for all the useful and fun discussions. Thanks also to Miguel Berg and the system administrators in both Stockholm and Sundsvall for helping me with computer problem and, of course, a big thank you to my golfing partners Claes Lindkvist, Stefan Pettersson and Per-Erik Näsholm, who together with the floorball team Sundsvall Academics helped take my mind off work.

Ingalill Arnfridsson, you're a well of patience and smiles. Thanks for all the help with arranging my travels and other boring administrative stuff during my time as a PhD student.

I would like to thank the following for their financial support: Mid Sweden University, the Swedish National Board for Industrial and Technical Development – NUTEK, and the Swedish Foundation for Strategic Research – SSF.

Lastly I would like to take the opportunity to thank my loving girlfriend Susanne for all her support and love. Without you, life would be so boring.

Sundsvall, May 1999

Mattias O'Nils

Contents

Abbreviations and Acronyms

General

ADC. . . . . Analog to Digital Converter
AMPS . . . . Advanced Mobile Phone System
API. . . . . Application Program Interface
ARM . . . . Advanced RISC Machines
ASIC . . . . Application Specific Integrated Circuit
ATM. . . . . Asynchronous Transfer Mode
CAD. . . . . Computer Aided Design
CAN. . . . . Controller Area Network
CISC . . . . Complex Instruction Set Computer
CMOS . . . . Complementary MOS
CORBA . . . . Common Object Request Broker Architecture
CSP . . . . Communicating Sequential Processes
D-AMPS . . . . Digital AMPS
DAC. . . . . Digital to Analog Converter
DMA . . . . Direct Memory Access
DMAC . . . . DMA Controller
DRAM. . . . . Dynamic RAM
DSP . . . . Digital Signal Processing
EDA. . . . . Electronic Design Automation
FIFO . . . . First In First Out
FSM. . . . . Finite State Machine
GSM . . . . Global System for Mobile Communications
HCCS. . . . . Host Code Cosimulation
HDL. . . . . Hardware Description Language
HLS . . . . High Level Synthesis
HW . . . . Hardware
I/O . . . . Input/Output

I^2C . . . . Inter Integrated Circuit bus
IEE . . . . Institution of Electric Engineers
IEEE . . . . Institute of Electrical and Electronic Engineers
IP . . . . Intellectual Property
ISR . . . . Interrupt Service Routine
ISS . . . . Instruction Set Simulator
LoC . . . . Lines of Code
MOS . . . . Metal-Oxide-Silicon
NUTEK . . . . Swedish National Board for Industrial and Technical Development
OAM . . . . Operation and Maintenance
OS . . . . Operating System
ProGram . . . . Protocol Grammar
RAM . . . . Random Access Memory
RISC . . . . Reduced Instruction Set Computer
ROM . . . . Read Only Memory
RPC . . . . Remote Procedure Call
RTL . . . . Register Transfer Level
RTOS . . . . Real-Time Operating System
PC . . . . Personal Computer
PDA . . . . Personal Digital Assistant
SDL . . . . Specification and Description Language
SSF . . . . Swedish Foundation for Strategic Research
SW . . . . Software
TCCS . . . . Target Code Cosimulation
TDMA . . . . Time Division Multiple Access
UART . . . . Universal Asynchronous Receiver Transmitter
USART . . . . Universal Synchronous/Asynchronous Receiver Transmitter
VHDL . . . . VHSIC Hardware Description Language
VHSIC . . . . Very High Speed Integrated Circuit
VLSI . . . . Very Large Scale Integration
VSI . . . . Virtual Socket Initiative
YACC . . . . Yet Another Compiler Compiler

Research groups and approaches

AKKA . . . . Royal Institute of Technology's early codesign approach
CASTLE . . . . Codesign and Architecture driven Synthesis TooL Environment
CCG. . . . . Communication Classification Graph, defined in [89]
CFSM . . . . Codesign FSM, defined in [28]
Chinook . . . . University of Washington's codesign approach
CodeSign . . . . Object oriented codesign approach, ETH, Zurich, Swiss
ComSyn . . . . Royal Inst. of Techonlogy's HW/SW interface synthesis tool
COSMOS. . . . . An SDL-based Codesign Tool, TIMA, Grenoble, France
COSYMA . . . . COSYnthesis for eMbedded Architectures
C^x. . . . . Super set of C, defined in [50]
ESG . . . . Extended Syntax Graph, defined in [19]
FGM . . . . Flow Graph Model, defined in [83]
HardwareC. . . . . Modified C dialect for hardware synthesis
LYCOS . . . . LYngby COSynthesis system
Polis . . . . A codesign system for control application, Berkley, USA
PSM . . . . Program State Machine, defined in [55]
RTC . . . . Register Transfer C, defined in[21]
SOLAR . . . . FSM based design representation, used in COSMOS
SpecSyn . . . . University of California - Irvine's codesign approach
STG . . . . Signal Transition Graph, defined in [80]
Tosca . . . . Politecnico di Milano's codesign approach
Vulcan . . . . Early codesign approach presented in [58]

List of Figures

List of Papers

Papers included in this thesis:

1. Mattias O'Nils, Axel Jantsch, "Communication in Hardware/Software Embedded Systems - A Taxonomy and Problem Formulation", *Proceedings of the 15th IEEE Norchip Conference*, pp. 67-74, 1997.

2. Mattias O'Nils, Johnny Öberg, Axel Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", *Proceedings of EUROMICRO Conference*, pp. 55-58, 1998.

3. Mattias O'Nils, Axel Jantsch, "Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification", *Proceedings of Design, Automation and Test in Europe (DATE)*, pp. 562-567, 1999.

4. Mattias O'Nils, Axel Jantsch, "Generation of DMA Controllers from Device Driver Descriptions", *Proceedings of IEEE VLSI Design Conference*, pp. 138-145, 1999.

5. Mattias O'Nils, Axel Jantsch, "Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications", submitted to *Design Automation for Embedded Systems*, Kluwer Academics Publisher, 1999.

6. Mattias O'Nils and Axel Jantsch, "Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models", *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, pp. 40-46, 1998.

7. Mattias O'Nils, Axel Jantsch, "HW/SW Interface Validation in IP based System Design", *Proceedings of International. Workshop on IP based Synthesis and System Design*, pp. 79-84, 1998.

8. Mattias O'Nils, Axel Jantsch, "Refinement of HW/SW Communication Channels: Case Study and Comparison", *Proceedings of the 16th IEEE Norchip Conference*, pp. 230-237, 1998.

Other papers:

1. Mattias O'Nils, Kalle Tammemäe, Axel Jantsch, Ahmed Hemani, Hannu Tenhunen, “Experiences using Akka: A Hardware-Software Codesign Tool Kit in design of Telecommunication systems”, Poster session of CAVE'95 Workshop, 1995.

2. Kalle Tammemäe, Mattias O'Nils, Axel Jantsch, Ahmed Hemani, “AKKA: A Codesign Environment”, *Proceedings of 13th Norchip Conference*, pp. 249, 1995.

3. Mattias O'Nils, Axel Jantsch, Ahmed Hemani, Hannu Tenhunen, “Interactive Hardware-Software Partitioning and Memory Allocation Based on Data Transfer Profiling”, *Proceeding of International Conference on Recent Advances in Mechatronics*, pp. 447-452, 1995.

4. Kalle Tammemäe, Mattias O'Nils, Anders Tornemo, Hannu Tenhunen, “VLSI System Level Codesign Toolkit AKKA”, *Proceedings of the 14th IEEE Norchip Conference*, pp. 196-202, 1996.

5. Mattias O'Nils, Kalle Tammemäe, Axel Jantsch, Ahmed Hemani, “Design of D-AMPS Channel Decoder with Codesign Methodologies”, *Proceedings of Baltic Electronic Conference*, pp.397-400, 1996.

6. Kalle Tammemäe, Mattias O'Nils, Ahmed Hemani, “Flexible Codesign Target Architecture for Early Prototyping of CMIST Systems”, *Field-Programmable Logic Smart Application, New Paradigms and Compilers*, Springer-Verlag, ISBN 3-540-61730- 2, pp.193-199, 1996.

7. Kalle Tammemäe, Mattias O'Nils, Axel Jantsch and Ahmed Hemani, “AKKA: A Toolkit for Cosynthesis and Prototyping”, *IEE Digest no:96/036 of Colloquim on Hardware-Software Cosynthesis for Reconfigurable Systems*, pp.8/1-8/8, 1996.

8. Bengt Oelmann, Mattias O’Nils, “Asynchronous Control of Low-Power Gated-Clock Finite-State Machines”, *Proceedings of IEEE International Conference on Electronics*, Circuits and Systems, 1999.

9. Bengt Oelmann, Mattias O'Nils, “A Low Power Hand-Over Mechanism for Gated-Clock FSMs”, *Proceedings of the European Conference on Circuit Theory and Design*, 1999.

1 Introduction

Many types of electronic systems exist and they can be found almost everywhere, for example, cellular telephones, toys, cars, and so on. Looking at the evolution of electronic products one can identify several characteristics that are becoming more important, and will therefore also influence the design process. The characteristics are increased system functionality, more-portable systems (that is, battery operated), and shorter *product lifetime*. Table 1.1 outlines how the different characteristics will influence the design process. The first and second columns indicate the product trend. Columns 3 and 4 show how each trend affects the design and the design process.

Table 1.1: Electronic product trends and their effects on product design.

Product trend	Δ	This leads to	Δ
Increasing complexity	$\nearrow$	Increasing design time	$\nearrow$
		Increasing power	$\nearrow$
Decreasing power	$\searrow$	Increasing design time	$\nearrow$
Decreasing product lifetime	$\searrow$	Decreasing design time	$\searrow$

Both the increase in system complexity and the requirements for low-power operation lead to increased design time. Increased complexity can also lead to increased power consumption. Design time will therefore increase because the designer needs to put more effort into low-power design. At the same time, the decreased *product lifetime* requires a reduction in design time. The design task will thus become more complex and the design time for these systems will need to be reduced. In short, the electronic systems design process has to be made more efficient by using better design methods.

In the past the driving force behind increased electronic system complexity has been the advances in silicon technology. Figure 1.1 shows that the gap is increasing between silicon capacity, in terms of the number of gates on a chip, and the size of designs. Thus the limitations are moving towards design productivity rather then technology capacity. This is becoming a problem for silicon vendors because their production capacity is not fully used, leading them to take more interest in design issues in recent years [6,15].

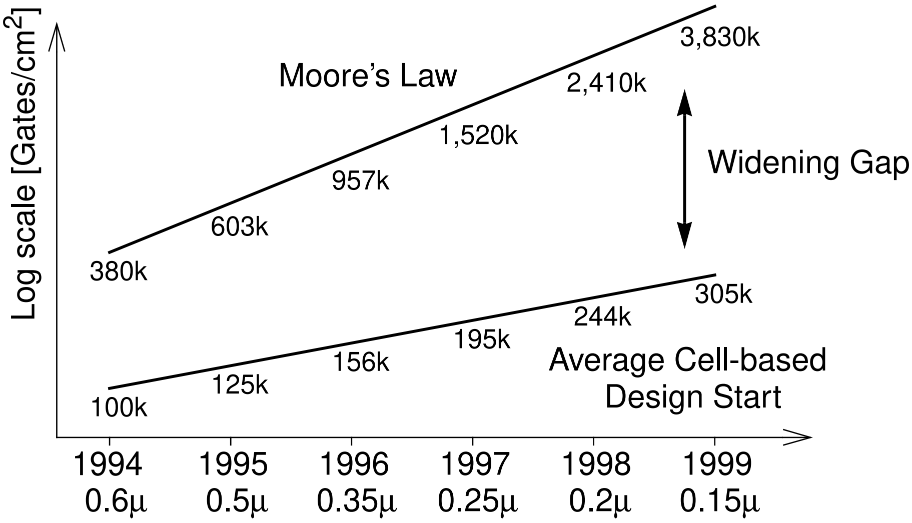


Figure 1.1: The growing gap between what engineering teams can design and what fabrication lines can economically produce [53].

In conclusion we see that there is a demand for more-efficient design methods from both consumers and producers of electronic systems. This demand also includes efficient design methods for low power. There are several ways of increasing electronic design productivity. This thesis concentrates on the system design of the digital parts of embedded systems. However, there exist several unsolved problems in other areas such as analog design [78], electronic system packaging [9] and the integration/testing of electronic systems [3,68].

There are two major ways of achieving increased design productivity: (1) using computer aided design (CAD) tools, and (2) using design based on intellectual property (IP) components. There are several techniques to achieve low-power operation for a system. This is also reflected in several research approaches for low-power implementations, for example: asynchronous logic [96], highly parallel low-speed structures [22,26], isosynchronous techniques [5,61] or power management techniques [17].

The predominant computer aid for hardware design is still schematic capture and logic synthesis [39]. Design tools for higher levels of abstractions are becoming more mature and have resulted in tools for graphical design [103] and tools for high-level synthesis (HLS) [24,54].

The reuse of hardware IP components is starting to gain recognition in the designer community. The VSI Alliance [117] for the documentation and exchange of hardware IP components is a good start, but there are still many issues that have to be solved to enable the complete success of IP-based hardware design, for example, copyright protection [27], the validation of IP components [38,93] and CAD tools for IP-based design (communication synthesis) [100,115,124].

Tools accepted and widely used by the software-designer community are C cross compilers and debuggers for particular microprocessors. These tools are usually not portable, that is, a software development system will only support one processor family (processor architecture). Only a few software CAD tools support different optimization strategies and analysis methods for size/performance/power constraints. Tools exists for higher levels of abstractions for software synthesis, that is, case tools [34,42]. These tools enable designers to work on a higher level of abstraction while the tools themselves handle the low-level implementation details. These tools have not yet gained widespread recognition. The reuse of software IP components (code) has been proposed and debated for a long time in the software community. It is only recently, with approaches like Corba [35] and JavaBeans [73], that IP-based software design has gained acceptance among designers.

Hardware/software codesign[1] is the next evolutionary step for embedded systems CAD tools. Hardware/software codesign research has been going on for about 10 years [46,55,82,118]. Although the joint design of hardware and software is much older than this, the concept of hardware/software codesign is to merge the design flows for hardware and software into one hardware/software design flow. In recent years codesign research has resulted in a few commercial design tools like Coware [21], Arexsys [105] and Felix [104]. Most approaches to hardware/software codesign focus on top-down design, that is, starting from a high-level specification that is then refined into an implementation. Only a few research groups focus on methods for IP-based design [47,92,122].

1.1 Thesis outline

The next section goes through embedded systems, embedded systems' design and the research in hardware/software codesign. Section 3 analyses the task of hardware/software interfacing and describes the related work in this area. It also describes the ComSyn system, that is, the hardware/software communication synthesis system presented in the appended papers. Section 4 summarizes the work covered by all papers included in this

1. "HW/SW codesign means meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design." – Giovanni De Micheli, Stanford University.

thesis. Section 5 summarizes and concludes the contributions of this thesis. It also indicates unresolved problems in communication synthesis. The eight papers that present the original contribution are appended to the end of the thesis.

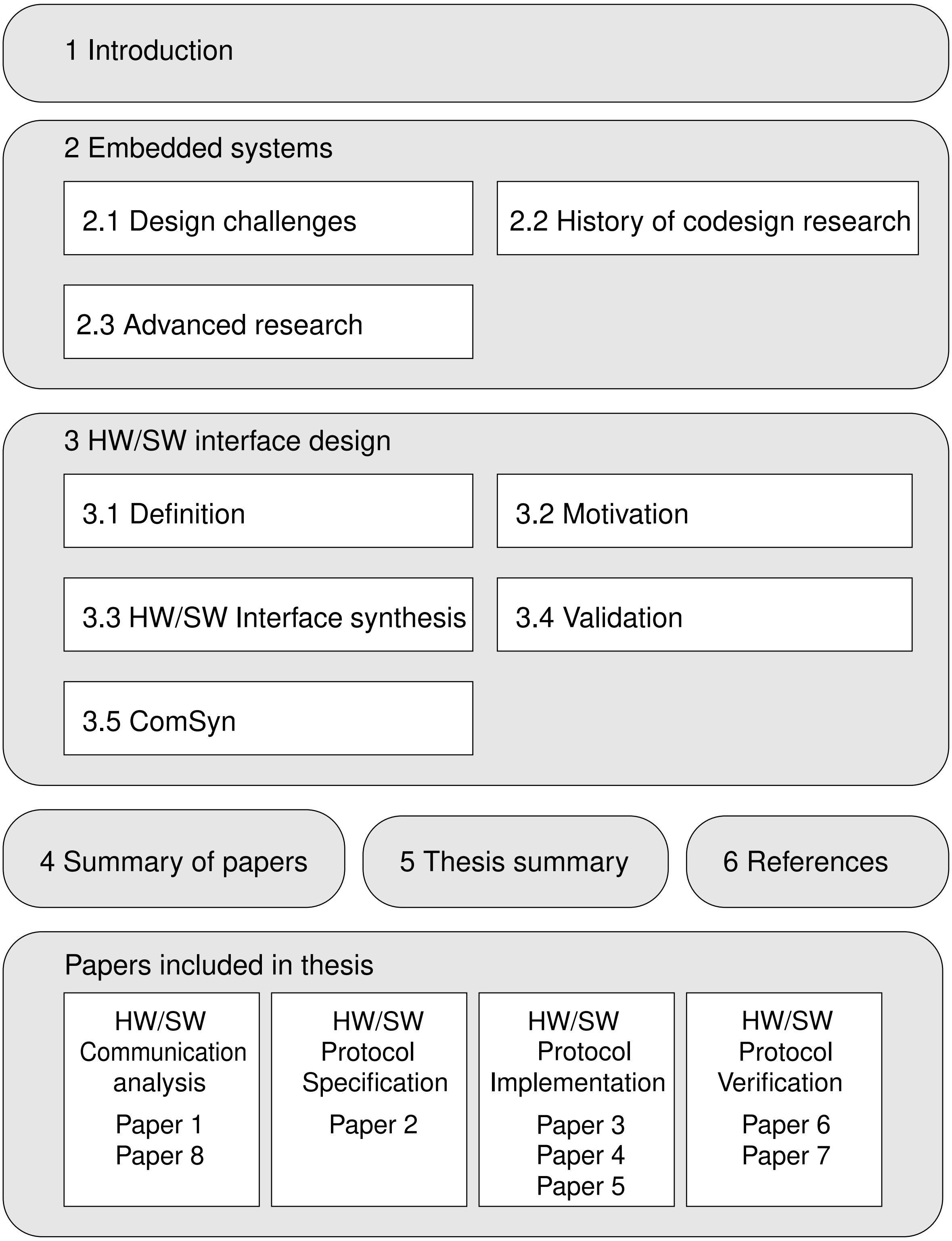


Figure 1.2: Organization of the thesis.

2 Embedded systems

Today embedded computer systems have become everyday gadgets for most people, who use them even without even knowing it. Figure 2.1 and the bulleted items below show some examples of things, taken from the author's daily life, that contain one or more embedded computer systems. Each example includes a description of tasks that can be carried out by embedded systems.

- *microwave oven* – providing the user interface and controlling the cooking of food.
- *audio equipment* – providing the user interface and processing the audio data.
- *engine preheater* – providing the user interface and controlling the power to the preheater in accordance with the user's wishes and the climatic conditions.
- *cars* – today cars usually contain several embedded systems: anti-lock brake controller, ignition control, air conditioning controller, etc.
- *cellular phone* – providing the user interface, encoding data, speech compression, etc.
- *TV set-top box* – controlling the on-screen user interface, image processing, etc.
- *Answering machine* – compressing and storing speech data.

One reason why embedded computer systems are so popular is the combination of high-performance hardware with flexible software that has a short design time. The first embedded systems were banking and transaction systems running on mainframes and

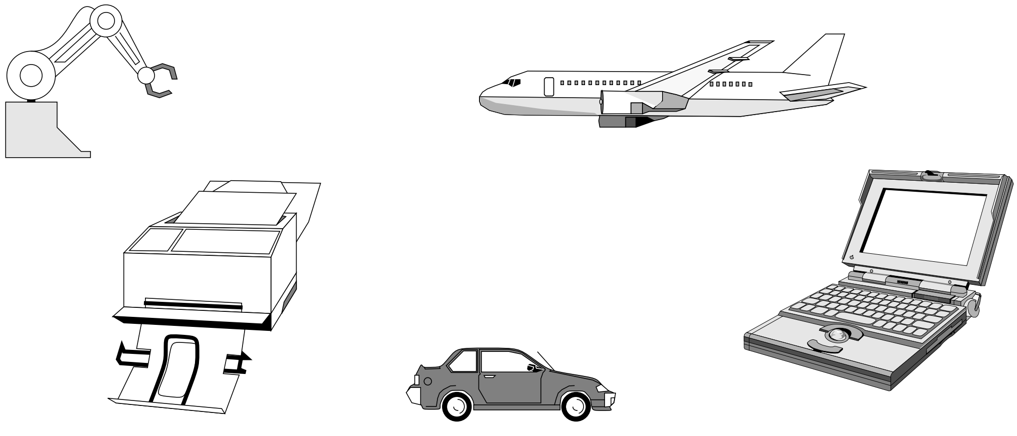


Figure 2.1: Products that include embedded systems.

arrays of disks. Since this sort of equipment was built from very expensive components, it was used for relatively few but important applications. When equipment was as expensive as the early mainframes were, it was easy to motivate high costs for the design and maintenance of systems.

What is an embedded system? There are probably as many definitions of what an embedded system is as there are designers. The word embedded implies that the system will not change after it has been designed, and that it has a specific task. This makes sense for the examples above, but there is actually no generally accepted definition of what an embedded system is. For example, is a PC that only runs a phone book application an embedded system? Or is a PDA that uses the same processor running the same phone book application an embedded system? All embedded systems mentioned in this thesis refer to hardware/software systems with one or more microprocessors, similar to the system shown in figure 2.2.

An embedded system has several components (see figure 2.2), but the most important device, which has enabled the evolution of embedded systems, is the microprocessor [64]. The microprocessor has evolved from the simple 4004 [33] from Intel with only a few thousand transistors, to the microprocessors of today (1999) with 6.5 to 100 million transistors on a single chip, for example, the Alpha™ 21364 [13], the PowerPC™ 750 [102], the Pentium II™[101]. The invention of the microprocessor enabled the use of embedded systems in low-cost consumer products like the ones listed above. The complexities of the processors mentioned above, together with memory devices containing up to 256 million devices on a single chip [1], have made it possible for chips to contain entire systems. This in turn enables system functionality well beyond traditional ASIC-based systems, for example, a whole GSM mobile phone can now be integrated on a single chip.

The complexities of today's systems mean that many problems will emerge during the design process, as mentioned in the introduction. First of all, the design of embedded systems embraces several design areas such as distributed system design for linking the system's network of communicating microprocessors and real-time design to ensure that the embedded system operates within the time constraints set by the user. These constraints may be soft, as with a laser printer, where a delay in printing a page is not critical, or they may be hard, as with an autopilot, which must without fail respond in real time. Many embedded systems include at least a few hard real-time constraints. The designer's job is to choose the engine that does the job most economically, but at the same time satisfy such constraints as physical size and power consumption. However, it is performance

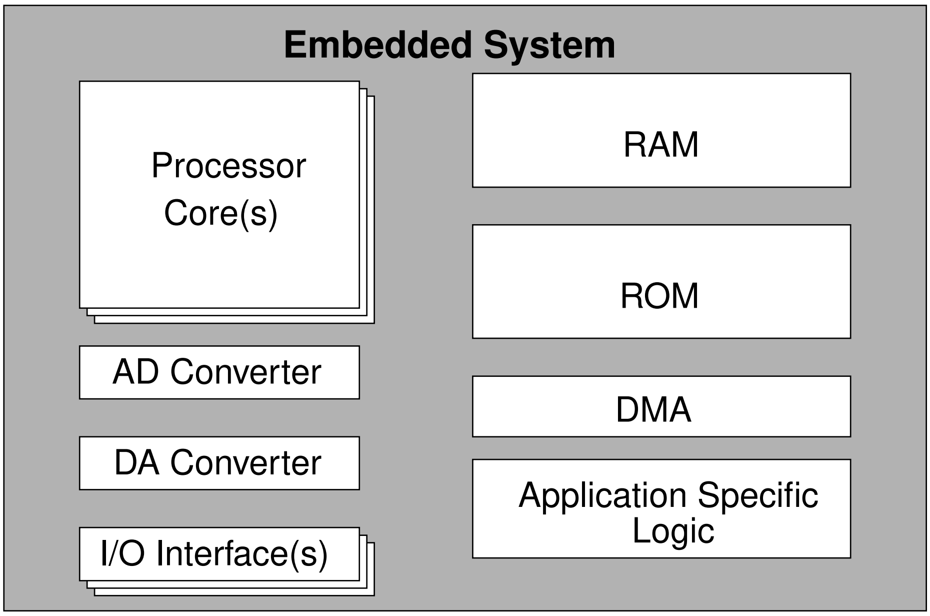


Figure 2.2: Typical embedded system.

constraints, particularly the hard real-time deadlines, that determine the basic requirements of a hardware engine.

2.1 Design challenges

To refine an embedded system from idea to finished product, a designer has to perform a huge number of design and validation tasks. A great deal of effort is being put into developing a design tool that offers computer aided design support for the whole design process, but several design sub-problems have yet to be analysed and researched before a tool for system synthesis becomes reality. Figure 2.3 shows a design flow for the design of hardware/software embedded systems. The tasks within this design flow are identified below. The different design tasks are divided into three different classes: specification, design and validation.

2.1.1 Specification

Many different aspects of a system have to be captured in a specification, that is, behaviour, architecture and requirements. Writing a specification is a very challenging task since the specification should be unambiguous, consistent and complete. Specifications have traditionally been captured, if at all, in natural language. The challenge for the future in the area of specification is to find formal methods of capturing the information needed to specify an embedded system.

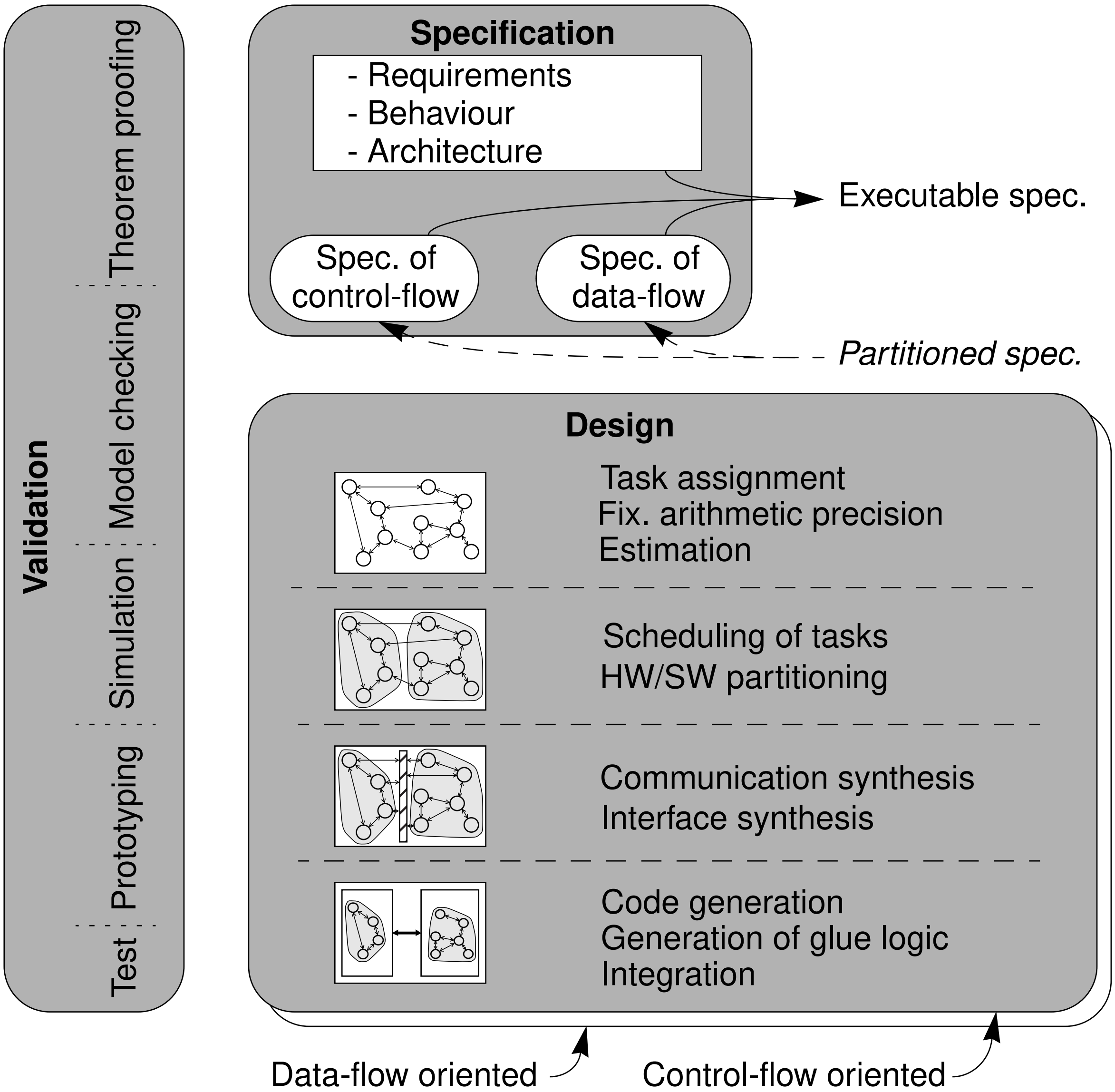


Figure 2.3: System synthesis design flow.

- *Behavioural specification*. Captures the functional parts of a system, such as algorithms and control flow.
- *Architectural specification*. Describes the architectural parts of the system, that is, describes all parts of the system that are reused (for example, processors and IP Components).
- *Requirements specification*. Describes non-functional system requirements, for example, throughput constraints, power consumption constraints and descriptions of external interfaces.
- *Executable specification*. Captures the behaviour of the system in a format that can be executed on a computer, thus enabling validation and elaboration of system properties.

- *Specification partitioning*. Partitions a specification into, for example, control- and data-flow dominated parts, that is, individual specifications of parts of the system that need to be handled in different design flows.

2.1.2 Design

To transform a specification into an implementation, a designer has to go through several refinement steps. The design task can be divided into four stages: (1) applying the specification to a specific implementation and estimating design properties, (2) assigning the specification to execution units and scheduling it, (3) synthesizing the interconnections, and (4) code generation and integration.

- *Task assignment.* Break down the specification into a set of tasks that perform the system functionality to enable concurrency and optimization.
- *Fix arithmetic precision*. For fixed-point implementation, the arithmetic width has to be defined and verified so that the implementation meets the initial specification.
- *Estimation*. Estimate cost parameters for both hardware and software. Cost parameters can be, for example, timing performance, gate count (HW), code size (SW), power, memory size, and statistics on execution frequency. Estimated values are used to aid designers/tools to make design decisions, thus decreasing the number of design iterations.
- *Allocation*. Allocate a set of processors and execution units that will deliver enough computation power to meet all cost, performance and other constraints.
- *Hardware/software partitioning*. Partition a system into multiple processors and other execution units. Partitioning decisions should consider timing, power and other cost constraints.
- *Scheduling*. Find a system schedule and allocation, taking into consideration timing constraints and the need to minimize system costs.
- *Communication synthesis*. Define the communication protocol from the system specification, select a suitable communication architecture that fulfils a set of communication constraints, and map the communication protocol onto the selected architecture.
- *Code generation*. Generate code that is optimized for each specific execution unit.
- *Integration*. Integrate hardware, software, memory, etc. into a complete system.

Most of the design tasks will affect each other's results. For example, the allocation will affect the partitioning, scheduling and communication synthesis, but at the same time the communication synthesis can affect all the others. The quality of all design decisions depends on the quality of the estimation values.

2.1.3 Validation

System validation, which is carried out at the same time as the design and specification activities, includes verifying the specification, checking that models are consistent with each other and testing the final system.

- *Formal verificatio*n. Check models between abstraction levels and proof specifications.
- *Simulation*. Generate accurate simulation models. Provide simulation models for cosimulation, for example, hardware/software cosimulation and data-flow/control-flow cosimulation.
- *Prototyping*. Rapidly develop an accurate implementation for evaluation, validation and demonstration purposes.
- *Test*. Verify that the final implementation of the system meets the initial system specifications.

2.2 History of codesign research

As mentioned in the introduction, the research field of hardware/software codesign is about 10 years old. That is, the research for design methods where there is no bias towards a hardware or software implementation in the initial specification and the design processes for hardware and software have been merged.

Initially almost every group that worked on hardware/software codesign developed their own hardware/software partitioning technique. There were approaches that attempted to speed up the initial software implementation by integrating computationally intensive parts into hardware [8,45,49,69,94]. Others attempted to partition a set of parallel processes into hardware and software [14,58,113]. Most of these approaches had evolved from high-level synthesis.

As the research field has matured a wider range of research topics has been dealing with, for example, specification [15,71,122], architectural modelling [47,123], partitioning [48,63,66], scheduling [32,108,110], estimation [52,56,119], communication synthesis [36,97,115], validation [41,114], code generation [16,77,79] and prototyping [25,60,109]. The most mature topic is hardware/software cosimulation, which has earned a wide recognition in the designer community with tools like Seamless and the Eagle [41]. Another area that has resulted in a few start-up companies is communication synthesis, that is, CoWare [21], Arexsys [105] and Felix [104]. Even though these tools have more features, it is the cosimulation and communication synthesis techniques that are most important.

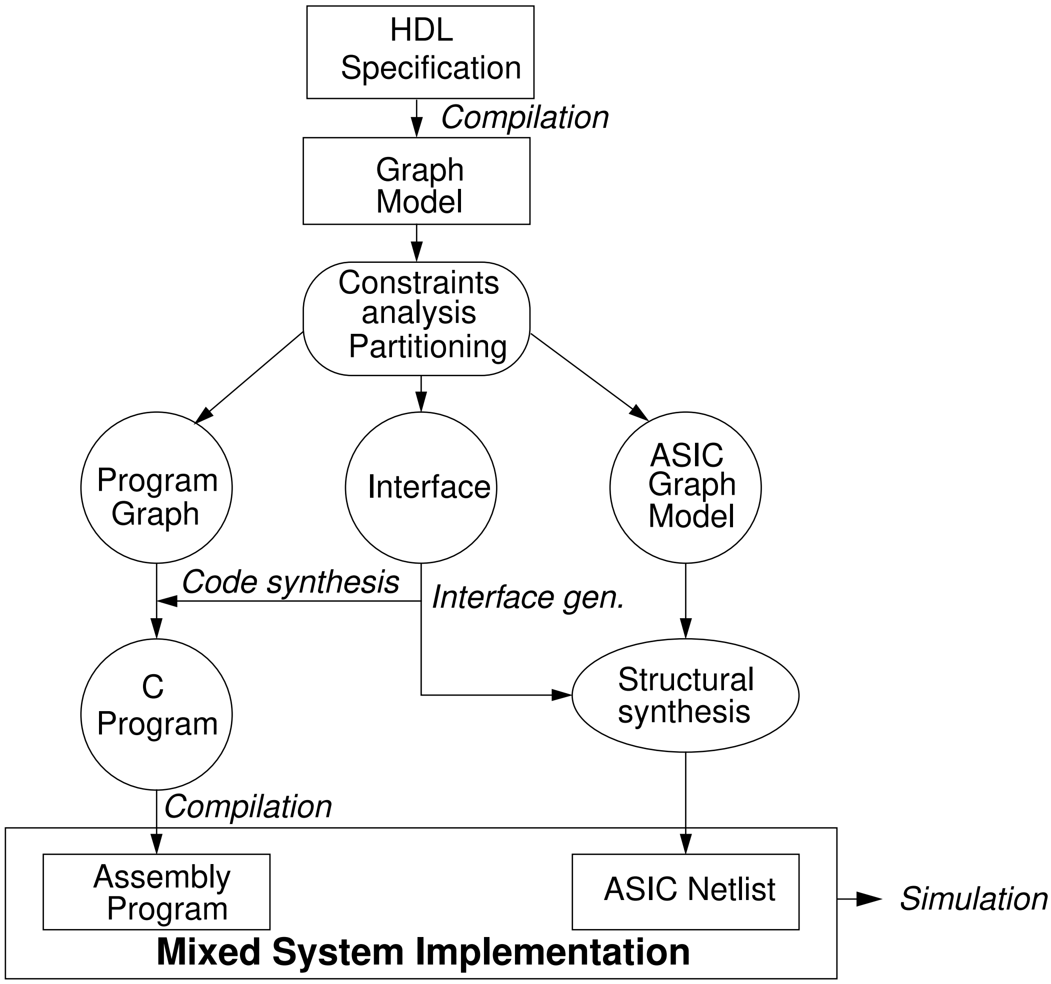


Figure 2.4: Vulcan design flow.

To illustrate the initial research efforts, the following two subsections present two different research projects. Each represents one class of approaches: COSYMA [49] (speed-up of software) and Vulcan [58] (evolved from HLS research). The remaining parts of the section describe the *leading approaches* in hardware/software codesign research.

2.2.1 Vulcan

VULCAN takes a HardwareC specification [83] (hardware description language that is a subset of C with processes) as input. The description is compiled to a data flow graph model, FGM, that allows multiple threads. Starting with the whole system in hardware, functionality is successively moved to software. Global and local timing constraints can be expressed in the specification and a key feature of the partitioning is that it carefully analyses these timing constraints to meet them while minimizing design costs.

Interprocess communication is modelled with send/receive primitives. The tool accepts blocking, non-blocking and buffered communication. All communication between processes is performed through static channels. The tool generates a buffered structure for synchronization and scheduling between processes.

The first assumption made is that non-deterministic operations related to data-dependent loop operations define the beginning of program threads in software, while all other operations are implemented in hardware. If the initial assumption is feasible, iterative improvement is achieved by migrating deterministic operations between the partitions. Operations for migration are selected so that the move lowers the communication costs while maintaining satisfactory timing constraints. In addition, communication feasibility is checked by verifying the process communication for each thread and ensuring that processor and bus use constraints are satisfied [58]. The output from the system is C code for software and FGM for hardware (see the design flow in figure 2.4).

2.2.2 COSYMA

The front-end language to the COSYMA system is a C^x (C extended with processes) description that is compiled into an internal graph-based representation, extended syntax graph (ESG) [19]. The graph is scheduled [18] into a single serialized process, and a system speed-up constraint is calculated to meet the timing constraints. The objective of the partitioner in this system is to move basic blocks from software to hardware until the system speed-up constraint is fulfilled [62]. The partitioning process is divided into two steps: firstly, an estimation-based partition that uses simulated annealing (inner loop) and, secondly, an evaluation of the partition feeds back the real values to the inner loop (outer loop). The simulated annealing in the inner loop starts with an all-software solution and extracts hardware iteratively until all timing constrains can be met. The objects extracted for hardware are basic blocks. To calculate the speed-up for each block the difference between the software execution time and the sum of the hardware execution time and the hardware/software communication time is measured. Software execution time is captured using execution profiling refined through static analysis. For the hardware execution time, each basic block is scheduled to a fixed data path. A hardware cost is generated from the schedule. The selected partition from the inner loop is evaluated in a cosimulation environment. If the constraints are still fulfilled the partition is accepted, otherwise the real values from synthesis are fed back to the inner loop. The output of the system is C code for software and an RTL VHDL for the hardware (see figure 2.5 for the design flow).

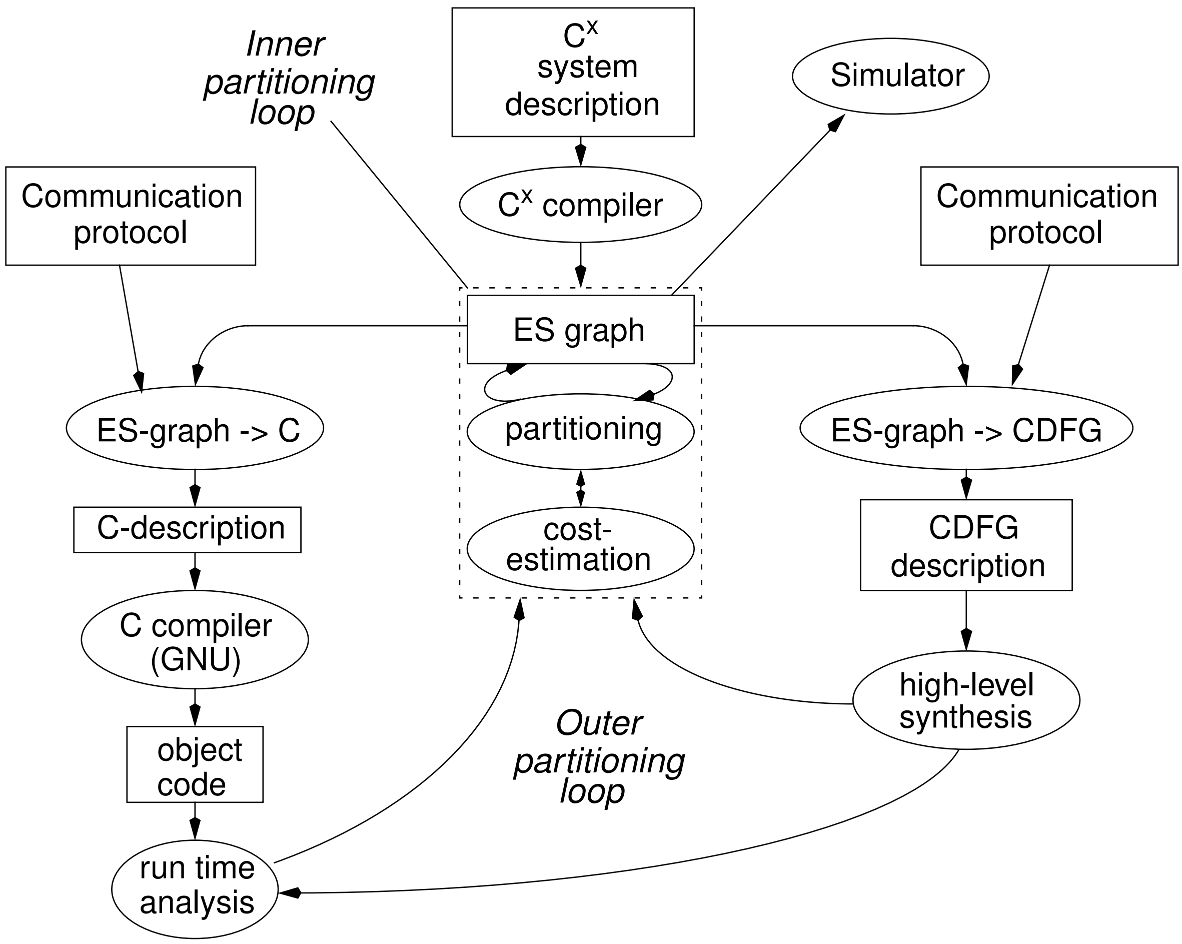


Figure 2.5: COSYMA design flow.

Communication in the C^x specification is modelled as logical static channels and bundles of channels. These logical channels can be mapped in the specification onto some predefined communication device/resource. The channel service functions are blocking and non-blocking send/receive, taken from a communication library [50].

2.3 Advanced research

2.3.1 COSMOS

Jerraya et al. use SOLAR [74] as an intermediate language. It supports system level modelling and synthesis. SOLAR is based on a finite state model extended with concepts of hierarchy and parallelism and can be generated from C, VHDL and SDL. The design specification is partitioned by an interactive toolbox called Partif [67]. Partitioning starts with a set of hierarchical and communicating processes that are characterized by a cost function and performance. The partitioning tool allows the user to apply system level transformations to the design: *move*, *merge*, *split*, *cut* and *map*. These transformations are applied manually by the designer. The designer chooses which transformation should be applied based on information about interconnection, variable sharing, states, operators in the data path and local variables. The result from the partitioning tool is a set of design units connected by abstract channels.

The communication semantics are based on the concept of rendezvous channel communication via send/receive operations. The types of protocols supported include blocking and non-blocking communication. A channel is implemented by allocating communication units from a library [36]. These library units are composed to fulfil a set of constraints put on the channel. The library consists of a set of communication services and protocols together with their implementations (a mixture of hardware and software). Interface synthesis techniques are used to permit communication between the processor and the chosen communication units. The research efforts in the COSMOS system have led to a commercial set of tools called Arexsys [105].

2.3.2 Polis

Polis [10] is focused on control-dominated applications with system architectures composed of a single processor surrounded by custom or library hardware. Polis uses Codesign Finite State Machines (CFSM) [28] as the internal representation for a system description, separating communication, behaviour and timing of the system. The CFSM is a finite state machine extended with a data path. The communication model is globally asynchronous, locally synchronous, with non-blocking finite buffers between CFSMs. So far, translation from Esterel [59] to CFSM has been reported. The description is manually partitioned with guidance of estimation values. C code and HDL code is generated from the CFSMs mapped to software and hardware respectively.

Except for the I/O drivers and code generated from the CFSMs, the software code consists of a generated application-specific operating system for the selected processor. All communication within software or between software and hardware occurs through shared memory, I/O ports or memory-mapped I/O. The synthesized hardware includes the address decoders, multiplexers, latches and glue logic. Special-purpose hardware must follow a simple, data/strobe-based protocol to make it suitable for interfacing with other CFSMs. The Polis project has led to a commercial set of tools called Felix [104].

2.3.3 CoWare

CoWare [21] is a design environment for heterogeneous hardware/software systems on a chip. CoWare's main focus is system integration and the handling of communication in embedded systems. CoWare accepts heterogeneous communicating processes, which are specified in a super set of C. This description is then manually refined down to an RTL description in register transfer C (RTC) or in VHDL/DFL.

The heterogeneous specification is mapped onto different processors (DSP, microcontroller or hardware). Software is generated with either an application-specific real-time kernel or generated by using software synthesis techniques [116]. Interprocess communication is done with point-to-point communication channels. The communication semantics are based on the concept of rendezvous channel communication via send/receive operations. Hardware/software communication channels are mapped onto a fixed architecture. This architecture is based on several library models. For software, the communication procedures are captured as parametrized C functions that are mapped onto a software model, that is, they adapt to processor-specific I/O handling, interrupt handling, etc. For hardware, a hardware interface cell is generated to connect via a handshake protocol to an I/O control unit [80,115]. This I/O control unit is a link between the processor and the handshake protocol.

2.3.4 Chinook

The Chinook [30,31] approach extends the commonly used concept of device driver to include the bus interface as well. The driver/interface is described in a timing diagram description (SEQ), which captures both the behaviour and timing constraints for the interface. The description is synthesized into low-level software code that accesses the device via the ports of a microcontroller (for example, Intel 87C51 [2]) together with the required glue-logic. The synthesis procedure tries to find the cheapest implementation (smallest amount of hardware) regarding both timing and resource constraints.

In their later work, Ortega et al. [29,97] expand the approach to communication synthesis for a distributed system. Input to the system is a behavioural description consisting of communicating processes. The processes communicate with each other via messages sent through ports connected by channels. All communication is of broadcast type, that is, one to many unidirectional. For each output port, the designer selects a high-level abstract protocol, that is, blocking or non-blocking, along with a deadline constraint. For each input port, the designer selects the appropriate queuing semantics. The designer also defines a system architecture that consists of heterogeneous processors connected by standard bus protocols (for example, CAN [121] and I^2C [65]). Processes and communication channels are then mapped onto the selected architecture. The system then generates the communication interfaces. If there is no direct connection between two communicating processes, intermediate hop processes are automatically inserted to route a message from one bus to another bus. For each processor, a customized real-time operating system is generated that includes a real-time scheduler, a message routing process and device drivers.

2.3.5 SpecSyn

SpecCharts [55] is used for capturing the system's behaviour in SpecSyn. It is based on hierarchical program state machine (PSM). The PSM description allows VHDL code within the states. Communication in SpecCharts is modelled with shared variables. In later publications the group presents a specification language, SpecC [122], that has promising characteristics for IP-based design, that is, better support for interfacing of components, hierarchy and communication. So far, only specification methods using SpecC have been reported.

In SpecSyn the SpecChart specification is compiled into an internal representation. Based on an estimation of cost and performance [56] several design refinement methods are applied to the system representation, that is, the allocation of processing elements and partitioning [113]. SpecSyn generates buses for communication between two processes using a technique called interface refinement. By analysing the size of the data communicated along with the rate of data generation, the bandwidth of the generated bus (number of wires and rate of transfer) is determined and the communication channels merged onto the bus. An abstract protocol is then directly implemented using additional control wires. Arbiters are synthesized to control access to the bus. The software send and receive subroutines are modified to use the synthesized bus.

2.3.6 Tosca

Functionality in TOSCA [11] is captured in an OCCAM [75] description. OCCAM is built on communicating sequential processes (CSP). High-level languages like C (software) and VHDL (hardware) can be mapped/connected to the OCCAM description. Only processes that are written in OCCAM can be used in the exploration of the design space. Processes connected to the OCCAM description, for example, C, VHDL, are mapped to software and hardware respectively. The partitioning process starts with expanding the description to parallel processes at the top level. Processes suitable for hardware or software are preallocated to the best implementation. The last step consists of pair-wise collapsing processes to new processes, considering user-defined closeness criteria. This is done until a user-specified cost constraint is fulfilled [12].

For software synthesis, a virtual machine code (VIS [7]) is generated from the OCCAM code. The implementation assembler code for the selected processor is then generated by translation from the virtual machine code. To verify and evaluate the design quality the TOSCA environment also provides a cosimulation facility, where the virtual machine code is executed in a VHDL model. The virtual processor accesses the system bus through a bus interface for the selected processor. The hardware/software interfaces are implemented with library functions on the software side and a fixed interface structure on the hardware side.

2.3.7 Miscellaneous approaches

The seven approaches presented in detail in this section represent the most recognized and successful research groups. However, they still represent only a fraction of the work being done in hardware/software codesign. Some other research approaches are described in brief below. The author refers readers interested in hardware/software codesign to [46,55,82,118].

Lycos [81] is a system for speeding up C programs by implementing computationally intensive parts in hardware. The main application area has been image processing. Camposano et al. [23] present a similar approach called CASTLE. One system that has focused on data-flow-dominated systems is Cool [86]. Cool uses a subset of VHDL for specification. The main objective of Cool is heterogeneous implementation and its creators present several partitioning algorithms. Eles et al. [48] also present several different hardware/software partitioning algorithms.

AKKA accepts system specification in C++ and partitions it into hardware and software components and synthesizes interfaces between them. The hardware component is coded in behavioural VHDL. Principal features of AKKA [72] are as follows: performance analysis, hardware/software partitioning [94] and cosimulation and coemulation environment [70,111].

The CodeSign [47] project presents an approach that starts with a heterogeneous specification. The tool supports several design refinements. The project also proposes methods for object-oriented modelling of the architectural parts. The architectural model is used during interface synthesis.

Programming language Java is a hot topic in the information technology industry, which has led to several proposals for using Java for hardware/software codesign. Fleischmann et al. [51] use Java for specifying dynamically reconfigurable embedded systems. Young et al. [120] present a method for using JavaTime (extended Java) for specifications and their successive refinement.

2.3.8 Summary

Even though a considerable amount of work has been done in the field of hardware/software codesign there are still several problems to be solved. One can observe that from the initial approaches, in which research was often aimed at automatically transforming a specification into an implementation, researchers now tend to focus on more specific problems. This is also the case in this thesis, which concentrates on the design of hardware/software interfaces or, more specifically, the specification, synthesis and design of the device-driver part of hardware/software interfaces.

On studying the work done in the area of hardware/software with respect to interface design, one can observe that tools like CoWare [21], Polis [10] and Cool [86] are primarily designed for situations in which the whole design functionality is captured in its environment, with communication then being refined during system synthesis, that is, the device drivers are generated together with the custom hardware and an application-specific operating system. If designers use IP blocks and off-the-shelf real-time operating systems (RTOS), they are faced by the same problems that occur in manual design as described by Tuggle in [112]. COSYMA [49] and Lycos [81] use libraries for the hardware/software interfaces, which only moves the problems to the provider of the library. CodeSign [47] and MakeApp [57] are two tools that support the interfacing of IP components. CodeSign generates the interface code and an application-specific real-time kernel for the selected processor. It even supports some DMA access from data-flow specifications. MakeApp is a tool for generating device drivers for different devices and processors matching user-defined configurations.

The work presented in this thesis, ComSyn, deals with hardware/software interface synthesis that ranges from the specification to the implementation and validation of hardware/software interface protocols. ComSyn takes an architecture- and implementation-independent description of the hardware/software interfaces as well as libraries capturing the architecture. From these inputs a synthesis tool generates (a) device driver functions, (b) a combination of device driver functions and a DMA controller, or (c) simulation models, depending on what the designer decides. The clean separation of information facilitates (1) efficient design space exploration with combinations of different processors, operating systems and protocols, and (2) efficient maintenance of a large number of different versions and variants of hardware/software interfaces. This is not achieved by any of the other approaches.

A careful examination of the problem and a review of the work done in hardware/software interfacing is presented in the next section, which also contains a thorough presentation of ComSyn, the tool for handling of hardware/software interface design developed by the author.

3 HW/SW interface design

In [89] we discussed the specification and implementation of communication in an embedded system. There we identified several different communication routes in a hardware/software embedded system, which includes both hardware and software library components (see figure 3.1). A communication specification can be mapped onto a large set of heterogeneous implementations (see table 3.1). If the specification can be mapped onto a distributed system the implementation set will be even larger and the communication synthesis task even more complicated.

In [89] we also analysed the research dealing with communication in embedded systems. This analysis led us to focus on the hardware/software interfacing of IP components, since this was not well covered in the research work we found.

3.1 Definition

A hardware/software interface can consist of up to four parts: (1) device driver functions, (2) direct memory access (DMA) controller, (3) bus interface, and (4) the register file and handshake behaviour of the device. A device driver is the wrapper for a hardware device

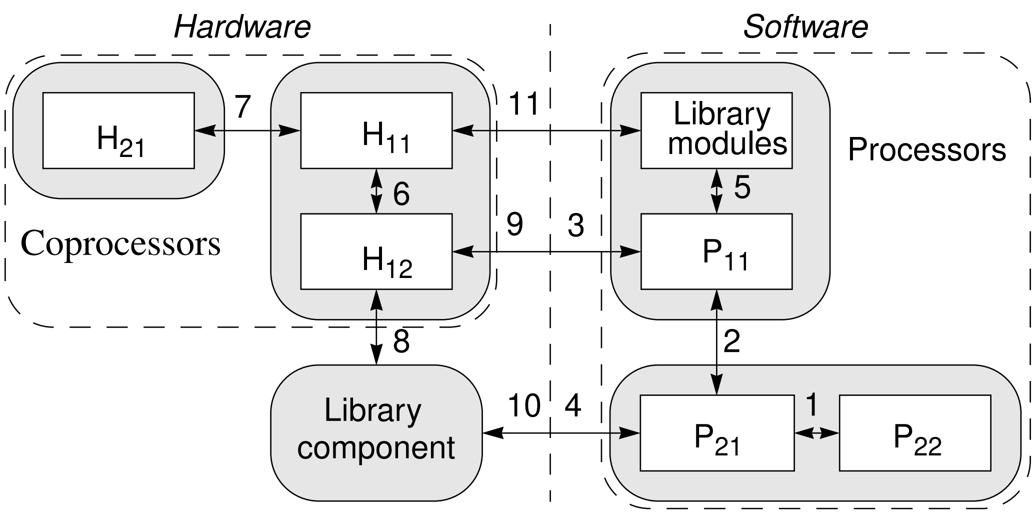


Figure 3.1: Communication routes in a heterogeneous embedded system.

Table 3.1: Communication routes in an embedded system.

#	Description
1.	SW process connected to another SW process on the same processor.
2.	SW process connected to another SW process on a different processor, with the same or a different operating system.
3.	SW process connected to an HW process (coprocessor).
4.	SW process connected to a peripheral, for example, off-the-shelf component, library module, etc.
5.	SW process connected to a library SW module.
6.	HW process connected to another HW process on the same chip (partition).
7.	HW process connected to another HW process on another chip (different partitions).
8.	HW process connected to a peripheral, for example, off-the-shelf component, library module, UART, etc.
9.	HW process connected to an SW process.
10.	Peripheral connected to an SW process.
11.	HW coprocessor connected to a library SW module.

accessed from software. The device driver's behaviour is essentially that of a protocol defining how the device is accessed and synchronized with software. A device driver can perform all accesses to the device directly or it can use DMA to transfer data between two places in the address space. In the last case the device driver controls the DMA controller, which in turn accesses memory and other devices.

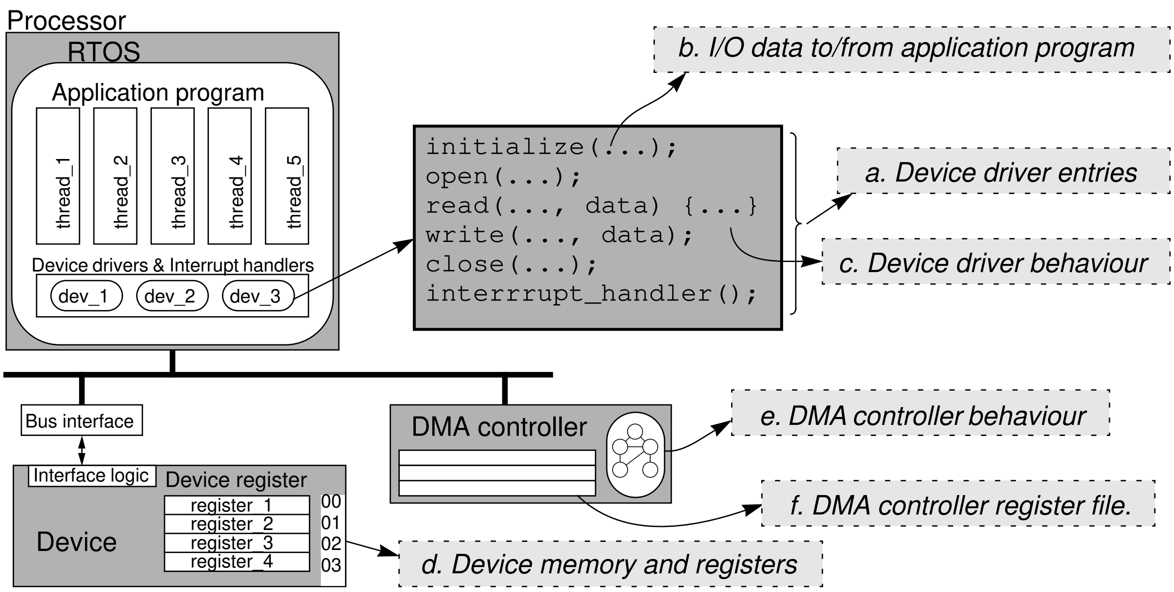


Figure 3.2: Break down of the HW/SW communication into different parts.

Figure 3.2 illustrates the dependence of a device driver on various other parts, both hardware and software. We can observe that a device driver contains information on: (a) processor architecture and behaviour, (b) real-time kernel behaviour, (c) device architecture and behaviour, (d) access protocol of the device, (e) application programmer interface (API) rules, and (f) DMA controller architecture and behaviour. Device drivers thus accommodate a very high information density and are dependent on many parts that can appear in a large number of combinations. This fact is the reason for the low productivity of device driver modelling, which is four times lower than for ordinary software code [20].

3.2 Motivation

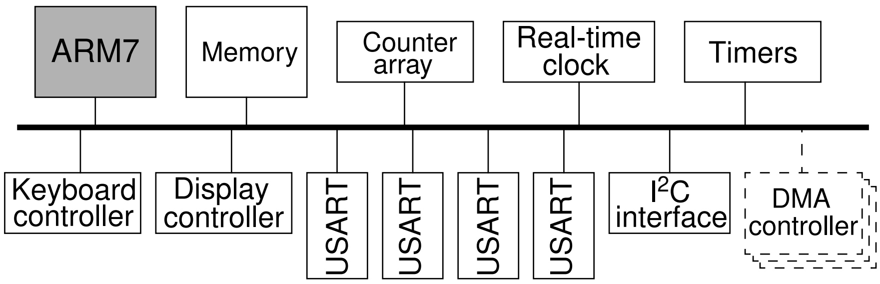


Figure 3.3: Simplified schematic of the pay phone controller system.

Días et al. [40] present a good example of design using IP components, in which more than 60% of the design consists of IP components. It illustrates how time-consuming the task of interfacing IP components can be, that is, design of the hardware/software communication parts of the system. Figure 3.3 shows a simplified schematic of the system, which is a pay phone controller. Consider the following design scenario: A system designer needs to evaluate the impact DMA controllers have on the system performance by comparing two alternative architectures of interfaces with the USARTs in figure 3.3: (1) no DMA controller, and (2) DMA controllers to interface the USARTs. The assessment is difficult because of the many dependencies, and a simple calculation is not accurate enough. This is supported by the observation that in many practical situations the traffic on the bus is identified as a performance bottleneck only after the system has been built. Ideally the two alternative architectures should be compared using simulation with all the interface details included in the simulation models. However, without tool support this is a formidable task because the designer must develop an entirely different interface code for each configuration and component. This is necessary because of the huge difference between implementing a hardware/software communication protocol in pure software and implementing it with a DMA controller. This task is so time-consuming that system designers typically consider a very limited number of design alternatives, and frequently only one. On the other hand a tool is perfectly capable of generating efficient models for this purpose from a high-level specification of the communication protocols, as this thesis shows.

In addition to the evaluation of different architectures, there is the task of configuring a system for different product versions and product generations. For example, in a second version of the product it is decided to use a different processor and real-time kernel, but the functionality is the same. If the interface code was written with reuse in mind, that is, macros for real-time kernel functions and processor instructions are used in the code, most of the code can be reused. Only the APIs of the code have to be adapted to fit the new architecture. However, a new processor will require another DMA controller. The interface code dependent on the DMA controller therefore has to be redesigned.

Now consider the evaluation and design architecture upgrade tasks using a tool for hardware/software communication. The designer maps a protocol description of the interfaces onto an architecture, that is, real-time kernel, processor and DMA controller (see figure 3.4). The communication synthesis procedure transforms the protocol descriptions into

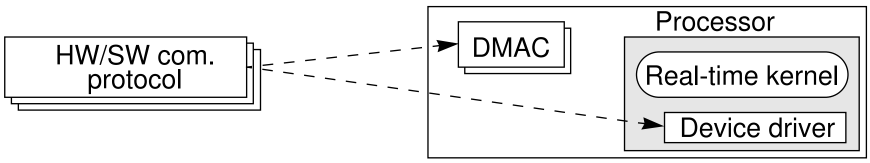


Figure 3.4: Mapping of hardware/software communication protocols onto selected architecture.

architecture-specific device driver code and application-specific DMA controllers. This enables IP providers to supply a wrapper for IP components. A designer that uses these IP components therefore only has to use the tool to generate the interface code for the specific architecture.

Figure 3.5 illustrates the differences in design flow between manual and computer aided hardware/software interface design.

3.3 HW/SW Interface synthesis

Many research groups have recently focused on the problem of communication synthesis for embedded real-time systems [98]. However, none of them has fully dealt with the problem described above. Tools like CoWare [21], Polis [10] and Cool [86] are primarily designed for cases in which the whole design functionality is captured within the tool's

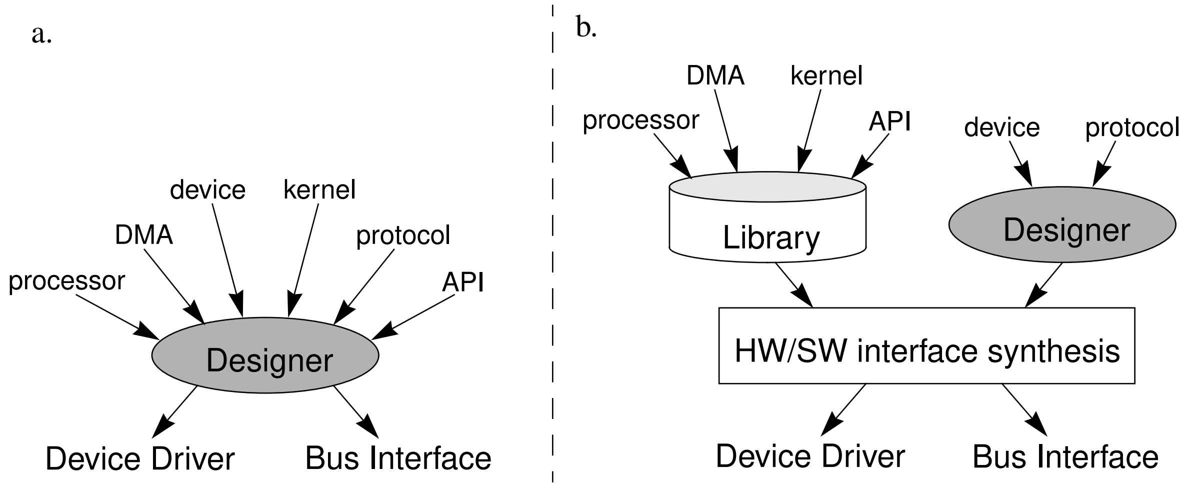


Figure 3.5: HW/SW interface design flows. (a) Without HW/SW interface synthesis tool and (b) with a HW/SW interface synthesis tool.

environment and communication refined during system synthesis, that is, the device drivers are generated together with the custom hardware and the operating system. However, if users want to use IP blocks and off-the-shelf real-time operating systems (RTOS), they will face the same problems that occur in manual design [112].

As described above, a hardware/software interface consists of several parts that have to be generated during hardware/software interface synthesis. When generating hardware/software interfaces between software and hardware IP components, however, the interface synthesis task is reduced to the generation of device drivers (including DMA controllers) and bus interfaces.

3.3.1 Device driver synthesis

Most codesign approaches support the generation of device drivers when a system description is synthesized down to an implementation. The interfacing of hardware library components is only supported in a few research projects [47,57,91]. Most approaches to hardware/software interface synthesis use a parametrized library for the device driver generation. This is the case for CoWare [21], where communication channels are mapped onto library procedures (device drivers). These device drivers are captured as parametrized C functions that are mapped onto a software model, that is, they adapt to processor-specific I/O handling and interrupt handling. A library component is therefore interfaced by writing parametrizable C functions. This is also the case for the Polis system [10].

The Chinook [30,31] approach extends the commonly used concept of device driver to include the bus interface. The driver/interface is described in a timing diagram description (SEQ), which captures both the behaviour and timing constraints for the interface. The description is synthesized into low-level software code that accesses the device via the ports of a microcontroller (for example, Intel 87C51 [2]) together with the required glue-logic. The synthesis procedure tries to find the cheapest implementation (smallest amount of hardware) regarding both timing and resource constraints. In their more recent work, in which communication routes are synthesized onto a distributed architecture [97] with standard bus protocols, the device drivers are captured as library elements.

In COSMOS [66] a communication channel is implemented by allocating communication units from a library [36]. These library units are composed to fulfil a set of constraints put on the channel. The library consists of a set of communication services and protocols together with their implementations (a mixture of hardware and software).

MakeApp [57] is a tool for generating device drivers for different devices and processors matching user-defined configurations. MakeApp solves only part of the problems described in [112] since it does not support operating systems. This means that for example semaphores sometimes needs to be inserted when an operating system is used.

Eisenring et al. [47] present a method for modelling the target architecture by means of object-oriented methods. The architectural model is used to generate hardware/software interfaces from data-flow descriptions. This is the only approach, besides ours, that handles the mapping of hardware/software communication onto DMA, but their approach is limited to a set of standard APIs and cannot handle more complex protocols, as in our approach [87,91].

The approach to device driver synthesis presented in this thesis is unique regarding the clean separation of behaviour and architecture [91,92], the efficient techniques for modelling, and the full support of direct memory access (DMA) [87,91]. The separation of behaviour and architecture enables the efficient reuse of the protocol descriptions since they can be mapped onto any combination of processors and real-time kernel. Protocols can also be mapped to an implementation using DMA. The information and size of the interface descriptions are much smaller than other modelling techniques [90,91,92,95].

3.3.2 Bus interfaces synthesis

Bus-level protocols for components are usually described graphically by means of timing diagrams in a data sheet. For this reason [31] and [85] have developed specification languages based on timing diagrams. One problem with these approaches is that it is not possible to specify high-level behaviour. Lin et al. model the interface with an extended STG (signal transition graph) [80] based on Petrinets. They can use this modelling technique to model asynchronous events, and an extended-STG model to perform asynchronous synthesis. This approach enables the modelling of more complex behaviours. [44] and [55] use hardware description languages to model bus interfaces. Gajski et al. use a program state machine (PSM) to specify interfaces, but in all examples they use the HDL capability of the PSM to model interfaces. Öberg et al. present a communication protocol synthesis tool that specifies the design in a special-purpose language, ProGram [124], which is based on context-free grammar. The synthesizable subset is limited to a regular grammar with attributed conditions.

Table 3.2 shows a comparison between some of the approaches described above and the extended version of ProGram presented in this thesis. The table shows that interfaces presented by other research groups are modelled more efficiently in ProGram. The example for SEQ in row 4 is an asynchronous circuit, which is the primary target for SEQ but not for ProGram. The ProGram model is still comparable with the SEQ model, however, and better suited to synchronous interfaces than SEQ.

Table 3.2: Comparison of Lines of Code (LoC).

Reference model	LoC	ProGram	Ratio
HDL (VHDL), list 1 in [44]	20	5	4.0
Extended TD, fig. 1 in [85]	16[a]	7	N.a.
PSM, fig. 8.10 in [55]	21	12	1.8
Timing diagram (SEQ), [32]	6	6	1.0
Petrinets (E-STG), fig. 5 in [80]	11	6	1.8

a. Number of conditions specified graphically.

3.4 Validation

Validation is as important as design, since a design does not have any value until the designer knows that it fulfils the desired behaviour. In the traditional method of validating hardware and software in embedded system design, software is typically developed after a hardware design has begun to stabilize and prototypes are available for the integration of the hardware and software. This method provides limited visibility of both hardware and software operations. Software can also be prototyped in a host environment in which hardware/software communication is replaced by device drivers that emulate the hardware behaviour [84]. The problem with this approach lies in ensuring that the emulation models are consistent with the implementation.

The alternative to the sequential validation of hardware/software communication is the emerging technology of hardware/software cosimulation [41]. Cosimulation means the simulation of a model of the hardware executing the software. This simultaneously provides both visibility and control of the hardware model while allowing software execution to be controlled and observed at all the levels of detail required to understand the behaviour of the system. Cosimulation leads to radical improvements in productivity because of the decreased effort required to model different levels of abstractions. This increased productivity comes at the cost of low simulation performance caused by the cost of the synchronization between the software and hardware simulation. Cosimulation can be performed at several different levels of abstraction and can be implemented in several different ways.

Host code cosimulation: Host code cosimulation (HCCS) is when software is executed on the host workstation and the hardware/software communication is linked to a hardware simulator. HCCS can be performed with or without a bus interface. With a bus interface, a simulation device driver connects the software to the hardware simulator via the bus interface and addresses the device in the same way as it does in an implementation [107]. In cases where there is no bus interface, the hardware and software, for example, C and VHDL code, communicate with a handshake protocol [114].

RTL model of processor: The simplest method for hardware/software cosimulation is to develop an RTL model in HDL of the processor on which the target code is executed. This method can be improved by implementing an instruction set simulator connected to an HDL interface [4].

Cosimulation bridge tool: A bridge tool for cosimulation is a tool that connects two or more simulators, that is, it handles the synchronization and interaction between them. In the case of hardware/software cosimulation it is connecting an instruction set simulator (ISS) with an HDL simulator or a gate level simulator [43,107]. More general bridge tools also exist, such as Ptolomy [37], which can connect a large set of heterogeneous simulators. Valderrama et al. [114] present an approach in which they generate the simulation models from an interface description. This approach is similar to ours [88,93], with the exception that we generate simulation models for commercial tools like Seamless [107] and support higher abstraction levels.

Commercial hardware/software cosimulation tools like Seamless and the Eagle tools support selective focus as a mechanism to reduce simulation time. Simulation speed-up is achieved by filtering the data passed between the processor and the hardware. These filters can, for example, store the instruction memory locally in the ISS, that is, no code data is sent over the cosimulation interface. Another way of speeding up simulation is to activate the bus only during access and disable it at other times (saves HDL simulator time).

Table 3.3: Commercial cosimulation tools and tools with cosimulation support.

Tool	Provider	Bridge tool			RTL model
		HCCS[a]	TCCS[b]	Multilingual	
Seamless	Mentor Graphics	x	x		
Eagle	Synopsys	x	x		
Virtual-CPU	Summit Design	x	x		
Virtual ICE	Yokogawa Electric				x
VMLink	Yokogawa Electric			x	
Armulator	ARM				x
Arexsys	Arexsys	x	x	x	

a. Host code HW/SW cosimulation.
b. Target code HW/SW cosimulation.

Columns 1–2 in table 3.3 respectively contain the name of the cosimulation tool and the name of the company that provides it. Columns 3–6 indicate what type of cosimulation each specific tool supports. Columns 3–5 indicate different types of bridge tools: (3) host code cosimulation, (4) target code cosimulation, and (5) multilingual simulator connections. The RTL model of processor is indicated in column 6.

The multilevel validation approach presented in section 3.5.5 and [88,93] does not propose any new validation techniques, rather an improvement of well-established techniques. Both consistency between models and design productivity are improved. Consistency is improved because all simulation models and implementation are generated from the same interface protocol description. Design productivity is improved because the interface protocol descriptions can be reused over and over again and all simulation and implementation models are derived from the same interface description. The designer therefore only needs to write one description and then generate simulation models and implementation.

3.5 ComSyn

ComSyn is the informal name of the project that is covered in this thesis. This section summarizes the objective, design flow and refinement methods presented in the appended papers [87,88,90,91,92,93,95].

3.5.3 Objective

The objective of ComSyn is to reduce the amount of effort required by the designer to reuse hardware IP components in hardware/software systems by reusing the hardware/software interface. So far the ComSyn project has only concentrated on the access protocols of hardware/software interfaces, also known as the device drivers. To enable the easy reuse of hardware components the designer must access the device through an application programmer interface (API) suitable for the specific project. The designer should not have to be concerned with the implementation details of the interface. This is achieved in ComSyn by separating the behaviour of the protocol from the architecture and capturing the architecture in two different libraries: one that captures the processor-specific parts and another that captures the parts specific to the operating system. Implementations are generated from the behavioural description and the two libraries.

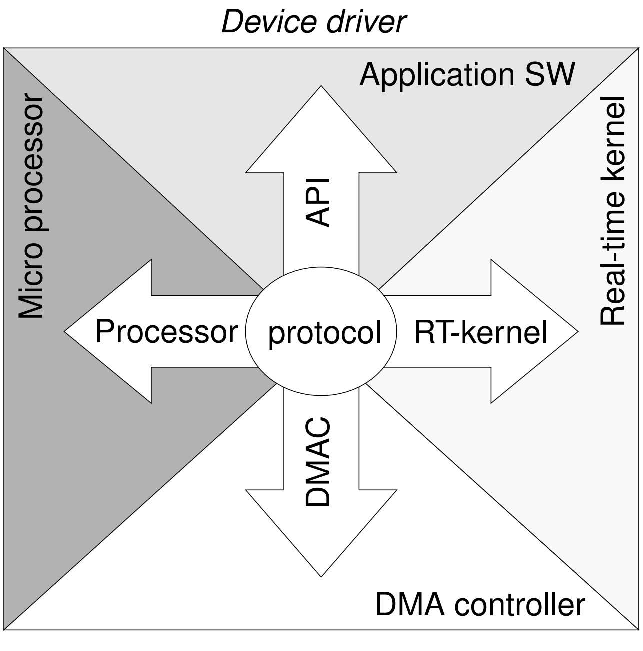


Figure 3.6: Device driver protocol mapped onto an architecture.

The information density of a device driver is very high, which is also the reason for the low reuse and lower productivity (about four times lower) in writing device drivers compared with application software [20]. In essence, a device driver is an interface protocol for synchronizing and transferring data between two parties, that is, application software and the interface device. The device driver can use a real-time kernel and a DMA controller to increase the performance of the implementation. During the implementation the device driver protocol therefore has to adapt to the application software, microprocessor, real-time kernel and a DMA controller (see figure 3.6). A device driver contains information on:

- processor: a device driver can have processor-specific assembler code in it. Address calculation for accessing device registers will also depend on the processor.
- real-time kernel: a device driver can have real-time kernel functions for synchronization, execution delay and handling of mutual exclusion.
- device: the behaviour of the device and the structure of the interface will affect the device driver.
- access protocol: the protocol determines how the comminution should be performed.
- API rules: determine how the communication between the device driver and the application software is performed.
- DMA controller (when DMA access is required): each DMA controller has its own unique architecture and behaviour, that is, the device driver has to adapt its behaviour depending on the DMA controller used.

Using ComSyn the designer only needs to model the access protocol to the device, the centre in figure 3.6, and let the tool generate the architecturally dependent parts, that is, the arrows in figure 3.6. To further illustrate the benefits of ComSyn, one can look at the example in figure 3.7. In this example we want to provide a library of device drivers for accessing a device. The library is based on two access protocols, and it should support two processors and two real-time kernels, as well as both DMA and non-DMA access and two different APIs. To write this library manually (see figure 3.7 (b)), the designer of the library has to write 16 models for each protocol (that is, a total of 32 models). This does not include the models necessary for verification. This can of course be improved if the device drivers are written for reuse with the help of macros, that is, removing the dependence on the processor and real-time kernel. If this is done only 4 models are required for each protocol, not counting the verification models (that is, a total of 8 models). Applying the same example to ComSyn one would need 2 models, each describing one access pro-

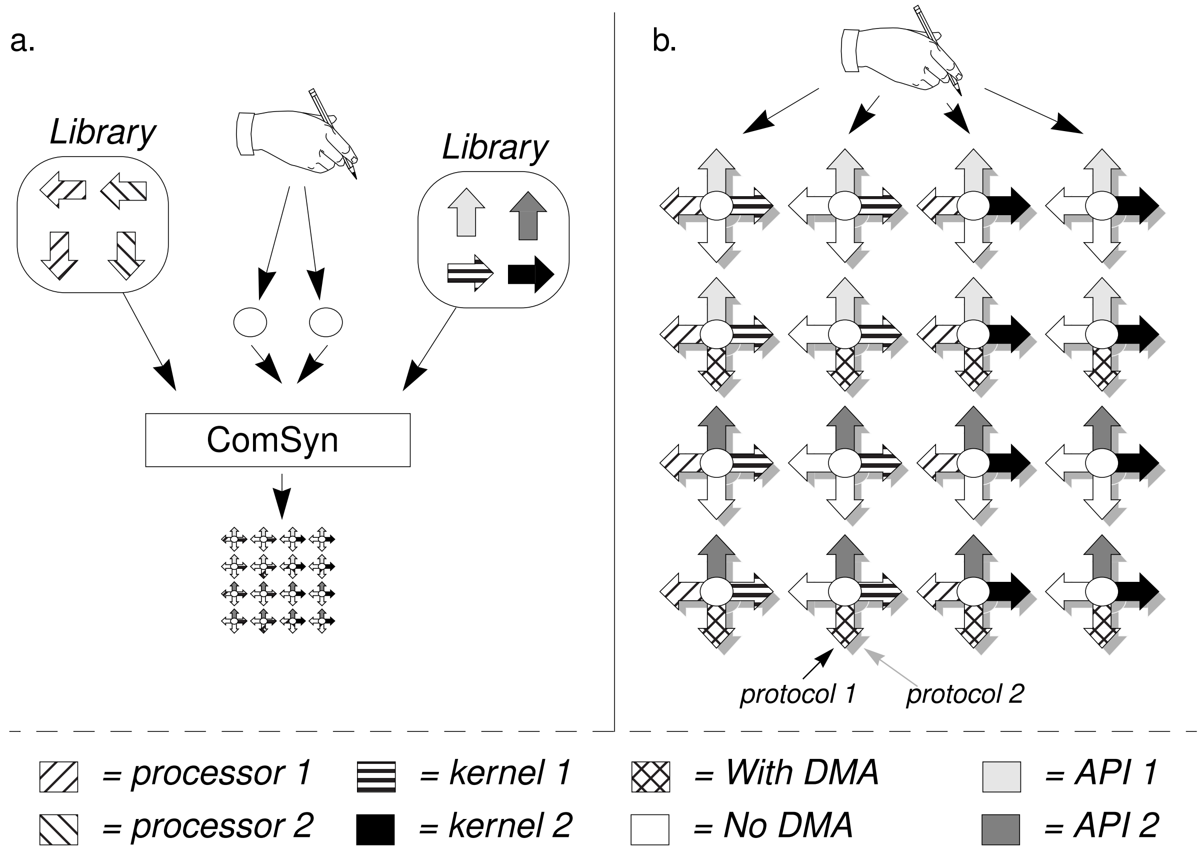


Figure 3.7: Providing device driver implementation for two device driver protocols mapped onto two RT-kernels and two processors, using two different APIs, with and without DMA. (a) describes the generation of implementation with the ComSyn approach, and (b) with a handwritten library.

tocol. From these descriptions the ComSyn tool would generate the desired implementations (see figure 3.7 (a)). This also includes the generation of verification models. One could argue that if using ComSyn the designer needs to capture the library entries for the different architectures, but this is almost the same amount of information that needs to be captured for manually written models using macros.

3.5.4 Design flow

We model device drivers independently of the architecture using ProGram [124]. ProGram is a grammar-based notation for protocol applications, which is inspired by YACC [76]. Specifications in ProGram deal with sequences of allowed events as opposed to states and state transitions in an FSM model. In contrast to parsers for compilers ProGram allows several concurrent input and output streams. The ProGram description is synthesized into a set of untimed extended state machines [95,124], which is the input to the architecture mapping procedure described in [87] and [92].

There are two types of state machines: access functions and interrupt routines. As shown in figure 3.8, the mapping procedure uses data from two libraries to generate the architecture-specific code. The first library captures the information on the operating system architecture (OSLib) and the second library captures the processor-specific characteristics (ProcLib). This organization carefully separates protocol specification from operating system and processor dependencies. Using this input the tool can generate three alternative implementations: (I) pure software implementation, (II) code for mixed hardware/ software solutions with a DMA controller, and (III) multilevel simulation models.

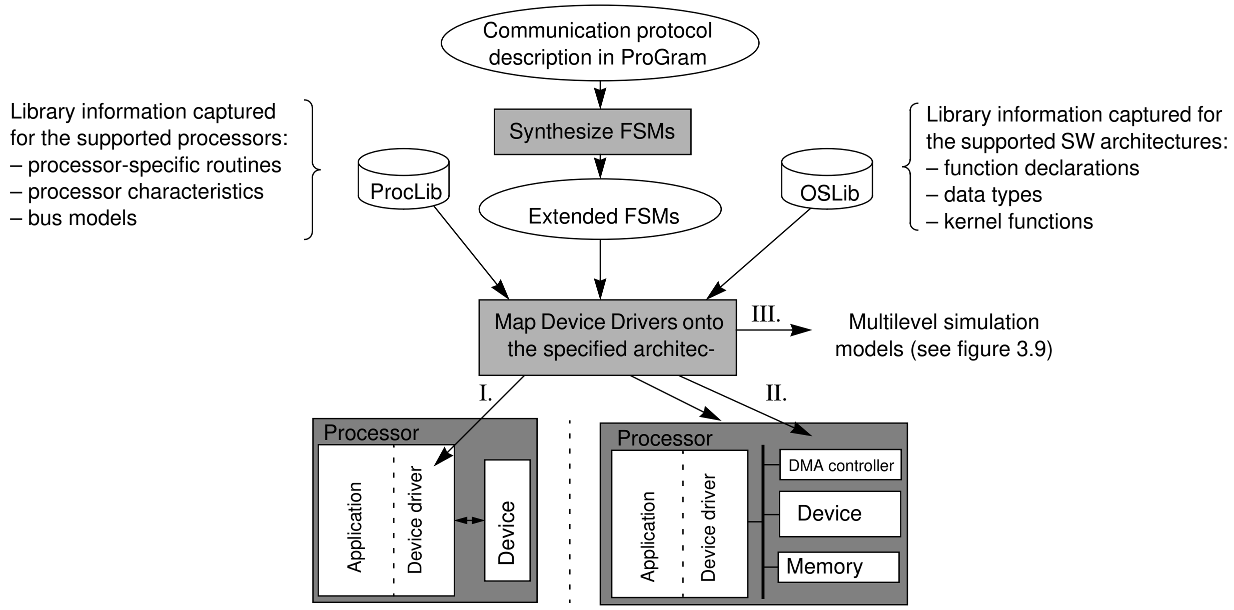


Figure 3.8: Overview of the ComSyn synthesis system.

3.5.5 Target architecture

As indicated in figure 3.8, our approach to device driver synthesis supports two alternative target architectures: (1) pure software implementation, and (2) software together with a generated DMA controller.

Software implementation: For pure software implementation, access functions and interrupt routines are implemented as software functions in C, which are mapped to fit the selected real-time kernel and processor. An interrupt handler is generated for the appropriate call of the interrupt routine.

With DMA controller: The combined DMA and software-based architecture is used when the designer has selected one of the state machines to be implemented as a DMA controller, which controls the data transfer between a hardware device and memory. This is the preferred solution if high peripheral data rates must be achieved, and performance and bus use is critical.

HW/SW simulation models: ComSyn supports the generation of three different simulation models (illustrated in figure 3.9): (b) hardware/software stub simulation models, (c) device drivers for hardware/software host code cosimulation, and (d) device drivers for hardware/software target code cosimulation. In stub simulation, hardware and software are simulated separately. For software simulation ComSyn generates a device driver that emulates the device behaviour, and for hardware simulation it generates a testbench with the device driver behaviour. For host code cosimulation, in which software is executed on the host workstation and hardware simulated in an HDL simulator, ComSyn generates a device driver that connects the software to the hardware via the HDL simulator interface instead of reading and writing registers in the same way as in the implementation. Device drivers for target code cosimulation are the same as the ones used for implementation.

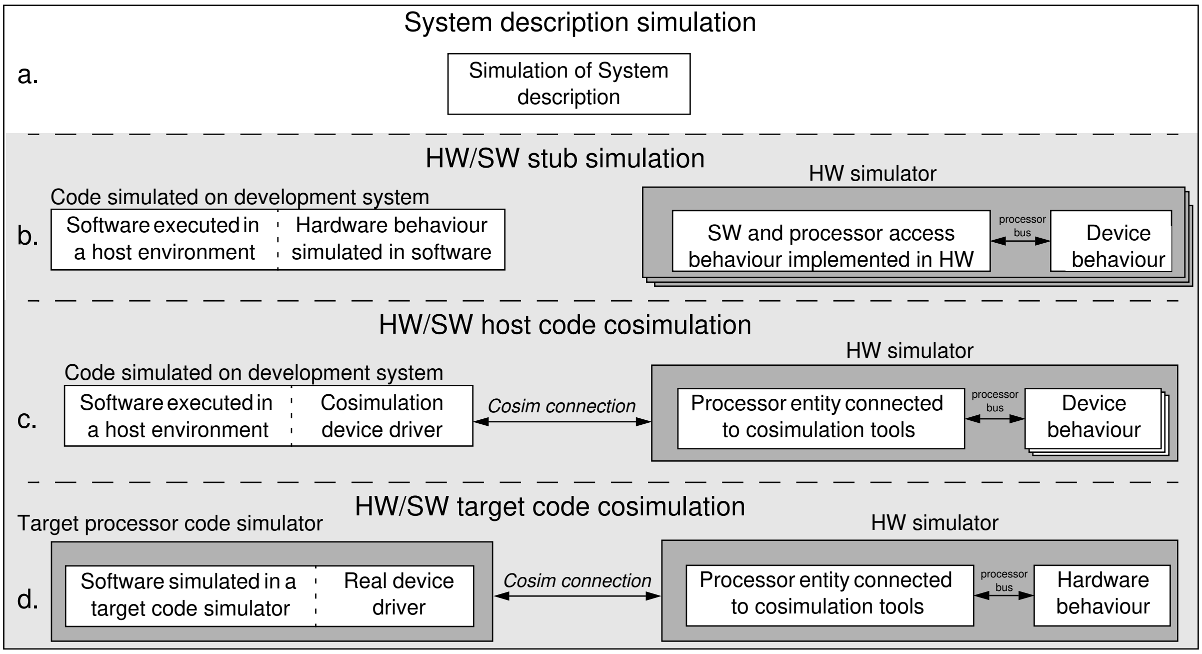


Figure 3.9: Simulation levels in embedded hardware/software systems development: (a) system description simulation, (b) HW/SW stub simulation, (c) HW/SW host code cosimulation, and (d) target code cosimulation.

3.5.6 Protocol modelling in ProGram

A ProGram model defines the possible event sequences for several parallel inputs in terms of a grammar. Actions, which are associated with grammar rules, define the reaction to input events and the event sequence on the output ports. A ProGram description consists of three parts: (1) interface declaration, (2) tokens and memories, and (3) grammar rules and actions.

Interface declaration: The interface declaration describes the interface to the application and the device. There are four types of ports to describe the device driver interface: (1) ports to the application (SW), (2) registers of the interfaced device (HW), (3) internal signals (internal), and (4) interrupt signals from the device (interrupt). In addition to the type, a port declaration contains name, direction and bit width. A port declaration of a device register also requires the register's relative address.

Tokens and memories: A Token is a pattern of bits read from an input stream or written to an output stream. It can also be viewed as a constant. Reading and writing tokens are the primary events. Memories and variables can be defined to maintain a state and to communicate between concurrent activities of the protocol implementation. The size and layout of memory fields can be defined.

Grammar rules and actions: The grammar description begins with one or more start rules. They define concurrent activities and work in practice as process declarations that define the signals used for the start condition. The synthesis process generates an access function or an interrupt routine for each start rule. Actions in the grammar specify the assignment of values to signals. Expressions to compute these values may be put directly in the assignments or may be associated with some symbols in the action value section. The assignments can then simply refer to these symbols. The expressions allow concatenation and conditionals in addition to the usual arithmetic and logic operations. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols. A grammar rule consists of a grammar symbol that serves as a rule identifier and a list of alternatives. Each alternative is a sequence of non-terminal symbols, terminals and actions. Passing the new signal stream as a parameter to the subtree of productions redirects the input stream. Actions are enclosed in curly brackets (see figure 3.10).

```
grammar_rule_1(sig):symbol_1 grammar_rule_2
    | ...
    | symbol_n {action_section;};
```

Figure 3.10: Grammar rule with actions.

4 Summary of papers

The eight papers in this thesis can be categorized in four groups: (section 4.1) analysis and problem formulation, (section 4.2) specification of interface protocols, (section 4.3) synthesis of interface protocols, and (section 4.4) validation of interface protocols. Sections 4.1 to 4.4 outline the content and contribution of each paper. The content of each paper is identified in the design flow of our device driver synthesis tool, ComSyn, depicted in figure 4.1. Section 4.5 presents an outline of each author's contribution to the papers.

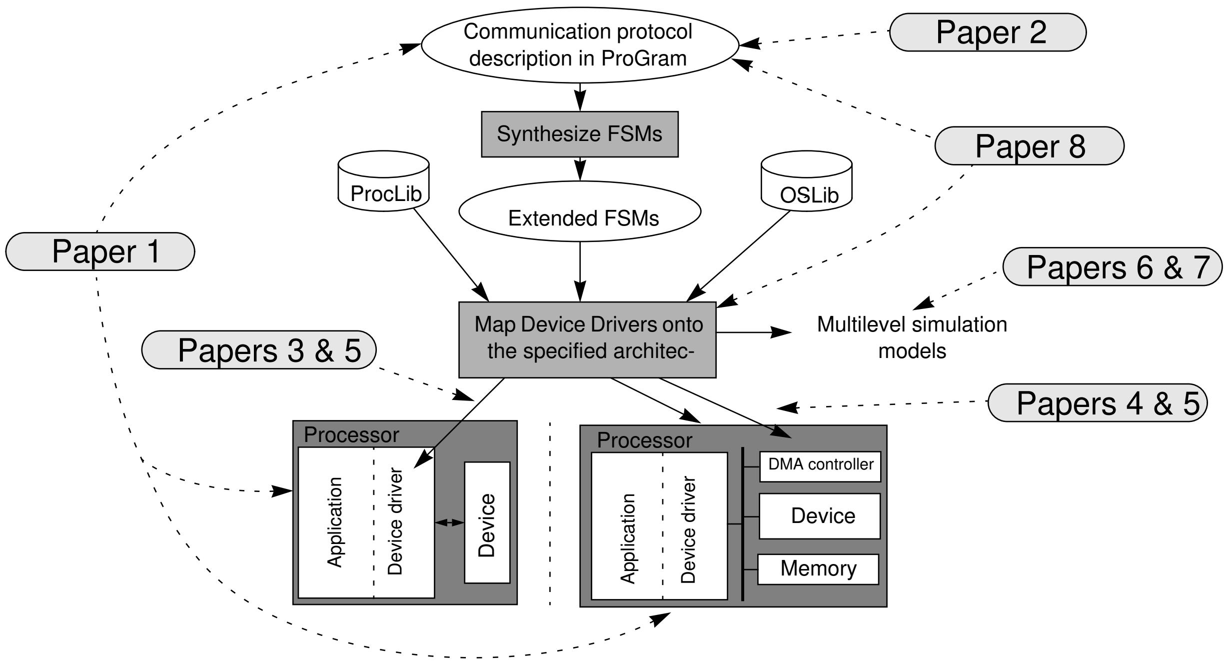


Figure 4.1: Identification of paper contents in the ComSyn design flow.

4.1 Analysis and problem formulation

4.1.1 Paper 1, Norchip 1997

This paper formulates the problem of communication synthesis in hardware/software embedded systems (a) by investigating the required concepts at the specification level and (b) by analysing potential communication routes in heterogeneous hardware/software architectures, thereby defining the input and output for communication synthesis. We also formulate a taxonomy for a complete classification of communication in hardware/software embedded systems.

4.1.2 Paper 8, Norchip 1998

This paper compares four different approaches to the specification and refinement of software parts in a hardware/software communication channel with respect to design productivity. The four approaches are CoWare [21], MakeApp [57], ComSyn [87,95,88,92] and manual refinement. This case study shows how the different tools can improve the design process. It also shows the deficiencies of the different approaches. The comparison is done by analysing the impact different techniques have on the design process. The analysis is based on two designs: a channel decoder of a transceiver in a D-AMPS base station and an operation and maintenance block of an ATM network. We compare design productivity by comparing the amount of effort required for development, validation and library maintenance.

4.2 Protocol description

4.2.3 Paper 2, Euromicro 1998

This paper presents an adaptation of the interface specification language ProGram to capture hardware/software interfaces, that is, device drivers and bus interfaces. ProGram is used for the architecturally independent modelling of device drivers and bus interfaces for mixed hardware/software systems. The specification of the protocol is separated from the description of processor bus interfaces and operating system device driver interfaces, which ensures high efficiency in device driver development and maintenance. A device driver synthesis method for single threaded architectures is presented together with the results of modelling and implementation efficiency for both device drivers and bus interfaces.

4.3 Protocol synthesis

4.3.4 Paper 3, DATE 1999

This paper presents a method for generating the software parts of a hardware/software interface (that is, the device drivers), which separates the behaviour of the interface from the architecture-dependent parts. The generation procedure targets a real-time kernel-based architecture. The synthesis procedure presented uses a ProGram protocol description and two libraries as input. The first library captures the processor-specific parts and the second captures the operating system kernel parts. The paper also presents an analysis and specification of the contents for the two libraries.

4.3.5 Paper 4, VLSI Design 1999

This paper presents a protocol synthesis method that generates a mixed hardware and software implementation, that is, a complement to the approach in paper 3. For the hardware part, the synthesis method will generate an application-specific DMA controller for each synthesized protocol specification. The software parts of the generated implementation are components for the initialization and synchronization of and communication with the DMA controller. Since this approach is based on a device driver synthesis system for software solutions (in paper 3), which adapts the device drivers generated to a selected processor and kernel, the generated hardware/software solution can also be adapted to any processor and OS kernel.

4.3.6 Paper 5, Kluwer DAES Submitted 1999

This paper presents an improvement of the approaches proposed in papers 3 and 4. The improvements are:

- Improved description of the synthesis procedures.
- Improved software synthesis procedure.
- DMA controller synthesis extended to handle access functions.
- Wider range of examples applied to the synthesis procedures.
- Complexity analysis of the synthesis methods.

4.4 Protocol validation

4.4.7 Paper 6, HLDVT 1998

This paper presents a multiphase validation approach that facilitates the simulation of interfaces at several steps during development. The technique is base on the generation of simulation models from an interface protocol description. The presented validation method keeps the simulation models consistent with both the specification and the implementation by generating the models with a technique derived from interface synthesis (in

Table 4.1: Author's contribution to each individual paper (M = main contributor, C = co-author).

Paper #	Author contribution			Contribution
	MO[a]	**AJ**[b]	**JÖ**[c]	
1	M	C		**MO**: Analysis of communication in embedded systems. **MO**: A taxonomy for classification of communication. **AJ**: Supervisor.
2	M	C	C	**MO** & **JÖ**: Adaptation of ProGram to specification of HW/SW interface with ProGram. **MO**: A method for generating SW code for single-threaded software. **AJ**: Supervisor.
3	M	C		**MO**: Analysis and proposal of library structure for capturing architecturally dependent parts of device drivers. **MO**: SW code generation methods for real-time kernel-based embedded systems. **AJ**: Supervisor.
4	M	C		**MO**: Proposal of a DMA architecture for synthesis. **MO**: A DMA controller synthesis method. **AJ**: Supervisor.
5	M	C		**MO**: Improved description of the synthesis procedures. **MO**: DMA controller synthesis expanded to handle access functions. **MO**: Wider range of example applied to the methods. **MO**: Complexity analysis. **AJ**: Supervisor.
6	M	C		**MO**: Multilevel validation technique based on generated simulation models. **AJ**: Supervisor.
7	M	C		**MO**: Adaptation of the validation approach to validation of HW/SW interfaces in IP-based design. **AJ**: Supervisor.
8	M	C		**MO**: A comparison between three different approaches with ours. **MO** & **AJ**: Case studies. **AJ**: Supervisor.

a. Mattias O'Nils
b. Axel Jantsch
c. Johnny Öberg

papers 3 and 5). This method enables the interface specification, simulation models and implementation to be consistent with each other. The paper shows that several validation phases are justified (a) by the faster simulation of early phases (up to four times faster than late phases), and (b) by allowing both hardware designers and software developers to work in their familiar tool environments as long as possible.

4.4.8 Paper 7, IPSDP 1998

This paper applies the approach taken in paper 6 to the verification of the interface protocols used for IP components in IP-based design. The method of generating simulation models results in significantly less effort being required to verify interfaces between software and IP components.

4.5 Author's contributions

The contribution of the author of this thesis has been essential to all the papers presented in the thesis. The exact contribution of each author is described in table 4.1. As described above, these papers cover the analysis, refinement, design and verification of hardware/ software interfaces.

5 Thesis summary

Aspects of modelling, synthesis and validation of hardware/software interfaces for low effort reuse have been studied and analysed in this thesis. Sections 2 and 3 give an extensive review of the related work. Section 3 also gives a motivation for the work together with a summary of the work presented in the eight appended papers. Section 4 identifies the original contribution for each paper.

The following section summarizes the conclusions reached during work on this thesis, followed by suggestions for improvements of the approach presented.

5.1 Conclusions

Paper 1 [89] presents an approach to the classification of communication within system specifications. It also defines the solution space and problem formulation of communication synthesis for embedded hardware/software systems. The problem formulation works as a base for the work covered by this thesis.

The thesis presents an approach to the generation of hardware/software interfaces between application software and hardware IP components. This approach reduces the effort required in designing the hardware/software interfaces to the generation time of the interface (see paper 5 [91]). The reuse and generation of device drivers is enabled by modelling the interface protocols using an architecture- and implementation-independent description (ProGram) (see paper 2 [95]). ProGram is a grammar-based notation extended to model device drivers. The protocol description contains less information than comparable models in C because no architectural information is captured in the protocol. Results from the modelling of design examples indicate that the size of the grammar-based description is half that of an ordinary C description (see papers 2, 3, 5 and 8 [95,92,91,90]).

Methods for translating the protocol description into software implementation are presented (see papers 2, 3 and 5 [95,92,91]), as well as into a DMA-based implementation (see papers 4 and 5 [87,91]). Results from these methods show that the performance (execution time) and size (code size) for these are similar to those modelled by hand in C (see paper 5 [91]).

Papers 6 and 7 [88,93] present methods for the generation of hardware/software co-verification models. These models can be generated for three levels of abstraction, each level differs in the details covered and simulation speed. The highest level of abstraction (stub simulation) gives up to four-times higher simulation performance and allows hardware and software developers to work in their familiar tool environments. The approach leads to better consistency between specification and implementation/simulation models for all abstraction levels, since all models are generated from the same protocol description.

5.2 Future work

5.2.1 Specification

ProGram is the grammar-based description language used to describe communication protocols. The initial target for ProGram was to capture protocols that were intended for hardware implementations. In this thesis, ProGram has been adapted to capture interface protocols intended for a hardware/software implementation. Although ProGram has been adapted to model hardware/software interface protocols, it still requires awkward descriptions to model certain behaviours, for example, unbounded loops and buffers.

Task: Learn from the results and do a thorough analysis of device drivers, and from this improve the description language.

5.2.2 Synthesis

The synthesis procedure presented here starts from where the synthesis procedure for hardware [124] has generated a set of FSMs. The translation from ProGram to state machines is done with a hardware implementation in mind. The FSMs are then transformed and optimized for software or DMA-based implementation. As shown in this thesis, the quality of the generated code is similar to that for handwritten code with respect to performance and size. It is likely that the code quality will increase if the translations from specification to implementation are merged, that is, translation from ProGram to FSMs would be optimized for device driver synthesis. In addition, heuristics for the trading of size and performance of the generated code could be applied.

Task: Merge the tasks of protocol description synthesis, architectural mapping and code generation by developing new methods for translating from description to FSMs.

5.2.3 Target architecture

As presented in this thesis, the methods for architectural mapping only support a simple bus architecture, as found in an embedded system. No work has been done to study the generation of device drivers for more complex systems with standard buses such as PCI [99] or SBus [106].

Task: Analyse the requirements for device drivers in a standard bus structure such as PCI and SBus, that is, multibus structure. Improve the library structure and the architectural mapping procedures to handle multibus architectures.

The generation of application-specific DMA controllers presented in this thesis supports only a read/write architecture. That is, if data is moved from source A and moved to sink B, the DMA controller first has to read the data from A and then write the data to B. Most off-the-shelf DMA controllers supports a *write-through* mode, that is, the DMA controller writes to B at the same time as it reads A. This performance improvement comes at the cost of changes to the implementation since the DMA controller has to control the sink B instead of having B controlled from the bus.

Task: Improve the methods for generating application-specific DMA controllers to handle write-through operation.

The granularity of parts of the protocol that can be implemented as a DMA controller is defined by whole access functions or interrupt handlers. This is not always the optimal solution since the data dependencies in the access functions can cause several bus operations.

Task: Develop methods for partitioning device driver behaviour between software implementation and DMA controller implementation to fit a set of performance and cost constraints.

6 References

[1] *256 Mbit SDRAM - 16M x 4 bit x 4 Banks Synchronous DRAM*, Data sheet KM416S16230A, Rev. 0.2. Samsung Electronics Inc., January 1999.

[2] *80C51-based 8-bit microcontrollers*, Philips Semiconductors, Netherlands, 1997.

[3] M. Abramovici, M. A. Bruer, A. D. Friedman, *Digital Systems Testing and Testable Design*, Rockville, Computer Science Press, 1990.

[4] Advanced RISC Machines (ARM) Ltd., 90 Fulbourn Road, CAMBRIDGE, Cambridgeshire, CB1 9JN, England.

[5] M. Afghahi and C. Svensson, “Performance of synchronous and asynchronous schemes for VLSI systems”, *IEEE Transactions on Computers*, Vol. 41, No. 7, pp. 858-872, July 1992.

[6] P. Ainsley, “The Embedded Micro Controller ASIC”, *Proceedings of the International Workshop on IP Based Synthesis and System Design*, pp. 97-100, 1998.

[7] A. Allara, S. Filipponi, W. Fornaciari, F. Salice, D. Sciuto, “A Flexible Model for Evaluating the Behavior of Hardware/Software Systems” *Proceedings of the 5th International Workshop on Hardware/Software Codesign*, March 1997.

[8] P. M. Athanas, H. F. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis”, *IEEE Computer*, 1993.

[9] H. B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley Publishing Company, ISBN 0-201-06008-6, 1990.

[10] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A, Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.*, Kluwer Academic Press, 1997.

[11] A. Balboni, W. Fornaciari, D. Sciuto, “Co-synthesis and Co-simulation of Control-Dominated Embedded Systems”, *Design Automation for Embedded Systems*, Vol. 1, No. 3, pp. 257-289, 1996.

[12] A. Balboni, W. Fornaciari, D. Sciuto, "Partitioning and Exploration Strategies in the TOSCA Co-Design Flow", *Proceedings of the 4th International Workshop on Hardware/Software Codesign*, March 1996.

[13] P. Bannon, "Alpha EV7: A Scalable Single-chip SMP", *MicroProcessor Forum*, October 1998.

[14] E. Barros, W. Rosenstiel, X. Xiong, "A method for partitioning UNITY language in hardware and software", *Proceedings of the European Design Automation Conference*, 1994.

[15] D. L. Barton, S. C. Fortier, "Using a systems description language for complete avionics systems", *Proceedings of the IEEE Aerospace Conference*, pp. 249-254, 1998.

[16] S. S. Battacharyya, J. T. Buck, S. Ha, E. A. Lee, "Generating compact code from dataflow specifications of multirate signal processing algorithms", *IEEE Transaction on Circuits and Systems*, Vol. 42, No. 3, pp. 138-150, 1995.

[17] L. Benini, G. De Micheli, *Dynamic Power Management of Circuits and Systems: Design Techniques and CAD Tools*, Kluwer, 1997.

[18] Th. Benner, R. Ernst, A. Österling, "Scalable Performance Scheduling for Hardware-Software Cosynthesis", *Proceedings of the European Design Automation Conference*, 1995.

[19] Th. Benner, J. Henkel, R. Ernst, "Internal Representation of Embedded Hardware-/Software-Systems", *Proceedings of the 1st International Workshop on Hardware/Software Codesign*, 1993.

[20] B. Bohem, B. Clark. E. Horowitz, C. Westland, R. Madachy, R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0", *Annals of Software Engineering*, Vol. 1, pp. 57-94, 1995.

[21] I. Bolsens, H. J. De Man, B. Lin, K. van Rompaey, S. Vercauteren, D. Verkest, "Hardware/Software Co-Design of Digital Telecommunication Systems", *Proceedings of the IEEE*, Vol. 85, No. 3, pp. 391-418, 1997.

[22] R. Brodersen, A. Chandrakasan, S. Sheng, "Design Considerations for Portable Systems," *IEEE International Solid-state Circuits Conference*, February 1993.

[23] R. Camposano, J. Wilberg, "Embedded System Design", *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 1, No. 1-2, March, pp. 5-50, 1997.

[24] R. Camposano, W. Wolf, *High-Level VLSI Synthesis*, ISBN 0-7923-9159-4, Kluwer Academic Publishers, 1991.

[25] S. Cardelli, M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, C. Sansoè, A. Sangiovanni-Vincentelli, "Rapid-Prototyping of Embedded Systems via Reprogrammable Devices", *Design Automation for Embedded Systems*, Kluwer Academics Publisher, Vol. 3, No. 2/3, pp. 149-162, March 1998.

[26] A. Chandrakasan, S. Sheng, R.W. Brodersen, "Low-Power CMOS Digital Design," *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 4, pp. 473-484, April 1992.

[27] E. Charbon, I. Torunoglu, "Intellectual Property Protection Via Hierarchical Watermarking", *Proceedings of the International Workshop on IP based Synthesis and System Design*, 1998.

[28] M. Chiodo, P. Giusto, A. Jurecska, M. Marelli, "A Formal Specification Model for Hardware/Software Codesign", *Proceedings of the 1st International Conference on Hardware/Software Codesign*, 1993.

[29] P. Chou, K. Hines, K. Partridge, G. Borriello, "Control generation for embedded systems based on composition of modal processes", *Proceedings of the International Conference on Computer Aided Design*, 1998.

[30] P. H. Chou, R. B. Ortega, G. Borriello, "The Chinook Hardware/Software Co-Synthesis System", *Proceedings of the International Symposium on System Synthesis*, 1995.

[31] P. H. Chou, R. B. Ortega, G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems", *Proceedings of the International Conference on Computer Aided Design*, pp. 488-495, 1992.

[32] P. Chou, E. A. Walkup, G. Borriello, "Scheduling for reactive real-time systems", *IEEE Micro*, Vol. 14, No. 4, pp. 37-47, August 1994.

[33] Computer on a Chip, *IEEE Computer*, January, 1971.

[34] *Control Design Automation Solution*, The MathWorks Inc., Watertown, MA, USA.

[35] *CORBA/IIOP 2.2 Specification*, Object Management Group, Inc., 492 Old Connecticut Path Framingham, MA 01701, USA, 1998.

[36] J-M Daveau, G. F. Marchioro, T. Ben Ismail, A. A. Jerraya, "Protocol Selection and Interface Generation for HW-SW Codesign", *IEEE Transaction on Very Large Scale Integration*, Vol. 5, No. 1, pp. 136-144, 1997.

[37] J. Davis, R. Galicia, M. Goel, C. Hylands, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie N. Smyth, J. Tsay and Y. Xiong, *Heterogeneous Concurrent Modeling and Design in Java*, Memorandum UCB/ERL M98/72, EECS, University of California, Berkeley, CA, USA 94720, November 23 1998.

[38] A. E. K. Dekdouk, O. Ait-Mohamed, M. S. Jahan, E. Cerny, “What About Formal Verification of IP-based SoC Designs?”, *Proceedings of the International Workshop on IP based Synthesis and System Design*, 1998.

[39] S. Devadas, A. Ghosh, K. Keutzer, *Logic Synthesis*, ISBN 0-07-016500-9, McGraw-Hill Inc., 1994.

[40] J. C. Días, J. Riesco, P. Plaza, “Design of an ARM based System-on-a-Chip for Pay Phones”, *Proceedings of the International Workshop on IP Based Synthesis and System Design*, pp. 101-105, 1998.

[41] P. Dreike, J. McCoy, “Co-Simulating Software and Hardware in Embedded Systems“, *Embedded Systems Programming*, Vol. 10, No. 6, June 1997.

[42] *DSP Workshop*, The MathWorks Inc., Watertown, MA, USA.

[43] Eagle Tools, Synopsys Inc., 700 East Middlefield Rd. Mountain View, CA 94043, USA.

[44] W. Ecker, M. Glesner, A. Vombach, “Protocol Merging: A VHDL-Based Method for Clock Cycle Minimizing and Protocol Preserving Scheduling of IO-Operations”, *Proceedings of the European Design Automation Conference*, pp. 624-629, 1994.

[45] M. Edvards, J. Forrest, “A Development Environment for the Co-synthesis of Embedded Software/Hardware Systems”, *Proceedings of ED & TC, E-DAC, EUROASIC*, pp. 469-473, 1994.

[46] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli, “Design of Embedded Systems: Formal Models, Validation, and Synthesis”, *Proceedings of the IEEE*, Vol. 85, No. 3, pp. 366-390. March 1997.

[47] M. Eisenring, J. Teich, “Domain-Specific Interface Generation from Dataflow Specifications”, *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, pp. 43-47, 1998.

[48] P. Eles, Z. Peng, K. Kuchinski, A Doboli, “System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search”, *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 2, No. 1, pp. 5-32, 1997.

[49] R. Ernst, J. Henkel, Th. Benner, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design and Test of Computers*, December 1993.

[50] R. Ernst, Th. Benner, *Communication, Constraints and User Directives in COSYMA*, Technical Report CY-94-2, Institut für DV-Anlagen, Technische Universität Braunschweig, 1994.

[51] J. Fleischmann, K. Buchenrieder, R. Kress, "Codesign of Embedded Systems Based on Java and Reconfigurable Hardware Components", *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pp. 768-769, March 1999.

[52] W. Fornaciari, P. Gubian, D. Sciuto, C. Silvano, "Power estimation of embedded systems: a hardware/software codesign approach", *IEEE Transaction on VLSI Systems*, Vol. 6, No. 2, pp. 266-275, June 1998.

[53] R. Foster, "A Design Style to simplify IP Integration and Verification", *Proceedings of the International Workshop on IP Based Synthesis and System Design*, pp. 39-44, 1998.

[54] D. Gajski, A. Wu, N. Dutt, S. Lin, *High-Level Synthesis*, ISBN 0-7923-9194-2, 1992.

[55] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, 1994.

[56] J. Gong, D. D. Gajski, A. Nicolau, "A performance evaluator for parameterized ASIC architectures", *Proceedings of the European Design Automation Conference*, pp. 66-71, 1994.

[57] R. Grehan, "Driver Assistance", *Computer design*, vol. 36, no. 1, pp. 75-80, 1997.

[58] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, PhD Thesis, Stanford, Dec. 1993.

[59] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

[60] W. Hardt, W. Rosenstiel, "Prototyping of Tightly Coupled Hardware/Software Systems", *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 2, No. 3/4, pp. 283-318, May 1997.

[61] A.Hemani, T.Meincke, P.Nilsson, T.Olsson, S.Kumar, P.Ellervee,J.Öberg, A.Postula, "Lowering power consumption in clock by using Globally Asynchronous, Locally Synchronous Design Style", *Proceedings of the 36th Design Automation Conference*, June 1999.

[62] J. Henkel, R. Ernst, U. Holtmann, Th. Benner, "Adaptation of Partitioning and High-Level Synthesis in Hardware/Software Co-Synthesis", *Proceedings of International Conference on Computer Aided Design*, pp. 96-100, 1994.

[63] J. Hou, W. Wolf, "Partitioning methods for hardware-software co-design", *Proceedings of the 4th International Workshop on Hardware/Software Codesign*, pp. 70-76, 1996.

[64] G. P Hyatt, United State Patent, Pat. no. 4 942 516, July, 1990.

[65] *I^2C Peripherals for Microcontrollers*, Philips Semiconductors, 1992.

[66] T. Ben Ismail, M. Abid, A. Jerraya, "COSMOS: A CoDesign Approach for Communicating Systems", *Proceedings of 2nd International Workshop on Hardware/Software Codesign*, 1994.

[67] T. Ben Ismail, K. O'Brien, A. A. Jerraya, "Interactive System-level Partitioning with PARTIF", *Proceedings of the European Design Automation Conference*, Paris, France, pp. 464-468, Feb. 1994.

[68] *INTEGRATION: The VLSI Journal*, Quarterly, North Holland Publishing, Amsterdam, Netherlands.

[69] A. Jantsch, P. Ellervee, J. Öberg, A. Hemani and H. Tenhunen, "Hardware-Software Partitioning and Minimizing Memory Interface Traffic", Proceedings of the European Design Automation Conference, pp. 226-231, 1994.

[70] A. Jantsch, J. Isoaho, "A Versatile Design Validation Environment by Means of Software Execution, Hardware Simulation, and Emulation", *Proceedings of the 36th SIMS Simulation Conference*, pp 322-325, 1994.

[71] A. Jantsch, S. Kumar, I. Sander, B. Svantesson, J. Öberg, and A. Hemani, "Comparison of Six Languages for System Level Descriptions of Telecom Systems", *Proceedings of the Forum on Design Languages*, vol. 2, pp 139 - 148, 1998.

[72] A. Jantsch, J. Öberg, P. Ellervee, A. Hemani, H. Tenhunen, "A software oriented approach to hardware-software co-design", *Proceedings of the International Conference on Compiler Construction*, pp. 93-102, 1994.

[73] *JavaBeans - Specification*, Sun Microsystems, Inc., 901 San Antonio Rd., Palo Alto, CA 94303 USA, 1998.

[74] A. A. Jerraya, K. O'Brien, "SOLAR: An Intermediate Format for System-Level Modeling and Synthesis", *Computer Aided Software/Hardware Engineering*, Ed. J. Rozenblit, IEEE Publisher, chap. 10, 1994.

[75] H. Jifeng, I. Page, J. Bowen, *Towards a Provably Correct Hardware Implementation of OCCAM*, Technical Report, Oxford University Computing Laboratory, 1994.

[76] S. C. Johnsson, *Yet another compiler compiler*, Computing Science Tech. Rep. 32, AT&T Bell Lab. Murray Hill, 1975.

[77] L. Lavagno, J. Cortadella, A. Sangiovanni-Vincentelli, "Embedded code optimization via common control structure detection", *Proceedings of the 5th International Workshop on Hardware/Software Codesign*, March 1997

[78] T. Lee, *The Design of CMOS Radio-Frequency Integrated Circuits*, Cambridge University Press, 1998.

[79] C. Liem, P. Paulin, A. Jerraya, "Compilation Methods for the Address Calculation Units of Embedded Processor Systems", *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 2, No. 1, pp. 61-78, January 1997.

[80] B. Lin, S. Vercauteren, "Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation", *Proceedings of the International Conference on Computer Aided Design*, pp. 101-108, 1994.

[81] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, A. Haxthausen, "LYCOS: the Lyngby Co-Synthesis System", *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 2, No. 2, March, pp. 195-235, 1997.

[82] G. De Micheli, R. K. Gupta, "Hardware/Software Co-Design", *Proceedings of the IEEE*, Vol. 85, No. 3, pp. 349-365, March 1997.

[83] G. De Micheli, D. C. Ku, F. Mailhot, T Truong, "The OLYMPUS Synthesis System for Digital Design", *IEEE Design & Test of Computers*, Vol. 7, No. 5, pp. 37-53, 1990.

[84] L. Mittag, "Device Drivers for Nonexistent Devices", *Embedded System Programming*, Aug. 01 1996, vol. 9, no. 8, pp. 30-32,34,36,38,40.

[85] P. Moeschler, H. P. Amann, F. Pellandini, "High-Level Modeling using Extended Timing Diagrams, A formalism for the behavioral specification of digital hardware", *Proceedings of the European Design Automation Conference*, pp. 494-499, 1992.

[86] R. Niemann, P. Marwedel, "Synthesis of Communicating Controllers for Concurrent Hardware/Software Systems", *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, pp. 912-913, 1998.

[87] M. O'Nils, A. Jantsch, "Synthesis of DMA Controllers from Architecture Independent Descriptions of HW/SW Communication Protocols", *Proceedings of the IEEE VLSI Design Conference*, pp. 138-145, 1999.

[88] M. O'Nils, A. Jantsch, "Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models", *Proceedings of the IEEE Workshop on High Level Design, Validation and Test*, November, 1998.

[89] M. O'Nils, A. Jantsch, "Communication in Hardware/Software Embedded Systems - A Taxonomy and Problem Formulation", *Proceedings of the 15th IEEE Norchip Conference*, pp. 67-74, 1997.

[90] M. O'Nils, A. Jantsch, "Refinement of HW/SW Communication Channels: Case Study and Comparison", *Proceedings of the 16th IEEE Norchip Conference*, pp. 230-237, 1998.

[91] M. O'Nils, A. Jantsch, "Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications", submitted to *Design Automation for Embedded Systems*, Kluwer Academics Publisher, 1999.

[92] M. O'Nils, A. Jantsch, "Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification", *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, Germany, pp. 562-567, March 1999.

[93] M. O'Nils, A. Jantsch, "HW/SW Interface Validation in IP based System Design", *Proceedings of the International Workshop on IP based Synthesis and System Design*, pp. 79-84, 1998.

[94] M. O'Nils, A. Jantsch, A. Hemani, H. Tenhunen, "Interactive Hardware-Software Partitioning and Memory Allocation Based on Data Transfer Profiling", *Proceedings of the International Conference on Recent Advances in Mechatronics*, pp. 447-452, 1995.

[95] M. O'Nils, J. Öberg, A. Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", *Proceedings of the Euromicro Conference*, pp. 55-58, August, 1998.

[96] B. Oelmann, *Design and performance evaluation of asynchronous micropipeline circuits for digital radio*, Lic. thesis, ISSN 1104-8697, Royal Institute of Technology, Stockholm, Sweden, 1996.

[97] R. B. Ortega, G. Borriello, "Communication Synthesis for Distributed Embedded Systems", *Proceedings of the International Conference on Computer Aided Design*, 1998.

[98] R. B. Ortega, L. Lavagno, G. Borriello, “Models and methods for HW/SW intellectual property interfacing”, NATO ASI on System-level Synthesis, 1998.

[99] *PCI Specification*, Version 2.2, PCI Special Interest Group, Oregon, USA, 1999.

[100]R. Passerone, J. A. Rowson, “Automatic Synthesis of Interfaces between Incompatible Protocols”, *Proceedings of the 35th Design Automation Conference*, June 1998.

[101]*Pentium II Processor Developer’s Manual*, Intel Inc., October 1997.

[102]*PowerPC - MPC 750 RISC Microprocessor Technical Summary*, Motorola Inc., 1997.

[103]*Renoir - Reference Manual*, Mentor Graphics Corporation, 1998.

[104]J. Rowson, A. Sangiovanni-Vincentelli, “Felix initiative pursues new codesign methodology”, *Electronic Engineering Times*, pp. 50,51, 74, June 1998.

[105]M. Santarini, “French allies to blend CASE, EDA in SoC tool”, *Electronic Engineering Times*, Issue: 1035, November 1998.

[106]*SBus Specification*, IEEE Std 1496-1993, 1993.

[107]*Seamless Co-Verification Environment User’s and Reference Manual*, Ver. 2.3, Mentor Graphics Inc., 1998.

[108]G. C. Sih, E. A. Lee, “Declustering: a new multiprocessor scheduling of periodic, real-time tasks”, *IEEE Transaction on Parallel and Distributed Computing*, Vol. 4, No. 6, pp. 625-637, June 1993.

[109]A. Smailagic, D. P. Siewiorek, R. Martin, J. Stivoric, “Very Rapid Prototyping of Wearable Computers: A Case Study of Custom versus Off-the-Shelf Design Methodologies, *Design Automation for Embedded Systems*, Vol. 3, No. 2/3, pp. 217-230, March 1998.

[110]J. A. Stankovic, M. Spuri, M. Di Natale, G. C. Buttazzo, “Implications of classical scheduling results for real-time systems,”, *IEEE Computer*, Vol. 28, No. 6, pp. 16-25, June 1995.

[111]K. Tammemäe, M. O'Nils, A. Hemani, “Flexible Codesign Target Architecture for Early Prototyping of CMIST Systems”, *Field-Programmable Logic Smart Application, New Paradigms and Compilers*, Springer-Verlag, ISBN 3-540-61730-2, pp.193-199, 1996.

[112]E. Tuggle, “Writing Device Drivers”, *Embedded Systems Programming*, Jan. 1993, pp. 42-65.

[113]F. Vahid, J. Gong, D. D. Gajski, “A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning”, *Proceedings of the European Design Automation Conference*, 1994.

[114]C. Valderrama, F. Naçabæl, P. Paulin, A. Jerraya, “Automatic VHDL-C Interface Generation for Distributed Cosimulation: Application to Large Design Examples”, *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 3, No. 2/3, pp. 199-216, March 1998.

[115]S. Vercauteren, B. Lin, “Hardware/Software Communication and System Integration for Embedded Architectures“, *Design Automation for Embedded Systems*, Kluwer Academic Publisher, Vol. 2, No. 3/4, pp. 359-382, May 1997.

[116]S. Vercauteren, D. Verkest, J. Van Der Steen, “Combining Software Synthesis and Hardware/Software Interface Generation to Meet Hard Real-Time Constraints”, *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, pp. 556-561, March 1999.

[117]*VSI Alliance - Architecture Document*, Version 1.0, 1997.

[118]W. H.Wolf, “Hardware-Software Co-Design of Embedded System”, *Proceedings of the IEEE*, vol. 82, no. 7, July 1994.

[119]T. Y. Yen, W. Wolf, “Performance estimation of distributed embedded systems”, *Proceedings of the International Conference on Computer Design*, 1995.

[120]J. S. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, A R. Newton, “Design and Specification of Embedded Systems in Java Using Successive, Formal Refinement”, *Proceedings of the 35th Design Automation Conference*, 1998.

[121]H. Zeltwanger, “An inside look at the fundamentals of CAN, *Control Engineering*, 42(1), January 1995.

[122]J. Zhu, R. Dömer, D. D. Gajski, “Syntax and Semantics of the SpecC Language”, *Proceedings of the Synthesis and System Integration of Mixed Technologies*, December 1997.

[123]J. Zhu, D. D. Gajski, “A Retargetable, Ultra-fast Instruction Set Simulator“, *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, pp. 298-302, March 1999.

[124]J. Öberg, A. Kumar, A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, *Proceedings of the 9th International Symposium on System Synthesis*, pp. 14-19, Nov. 1996.

Paper 1

Communication in Hardware/Software Embedded Systems - A Taxonomy and Problem Formulation

Communication in Hardware/Software Embedded Systems - A Taxonomy and Problem Formulation

Mattias O'Nils, Axel Jantsch
Royal Institute of Technology, Electronic System Laboratory
Electrum 229, S-164 40 Kista, Sweden
Receipt: +46 (0)8 752 1417, Fax: +46 (0)8 752 1240
E-mail: mattias | axel@ele.kth.se

ABSTRACT

We formulate the problem of communication synthesis in HW/SW embedded systems by (a) investigating the required concepts at the specification level and (b) by analyzing potential communication routes in heterogeneous HW/SW architectures. Thus, we define the input and output for communication synthesis and we formulate a taxonomy for a complete classification of communication in HW/SW embedded systems.

INTRODUCTION

Communication in a digital embedded hardware/software system tend to cause many errors and force designers to spend a lot of time designing the communication module. This is the motivation to research effective ways to model and refine communication within an embedded system and to transform these models into an implementation for some specific architecture.

This paper provides a frame to analyze the communication synthesis problem for mixed hardware/software embedded systems. There exists a wide variety of communication concepts used in system specification, and there exists an equally large diversity in the possible ways to implement communication. All existing communication synthesis approaches take a narrow view and consider only one or a few communication concepts which are then mapped on one out of very few implementation alternatives. As a first step to generalize existing approaches we unfold the full problem by exhaustively characterizing communication concepts and communication routs of the implementation. In figure 1 a general system for communication synthesis is illustrated, starting from a specification of the communication to the detailed implementation. This analysis leads to a taxonomy in form of a graph for classification of communication constructs. Then we present an analysis on the communication at architectural level of a heterogeneous hardware/software system. These two investigations together with a thorough research of the related work form an outline of the communication synthesis problem and work as a starting point for a future research.

The remaining part of this paper is organized so that the next two chapters contain the analysis of the input and output for communication synthesis respectively. The last chapters analyze the parts of communication synthesis that have been solved and which parts are still open for research.

COMMUNICATION WITHIN A SPECIFICATION

In executable specifications for embedded hardware/software systems, there are two principle ways to model communication: shared variable and channel based. We propose to use channels to achieve inter process communication in an embedded system, since we consider shared variables to be a technique to be used for implementation and not for specification. This we base on the fact that shared variables will force the designer to add all control behavior for the communication into the functional specification.

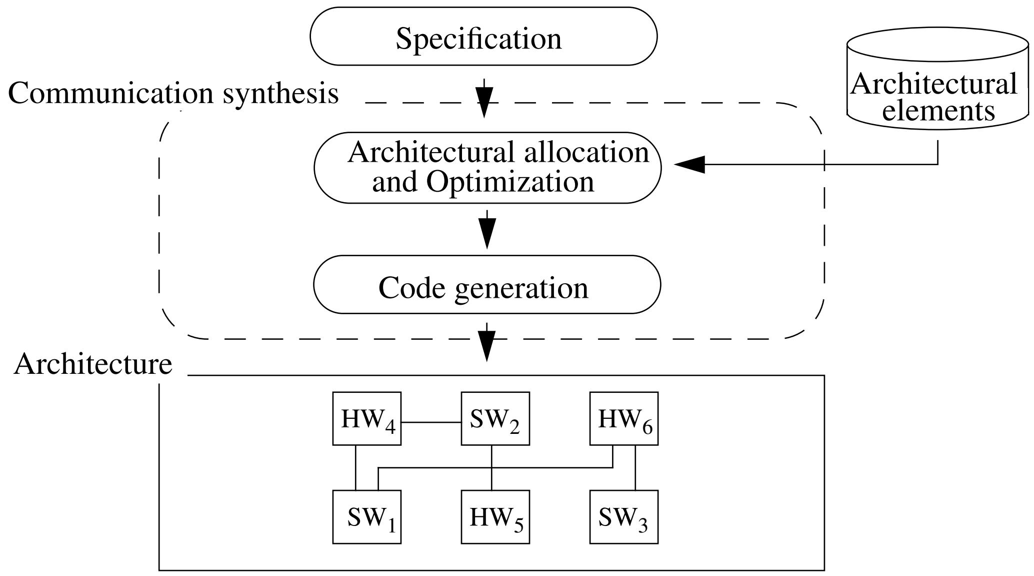


FIGURE 1. Interface design flow for embedded hardware/software system.

By using channel based communication we can ignore implementation and control behavior for the communication, this enables the designer to concentrate on the functionality. But with the notation we propose it is still possible to model shared variables.

Channel classification

To be able to completely classify the communication construct at systems specification level, we have to investigate all possible communication situations needed during specification of an embedded system. We do this by listing a set of situations that need explicit constructs for describing channels at the specification level. We also look at constructs existing in specification languages and look at design examples for motivation of certain constructs.

Life time: Depending on when a channel is constructed and destructed, we need constructs to determine the life time of the channel:

a. Communication primitives refer to a logical channel, the channel is constructed at compile time. This kind of construct is often used in programming languages, e.g. C^x [8], SDL [18]. In our notation we call this a *static channel*.

b. Communication primitives refer to a logical channel, the channel is constructed during run time. This construct is used in SDL [18] when a process is constructed during run-time, the channel is constructed at the same time. In our notation we call this a *dynamic channel*.

c. Communication primitives address the process it wants to communicate with, i.e. do not refer to a routed channel. This kind of method is often used for communication between processes in software systems. E.g. in Erlang [1] every message directly addresses the receiving process and no connection exists between processes when no message is transmitted. In our notation we call this a *virtual channel*.

Connectivity: In an embedded system there is not only a need to describe a channel between only two processes, sometimes there is a need to describe a channel that is shared between several producer and consumers of data.

a. The first and most simple case, is a channel between one producer and one consumer of data, i.e. a connection between two processes. Called *1to 1 unicast* in our notation.

b. Next case is the connection with one producer and several consumers of data, i.e. a connection from one to several processes. An example of this is the broad cast of a initialization signal at system start up. Called *1 to n multicast* in our notation.

c. Third case is the situation were we have several producers and one consumer of data, called *n to 1 unicast*. Several problems can occur with this kind of construct since collisions can occur. This construct have to model a scheme that will handle collisions. There are two major schemes that could be used: all messages are sent but the messages have to be ordered according some algorithm or messages are lost according to some priority scheme.

d. Last case and the most general case is when we have several producers and several consumers, a design example of this could be Ethernet. We call this *n to m multicast* in our notation. This construct have to handle similar problems to the one in c.

Control type: The third category requires constructs to model control behavior for the communication.

a. The simplest kind of control behavior for a transfer is, when the producer sends data and does not wait on the consumer to consume it. This kind of construct could be used to model polling of data in a system. This is called *non-blocking* transfer in our notation.

b. Another simple control behavior is when a producer waits for the consumer to consume the data. Remote procedure call (RPC) use this behavior, that is called *blocking* transfer in our notation.

c. The last and most general control type is the buffered control behavior, the buffered channel has to have a buffer size associated to it. Within the construct for a buffered channel there has to be a description how to handle the situation when the buffer is full. In principle there are two ways to handle buffer overflows: let data get lost according to some scheme (e.g. oldest data is lost) or no data will be lost and the producer of data has to wait for the consumer to empty the buffer.
A non-blocking and blocking transfer are a special cases of buffered transfer with one as buffer size. Non-blocking uses the scheme that the oldest data is overwritten when the buffer is full. For blocking, buffer-full is handled so that no data is lost and when the buffer is full the producer will wait until the buffer is emptied.

Channel type: In the three classification types above, the communication channel are performed with send/receive primitives, i.e. it means the communication channel is *discontinues*. Sometimes there is a need to describe a channel that transmits continuously over time without using send/receive primitives (e.g. Reveal/View in SDL [18]), called *continues* channel in our notation. This channel can either be unicast or multicast, if there is more than one producer of data there is a need to handle message collision. In the continues case the collision handling will be similar to resolution functions in VHDL. A continues channel is either static or dynamic.

Communication Classification Graph (CCG)

When all communication constructs are listed and motivated we have gathered all the constructs into a graph, figure 2. This graph forms a taxonomy of specification constructs for describing communication. This graph can be used to classify communication and to define channel specifications. The graph is also used in the related work chapter to evaluate the different approaches.

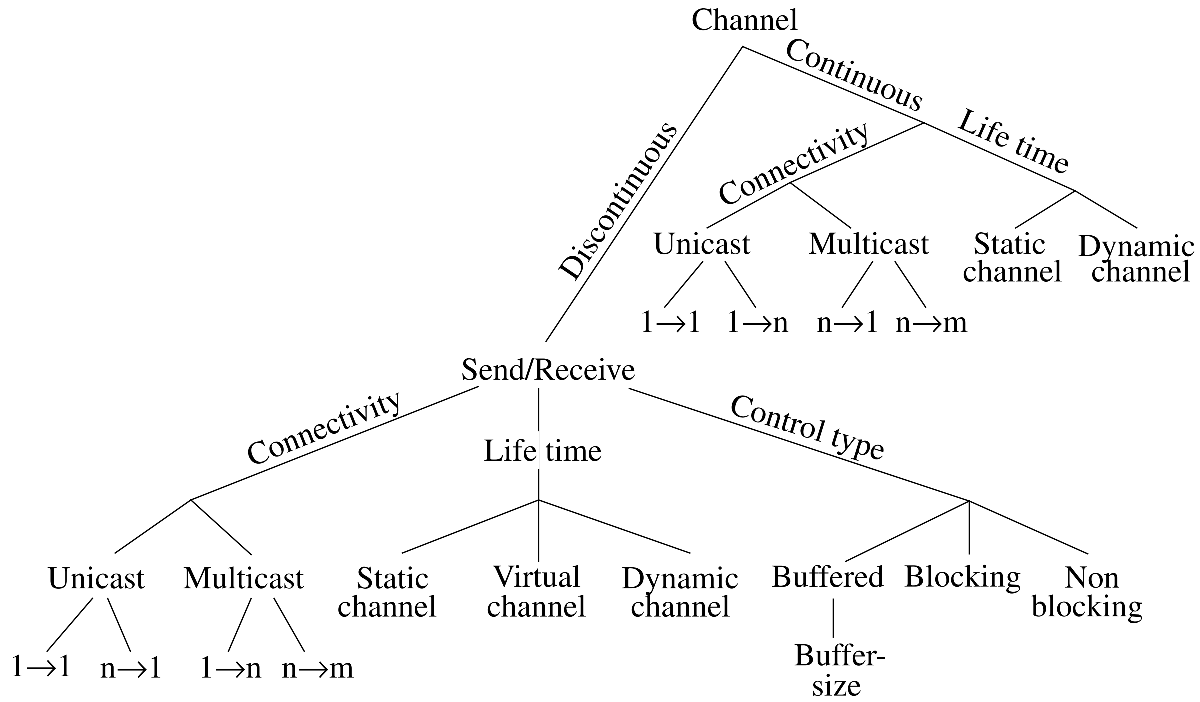


FIGURE 2. Communication Classification Graph

ARCHITECTURAL COMMUNICATION ROUTES

When the communication at specification level has been determined, the second step to outline a technique for translation of a behavioral description of the communication into an implementation is to identify and characterize the different cases of communication that can occur in a heterogeneous embedded system.

The hardware part of the system can consist of several different hardware blocks that can either be designed by the designer (co-processor, IO) or be library elements like co-processors, IO and processors. The processors (RISC, CISC, DSP, micro controller) execute the software part of the system functionality. The software also consist of several different modules, operating system (often an real-time operating system), library functions, and application code.

Considering all this we can define the possible routes for which communication can occur and this way get an overview over the communication at the architectural level.

The different routes in a heterogeneous system are listed below and the each route is also described in figure 3. The reason for separating communication between two different modules depending on which of them are initiating the communication is that the control behaviors are so different. For example, consider the communication between a co-processor and a software process: When the software wants to start a transfer to hardware it only addresses the hardware address space but when the hardware wants to start a transfer it has to get the attention from the software by interrupt the software execution in some way.

1. SW process connected to another SW process on the same processor.
2. SW process connected to another SW process on a different processor, with the same or a different operating system.
3. SW process connected to a HW process (co-processor).
4. SW process connected to a peripheral (E.g. off-the-shelf-component, library module, etc.).
5. SW process connected to a library SW module.

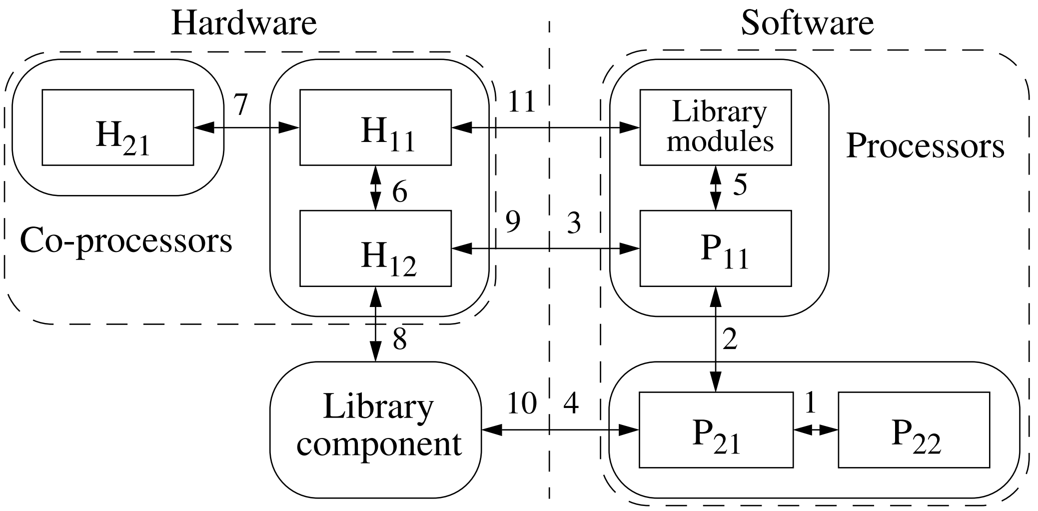


FIGURE 3. Communication routes in a heterogeneous embedded system

6. HW process connected to another HW process on the same chip (partition).
7. HW process connected to another HW process on another chip (different partitions).
8. HW process connected to a peripheral (E.g. off-the-shelf-component, library module, UART, etc.).
9. HW process connected to a SW process.
10. Peripheral connected to a SW process.
11. HW co-processor connected to a library SW module.

RELATED WORK

Communication primitives in specifications

Most research groups use channel based communication at the specification level in some way, with exceptions like LYCOS [15] which does not support parallelism and communication. When this is partitioned into hardware and software, an interface is put between the hardware software boundary (similar to the AKKA system [9]). SpecChart/SpecSyn [10] does not use channels either, it represents communication with shared variables.

In the COSYMA cosynthesis environment [9], communication routes in the C^x [8] specification is declared as logical static channels and bundles of channels. These logical channels can be mapped in the specification onto some predefined communication device/resource. The channel service functions are blocking and non-blocking send/receive. The COSYMA approach is similar to the one used in the CoWare system [20], which also use a super set of C as specification language. In CoWare channels are declared to connect two ports one master and one slave (static connection). All communication over these channels are done with remote procedure calls initiated from a master, the master can either be a send or a receive function. The difference between these two approaches is that CoWare has a well defined strategy for refinement of the channels.

The two codesign systems TOSCA [2] and Vulcan [11] have also similar communication strategy, both are using static channels and send/receive service functions to access the channel. The difference is that TOSCA handles only buffered and unbuffered non-blocking send/receive functions while Vulcan also supports blocking send/receive.

In Polis [3] communication is performed between ports that form implicit channels. The communication on these channels are performed with read and write statements controlled by event statements. With these statements blocking, non-blocking and buffered send and receive

operations can be constructed. But this forces the designer to handle the communication control within the functional specification.
COSMOS [21] uses SDL as specification language and uses the communication technique used in SDL. Thus, it uses static or dynamic channels on which send and receive operation can work on. The send operations are non-blocking and receive operations are blocking. Send/receive can be dynamically also by specifying the process identification you want to communicate with, what we call virtual channel. But this is not supported when the SDL specification is translated to the intermediate format SOLAR [14]. For the COSMOS row in table 1, the dots indicating not fully supported indicates that the construct is supported in the specification language but not by the tool.

Research group	Communication constructs											
	Connectivity				Life time			Control type			Channel	
	1→1	1→n	n→1	n→m	Static	Virtual	Dynamic	Non-Blocking	Blocking	Buffered	Continuous	Discontinuous
LYCOS	-	-	-	-	-	-	-	-	-	-	-	-
COSYMA	l	m	m	m	l	m	m	l	l	l	m	l
CoWare	l	m	m	m	l	m	m	m	l	l	m	l
TOSCA	l	m	m	m	l	m	m	l	m	l	m	l
Vulcan	l	m	m	m	l	m	m	l	l	l	m	l
Polis	l	m	m	m	l	m	m	l	l	l	m	l
COSMOS	l	w	m	m	l	w	w	l	l	l	w	l
SpecChart	l	l	w	w	l	-	m	-	-	-	l	m
l construct is supported, w construct not fully supported, m construct is not supported, - not applicable												

TABLE 1. Comparison of system level expressiveness for different research groups.

Specification constructs lacking in all research approaches above are: Virtual channels and multi cast channels.

Implementation elements

Since most research groups use direct mapping of the channels on to an architecture, the implementation elements consist of design modules that implements the channel and its service functions. In the case of manual refinement the channel and service functions are written without formal rules, so these could either use library elements or only self designed implementations.
The COSMOS group has specified three implementation elements for the allocation/mapping algorithm [7]: a bidirectional handshake protocol, a single FIFO and a dual FIFO.

Mapping algorithms

The COSMOS group is the only research group so far, that has developed mapping algorithms to allocate and bind architectural elements to implement the logical communication channels [7]. Other groups use direct mapping of the service functions on some implementation [2,4,8,11,13] or manual refinement [19,20] instated of some heuristic.
For lower level transformations several groups have reported algorithms [5,16,17,19] to translate low level protocols. The Chinook system includes algorithms for generating the control signals on a micro controller to interface components. This algorithm starts with a description of the wave forms to perform read and write operation to a certain component. CoWare has a

similar low level protocol translation algorithm [17] but that is not limited to micro controllers.

Architecture support

Codesign systems [2,3,5,9,15,21] often handle a hardware/software system with one processor and a set of co-processors. Gajski et. al. [10] do consider multi processor architecture during partitioning but perform very limited optimization on the communication structure at the architectural level.

CONCLUSION

In this paper we have investigated what constructs we need to describe communication at specification level. We have done this by explaining situations where an explicit construct would be useful, point to constructs used in different specification languages and constructs needed to describe situations that can occur for certain design examples. This investigation has led to a communication classification graph. Channels that can be defined from the graph (figure 2) are the desired input for the ideal communication synthesis environment. As can be seen in the chapter on related work each research group covers a small fraction of the classification tree (figure 2) and not even a union of the research efforts cover the whole tree, virtual and multi cast channels are still not covered. This leads to the conclusion that there is a need to develop techniques and constructs so it is possible to specify all constructs that can be extracted from the classification. The modeling concepts for communication should be comparably rich to modeling concepts for computation we find in any programming or hardware description language.

The paper also covers an investigation on the output from the ideal communication synthesis system. This is done by describing all the different communication routes that can occur at architectural level. One can see that the support for heterogeneous hardware/software architectures are very limited in the current approaches. This is contradictory to the idea in codesign of using a highly flexible and inexpensive solution. A codesign system with communication synthesis should support a heterogeneous system with different processors, different operating systems and different kind of custom hardware.

By outlining input and output for an ideal communication synthesis environment we have defined the problem of translating a system level communication constructs to a communicating architecture.

Open areas

With the information in the related work we can point on areas still open for research in the field of communication synthesis, these are:

- general concepts for communication specification
- efficient architectures for communication synthesis
- methods for optimizing the communication in heterogeneous systems which support a rich set of specification concepts and architectural routes. Specially optimizing techniques that also include the software and operating system aspects.

REFERENCES

1. J. Armstrong, R. Virding, M. Williams, Concurrent Programming in Erlang, Prentice Hall, 1993.
2. A. Balboni, W. Fornaciari, D. Sciuto, Co-synthesis and Co-simulation of Control-Dominated Embedded Systems, Design Automation for Embedded Systems, 1(3), pp. 257-289, 1996.

3. M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, A. Sangiovanni-Vincentelli, A Case Study in Computer-Aided Co-design of Embedded Controllers, Design Automation for Embedded Systems, Kluwer Academic Publisher, 1(1) pp. 51-67, 1996.
4. M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli, L. Lavagno, Hardware-Software Codesign of Embedded Systems, IEEE Micro 14(4), pp. 26-36, August 1994.
5. P. Chou, R. Ortega, G. Borriello, Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems, in Proc. of the IEEE/ACM ICCAD, Santa Clara, pp. 488-495, CA, November 1992.
6. P. Chou, R. Ortega, G. Boriello, The Chinook Hardware/Software Co-Synthesis System, in Proc. of the 8th ISSS, Cannes, France, September1995.
7. J-M. Deveau, G. Fernades Merchioro, T. Ben Ismail, A. Jerraya, Protocol Selection and Interface Generation for HW-SW Codesign, In Transaction on VLSI Systems, 5(1), March 1997.
8. R. Ernst, Th. Benner, Communication, Constraints and User Directives in COSYMA, Technical Report CY-94-2, Institut für DV-Anlagen, Technische Universität Braunschweig, June 1994.
9. R. Ernst, J. Henkel, Th. Benner, Hardware-Software Cosynthesis for Microcontrollers, in Proc. of IEEE ED&TC, December 1993.
10. D. Gajski, F. Vahid, S. Narayan, J. Gong, Specification and Design of Embedded Systems, Prentice Hall, ISBN 0-13-150731-1.
11. R. K. Gupta, Co-Synthesis of Hardware and Software for Digital Embedded Systems, Kluwer Academic Publisher, ISBN 0-7923-9613-8
12. A. Jantsch, J. Öberg, P. Ellervee, A. Hemani, H. Tenhunen, A software oriented approach to hardware-software co-design, International Conf. on Compiler Construction, CC-94, pp. 93-102, Edinburgh, Scotland, 1994.
13. D. Jensen, J. Madsen, S. Pedersen The Importance of Interfaces: A HW/SW Codesign Case Study, in Proc. of th 5th Codes/CASHE'97, Braunschweig, Germany, March 1997.
14. A. A. Jerraya, K. O'Brien, SOLAR: An Intermediate Format for System-Level Modeling and Synthesis, Ed. J. Rozenblit, Computer Aided Software/Hardware Engineering, IEEE Publisher, chap. 10, 1994.
15. J. Madsen, J. Grode, P.V. Knudsen, M.E. Pedersen, A. Haxthausen, LYCOS: the Lyngby Co-Synthesis System, Design Automation for Embedded Systems, 2(2), Kluwer Academic Publisher, pp. 195-235, 1997.
16. J. Madsen, B. Hald, An Approach to Interface Synthesis, in Proc. of the 8th ISSS, Cannes, France, September1995.
17. B. Lin, S. Vercauteren, Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation, in Proc. of ICCAD'94, 1994.
18. A. Olsen, O Faergemand, B. Moeller-Pedersen, R. Reed, and J.R.W Smith, Systems Engineering with SDL-92, North Holland, 1995.
19. R. Ortega, G. Boriello, Communication Synthesis for Embedded Systems with Global Considerations, in Proc. of the 5th Codes/CASHE, Braunschweig, Germany, March 1997.
20. D. Verkest, K. Van Rommpaey, I. Bolsens, H. de Man, CoWare - A Design Environment for Heterogeneous Hardware/Software Systems, Design Automations for Embedded Systems, 1(4), 357-386, 1996.
21. M. Voss, T. Ben Ismail, A. Jerraya, K-H. Kapp, Towards a Theory of Hardware/Software Codesign, In Proc. of th 2nd Codes/CASHE, Grenoble, France, 1994.

Paper 2

Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces

Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces

Mattias O'Nils, Johnny Öberg, Axel Jantsch
Royal Institute of Technology
Department of Electronics, Electronic System Design Lab.
Electrum 229, S-164 40 Kista, Sweden
Email: {mattias,johnny,axel}@ele.kth.se

Abstract

ProGram, a grammar based communication protocol description language, is used for architectural independent modelling of device drivers and bus interfaces for mixed hardware/software systems. The specification of the protocol is separated from the description of processor bus interfaces and operating system device driver interfaces, which ensures a high efficiency in device driver development and maintenance. A synthesis method for device drivers is presented together with results on modelling and implementation efficiency for both device drivers and bus interfaces.

1 Introduction

Embedded IPs integrated in large systems on a single chip, make interface design and validation to a dominating design problem. We address the problem of interfacing devices in embedded hardware/software systems. As elaborated by Tuggle in [9], for a given device, a device driver must be written for every operating system it will be used with. For hierarchical device drivers like SCSI systems, the number of device drivers needed for a device is the product of the number of supported operating systems and the number of supported host adapters. If we also add different implementation styles of a device driver, the number of possible implementations will be the product above, multiplied with the number of implementation alternatives. One quickly realizes, that verifying all these device drivers is a huge task. To solve this, we propose techniques for architecture independent modelling and refinement of hardware/software interfaces (device drivers) as well as for the hardware/hardware interfaces (bus interfaces).

Figure 1 identifies what has to be modelled to interface a device to an *of-the-shelf* processor. For device drivers one needs to declare the I/O to both the device and to the application software from where the device driver is called. The internal structure of device drivers have different layout for different software architectures [9]. But the general structure is common, a device driver is composed of several functions for initialization, access of data and interrupt handlers, see figure 1. To describe the functionality of the device driver functions, the specification language has to capture all the behaviours defined in figure 1, field c.

To our knowledge, there exists no architecture and implementation independent approach for device driver modelling. For the bus interface process, which translates the bus protocols of the two devices, exists several approaches to generate bus interfaces from architecture independent descriptions [2,3,4,7,8]. Our approach extends ProGram [10] to model both device drivers and bus interfaces. ProGram is a data communication protocol description language that

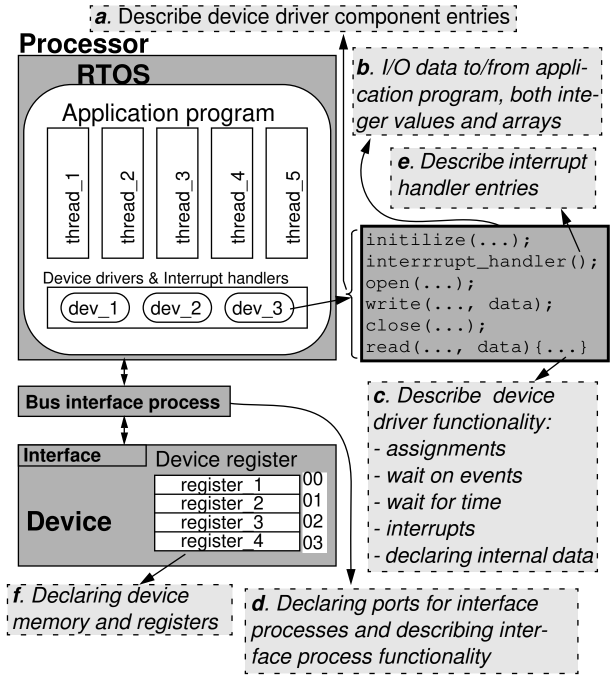


FIGURE 1. Identifying device driver and bus interface modelling issues.

has been proven efficient for the modelling and synthesis of telecommunication protocols [10].

The rest of the paper is organized in the following way. The next section goes through the related work and compares it to our approach. Then we present the ProGram language and propose extensions to model both device drivers and bus interfaces. The fourth section discusses synthesizing techniques for both device drivers and bus interfaces. Finally, we conclude the paper with results on modelling and implementation efficiency.

2 Related Work

The main focus in literature on device drivers is on writing implementations [9] or providing tools for library handling [5]. Chou et al. [1] models and generates software device drivers, but not in the context of an operating system as we do, but rather to emulate the bus interface in software. In our approach we provide the possibility to specify both the bus interfaces and device drivers in one architecture independent language, ProGram.

Bus-level protocols for components are usually described graphically by means of timing diagrams in a data sheet. For this reason, [1] and [8] have developed specification languages based on timing diagrams. A problem with these approaches is that it is not possible to specify high level behaviour. Lin et al. model the interface with extended STG (Signal Transition Graph) [7] which is based on Petrinets. With this modelling technique they can model asynchrony and events. On the extended-STG model asynchronous synthesis is performed. This approach enables the modelling of more complex behaviours. [3] and [4] use hardware description languages to model bus interfaces, Gajski et al. use a program state machine (PSM) to specify interfaces but in all examples they use the HDL capability of the PSM to model interfaces. Öberg et al. present a communication protocol synthesis tool that specifies the design in a special purpose language ProGram [10], which is based on context free grammar, the synthesizable subset is limited to a regular grammar with attributed conditions.

Table 1 shows a comparison between some of the approaches described above and the extended version of ProGram presented in this paper. The table shows that interfaces presented by other research groups are modelled more efficiently in ProGram. The example for SEQ in row 4 is an asynchronous circuit, which is the primary target for SEQ but not for ProGram, but the ProGram model is still comparable with the SEQ model. For synchronous interfaces, ProGram is better suited than SEQ.

TABLE 1. Comparison of Lines of Code (LoC).

Reference model	LoC	ProGram	Ratio
HDL (VHDL), List 1 in [3]	20	5	4.0
Extended TD, fig 1 in [8]	16[a]	7	N.a.
PSM, Fig 8.10 in [4]	21	12	1.8
Timing diagram (SEQ), [2]	6	6	1.0
Petrinets (E-STG), fig 5 in [7]	11	6	1.8

a. Number of conditions specified graphically

3 Interface modelling with ProGram

ProGram (Protocol Grammar), the proposed language for interface description, is inspired by YACC [6] with respect to its notations but has been designed to handle hardware synthesis. Specifications in ProGram deal with sequences of allowed events instead of states and state transitions as it would in a FSM model.

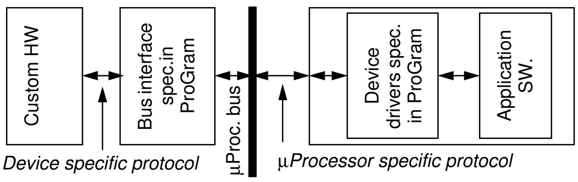


FIGURE 2. Identifying parts specified with ProGram.

ProGram as presented in [10] needs extensions to be able to model device drivers and bus interfaces, since ProGram in [10] targets protocol specification for hardware implementation. Figure 2 highlights the parts of an interface between a processor and a device we want to model with ProGram. The rest of this section is devoted to describing how device driver/bus interface functionality is captured with ProGram, at the end the ProGram language is described

3.1 Modelling of device drivers

The nature of a device driver is to assign and access registers/memories according to predetermined control flow, waiting for events to occur and/or waiting for some specified time interval to elapse. Hence, a device driver is characterized with assignments, control flow, event handling rather then data flow and computations, these characteristics fit very well to the spirit of ProGram. However to model device drivers with ProGram, we first have to analyse behaviours that are not captured with ProGram as specified in [10].

Port declarations. A device driver has two types of ports: the first port type is the communication with the device (i.e. declaration of device registers/memories) and the second type of port is for the communication with the application software. Both these types are captured with the ordinary port declaration in ProGram. The identification of where a port is connected, is done with the two key words: `hw` (device register) and `sw` (port to application software). For the port declared as `hw`, an address relative to the base address of the device has to be specified.

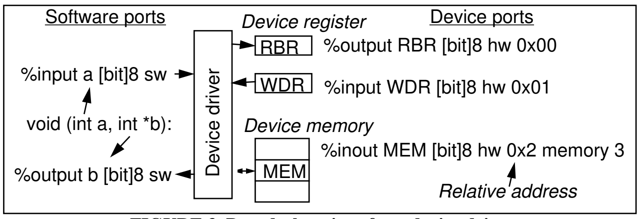


FIGURE 3. Port declarations for a device driver.

The only data type allowed in ProGram is unsigned bit vectors, this is not a problem since most of the functionality of a device driver is to control and move data of data types that can be derived from a bit vector type. Internal arrays are declared as memory in ProGram, the same key word is used to identify array signals (pointers) in the port declaration, i.e. `memory`. These arrays are treated in the same way as memories with the layout specified by the type in the port declaration.

Time. Real time has to be handled in ProGram in order to capture the full functionality of device drivers. Since within a device driver it can be necessarily to wait for some time interval instead of waiting for events, for this the key word `wait(`*`time`*`)` is used. Another use of time is to specify timing constraints for certain wait loops, i.e. define the wait's timeout time to prevent dead-locks.

TABLE 2. Device driver modelling examples.

	Examples of items in figure 1	ProGram
a & e	Declaration of device driver component entry	%start read() %start int_handler(intSignal)
b	Declaration of in parameter from SW (16-bit) and declaration of in/out array to/from SW (8-bit, size 4)	%input data [bit]16 sw %inout inData [bit]8 sw memory 4
c	Describing functionality (wait on event and assignment)	readData(RTR):00 wait |others {data=RDR;};
e	Waiting for an interrupt to occur	waitForInt: others waitForInt | interrupt(intSignal) int_handler;
f	Declaration of a write only device register (8-bit) with relative address 2, and declaration of read/write device memory (8-bit, size 16)	%output FCR [bit]8 hw 2 %inout buff [bit]8 hw 0 memory 16

Start conditions and interrupt. A start condition for hardware is evaluated as soon as the signals within the start condition change (i.e. same as a sensitivity list in VHDL), but since the concept of software and hardware are different the start conditions for device drivers are handled differently. Instead of start conditions that are trigged on signal changes, a start condition is trigged when it is called (i.e. like a function call). This means that the condition fields for device driver rules are empty. Each start condition will be a component of the device driver, as described in figure 1. For interrupt handlers the start condition is modelled as a hardware condition, i.e. sensitive to a signal (interrupt signal).

The keyword `interrupt` is used to indicate that one branch of a production rule is entered only if an interrupt condition occurs. If an interrupt is activated the other branches are aborted, see table 2. For example, this construct can be used to model a wait for an external event.

3.2 Modelling of bus interfaces

Since specification of a bus interface is to describe the conversion of two protocols (see figure 2) and [10] defines ProGram for specification of protocols, only small additions to ProGram are necessary to enable bus interface modelling.

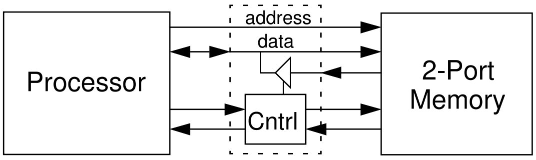


FIGURE 4. Interface between a single port processor and a two port memory.

Combinatorial. All reactions to ProGram grammar rules in bus interfaces are synchronized with a clock and thus the out ports will always be latched. In some cases this will lead to one extra clock cycle latency. As example consider an interface between a processor and a two port memory (figure 4). To model this interface, the address and data lines have only to be by passed through a buffer. To model this, we use a construct where combinatorial values can be assigned to signals (`%transparent`).

I/O. To describe the data port connected to the processor in figure 4, we need to declare an in/out port (`%inout`). For an in/out port the drivers for a signal has to be controlled, switching off the signal with help of assigning the constant `Z` to a signal as in VHDL.

Hierarchy. To model the whole interface in figure 4 with ProGram, we need constructs for declaring (`%component`) and instantiating components, i.e. hierarchy constructs. Components are instantiated as production rules.

3.3 Specification of ProGram

Interface declaration. The interface section starts with a component declaration which is the first part of the hierarchy construct (second is the instantiation), i.e.

```
%component 68toMem
```

The second part of the interface section declares ports for a component and the internal signals used for inter process communication within the component:

```
%input enable bit
%internal synch bit
%output dataOut [bit]32 sw
%inout RBR[bit]8 hw 0x00
```

An input declaration consists of the name, the width and for hardware it can also have a bit rate constraint for each signal. For device driver modelling an indication from where a signal originates: `sw`, is used for communication with software, `hw` indicate that it is a register in the device with a specified relative address. The interface section ends with start rules as in YACC. These start rules work as process declarations that define the signals used for the start condition. Hence, ProGram can specify concurrent processes in contrast to YACC:

```
%start transferStart(request)
%start p2s(enable)
```

Tokens and memories. A Token is a pattern of bits read from an input stream or written to an output stream and can be viewed as constant. Reading and writing tokens are the primary events. Memories are specified as:

```
%memory con_mem [[bit]8] con_stat
```

The size of the memory is given by the number in brackets as the number of address lines connected. Then the rule that specifies the memory field layout is given.

Unlatched assignments. Unlatched assignments (used to describe unlatched behaviour) are equivalent with concurrent assignment statements in VHDL, e.g.

```
%transparent outValue = (value AND enable)
```

Actions and grammar rules. Actions in the grammar specify assignment of values to signals. Expressions to compute these values may be put directly in the assignments or may be associated with some symbols in the action values section. The assignments can then simply refer to these symbols. The expressions allow concatenation and conditionals in addition to the usual arithmetic and logic operations. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols. A grammar rule consists of a grammar symbol that serves as a rule identifier and a list of alternatives. Each alternative is a sequence of non-terminal symbols, terminals and actions. Passing the new signal stream as a parameter to the subtree of productions does a redirection of the input stream. Actions are enclosed in curly brackets.

```
grammar_rule(sig):symbol_1 grammar_rule
           | ...
           | symbol_n {action_value_section;};
```

An example of a grammar rule with two alternatives selected with the two symbols 0 and 1 follows below. 0 transferStart, is a non-terminal symbol which recursively calls itself. The other alternative, first has an action value and is ended by a termination of the rule.

```
transferStart(rqst):0 transferStart
           | 1 {dataOut = dataIn;};
```

4 Synthesis techniques

4.1 Device driver synthesis

In the following we focus on the device driver synthesis to C code for a single threaded software architectures.

Port implementation. As mentioned in the section on ProGram, there exist two different types of ports for device drivers: ports to the device and ports to application software.

Ports to the device are translated to a pointer, with the pointer address set to the sum of the device base address and the port address (port address the product of the relative address form the port declaration and the byte width of the data bus).

Out ports to the application program, are translated to pointers to an object in the function parameter list, the pointer address is set by the caller of the device driver. In ports from the application program, are translated to objects in the function parameter list.

Handling of start conditions. As discussed in the ProGram section, start conditions without signals are treated as functions and with signals in the start condition are treated as interrupt handlers. Both are implemented as functions, but detailed implementation for specific software architectures of the interrupt handler is left for future work.

Code generation. The generation of C code is based on the algorithms for hardware synthesis presented in [10], i.e. a state machine is synthesized from the ProGram code. Where each start condition results in a state machine clocked with a clock signal specified in the start condition. When the state machine is in idle state, it waits for the start condition to become true. This does not efficiently translate to software execution, where the start condition is translated into a function and the state machine is not clocked, i.e. the time between to state transitions are not fixed.

```
mergeDelayStates(start_symbol)
begin
    for i=1 to Number_of_States loop
        if(Number_of_Successors(state[i])=1 AND
                    Number_of_Predeccessors(Successor(state[i])) = 1)
            mergeStates(state[i],Successor(state[i]));
    end loop;
end method;
```

FIGURE 5. The merge delay states algorithm.

A synthesis procedure optimized for software is derived from the synthesis procedure presented in [10]. To optimize this for software we have to remove output synchronization states (delay states), these are not necessary in software since there is not a fixed time delay between state transitions. The delay state are merged into super states where the code for a sequence of delay states are merged into one state, but making sure the execution order for the code is kept. The algorithm for merging of delay states is shown in figure 5.

Exit transitions (correspond to completion of grammar rules) in the state machine are extracted to generate *return-from-function*. The state machine in C is built with goto statements since these are executed fast on standard processors.

4.2 Bus interface synthesis

In [10] it has been shown, that ProGram can effectively be used for HW interface synthesis. In correspondence to our extensions to the ProGram language, we handle transparent signal assignment and hierarchy with a direct translation to the corresponding VHDL constructs, i.e. concurrent signal assignment and VHDL components since these constructs are well supported in logic synthesis tools.

5 Results

Methods for software generation from ProGram descriptions modelling device drivers have been presented. These methods are specific for single threaded software. A comparison of ProGram and C for different devices is presented in table 3, where it is shown that code generated from ProGram is comparable to hand written C code, with respect to size and speed. The software generation has to be further investigated in the future, to find how different software architectures like for instance real-time kernels will affect the code generation and how this should be modelled.

TABLE 3. Comparison of device drivers specified with ProGram and C.

Device driver	Lines of code[a]	Code size[b]	Code efficiency[c]
MAX530, DAC	6/11	1.33	0.90
6850, ACIA	17/17	1.33	1.54
TL16C552, DUART	121/222	1.13	0.99
MAX197, ADC	6/11	0.86	0.96
YM 3623	9/12	1.50	1.42

a. ProGram/C
b. Lines of assembler code for HP (hppa) processor, generated ÷ hand written.
c. exec_time(Generated code) ÷ exec_time(Hand written C).

Optimization techniques for bus interfaces are also discussed. In table 4 bus interfaces described in VHDL and in ProGram are compared with respect to lines necessary to describe the functionality and the synthesis result. This comparison shows that the result from synthesis is comparable to a hand written VHDL code even though the description is about one third of the VHDL description.

TABLE 4. Comparison of ProGram and VHDL.

Interface	Lines of Code[a]	Gate count[b]	Critical path[b]
MC68040 <->RAM	3/21	38/32	3/2
MC68040 <-> 4-ph	19/46	54/39	5/5
4-phase <-> 2-ph	14/32	72/54	5/5
2-phase <-> 4-ph	9/32	56/31	4/3
MC68040 <-> YM3623	21/70	1081/710	13/11

a. ProGram/VHDL
b. Generated VHDL code/hand written VHDL code for LSI 10k

6 Conclusion

We have shown how a grammar based communication protocol description language, ProGram, can be used for description of both bus interfaces and device drivers. We have compared ProGram to different bus interface modelling techniques presented by other research groups, and shown that ProGram results in compact descriptions, see table 1, and effective implementations (see table 3 and 4).

In addition, our approach addresses the maintenance problem of device drivers by significantly reducing the number of entities that must be maintained. Assume a company must support device drivers for 4 different operating systems on 8 variants of embedded processors and each device driver can be implemented in 3 different ways depending on performance and cost requirements. Furthermore we assume 4 protocols which differ in the detailed control and communication flow. Currently the company has to develop $4 \cdot 8 \cdot 3 \cdot 4 = 384$ device drivers and it has to develop $8 \cdot 3 \cdot 4 = 96$ new device drivers when an operating system is added. In our approach we separate the description of micro processor, operating system and protocol from each other and cover different implementation alternatives by the code generation tool based on user directives. Thus, we have only 4 descriptions for the operating system, 8 for the micro processor and 4 protocol descriptions in ProGram. Consequently, we have only $4+8+4 = 16$ independent entities to maintain each of which is significantly smaller than a fully coded device driver. Moreover, some of these entities can be used also in the hardware design of the system, further increasing the development and maintenance efficiency.

7 References

[1] P. Chou, R. Ortega, G. Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, In Proc. of the IEEE/ACM ICCAD, Nov. 1992, pp. 488-495.
[2] P. Chou, E. Walkup, G. Borriello, “Scheduling Issues in the Co-Synthesis of Reactive Real-TIme Systems, IEEE Micro, Aug. 1994, pp 37-47.
[3] W. Ecker, et al., “Protocol Merging: A VHDL-Based Method for Clock Cyckle Minimizing and Protocol Preserving Scheduling of IO-Operations”, In Proc. of Euro-DAC, 1994, pp. 624-629.
[4] D. D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, ISBN 0-13-150731-1, 1994.
[5] R. Grehan ,“Driver assistance”, Computer design, Nov. 01 1997, vol. 36, no. 1, pp. 75,76,78,80.
[6] S.C. Johnson, “YACC, Yet another compiler compiler”, Computing Science Tech. Rep. 32, AT&T Bell Lab., Murray Hill, 1975.
[7] B. Lin, S. Vercauteren, “Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation”, In Proc. of ICCAD, 1994, pp. 101-108.
[8] P. Moeschler, et al., “High-Level Modeling using Extended Timing Diagrams, A formalism for the behavioral specification of digital hardware”, In Proc. of the Euro-DAC, 1992, pp. 494-499.
[9] Tuggle E., “Writing Device Drivers”, Embedded Systems Programming, Jan. 1993, pp. 42-65.
[10] J. Öberg, A. Kumar, A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, In Proc. of the 9th ISSS, Nov. 1996, pp. 14-19.

3

Paper 3

Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification

Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification

Mattias O'Nils
Royal Institute of Technology
Department of Electronics, ESD Lab.
Electrum 229, S-164 40 Kista, Sweden
mattias@ele.kth.se

Axel Jantsch
Royal Institute of Technology
Department of Electronics, ESD Lab.
Electrum 229, S-164 40 Kista, Sweden
axel@ele.kth.se

Abstract

We present a method for generation of the software part of a HW/SW interface (i.e. the device drivers), which separates the behaviour of the interface from the architecture dependent parts. We do this by modelling the behaviour in ProGram (a grammar based protocol specification language) and capture the processor and OS kernel parts in separate libraries. By separating the behaviour from the architectural specific parts, compared to other approaches up to 50% development time can be saved the first time the component is used, and up to 98% for each time the interfaced component is reused.

1. Introduction

As elaborated by Tuggle in [10], for a given device, a device driver must be rewritten for every operating system it will be used with. For hierarchical device drivers like SCSI systems, the number of device drivers needed for a device is the product of the number of supported operating systems and the number of supported host adapters. If we also consider different implementation styles of a device driver, the number of possible implementations grows even more sharply. One quickly realizes, that development, verification, and maintenance of all these device drivers is a huge task.

Existing approaches address this problem only partially. CoWare [11] and Polis [1] concentrate on the case where the whole design functionality is captured within their environment and then during the system synthesis the communication is refined, i.e. the device drivers are generated together with the custom HW and operating system. But if the user wants to use IP blocks and an off-the-shelf RTOS, he/she will face the same troubles as for manual design [10]. In Chinook [2], the definition of a device driver also includes the bus interface, which makes it very architecture dependent. MakeApp [6] is a tool for generating device drivers for different devices and processors matching user defined configurations. Both Chinook and MakeApp solve only part of the problems described in [10], since they generate code only for a specific real-time kernel (Chinook) or no kernel (MakeApp).

To solve this problem we have proposed techniques for modelling of device drivers independent of both the real-time kernel and the processor [9]. In this paper we extend that with a synthesis technique to generate device drivers from three inputs: (1) an architecture independent protocl described in ProGram [12]; (2) a characterization of the operating system; (3) a characterization of the processor and its bus interface. With architecture we mean both the processor and the operating system.

The next section presents the proposed design flow for device drivers. Then we discuss the architecture dependent parts of a device driver. The fourth section presents a technique for modelling software architectures for device driver generation. In the fifth section we present the synthesis method. Finally, we conclude the paper with two case studies.

2. Design flow

As described in [9], we model device drivers independently of the architecture using ProGram [12]. ProGram, is a grammar based notation for protocol applications. Specifications in ProGram deal with sequences of allowed events as opposed to states and state transitions in FSM model.

The ProGram description is synthesised into an untimed extended state machine (described in [9,12]), which in turn is the input to the architecture mapping procedure described in section 5. The mapping procedure uses data from two libraries to generate the architecture specific code. The first of these libraries is the library that captures the information on the OS architecture, the second captures the processor specific characteristics, see figure 1.

3. Architecture dependent parts of a device driver

In the following sections we discuss architecture specific parts of device drivers, i.e. synchronization, execution delays, interrupt handling, mutual exclusion, and the device driver interface.

3.1. Synchronization with external events

The waiting for an external event, i.e. the synchronization of the device driver with an external event, can be implemented in three ways, as illustrated in figure 2: (a) polling of device signals, (b) wait for a fixed time (when

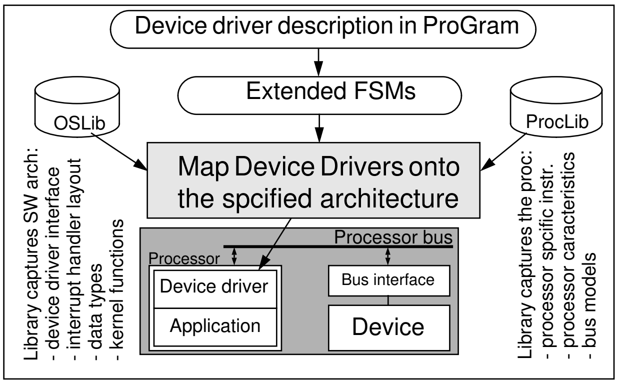


Figure 1.Device driver design flow.

the device is known to generate an event), or (c) wait for an operating system event generated by an interrupt that is triggered by the device. An example of where synchronization is needed is an analog to digital converter. There we first initiate a conversion by writing some control data, and then wait for the conversion to take place, i.e. synchronize with the conversion ready event. This is done by polling a status register, wait for the specified conversion time, or let the device generate an interrupt signal. After that, read the sampled value from the device. Polling is the only one of these three synchronization schemes that can be implemented independent of the software and OS architecture.

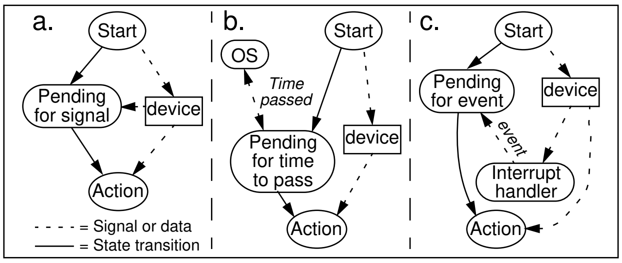


Figure 2.Synchronizing with external device.

3.2. Delay execution

As described in the previous section, in some situations a device driver has to halt itself for a certain amount of time. This could be necessary when waiting for an analog-digital conversion, but also to wait for coprocessors during the initialization process. The wait function is provided by some kernels, hence this function is architecture dependent.

3.3. Interrupt handling

Interrupt service routines can be used to synchronize a device driver with a hardware event as described above. Sometimes they are also independent routines with their own behaviour, e.g. when an interrupt handler receives data from a communication device and writes it to a buffer. There are two possible ways to implement this: (a) the interrupt service routine informs the task about the event when the interrupt is activated and the task handles the action. (b) The interrupt routine also performs the action itself, see figure 3.

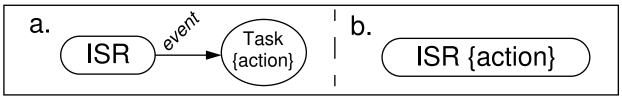


Figure 3.Implementation of an interrupt routine.

The implementation in figure 3b is more suited for actions with short execution time since alternative 3a introduces a communication overhead. Alternative 3b has also a shorter and deterministic response time but introduce indeterminism in the kernel behaviour. 3a's response time is dependent on the priority of the task responsible for the action.

3.4. Mutual exclusion

There are two possible ways to prevent several application threads accessing the same device simultaneously: (1) with help of semaphores and (2) to disable the interrupt mechanism. Both have there advantages and disadvantages. Disabling interrupts is fast and do not add overhead to the execution but it disables the normal behaviour of the kernel, i.e. introduce indeterminism. Semaphores do not affect the kernel behaviour but introduce communication overhead. So the conclusion is that one should use semaphores for complex device drivers and disable interrupt for simple ones.

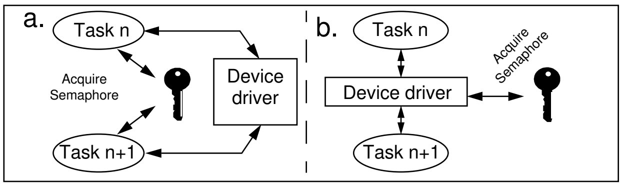


Figure 4.Handling mutal exclusion.

The control for mutual exclusion can be placed within or outside the device driver, see figure 4 for an example with a semaphore for handling mutual exclusion. The benefit of having it outside (i.e. within the application code) the device driver is that the device driver will not be dependent on the architecture for handling of mutual exclusion, the draw back is that the code will be less structured.

3.5. Device driver interface

For device drivers developed for single threaded software with no operating system, the interface to the application software (i.e. how in and out parameters to the device driver are handled, driver component naming style) is determined by designer, project or company coding style. There are systems with real-time kernels like the µC/OS or RT-kernel where the interface to the application is also defined by the designer. For other systems with kernels/ operating systems like UNIX, OS9, VxWorks and OS/2 the interface to the application/operating system is defined by the operating system [10].

4. Modelling of the architecture

To be able to map architecture dependent parts of a device driver to a specific architecture, the characteristics and behaviour of the architecture have to be captured. As seen in figure 1, the architecture is captured in two parts: the kernel specific parts (further described in section 4.1) and the processor specific parts (further described in section 4.2). Throughout this section, we will use the µC/OS [7] and MC68000 [8] as examples.

4.1. Software environment

The model of the software environment (i.e. the OS characteristics and functionality) is divided into three parts: (1) the environment characteristics, (2) the interface to the application/OS, and (3) the macro for accessing certain system functionalities.

4.1.1 SW environment characteristics

This part contains the environment name, type, timing, and include files needed for this architecture. The type is captured since it affects the behaviour of the interrupt routines. The characteristics for the model is defined in the table below.

Characteristics	Value(s)
name	Name of the architecture (string)
kernel_type	{preemptive, not-preemptive, single thread}
tick_time_period	Time between two clock ticks
interrupt_supported	{true,false}
header_files	{empty, list of header files}

Example 1 *A system with µC/OS kernel and a timing interrupt period of 10 ms, would have this characteristics model:*

```
name = uCOS;
kernel_type = preemptive;
tick_time_period = 10 ms;
interrupt_supported = true;
header_files = "uCOS.h";
```

4.1.2 SW environment interface

With SW environment interface we mean the naming style of the different device driver functions and how data is transferred to and from these functions. The interface model is composed of three interface definitions for generation of function declarations: (1) interrupt service routine, (2) device driver function, and (3) task function. The definitions describe how the function names and parameter declarations should be generated. There is an additional translation table for type conversion, i.e. that translates the bit vector type to a C data type with respect to the bit width.

Definition/Table	Outcome
interrupt_routine_def	interrupt routine declaration
driver_routine_def	driver routine declaration
task_def	task declaration
type_coversion_table	data typed signals

4.1.3 SW environment services

SW environment services are equal to system call functions offered by the kernel. Services that have to be modelled are discussed in section 3 and the system functions to provide these services are enumerated in the table below.

System call	Input
enable_interrupts	non
disable_interrupts	non
create_binary_semaphore	name (string)
delete_binary_semaphore	name (string)
pending_semaphore	non
set_semaphore	non
pending_time	no of ticks (integer)
enter_interrupt	non (*For preemptive kernels*)
exit_interrupt	non (*For preemptive kernels*)

Example 2 *A model of some of the uC/OS service functions:*

```
pending_time(n) = OSTimeDly(n);
disable_interrupts = OS_ENTER_CRITICAL();
enable_interrupts = OS_EXIT_CRITICAL();
```

4.2. Processor features

The processor features are captured by two parts: the processor characteristics and processor specific routines.

4.2.1 Processor characteristics

A processor can be characterized in many ways, but what is interesting for this approach is the internal and external data bus width since these will affect the address calculation of the device address. Also, the device access type has to be captured, i.e. if it uses a port mapped or memory mapped device access method. The table below defines the characteristics that need to be captured.

Characteristic	Value
name	Name of the architecture (string)
device_access_type	{memory mapped, port mapped}
internal_data__bus_width	{8,16,32}
external_data_bus_width	{8,16,32}

Example 3 *Model of characteristics for the MC68000:*

```
name = MC68000;
device_access_type = memory_mapped;
internal_data_bus_width = 32;
external_data_bus_width = 16;
```

4.2.2 Processor specific routines

For device driver generation some processor specific code must be provided. Specifically, the saving and restoring of registers must be captured. If enabling and disabling of interrupts is not supported by the kernel, then the processor specific code has to be used. The table below defines the processor specific code needed to be captured.

Assembler macro	Parameter
push_proc_register	non
pop_proc_register	non
enable_interrupts	non
disable_interrupts	non
return_from_interrupt	non
function_call	function label
read_port_mapped	data (*output*)
write_port_mapped	data (*input*)

Example 4 *Push the register onto the stack(MC68000):*

```
push_proc_register="move.l D0-D7/A0-A7,-(ssp)"
```

5. Mapping of device driver models

From the ProGram description of the device driver several untimed FSMs are synthesized with the same technique used for generating HW interfaces [12], have been shown to produce very good results. From there the implementation is created in two steps: (1) the transformation rules (section 5.1) perform optimizations and transformations which are independent of SW architecture and processor; (2) the code generation (section Figure 5.2) uses the libraries *OSLib* and *ProcLib* to generate the device driver in C.

5.1. Transformation rules

Each transformation rule described in this section is applied to all states in the synthesised state machines. These rules map the FSM behaviour onto a general software architecture.

5.1.1 External synchronization

External synchronization points in ProGram are translated to wait statements in the generated FSM, with a transition upon the receipt of the synchronization signal, i.e. the t_j transition in the figure 5 (a) is taken. The condition for t_i is complementary to the one for t_j, i.e. t_i fires when t_j does not and vice versa. Synchronization with external signals are implemented by means of an interrupt routine signalling an event to the waiting device driver.

Rule 1: State S_i has two transitions (t_i and t_j), t_i is a transition to S_i and t_j to S_j (figure 5, a). If the conditions for t_i and t_j are complementary and composed of external signals, then remove t_i, set the condition for t_j to true and insert a wait-for-event from the library into the code of S_i. Generate an interrupt routine, that sends this event upon activation (figure 5, b). The condition for t_j will be the interrupt signal that triggers the interrupt routine.

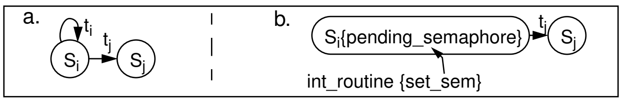


Figure 5.State transformation.

5.1.2 Identify internal synchronization

Internal synchronization points (i.e. synchronization between different parts of the device driver) are translated as the polling of an internal signal. As stated in [9] this construct can not be allowed between the different access functions of the device driver, since it could result in a deadlock. The only situation, where it will work and can be very useful, is, when the signal is driven by an explicitly declared interrupt routine.

Rule 2: State S_i has two transitions (t_i and t_j), t_i is the transition to S_i and t_j to S_j (figure 5a). If the conditions for t_i and t_j are complementary and composed of an internal signal (5a), then remove t_i and set the condition for t_j to true and replace the assignments of the signal by a set and clear event from the library (figure 5b).

5.1.3 Partitioning of interrupt routines

As mentioned in section 3.3, an interrupt routine can either be implemented as an ISR with behaviour (figure 3b) or as an ISR sending an event to a task (figure 3a). Here we let the designer to interactively select this, in the future this could be determined by some heuristic.

Rule 3: If state S_i is an entry state to an interrupt routine and the user marked the routine for partitioning then replace the state "entry" by a state "task entry" and insert a wait-for-event from the library into the code of S_i and generate an interrupt routine that sends this event upon activation.

5.2. Code generation

The first action in the code generation is to translate bit vector types to C data types, by using the translation table from OSLib. The code generation for the different parts of a device driver follows certain assumptions about the structure of each component.

5.2.1 Device driver function

First the head of the function declaration is generated, i.e. function name and parameters, by invoking the driver function definition. The body of the function consists of routines for mutual exclusion from the OSLib and the C code for the FSM. The FSM code contains OS and processor specific code, based on entries in OSLib and ProcLib, respectively.

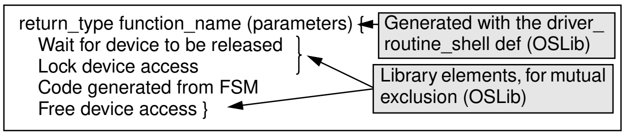


Figure 6.Code structure for an access function

5.2.2 Interrupt routine

Entry and exit of an interrupt routine must be in assembler language, since there is no support in C for interrupt handling. In the code generation of interrupt routines, the routine starts with storing and ends with restoring the processor registers. For a preemptive kernel the interrupt should also notify the kernel when it enters and leaves the interrupt routine, see figure 7.

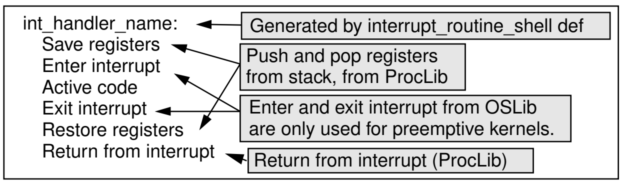


Figure 7.Code structure for an interrupt routine.

The code part of the interrupt routine is either code generated from an FSM that corresponds to the interrupt routine or just a library function sending an event. Either way, it will be a call to a C function.

5.2.3 Task

A task is generated only when an ISR is partitioned. In this context a task is only another implementation of an interrupt routine. Thus, its structure is very simple and basically an infinite loop containing a wait-for-event statement and the code for the corresponding FSM.

5.2.4 Miscellaneous

The enabling and disabling of interrupts can be captured in both libraries, OSLib and ProcLib, as described in section 4. If it exists in OSLib, the code generation uses it, otherwise the ProcLib is consulted.

Wait statements in the FSMs are replaced with the library macro from OSLib. But the real-time has to be translated into clock ticks, i.e. clock_ticks = time / clock_tick_period.

At the end of the code generation, a setup function is generated. This function creates and initializes the semaphores used by the other components. It also calculates the absolute addresses for the different device registers by using the base address and the declared relative addresses.

Example 5 *A device driver for the MAX197. We consider only the read value function, which is modelled in ProGram in figure 8a. This is synthesized into the state machine in figure 8b. After the transformation rules are applied we get the FSM in figure 8c and according to the optimization methods described in [9] we remove the redundant delay states and get the FSM in 8d. The coarse structure of the generated C code for the read function mapped onto µC/OS and MC68000 is illustrated in figure 8e.*

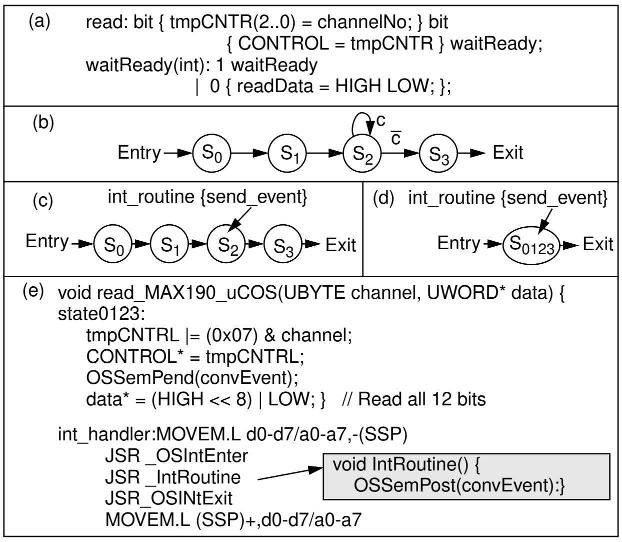


Figure 8.The read function of MAX197 from ProGram specification to C implementation.

6. Case study

We use two designs for the case study, a channel decoder of a transceiver in a D-AMPS base station and an operation and maintenance block (OAM) of an ATM network.

6.1. D-AMPS

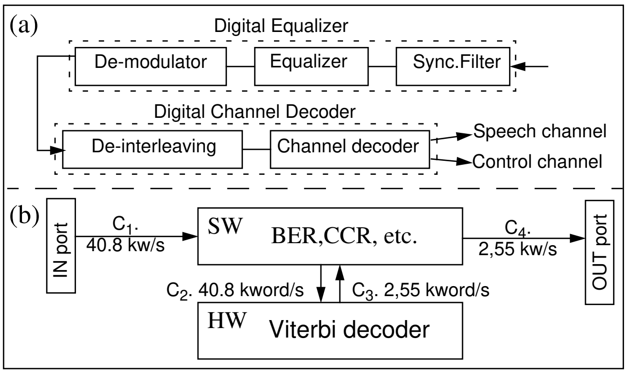


Figure 9.Receiver in IS-54B base station (a), and selected HW/SW partitioning (b).

D-AMPS [4] is a time division multiple access (TDMA) cellular standard. The standard has 832 channels per carrier in the 800 MHz range and a total of 1164 channels. The channel decoder block receives 272 bit frames that should be processed in 6.67ms. The process consists of: de-interleaving, Viterbi decoding, CRC-sum calculation, bit error rate estimation, sorting of speech data, and masking of bad speech frames.

We decided to implement the Viterbi decoder in hardware and the rest of the functionality in software. Figure 9b illustrates the hardware/software communication in the channel decoder design.

6.2. ATM Operation and Maintenance Block

The Operation And Maintenance (OAM) functionality

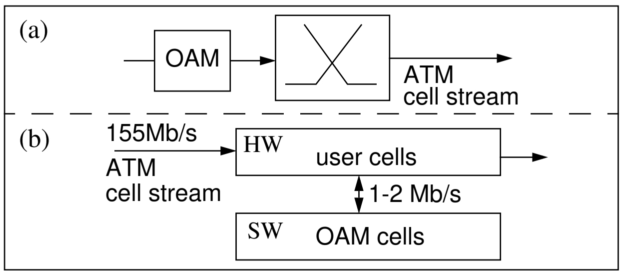


Figure 10.OAM block in an ATM network (a) and selected HW/SW partitioning (b).

[3] in an ATM network is situated near a switch. Specific ATM cells, OAM cells, are used to identify network problems, measure network performance, and communicate the state of the network between network nodes. OAM functions analyse, transform, and generate OAM cells. Figure 10b shows the selected HW/SW distribution, where ATM user cells are received and re-transmitted in custom hardware, while OAM cells are analysed and transformed in software. OAM cells typically constitute about 1% of the total traffic.

6.3. Results

We compare our approach to code development in C, since this is what a designer has to do if she/he wants to us an IP component and an RTOS in Coware [11], Polis [1] and MakeApp [6]. The comparison shows similar figures

Approach	C_1	$C_2 + C_3$	C_4	OAM HW/SW channel
C [LoC]	56	105	62	81
ProGram [LoC]	27	58	25	43
Difference [%]	48%	55%	40%	53%
Performance[a]	1.01	1.05	1.02	1.05

Table 1. D-AMPS channel decoder and ATM-OAM modelled in C and ProGram.

a. t_{exec}(C generated from ProGram) / t_{exec}(C)

for performance and code size, but a significant advantage of our approach in terms of designer productivity.

Table 1 shows the number of lines of C and ProGram code to describe the different protocols for the two designs. This shows that the amount of code in ProGram is typically 50% of the C code, because ProGram code does not contain any architecture dependent parts. The table also shows that the performance figures are comparable.

However, the true benefit comes from the reusability of a protocol specification in ProGram. If the same protocol should be implemented and maintained in several product versions with different RTOS and processors, the ProGram model can be fully reused, while the C code in any of the other approaches cannot. Figure 11 a and b give the amount of code to be written to implement a single protocol on a number of processor-RTOS combinations. On the X axes the processors and kernels are changed for even and odd generations, respectively. Figures 11 c and d translate these numbers into design time assuming a productivity of 16 LoC (Lines of Code) per day for ordinary systems and 4 LoC per day for embedded application code, left and right y-axis respectively, using the COCOMO model [13] for ordinary software systems and for embedded systems.

7. Conclusion

By separating architecture dependent from architecture independent parts of a device driver, we can significantly increase the reuse potential of models describing protocols, OS interfaces and processor interfaces. This addresses a critical step in IP based design, the specification and implementation of interfaces in the context third party HW blocks and SW components, e.g. RTOS.

We have presented a device driver synthesis technique which generates device driver implementations from a protocol, an OS and a processor description. The method is based on several interactive design decisions, e.g. the designer can select the implementation of interrupt routines and mutual exclusion handling. As shown in [9] and table 1 the efficiency of the generated code is close to hand written C code even though the code is not optimized.

Our approach will result in 50% less design time a protocol is used and 98% less for every product variant with a new processor or operating system, compared to CoWare, Polis and MakeApp.

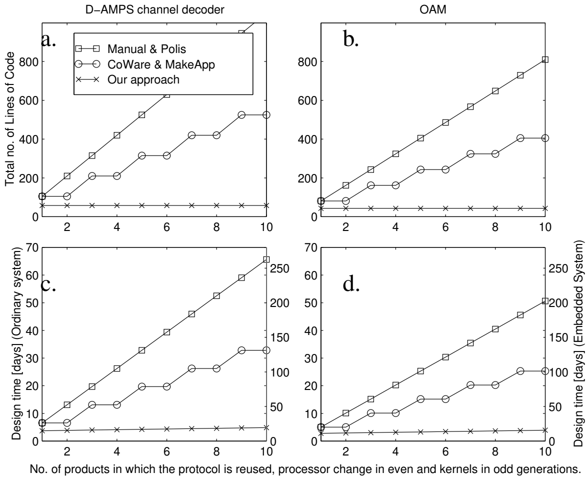


Figure 11.Lines of code and design time for channel decoder and OAM for reuse of protocol.

8. Reference

[1] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press , June 1997.

[2] P. Chou, R. Ortega, G. Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, In *Proc. of the IEEE/ACM ICCAD*, Nov. 1992, pp. 488-495.

[3] M. De Prycker, *Asynchronous Transfer Mode*, Prentice Hall, 1995.

[4] *EIA/TIA Interim Standard, IS-54-B*, April, 1992.

[5] S. Gal-Oz, A. Cohen, “The Hazards of Device Driver Design”, *Embedded Systems Programming*, May 1997, pp. 34-46.

[6] R. Grehan, “Driver assistance”, *Computer design*, Nov. 01 1997, vol. 36, no. 1, pp. 75,76,78,80.

[7] J.J. Labrosse, *μC/OS, The Real-Time Kernel*, R&D Publications, ISBN 0-87930-444-8.

[8] *MC68000 - Reference Manual*, Motoral Inc.

[9] M. O'Nils, J. Öberg, A. Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", In *Proc. of Euromicro Workshop*, Aug. 1998.

[10] E. Tuggle, “Writing Device Drivers”, *Embedded Systems Programming*, Jan. 1993, pp. 42-65.

[11] S. Vercauuteren, B. Lin, “Hardware/Software Communication and System Integration for Embedded Architectures”, in *Design Automation for Embedded Systems*, Vol 2, No. 3/4, May 1997.

[12] J. Öberg, A. Kumar, A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, In *Proc. of the 9th ISSS*, Nov. 1996, pp. 14-19.

[13] Boehm B. et al., “Cost models for future software life cycle processes: COCOMO 2.0”, *Annals-of-Software-Engineering*, vol. 1, 1995, pp.57-94.

Paper 4

4

Synthesis of DMA Controllers from Architecture Independent Descriptions of HW/SW Communication Protocols

Synthesis of DMA Controllers from Architecture Independent Descriptions of HW/SW Communication Protocols

Mattias O'Nils, Axel Jantsch
Royal Institute of Technology,
Department of Electronics, Electronic System Design Lab.
Electrum 229, S-164 40 Kista, Sweden
Email: {mattias,axel}@ele.kth.se
Voice(recept): +46 8 752 14 00, Fax: +46 8 752 12 70

Abstract

Starting from an architecture and implementation independent specification of hardware/software communication protocols, we present a protocol synthesis method that generates a mixed hardware and software implementation. For the hardware part, the synthesis method will generate an application specific direct memory access (DMA) controller for each protocol specification. Software parts of the generated implementation are components for initialization, synchronization and communication with the DMA controller. The protocol specification, with the grammar-based language ProGram, is used to model the HW/SW communication protocol. Since this approach is based on a device driver synthesis system for software solutions, which adopts the generated device drivers to a selected processor and kernel, the generated hardware/software solution can also be adopted to any processor and OS kernel. This lets the designer explore the design space for the communication protocols by trading off between performance and cost.

1. Introduction

The design and implementation of communication structures in digital embedded hardware/software systems tend to cause many errors and force designers to spend a lot of time designing and debugging the communication modules. The increasing importance of IP based design, will make specification and implementation of interfaces a dominant design problem for many embedded systems. This is the motivation to research effective ways to model and refine communication within an embedded system and to transform these models into an implementation for a specific architecture.

Hardware/software communication, i.e. transferring data between hardware and software, is basically reading and writing hardware registers and memory locations from software. There are three different ways to implement hardware/software communication: in software within an access function (in the normal execution flow), in software within an interrupt routine or in hardware by means of a direct memory access (DMA) controller. Synchronization between hardware and software can be implemented either by polling signals or with interrupts.

As seen in figure 1, the design flow for hardware/software communication starts with a description of the communication protocol in natural language or some specification language. This description is refined, by interpreting and mapping the protocol behaviour to some of the three different classes of implementations described above.

In [9,10,11] we addressed the problems of description, refinement and simulation of hardware/software communication protocols. We have done this by describing the hardware/software communication protocols with an architecture and implementation independent description [9], and show that these descriptions can be mapped [9,10] to several architectures of types (a) and (b) in figure 1. In [11] we present efficient techniques for simulation of hardware/software communication that are guaranteed to be consistent with the implementations. To cover all implementation alternatives presented in figure 1 in our approach, we address the mapping of communication descriptions to DMA structures (figure 1c) in this paper. By doing this we address the whole problem of description, simulation and implementation of hardware/software communication protocols.

The rest of the paper is organized in the following way. The next section goes through the related work and compares this to our approach. Then we present the method for mapping architecture independent communication protocols to DMA structures, by describing the target architecture, necessary library entries and synthesis methods. Finally we conclude with results from real life examples.

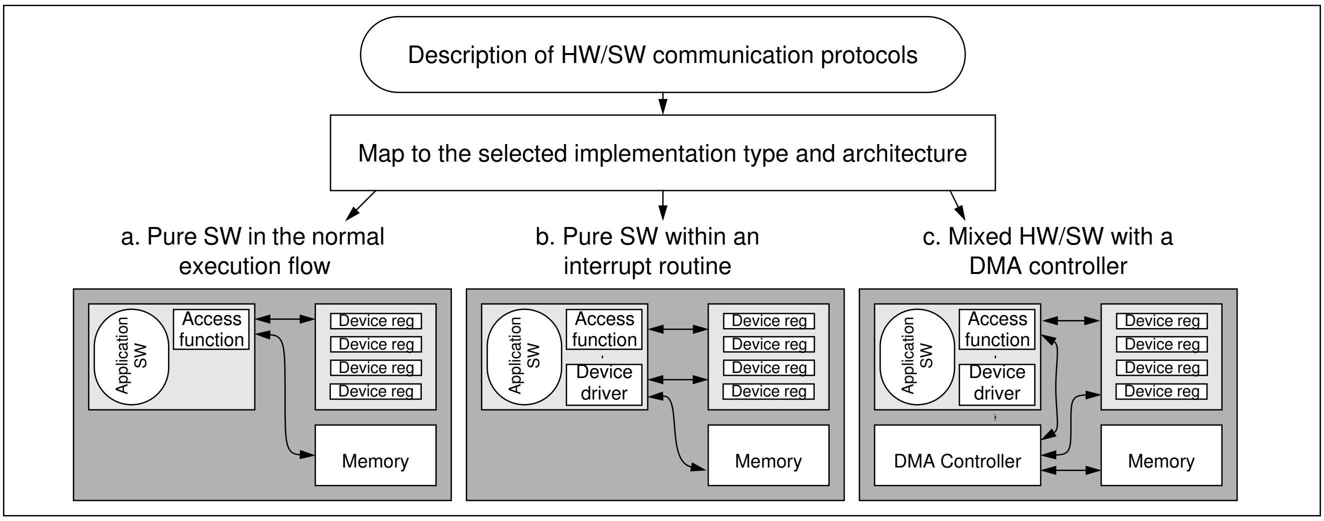


FIGURE 1. Implementation types of HW/SW communication.

2. Related work

Most research groups in the codesign field capture communication at the specification level with channels in some way [13,2,5], with exceptions like LYCOS [8], which does not support parallelism and communication. When these specifications are partitioned into hardware and software the communication channels between hardware and software have to be refined to an interface implementation. A hardware/software interface is composed of a software part (i.e. device driver), a bus protocol interface and the memory and register structure of the hardware. In addition, a direct memory access (DMA) controller can be used in the interface.

The main focus in literature on device drivers is on writing the implementation [12] or providing tools for library handling [6]. Chou et al. [1] models and generates software device drivers, but not in the context of an operating system as we do, but rather to emulate the bus interface in software. Eisenring et al. [4] present a system where the hardware/software protocols can be mapped to either software or software and general purpose DMA controller, but they only support one specific software architecture.

Bus-level protocols for components are usually described graphically by means of timing diagrams in a data sheet. For this reason, Chou et al. [1] have developed specification languages based on timing diagrams. A problem with these approaches is that it is not possible to specify high level behaviour. Lin et al. models the interface with extended STG (Signal Transition Graph) [13], which is based on Petrinets. With this modelling technique they can model asynchrony and events. On the extended-STG model asynchronous synthesis is performed. This approach enables the modelling of more complex behaviours. Ecker et al. [3] use hardware description languages to model bus interfaces. Öberg et al. present a communication protocol synthesis tool that specifies the design in a special purpose language ProGram [14], which is based on context free grammar. The synthesizable subset is limited to regular grammar with attributed conditions.

For the hardware part of the interface behaviour, there are approaches like from Daveau et al. [2] that address the generation of a communication components as allocation problem where they try to allocate communication resources by merging channels together. Freund et al. [5] present an approach where the optimizations of the device memory structure is integrated into the partitioning task.

Our approach addresses the modelling and synthesis of bus protocol interfaces [14,9] and modelling [9], simulation [11] and synthesis [10] of hardware/software communication protocols. The protocols are modelled as architecture and implementation independent device drivers using ProGram. ProGram, inspired by YACC [7], is a grammar-based notation for protocol applications. Specifications in ProGram deal with sequences of allowed events instead of states and state transitions, as it would be in an FSM model. Since we have an architecture independent model of the protocol we can generate implementation code for any software architecture or code for a mixed architecture with software and DMA controllers. This is not achieved by any other approach.

The ProGram description is synthesized into an extended state machine (described in [9,14]). This extended FSM could also be entered manually or generated from some other description language. The extended FSMs are fed to the architecture mapping procedure described in [10]. The procedure for implementation uses data from two libraries to generate the architecture specific code. The first of these libraries is the library that captures the information on the OS

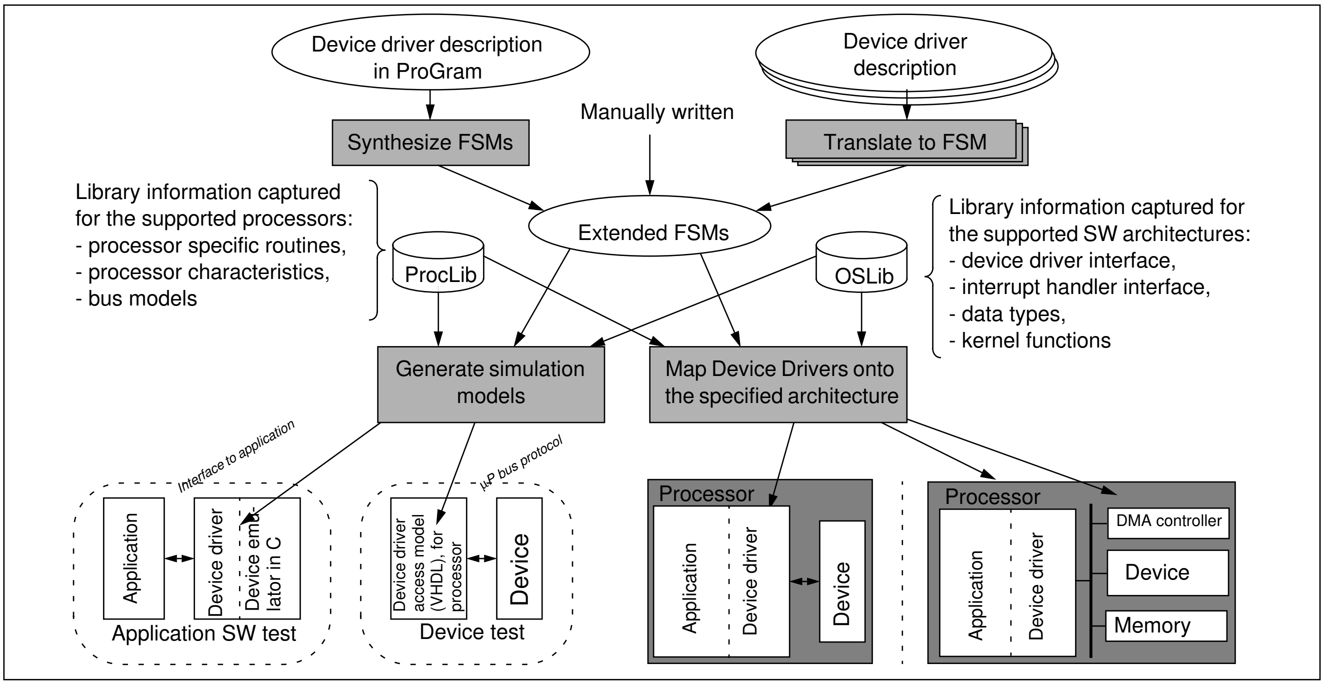


FIGURE 2. Device driver synthesis system overview.

architecture, the second captures the processor specific characteristics. In this paper the processor specific library is extended and is used in the generation of DMA controllers, which is further described in the next section.

3. DMA controller synthesis

General-purpose DMA controllers are complex devices with multiple behaviour and several different possible configurations. Communication by means of a DMA controller is used when a block of data is moved from one place to another, often from a device to memory, or vice versa. The data transfer is synchronized with an event generated by the device (an interrupt signal). The alternative to a DMA controller is an interrupt routine that moves the data to the memory from the device, or vice versa. Knowing that a DMA controller is basically an interrupt routine implemented in hardware, we have extended our device driver synthesis approach to also include synthesis of application specific DMA controllers from interrupt routine descriptions. I.e. our DMA controller is in fact an interrupt routine implemented in hardware. Thus, our DMA controller can have multiple behaviours: handle several types of data accesses, service different types of interrupt and even perform data processing.

3.1. Target architecture

As mentioned above, interrupt routine parts of a device driver can be implemented as either an ordinary software interrupt routine (figure 3a) or as an application specific DMA controller (figure 3b). The DMA controller consists of four parts: generated controller part, register file, master inter-

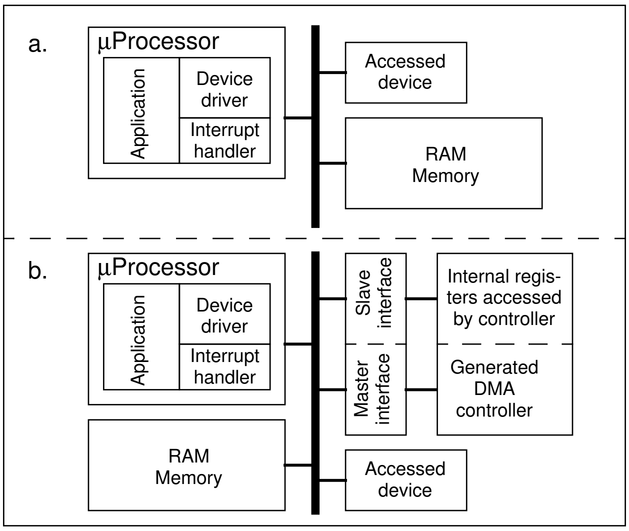


FIGURE 3. Target architecture for interrupt routines. (a) Implemented in SW as interrupt handler [9,10]. (b) Implemented in HW as a DMA controller.

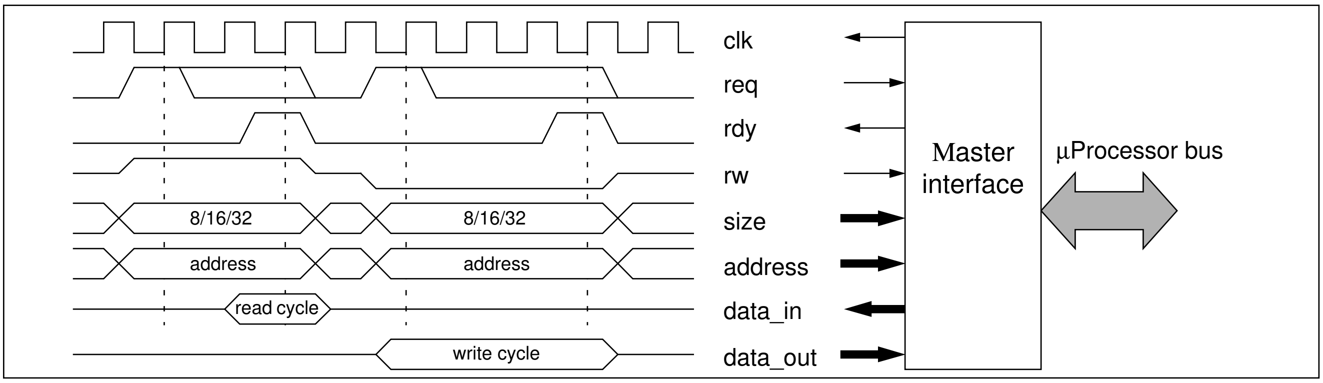


FIGURE 4. Master interface component and its access protocol.

face and a slave interface. The generated controller part (described in section 3.5) accesses the memory via the master interface (described in section 3.2) with help of the variables and pointers stored in the register file (described in section 3.3). The parts of the device driver still implemented in software read and write the register files through the slave interface.

3.2. Library elements

The processor library (ProcLib in figure 2) captures the characteristics of a processor. But as described in [10], the library has to be complemented with one entry to be used to generate DMA controllers. To this end a new entry in the ProcLib library describes the generation of a VHDL entity, in the following called the *master interface*. This is an interface between a fixed protocol (defined in figure 4) and the selected processor's bus protocol. The port with fixed protocol on the interface is the one connected to the generated part of the DMA controller. A data access is initiated from the DMA controller who activates the master interface, which then obtains control over the processor bus and performs the data access.

On one hand it implements the processor specific bus interface, and on the other hand it implements the master interface protocol to communicate with the generated DMA controller, see figure 3. The master interface communicates with the generated parts with a fixed protocol, which is defined in figure 4. Hence, our approach is independent of bus organization and arbitration controller, since arbitration handshake and bus specific behaviour is implemented by the master interface, not by the generated parts of the DMA controller.

The master interface component in figure 4 uses a 2-phase request/acknowledge protocol. The protocol is synchronized with the bus clock used for the processor bus. A read/write cycle starts with the request (req) signal driven high, the rw, size and address signals are driven to the appropriate values. A cycle is terminated by detection of the ready (rdy) signal at the positive clock (clk) edge. Read data (data_in) is valid only at the positive clock edges and when the rdy signal is high. Write data (data_out), address, size and rw signals should be valid before the positive clock edge when req is high and until after the positive clock edge when rdy is high. The size signal indicates the size of the transfer, i.e. it is up to the master interface to implement the reading and writing data of different size.

3.3. Generating the register file

A register file for accessing the device registers and memory data is generated, the register file contains: device addresses (figure 5a), internal variables, and vector addresses (figure 5b). An 8-bit control register for monitoring the state of the DMA controller is also generated (figure 5c).

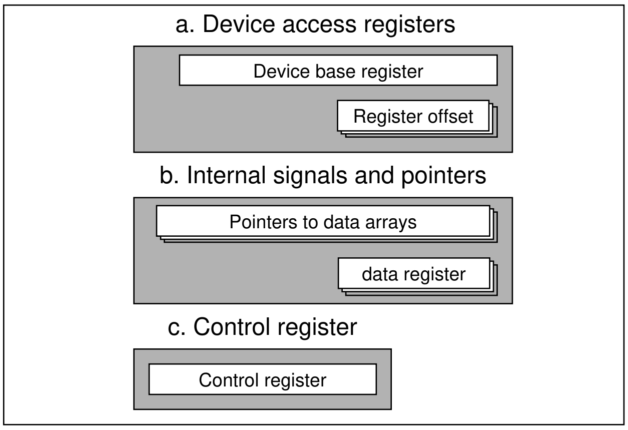


FIGURE 5. Register file for the DMA controller.

The device access registers are used to generate the device register addresses in the DMA controller during execution. There is one base register that stores the device base address and several offset registers (one for each device register). Address to a device register is the sum of the base reg-

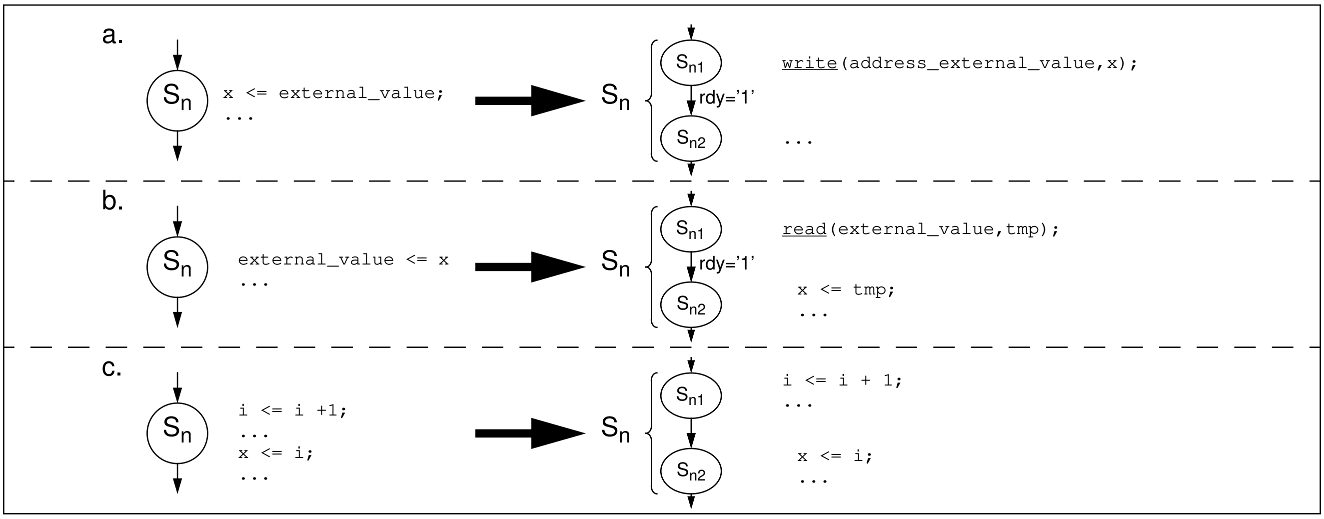


FIGURE 6. State transformation rules.

ister and the offset address for a specific device register. The base register has the same bit width as the processor address bus, and the register offset is set to $\lceil \log 2(\mathrm{size}(deviceRegFile)) \rceil$.

Internal signals are either internal variables or pointers to vectors, the internal variables have the same bit width as specified in the protocol description and the pointers have the same width as the processor address bus. Addresses to vector elements are generated in the same way as device register addresses, but as a sum of the vector pointer and the index variable.

The control register's least significant bit indicates if the base register address is written or not. If this address is not written, the controller can not service any interrupts from the device since the controller does not have the address to the device. This condition is put as an extra start condition together with the interrupt condition, to prevent the controller from service interrupts without the address to the device. The remaining seven bits are to indicate which synchronization signal generated an interrupt to the processor.

Base register, internal signals and pointers can be read or written from the software part of the device driver through an ordinary asynchronous memory interface (similar to a static RAM with read, write and chip_select signals). In this way the set-up functions can initialize the base address and the access functions can write and read back the internal variables and pointers.

3.4. Handling of synchronization points

In the synthesis of device drivers, sychronization points between access functions and device drivers are implemented with semaphores, described in [10]. This does not direct translate to the synthesis of DMA controllers since there is no way to send a semaphore from hardware directly to software. But the DMA controller can generate a synchronization event (interrupt) to the processor, this event is bound to a generated interrupt routine with the only behaviour to send a semaphore to the waiting access function. The generation of such an interrupt routine is described in [10].

3.5. Transformation rules

The DMA controller is generated from a finite state machine (FSM) describing an interrupt routine, the FSM is one of the generated FSMs (figure 2) that form a device driver. This state machine is optimized for device driver implementation in software [9], and has to be transformed for the implementation of a DMA controller. There are two types of transformations: (1) transform to memory access via master interface and (2) avoid intra-dependence for the assignments within one state.

a.

```
when state_n1 =>
   req <= '1'; rw <= '1'; size <= "00"
   address <= pointer + offset;
   if (rdy = '1') then
      req <= '0';
      next_state <= state_n2;
      tmp <= data_in;
end if;
```

b.

```
when state_n1 =>
   req <= '1'; rw <= '0'; size <= "00";
   address <= pointer + offset;
   data_out <= tmp;
   if (rdy = '1') then
      req <= '0';
      next_state <= state_n2;
   end if;
```

FIGURE 7. States for accessing external data.

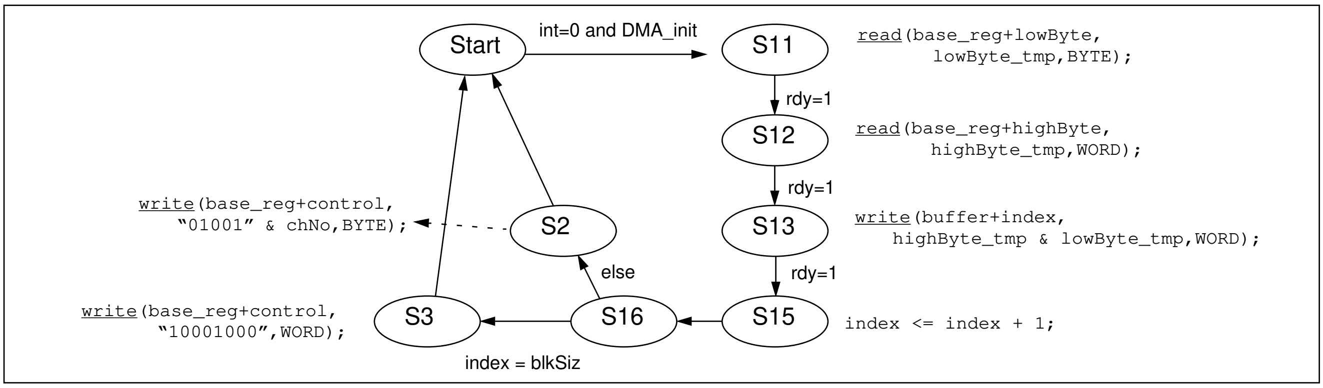


FIGURE 8. Transformed FSM.

First, in the state machine vectors and device register accesses are represented similar to the software implementation. In the DMA controller all memory and device access has to be done via the master interface, i.e. use the protocol specified in figure 4 to access the memory position. This is solved by inserting a state for reading (defined in figure 7a) the data from the memory/device or a state for writing (defined in figure 7b) for a write operation. These transformation rules are illustrated in figure 6 (a) and (b).

Second, code in the state machine for software implementation is sequential, i.e. there can exist a dependence between assignments and transition conditions, this is solved by moving the dependent parts to separate states. This transformation rule is illustrated in figure 6c.

When the state machines are transformed, the system generates the FSM in VHDL code optimized for RTL synthesis tools.

3.6. Case study

As an example we present the generation of both the DMA controller FSM and register file for the MAX197 analog to digital converter. The DMA controller is generated for Z80 system, 16 bit address bus and 8-bit data bus. Figure 9 shows the initial state machine, which is the input to the DMA controller synthesis.

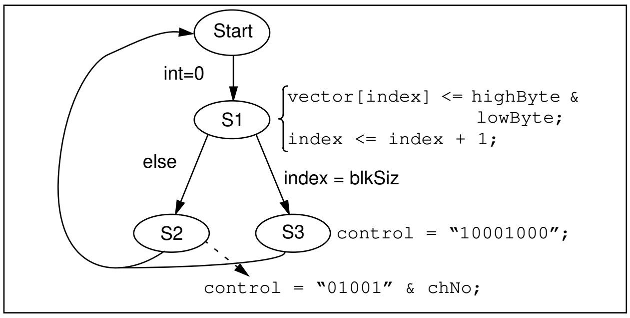


FIGURE 9. FSM description interrupt routine that reads 1 to 256 values from the MAX197 ADC and stores the values in a vector.

When the transformations in figure 6 have been applied to the state machine in figure 9 it will result in the state machine in figure 8. The `read` and `write` functions are used to increase readability, since these are really the read and write states described in figure in figure 7.

The register files for the generated FSM together with the slave interface is show in figure 10a. The only valid information in the control register is the least-significant bit indicating that the controller has been configured, i.e. the base address has been initialized (used in the start-state of the generated FSM). Figure 10b shows the offset constants for device register address calculation, these are 2-bit wide since $\log_2(3)$ rounded up to the nearest integer is 2.

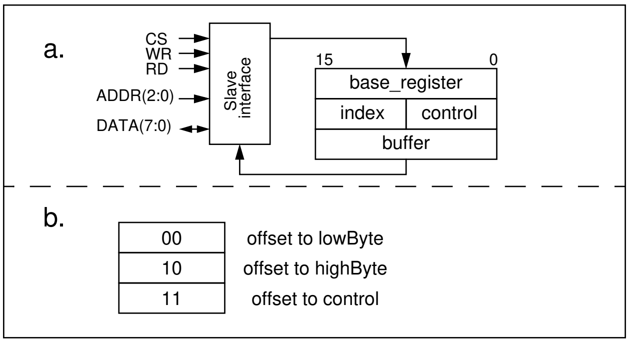


FIGURE 10. (a) Register file with slave interface. (b) Offset constants for calculation of the address to device register.

4. Results

A number of representative device driver examples have been used to evaluate the synthesis system [10]. Almost the same examples have been used to test and evaluate the generation methods presented in this paper. Table 1 shows the synthesis results for the generated DMA controllers. The

Device	Synthesized behaviour	Critical path	Controller gate count	Total gate count
MAX197	1. Read buffer (x values at channel ch)[a]	10 ns	1141	2100
	2. Read buffer (x values at channel ch)[b]	10 ns	1156	2120
	3. Read all channels and store in buffer[a]	10 ns	999	1970
	4. Read all channels and store in buffer[b]	8 ns	1018	1980
MAX530	5. Write x values from a buffer[a]	12 ns	1086	2150
	6. Write x values from a buffer[b]	12 ns	1101	2170
16552	7. Ring buffers (Rx buffer & Tx buffer)	14 ns	1997	2960

TABLE 1. Synthesis results for DMA controllers synthesised with LSI 10k connected to an Z80.

a. Asynchronous communication.
b. Synchronous communication.

first column is the name of the interfaced device; the second column describes the behaviour of the selected communication protocols. These protocol behaviours are composed of an access function and an interrupt routine. Tables 1 and 2 presents results regarding the interrupt routine part of the protocol, i.e. each DMA controller corresponds to an interrupt routine. The examples are all of the type read or write blocks of data, either of synchronous type (i.e. wait for data transfer to finish) or asynchronous type (no wait for the transfer to finish). In the columns three through five the present synthesis results from a RTL synthesis tool for the LSI 10k technology. The third column presents the critical path for the generated controller. Columns four and five present the gate count for the generated part DMA controller and the whole DMA controller (i.e. the master interface + generated controller), respectively

From these results we can see that the controller is quite fast (above 70 MHz) even for such an old technology as LSI 10k. The most likely application area for this is a system on a chip (SoC), but the DMA controller will work as good as a separate device on a board. Gate count for a system on a chip can be > 1M gates, i.e. with our approach can get up to 20 times increase in communication performance to a cost of less then 3‰ of the total system cost. This makes our solution an appropriate choice if communication performance is a bottleneck.

Table 2 compares the performance and bus usage between a pure software solution and the hardware solution with an application specific DMA controller. In the first column is the name of the interfaced device; the second column gives a reference to the behaviour description in table 1. In the columns four and six the performance (calculated as $performance = 1/t_{exec}$) numbers for two different configurations, software and DMA controller, respectively. Columns three and five present the numbers for system bus usage normalized to the all software solution, this is calculated as $usage_{HW} = mem_{access}(HW)/mem_{access}(SW)$. The sixth column shows the speed-up for a hardware solution compared to a software solution, calculated as $Speedup = t_{exec}(SW)/t_{exec}(HW)$. This illustrates that the hardware solution gives between 8 and 20 times higher performance. But in many cases more important is the reduced bus utilization by a factor between 10 to 20, because the processor bus is very often a scarce and highly utilized resource.

5. Conclusion

We have presented methods for synthesis of application specific DMA controllers from architecture independent device driver descriptions. By enabling this possibility, the designer can choose to implement parts off a hardware/software communication protocol either in software or in hardware based on requirements on cost and performance.

Both the software [9,10] and the hardware implementation as well as the simulation models [11] of the hardware/software communication protocol are guaranteed to be consistent with each other and the specification since they all originate from the same description. This is a clear benefit over a general purpose DMA controller, since in that case the designer has to refine the specification to an implementation by hand, with the risk that the specification and implementation will be inconsistent.

In table 2 we show that speed-up of the performance for an interrupt routine can be between 10 and 20 for the appli-

Device	Behaviour	Pure software		Generated DMA controller		Speedup
		Bus usage[a]	Performance[b]	Bus usage[a]	Performance[b]	
MAX197	1	100 %	$2.9{\cdot}10^4\ s^{-1}$	4.7 %	$5.6{\cdot}10^5\ s^{-1}$	19.4
	2	100 %	$2.9{\cdot}10^4\ s^{-1}$	4.9 %	$5.3{\cdot}10^5\ s^{-1}$	18.4
	3	100 %	$3.0{\cdot}10^4\ s^{-1}$	4.9 %	$5.6{\cdot}10^5\ s^{-1}$	18.7
	4	100 %	$2.9{\cdot}10^4\ s^{-1}$	8.5 %	$2.5{\cdot}10^5\ s^{-1}$	8.6
MAX530	5	100 %	$4.2{\cdot}10^4\ s^{-1}$	5.1 %	$6.7{\cdot}10^5\ s^{-1}$	16.0
	6	100 %	$4.2{\cdot}10^4\ s^{-1}$	5.4 %	$6.1{\cdot}10^5\ s^{-1}$	14.5
TL16552	7	100 %	$2.9{\cdot}10^3\ s^{-1}$	8.6 %	$4.4{\cdot}10^4\ s^{-1}$	15.2

TABLE 2. Maximum number of bytes moved per second and bus usage, in a Z80 10 MHz system.

a. Bus usage normalized the pure software implementation.
b. Maximum number of interrupts served per second.

cation specific DMA controller compared to an all software solution. Furthermore, the generated DMA solution occupies the system bus only about 5% of the time compared to the all software solution. This is a tremendous advantage for multi-processor systems, where the bus usage should be kept as low as possible. The speed-up and decrease in bus usage is achieved only at the cost of a few gates, which is in many cases justified.

6. Reference

[1] Pai Chou, Ross Ortega, Gaetano Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, in the *Proceedings of the IEEE/ACM ICCAD*, Nov. 1992, pp. 488-495.

[2] J-M Daveau, T. Ben Ismail, A. A. Jerraya, “Synthesis of System-Level Communication by an Allocation-Based Approach”, in the *Proceedings of the 8th International Symposium on System Synthesis*, 1995.

[3] W. Ecker, M. Glesner, A. Vombach, “Protocol Merging: A VHDL-Based Method for Clock Cyckle Minimizing and Protocol Preserving Scheduling of IO-Operations”, In Proc. of Euro-DAC, 1994, pp. 624-629.

[4] M. Eisenring, J. Teich, “Domain-Specific Interface Generation from Dataflow Specifications”, in the *Proceedings of International Workshop on Codesign*, Seattle, pp. 43-47, Feb. 1998.

[5] Laurent Freund, Denis Dupont, Michael Israel, Frederic Rousseau, “Interface Optimization During Hardware-Software Partitioning”, in the *Proceedings of International Workshop on Codesign*, pp. 75-79, Feb. 1997.

[6] R. Grehan, “Driver assistance”, *Embedded Systems Programming*, Nov. 1997, pp. 75-80.

[7] S.C. Johnsson, *Yet another compiler compiler*, Computing Science Tech. Rep. 32, AT&T Bell Lab., Murray Hill, 1975.

[8] Jan Madsen, Bjarne Hald, “An Approach to Interface Synthesis”, in the *Proceedings of the 8th International Symposium on System Synthesis*, 1995.

[9] Mattias O’Nils, Johnny Öberg, Axel Jantsch, “Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces”, in *Proceedings of the Euromicro Workshop on Digital System Design*, 1998.

[10] Mattias O’Nils, Axel Jantsch, “Refinement of HW/SW Communiation Channels: Case Study and Comparision”, in *Proceedings of IEEE Norchip* Conference, 1998.

[11] Mattias O’Nils, Axel Jantsch, “Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models”, in *Proceedings of IEEE High Level Design, Validation and Test Workshop*, 1998.

[12] Tuggle E., “Writing Device Drivers”, *Embedded Systems Programming*, Jan. 1993, pp. 42-65.

[13] D. Verkest, K. Van Rompaey, I. Bolsens, H. de Man, “CoWare - A Design Envrionment for Heterogeneous Hardware/Software Systems”, *Design Automation for Embedded Systems*, vol. 1, no. 4, 357-386, 1996.

[14] J. Öberg, A. Kumar, A. Hemani, “Grammar-based Hardware Synthesis of Data Communication Protocols”, In *Proc. of the 9th International Symposium on System Synthesis*, Nov. 1996, pp. 14-19.

Paper 5

Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications

5

Device Driver and DMA Controller Synthesis from HW/SW Communication Protocol Specifications

MATTIAS O'NILS mattias@ele.kth.se

Department of Electronics, Electronic System Design Laboratory, Royal Institute of Technology, Electrum 229, S-16440 Kista, Sweden
and
Department of Information Technology, Mid Sweden University, S-851 70 Sundsvall, Sweden

AXEL JANTSCH axel@ele.kth.se

Department of Electronics, Electronic System Design Laboratory, Royal Institute of Technology, Electrum 229, S-164 40 Kista, Sweden

Abstract. We separate the information required for HW/SW interface synthesis into three parts, the protocol specification, the operating system related information, and the processor related information. From these inputs a synthesis tool generates (a) device driver functions or (b) a combination of device driver functions and a DMA controller, depending on a designer's decision. The clean separation of information facilitates (1) efficient design space exploration with combinations of different processors, operating systems and protocols, and (2) the maintenance of a large number of different versions and variants of HW/SW interfaces.

Protocols are specified as a grammar, which is fully architecture and implementation independent. From this the synthesis tool generates device driver code in C and/or synthesizable RTL code in VHDL for DMA controllers. After the initial selection of implementation alternatives the presented methods are fully automated. Its computational complexity is $O(n^2)$. With real-life examples we show that the quality of the generated code is close to hand written quality in terms of performance, area and code size.

1. Introduction

Intellectual Property (IP) based design is emerging to close the gap between steadily increasing capacity, in terms of transistors on integrated devices, and the design productivity in terms of designed transistors per time unit. However, the integration of several IP blocks into a system on one chip makes specification and implementation of interfaces (e.g. bus interfaces and device drivers) a dominant design problem for embedded systems. Consequently, we need effective ways to model, refine, and implement communication within an embedded system. To take full advantage of the reuse potential of IP blocks, be it software or hardware, we must efficiently support a large number of operating systems, processors and processor cores, and communication protocols.

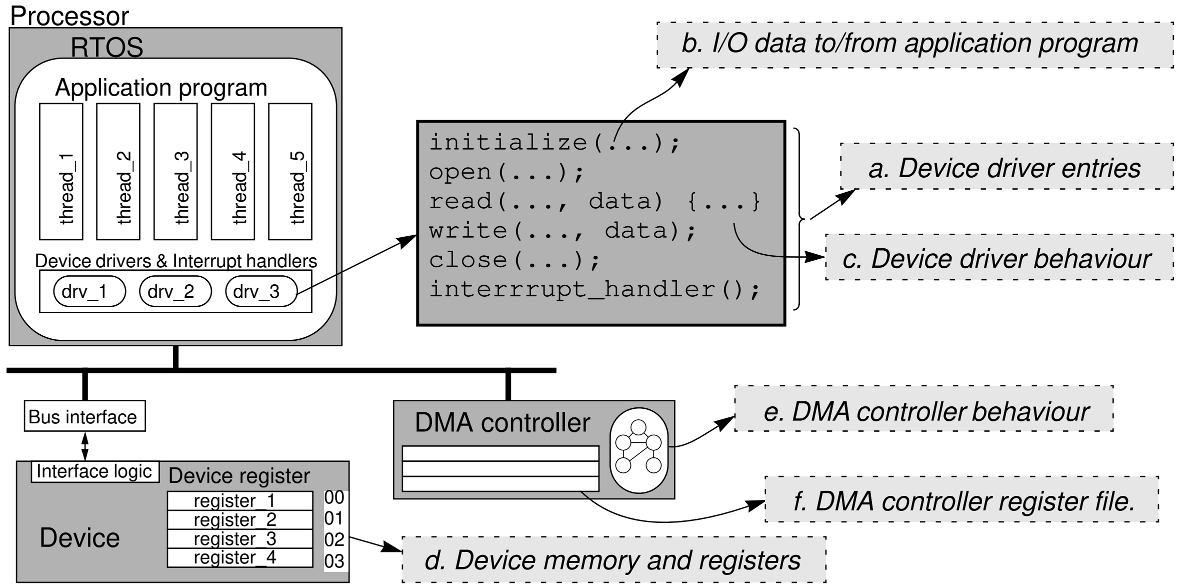


Figure 1. Break down of the HW/SW communication into different parts.

A hardware/software interface can consist of up to four parts: (1) device driver functions, (2) direct memory access (DMA) controller, (3) bus interface, and (4) the register file and handshake behaviour of the device. A device driver is the wrapper for a hardware device accessed from software. The device driver's behaviour essentially is a protocol defining how the device is accessed and synchronized with software. A device driver can perform all accesses to the device directly or it can use a DMA to transfer data between two places in the address space. In the latter case the device driver controls the DMA controller who in turn accesses memory and other devices.

Figure 1 illustrates the dependencies of a device driver on various other parts, both hardware and software. We observe that a device driver contains information on: (a) processor architecture and behaviour, (b) real-time kernel behaviour, (c) device architecture and behaviour, (d) access protocol of the device, (e) application program interface (API) rules, and (f) DMA controller architecture and behaviour. Hence, device drivers accommodate a very high information density and are dependent on many parts which can appear in a large number of combinations. This fact is the reason for the low productivity for device driver modelling, four times lower than ordinary software code [3].

Días et al. [9] present a good example of design using IP components, in which more than 60% of the design is composed of IP components. It illustrates how time-consuming the task of interfacing the IP components can be, i.e. design of the hardware/software communication parts of the system. Figure 2 shows a simplified schematic of the system, which is a pay phone controller. Consider the following design scenario: A system designer needs to evaluate what impact DMA controllers have on the system performance, by comparing two alternative architectures of interfaces to the USARTs in figure 2: (1) no DMA controller, and (2)

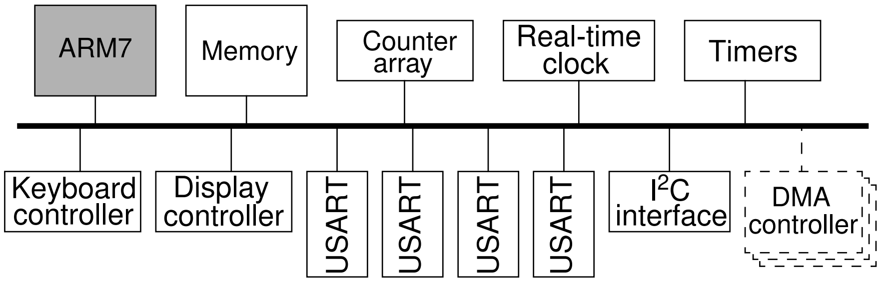


Figure 2. Simplified schematic of the pay phone controller system.

DMA controllers to interface the USARTs. The assessment is difficult due to the many dependencies and a simple calculation is not accurate enough. This is supported by the observation that in many practical situations the traffic on the bus is identified as performance bottleneck only after the system has been built. Ideally the comparison between the two alternative architectures would be done by simulation with all the interface details included in the simulation models. However, without tool support this is a formidable task because the designer must develop an entirely new interface code for each configuration and component, due to the huge difference between implementations of a hardware/software communication protocol in pure software and with a DMA controller. This task is so time-consuming that system designers typically consider a very limited number of design alternatives and frequently only one. On the other hand, a tool is perfectly capable of generating efficient models for this purpose from a high level specification of the communication protocols, as will be shown in this paper.

In addition to the evaluation of different architectures there is the task of configuring a system for different product versions and product generations. For example, in a second version of the product it is decided to use a different processor and real-time kernel, but the functionality is the same. If the interface code was written with reuse in mind, i.e. macros for real-time kernel functions and processor instructions is used in the code, most of the code can be reused. Only the APIs of the code has to be adopted to fit the new architecture. But a new processor will require another DMA controller. Thus, the interface code dependent of the DMA controller has to be redesigned.

Now consider the evaluation and design architecture upgrade tasks with a tool for hardware/software communication. Taking a protocol description of the interfaces the designer maps these onto an architecture, i.e. real-time kernel, processor, and DMA controller (see figure 3). The communication synthesis procedure

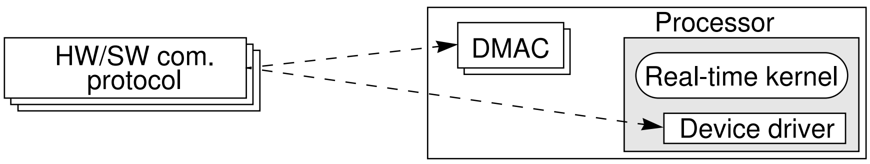


Figure 3. Mapping of hardware/software communication protocols onto selected architecture.

transforms the protocol descriptions into architecture specific device driver code and application specific DMA controllers. This enables IP providers to supply a wrapper for IP components. Thus, the designer that uses these IP components only has to use the tool to generate the interface code for the specific architecture.

Recently, much attention focused on the problem of communication synthesis for embedded real-time systems [27]. But non of them has fully addressed the problem described above. CoWare [4], Polis [2] and Cool [23] concentrate on the case where the whole design functionality is captured within their environment and then communication is refined during system synthesis, i.e. the device drivers are generated together with the custom HW and the operating system. But if the user wants to use IP blocks and off-the-shelf real-time operating systems (RTOS), he/she will face the same troubles as for manual design [30].

CoWare [4] is a design environment for heterogeneous hardware/software systems on a chip. CoWare accepts heterogeneous communicating processes, which are specified in DFL, VHDL or C. CoWare's main focus is system integration and handling of communication in embedded systems. The heterogeneous specification is mapped onto different processors (DSP, micro controller or hardware). Inter process communication is done with point-to-point communication channels. The communication semantics are based on the concept of rendezvous channel communication via send/receive operations. Hardware/software communication channels are mapped onto a fixed architecture. This architecture is based on several library models. For software, the communication procedures are captured as parameterized C functions that are mapped onto a software model, i.e. they adapt to processor specific I/O handling, interrupt handling, etc. For hardware, a hardware interface cell is generated to connect with a handshake protocol to an I/O control unit. This I/O control unit is a link between the processor and the handshake protocol.

Polis is focused on control-dominated applications with system architectures composed of a single processor surrounded by custom or library hardware. Polis uses Codesign Finite State Machines (CFSM) as the internal representation for a system description, separating communication, behaviour and timing of the system. The communication model is globally asynchronous, locally synchronous, with non-blocking finite buffers between CFSMs. C code and HDL code is generated from the CFSMs mapped to software and hardware, respectively. Except for the I/O drivers and code generated from the CFSMs, the software code consists of a generated application-specific operating system for the selected processor. All communication within software or between software and hardware occurs through shared memory, I/O ports or memory-mapped I/O. The synthesized hardware includes the address decoders, multiplexers, latches and glue logic. Special-purpose hardware must follow a simple, data/strobe based protocol in order to be interfaceable with other CFSMs.

In contrast to both CoWare and Polis, our approach uses an abstract, implementation independent protocol specifications, and is targeted specifically towards gluing IP blocks and third party RTOS together.

In Chinook [5,6], the definition of a device driver also includes the bus interface. Ortega et al. have in [26] expanded this approach to communication synthesis for distributed systems, i.e. from a behaviour description and an architectural mapping they synthesize interfaces for different bus structures, e.g. CAN and I^2C.

MakeApp [14] is a tool for generating device drivers for different devices and processors matching user defined configurations. Both Chinook and MakeApp solve only part of the problems described in [30], since they do not support DMA and generate code only for a specific real-time kernel (Chinook) or no kernel (MakeApp).

Jerraya et al. use SOLAR [17] as an intermediate language, it supports system level modelling and synthesis. The communication semantics are based on the concept of rendezvous channel communication via send/receive operations, similar to CoWare. Types of protocols supported include blocking and non-blocking communication. A channel is implemented by allocating communication units from a library [7]. These library units are composed to fulfil a set of constraints put on the channel. The library consists of a set of communication services and protocols along with their implementations (a mixture of hardware and software). Interface synthesis techniques are used to permit communication between the processor and the chosen communication units.

Eisenring et al. [11] present a method for modelling the target architecture by means of object oriented methods. The architectural model is used to generate hardware/software interfaces from data flow descriptions. LYCOS [16], COSYMA [12] and Vahid et al. [31] propose to solve the hardware/software communication by supplying a communication library. However, this only moves the problems of designing and maintaining device drivers from the designer to the provider of the library. The only approach that handle mapping of hardware/software communication on to DMA controllers is described by Eisenring and Teich [11], but their approach is limited to a set of standard APIs.

Öberg et al. present a communication protocol synthesis tool that specifies the design in a special purpose language ProGram [33], which is based on context free grammar, the synthesizable subset is limited to regular grammar with attributed conditions. The specified protocols are synthesized into hardware. Seawright et al. [28] have a similar approach but instead of grammar they specify the behaviour with a graphical version of regular expressions.

We adopt Öberg's approach to specify communication protocols in ProGram and apply it to the specification and synthesis of the software part and DMA controllers in hardware/software interfaces. The synthesis tool requires a behavioural description of the protocol and a mapping to a particular architecture. The relevant information about architecture elements, HW or SW, are captured in two libraries. All implementation details of the device driver and DMA controller are automatically synthesized. The communication synthesis tool is also capable to generate simulation code for commercial hardware/software co-simulators [25].

The next section describes the design flow, target architecture and protocol modelling in ProGram. Section 3 analyses the architecture dependent parts of device drivers and proposes a way to represent them. Section 4 and 5 present the proposed synthesis methods for software and for a software/DMA solution, respectively. Section 6 contains descriptions and results from the examples used to evaluate the methods.

2. Methodology

We model device drivers independently of the architecture using ProGram [24]. ProGram is a grammar based notation for protocol applications, which is inspired by YACC [18]. Specifications in ProGram deal with sequences of allowed events as opposed to states and state transitions in an FSM model. In contrast to parsers for compilers ProGram allows for several concurrent input and output streams. The ProGram description is synthesized into a set of untimed extended state machines [24,33], which is the input to the architecture mapping procedure described in sections 4 and 5.

As shown in figure 4, the mapping procedure uses data from two libraries to generate the architecture specific code. The first library captures the information on the operating system architecture (OSLib) and the second library captures the processor specific characteristics (ProcLib). This organization carefully separates protocol specification from operating system and processor dependencies. With this input the tool can generate three alternative implementations: (I) pure software implementation, (II) code for mixed hardware/software solutions with a DMA controller, and (III) multi-level simulation models. In this paper we discuss only alternative I and II, and refer the reader for further details about the multi-level simulation methodology to [25].

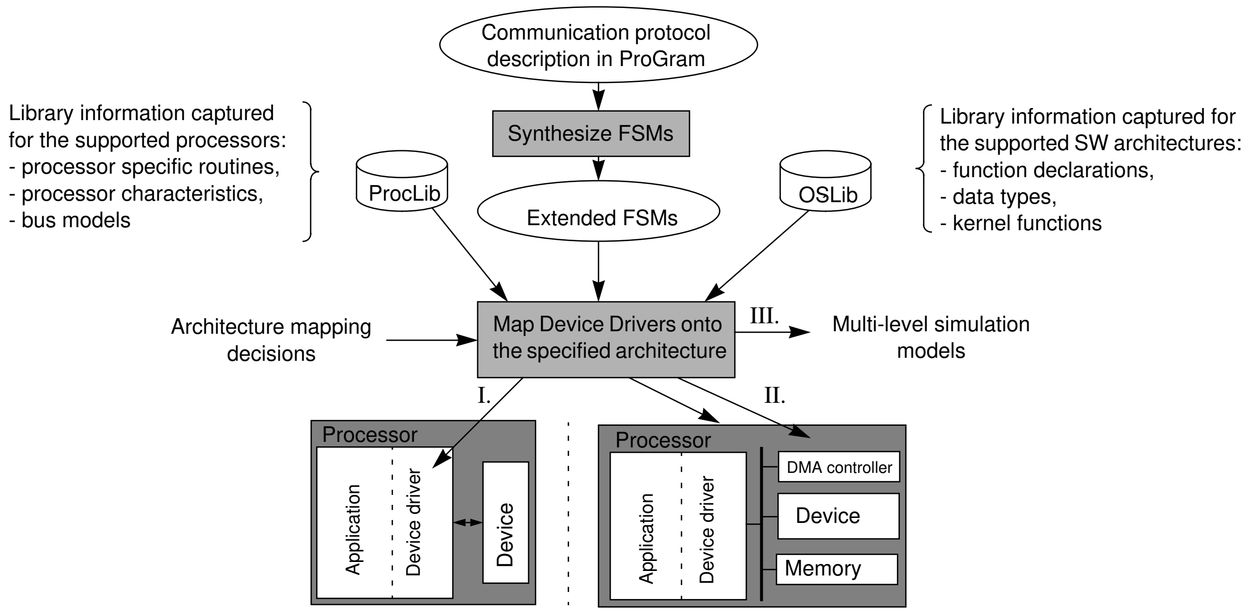


Figure 4. Overview of the device driver synthesis system.

2.1. Target architecture

As indicated in figure 4, our approach to device driver synthesis supports two alternative target architectures: (1) pure software implementation and (2) software together with a generated DMA controller.

Software implementation: For pure software implementation access functions and interrupt routines are implemented as software functions in C, which are mapped to fit the selected real-time kernel and processor. For the appropriate call of the interrupt routine an interrupt handler is generated.

With DMA controller: The combined DMA and software based architecture is used when the designer has selected one of the state machines to be implemented as a DMA controller which controls the data transfer between a hardware device and memory. This is the preferred solution if high peripheral data rates must be achieved, and performance and bus utilization are critical.

2.2. Protocol modelling in ProGram

A ProGram model defines the possible event sequences on several parallel inputs in terms of a grammar [33]. Actions, which are associated with grammar rules, define the reaction to input events and the event sequence on the output ports. A ProGram description consists of three parts: (1) interface declaration, (2) tokens and memories, (3) grammar rules and actions.

Interface declaration: The interface declaration describes the interface to the application and the device. There are four types of ports to describe the device driver interface: (1) ports to the application (sw), (2) registers of the interfaced device (hw), (3) internal signals (internal) and (4) interrupt signals from the device (interrupt). In addition to the type a port declaration contains name, direction and bit width. A port declaration of a device register requires also the register's relative address.

Tokens and memories: A Token is a pattern of bits read from an input stream or written to an output stream and can also be viewed as a constant. Reading and writing tokens are the primary events. Memories and variables can be defined to maintain a state and to communicate between concurrent activities of the protocol implementation. The size and layout of memory fields can be defined.

Grammar rules and actions: The grammar description begins with one or several start rules. They define concurrent activities and work in practise as process declarations that define the signals used for the start condition. The synthesis process generates for each start rule an access function or an interrupt routine. Actions in the grammar specify assignment of values to signals. Expressions to compute these values may be put directly in the assignments or may be associated with some symbols in the action value section. The assignments can then simply refer to these symbols. The expressions allow concatenation and conditionals in addition to the usual arithmetic and logic operations. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols. A grammar rule consists of a grammar symbol that serves as a rule identifier and a list of alternatives. Each alternative is a sequence of non-terminal symbols, terminals and actions. Passing the new signal stream as a parameter to the subtree of productions redirect the input stream. Actions are enclosed in curly brackets, see figure 5.

```
grammar_rule_1(sig):symbol_1 grammar_rule_2
   |  ...
   |  symbol_n {action_section;};
```

Figure 5. Grammar rule with actions.

3. Architecture characteristics and modelling

First we discuss architecture specific parts of device drivers, i.e. synchronization, execution delays, interrupt handling, mutual exclusion, direct memory access (DMA) and the device driver interface. Subsections 3.2 and 3.3 describe how operating system and processor characteristics are modelled to serve as input to the synthesis process,

3.1. Architecture dependent parts

Synchronization with external events: Synchronization of device drivers with external events can be implemented in three ways, as illustrated in figure 6: (a) polling of device signals, (b) wait for a fixed time (when the device is known to generate an event), or (c) wait for an operating system event generated by an interrupt that is triggered by the device. An example of where synchronization is needed is an analog to digital converter. There we first initiate a conversion by writing some control data, and then wait for the conversion to take place, i.e. synchronize with the conversion ready event. This is done by polling a status register, wait for the specified conversion time, or let the device generate an interrupt signal. After that, read the sampled value from the device. Polling is the only one of these three synchronization schemes that can be implemented independent of the software kernel.

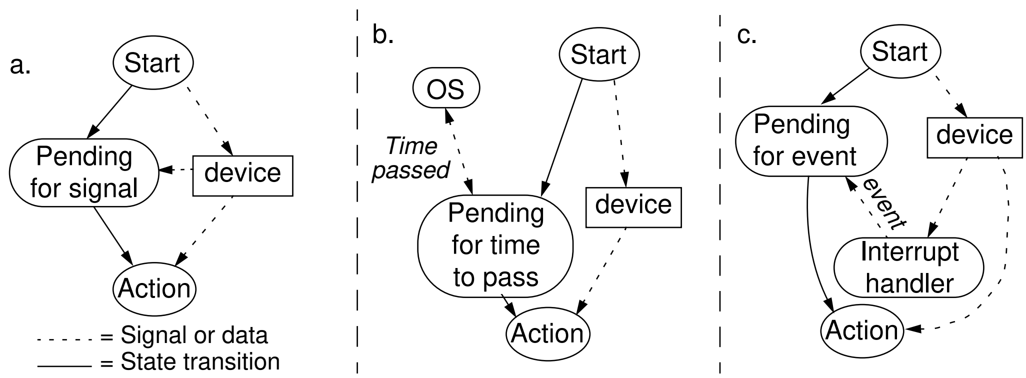


Figure 6. Synchronizing with external device.

Synchronization with internal events: Internal synchronization, i.e. synchronization between the device driver parts, is implemented using the semaphore facilities of the kernel. To prevent deadlocks the sender and receiver of the internal synchronization events has to be located in different execution flows. Hence, a sender has to be located in an interrupt routine and the receiver in an ordinary access function, since wait operations are not allowed in interrupt routines.

Delay execution: As described in the previous section, in some situations a device driver has to halt itself for a certain amount of time. This could be necessary when waiting for an analog-digital conversion, but also to wait for coprocessors during the initialization process. The wait function is provided by a kernel, hence this function is architecture dependent.

Interrupt handling: Interrupt service routines can be used to synchronize a device driver with a hardware event as described above. Sometimes they are also independent routines with their own behaviour, e.g. when an interrupt handler receives data from a communication device and writes it to a buffer. There are two possible ways to implement this: (a) the interrupt service routine informs the task about the event when the interrupt is activated and the task handles the action; (b) the interrupt routine performs the action itself. Interrupt handling is dependent on both the processor and the kernel. It is dependent on the processor because an entry to the interrupt routine must be written in assembler code since high-level languages can not model this. Dependent of the kernel in the case of a preemptive kernel, then the interrupt handler has to notify the kernel when entering an interrupt routine using kernel specific routines.

Mutual exclusion: There are two possible ways to prevent several application threads accessing the same device simultaneously: (1) by means of semaphores and (2) by disabling the interrupt mechanism. Both have their advantages and disadvantages. Disabling interrupts is fast and does not add overhead to the execution but it disables the normal behaviour of the kernel, i.e. introduces non-determinism. Semaphores do not affect the kernel behaviour but introduce communication overhead. Hence, one should use semaphores for complex device drivers and disable interrupts for simple ones. Both methods for controlling mutual exclusion are architecture dependent. To prevent this the control can be placed outside the device driver, as in figure 7a. The cost of this is less structured code compared to the solution in figure 7b, where the control is placed inside of

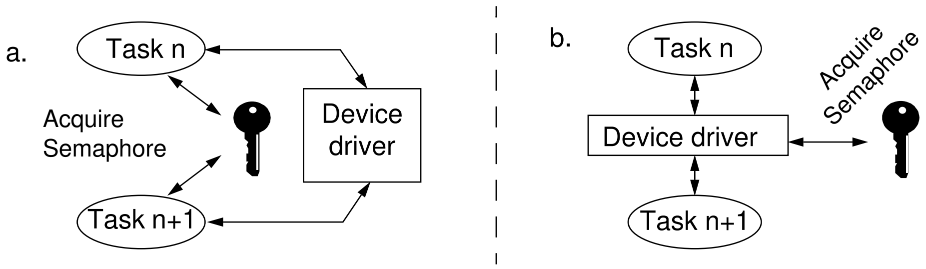


Figure 7. Handling mutual exclusion. (a) Mutual exclusion control outside device driver. (b) Mutual exclusion control inside device driver.

the device driver.

Device driver interface: The device driver interface, application program interface (API), is the definition of how the device driver is used, i.e. how in and out parameters to the device driver are handled, how the device driver should be used and driver component naming style. For device drivers developed for single threaded software with no operating system or systems with small real-time kernels like μC/OS or RT-kernel, the device driver interface is determined by the designer, project or company coding style. For other systems with kernels/operating systems like UNIX, OS9, VxWorks and OS/2 the interface to the application/operating system is defined by the operating system [30].

DMA controllers: DMA controllers are used to decrease the processor load, and to speed-up the transfer of large amounts of data. This is achieved at the cost of additional hardware and reusability of the device driver code, since the code not only dependents on the kernel and processor but also on the DMA controller, since each DMA controller has its own unique architecture and behaviour. A DMA circuit contains one or several controllers that can move data between memory-memory or memory-device. From the software view, the controller is a set of registers for setting address pointers and controlling the behaviour. The behaviour can either be to move a block of data in one step, or move it word wise synchronized by e.g. an interrupt signal from a device. When a DMA transfer is finished, the controller synchronizes with the software by triggering an interrupt.

3.2. Modelling the software environment

To map architecture dependent parts of a device driver to a specific architecture, the characteristics and behaviour of the architecture is captured in two libraries, OSLib and ProcLib. They capture the operating system specific and processor specific parts, respectively (figure 4).

The model of the software environment (i.e. the kernel's characteristics and functionality) used in OSLib is divided into three parts: (1) the environment characteristics, (2) the interface to the application/OS (API), and (3) the macro for accessing system services.

SW environment characteristics: This part contains the environment name, type, timing, and include files needed for this kernel. The type is captured since it affects the behaviour of the interrupt routines. Characteristics captured by OSLib are defined in table 1.

SW environment interface: With SW environment interface we mean the naming style of the different device driver functions and how data is transferred to and from these functions. The interface model is composed of three interface scripts for generation of function declarations: (1) interrupt service routine, (2) device driver function, and (3) task function. These scripts generate the function names and parameter declarations. There is an additional translation table for type conversion, i.e. for translation of bit vector types to C data types.

Type	Library entry	Comment
Characteristics	name	Name of the real-time kernel
	kernel_type	{preemptive, not-preemptive, single thread}
	tick_time_period	Time between two clock ticks
	interrupt_supported	{true, false}
	header_files	A list of header files needed by the kernel
API generation	interrupt_routine_def	Script for generating interrupt routine declaration
	access_routine_def	Script for generating access driver routine declaration
	task_def	Script for generating task declaration
	type_coversion_table	Table for converting bit vector to C data types
Services	enable_interrupts	Function for enabling interrupts
	disable_interrupts	Function for disabling interrupts
	create_binary_semaphore	Function that generates a binary semaphore
	delete_binary_semaphore	Function that delete a binary semaphore
	pending_semaphore	Function for pending a semaphore
	set_semaphore	Function for sending an event with a semaphore
	pending_time	Function for waiting for a specified time
	enter_interrupt	Function for informing kernel of interrupt service start
	exit_interrupt	Function for informing kernel of interrupt service end

Table 1. Library structure used in OSLib to capture a real-time kernel.

SW environment services: SW environment services are kernel functions for semaphore and interrupt handling. Modelled services are discussed in section 3.1 and the system functions to provide these services are enumerated in table 1.

3.3. Modelling of processor features

The model of the processor (i.e. the processors's characteristics and functionality) used in ProcLib is divided into three parts: (1) the processor characteristics, (2) processor specific routines, and (3) DMA components.

Processor characteristics: The captured processor characteristics are the (a) processor name, (b) the internal and external data bus width, since these will affect the address calculation of the device addresses, and (c) the device access type, i.e. if it uses a port mapped or memory mapped device access method (table 2).

Processor specific routines: Processor specific code necessary for device driver generation is code for saving and restoring of registers, code for enabling and disabling of interrupts. The last is used if the kernel does not support this feature. Table 2 defines the processor specific code captured in ProcLib.

Master interface: This is an interface between a fixed protocol (defined in figure 8) and the captured processor's bus protocol. The port with fixed protocol on the interface is the one connected to the generated part of the DMA controller, described in section 5. A data access is initiated from the DMA controller who activates the master interface, which then obtains control over the processor bus and performs the data access.

On one hand it implements the processor specific bus interface, and on the other hand it implements the master interface protocol to communicate with the generated DMA controller. The master interface commu-

Type	Library entry	Comment
Characteristics	name	Name of the processor
	device_access_type	{memory mapped, port mapped}
	internal_data__bus_width	{8,16,32}
	external_data_bus_width	{8,16,32}
Routines	push_proc_register	Instruction to store processor register on stack
	pop_proc_register	Instruction for restoring processor register from stack
	enable_interrupts	Instruction for enabling interrupts
	disable_interrupts	Instruction for disabling interrupts
	return_from_interrupt	Instruction for returning from interrupt
	function_call	Instruction for calling functions
	read_port_mapped	Instruction for reading port mapped
	write_port_mapped	Instruction for writing port mapped
DMA components	master_interface	Interface for the accessing the processor bus
	slave_interface	Memory interface for accessing a register file

Table 2. Library structure used in ProcLib to captured the processor specific parts.

nicates with the generated parts with a fixed protocol, which is defined in figure 8. Hence, this approach is independent of bus organization and arbitration controller, since arbitration handshake and bus specific behaviour is implemented by the master interface, not by the generated parts of the DMA controller.

The master interface component in figure 8 uses a 2-phase request/acknowledge protocol. The protocol is synchronized with the bus clock used for the processor bus. A read/write cycle starts with the request (*req*) signal driven high, the *rw*, *size* and *address* signals are driven to the appropriate values. A cycle is terminated by detection of the ready (*rdy*) signal at the positive clock (*clk*) edge. Read data (*data_in*) is valid only at the positive clock edges and when the *rdy* signal is high. Write data (*data_out*), *address*, *size* and *rw* signals should be valid before the positive clock edge when *req* is high and until after the positive clock edge when *rdy* is high. The *size* signal indicates the size of the transfer, i.e. it is up to the master interface to implement the reading and writing data of different size.

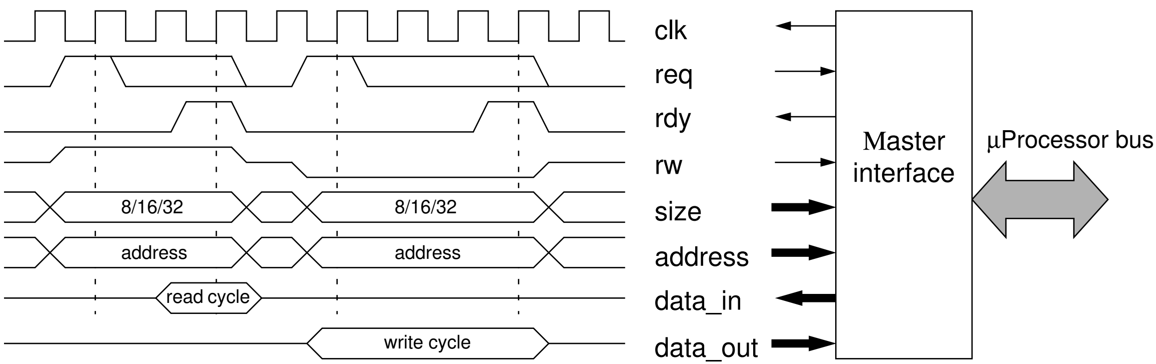


Figure 8. Master interface component and its access protocol.

Slave interface: The slave interface is a simple memory interface that is used to enable access to the generated DMA register file (described in section 5) from the software parts of the device driver. The register file is accessed through an ordinary asynchronous memory interface (similar to a static RAM with *read*, *write* and *chip_select* signals). In this way the set-up functions can initialize the base address and the access functions can write and read back the internal variables and pointers.

4. Software synthesis

From the ProGram description of the device driver several untimed FSMs are synthesized using the same technique used for generating HW interfaces [33]. From there the implementation is created in four steps: (1) translation of the bit vector data types used in the ProGram and FSM representations into C/C++ data types; (2) transformation of the state machines into state machines suitable for software implementation; (3) optimization of the state machines; (4) mapping of the state machines onto the selected processor/real-time kernel, generation of the application program interface (API), and generation of C code. These steps are indicated in the synthesis procedure depicted in figure 9.

```
procedure generate_sw_implementation
for all Signals
   signal.map_bit_vector_types(OSLib)                       1 Map data types
end for
for all FSMs
   for all FSM.States
      state.apply_transformation_rules(OSLib, ProcLib)      2 Transform
   end for
   FSM.minimize_no_states()                                 3 Optimize
   FSM.minimize_no_goto()
   case (type of FSM)
      ACCESS : FSM.gen_function(OSLib)
      INTERRUPT : FSM.gen_int_routine(OSLib)                4 Map and generate code
      TASK : FSM.gen_task_function(OSLib)
   end case
end for
end procedure
```

Figure 9. Synthesis procedure for transforming the FSMs to FSMs for SW implementation.

Map data types: Data types in the ProGram description and the generated FSM descriptions use a bit vector data type. As described in the previous section, a table in OSLib defines the translation of bit vectors into C types. This table describes how the translation from a bit vector of a specific width is represented in C.

Transform: The synthesis procedure has two transformation rules for identifying synchronization: (1) one for external synchronization and (2) one for internal synchronization. The objective of these rules is to transform the FSM behaviour to fit a general software architecture. As seen in figure 9 both these transformation rules are applied to all states in the synthesized state machines.

Synchronization points in a ProGram description are translated into self-loop states in the generated FSM, which is further transformed for a suitable software implementation (figure 10). The wait condition in the self-loop, *cond* in figure 10, and the conditions to exit the wait-state, $c_1 \ldots c_m$ in figure 10, determine if it is an ex-

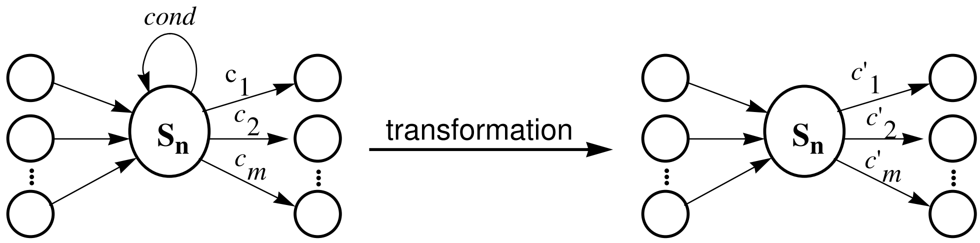


Figure 10. Template for transformation rules, where m is the number of transitions from state S_n.

ternal or internal synchronization point. With external synchronization in this context we mean synchronization with an interrupt signal generated by the interfaced device and with internal synchronization we mean synchronization between different parts of the device driver.

The constraints in table 4 determine if the self-loop in figure 10 is an external synchronization or an internal synchronization point. The signal type of the sensed signal within the wait-state is the only difference between the two sets of constraints. If the constraints for an internal or external synchronization point are fulfilled, the state machine is transformed according to the actions described in table 4 and partly illustrated in figure 10. For external synchronization these actions create another FSM entry for an interrupt routine that respond to the interrupt and send a software event to the FSM where the synchronization point was found. In the state that fit the transformation rule the self-loop transition is removed and replaced with a wait for software event routine in the code part of the state.

Transformation rule	Constraints	Transformations
External synchronization	• `width`(sig) = 1 • sig is an interrupt signal • cond ∈ [(sig=0) | (sig=1)] • cond = ¬[c1 ∨ c2 ∨ ... ∨ cm]	• Remove self-loop transition • Insert wait on semaphore in code of S_n • Add an interrupt routine that sends an event to wait statement
Internal synchronization	• `width`(sig) = 1 • sig is an internal signal • cond ∈ [(sig=0) | (sig=1)] • cond = ¬[c1 ∨ c2 ∨ ... ∨ cm]	• Remove self-loop transition • Insert wait on semaphore in code of S_n • Replace assignments to sig with semaphore manipulations

Table 3. Transformation rules used in the software synthesis process.

For internal synchronization points the self-loop transition in the state, fitting the transformation rule, is removed and replaced with a wait for software event routine in the code part of the state. Then all assignments to the event are replaced with semaphore manipulation routines.

Optimize: Apart from transforming the state machines to fit software architectures, the synthesis procedure optimizes the state machines for software implementation with respect to size and speed. Optimizations done are minimization of the number of states and *goto* operations. Both these optimization methods aim to minimize the code for the state transitions and, thus, decreasing the code size and typically increasing the performance. Minimization of the number of states in an FSM is done when the state machine is generated from the ProGram description. In the "optimize" step, state minimization goes further by removing states not necessary for software implementation, e.g. delay states. A delay state is a state with the only function to preserve execution order i.e. it has one entry and one exit transition. The execution order in software is ensured by merging the code of the removed state to the beginning or end of the code block of the succeeding or preceding state, respectively.

The synthesis procedure generates C code and the state machines are implemented with *if* and *goto* statements. By ordering the states in the code that implements the state machine the code for state transitions is minimized. By ordering states properly according to their execution sequence unnecessary *goto* statements can be avoided and code size and performance will improve. The *goto* minimization is performed such, that transitions that can be implemented by placing the states adjacent to each other, are placed first and remaining states are placed randomly.

Generate and map code: When the state machines are transformed and optimized for software implementation, the protocol description has to be mapped onto the processor and real-time kernel. This is done at the same time as the code is generated. The code generation has two different components to generate code for; (1) access functions and (2) interrupt routines. To activate the interrupt routines there is an interrupt handler that also has to be handled by the code generation. The application program interface (API) to a device driver is a set of functions used to access the device. As described in section 3 the layout of these APIs are determined either by the design rules or by the kernel. In the synthesis procedure the API is generated from the rules that determine how data should be passed between the application and device driver and the scripts how the function name should be generated. The rules and scripts are contained in the OSLib that is further described in section 3.2.

The function head of the access functions, i.e. function name and parameters, is generated as described above by using scripts in OSLib. The body code of the access function consists of routines for mutual exclusion taken from the OSLib and the C code for the FSM. The code implementing the FSM uses kernel specific routines for synchronization handling from the OSLib library.

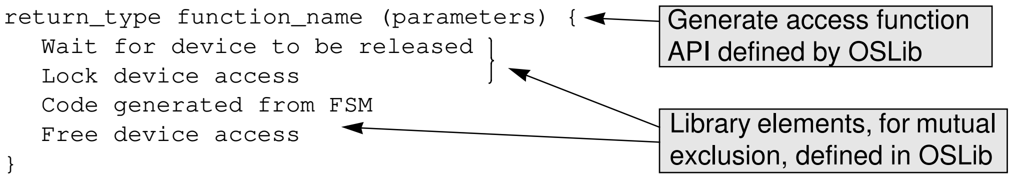


Figure 11. Code structure for an access function

The interrupt handler that responds to the interrupting device activates the interrupt routine. Interrupt type state machines are translated into interrupt routines. An interrupt routine has no parameters since the interrupt handler activates it without parameters. Thus, code for the interrupt handler is a parameterless function head and the code implementing the state machine.

Entry and exit of an interrupt handler must be in assembler language, since there is no support in C for the necessary operations. The generated code for an interrupt handler starts with storing and ends with restoring the processor registers. For a preemptive kernel the interrupt should also notify the kernel when it enters and leaves the interrupt routine.

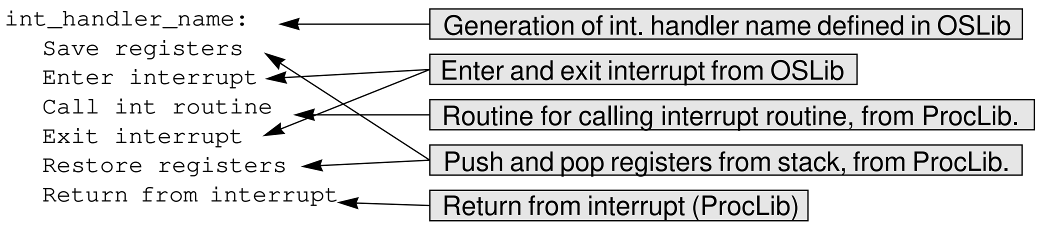


Figure 12. Code structure for an interrupt handler.

The code generation ends with the generation of an initialization function. This function creates and initializes the semaphores used by the other components. It also sets the addresses of the different device registers. The initialization function takes the device address as a parameter. The code generation also generates the declarations of semaphores and device register pointers.

A complexity analyses of the implementation of pseudo code for the synthesis procedure in figure 9 shows a run time complexity of $O(n^2)$, where the n is the number of states for the whole description. To verify this, we have exercised the synthesis procedure with a set of random examples of different complexity. Figure 13

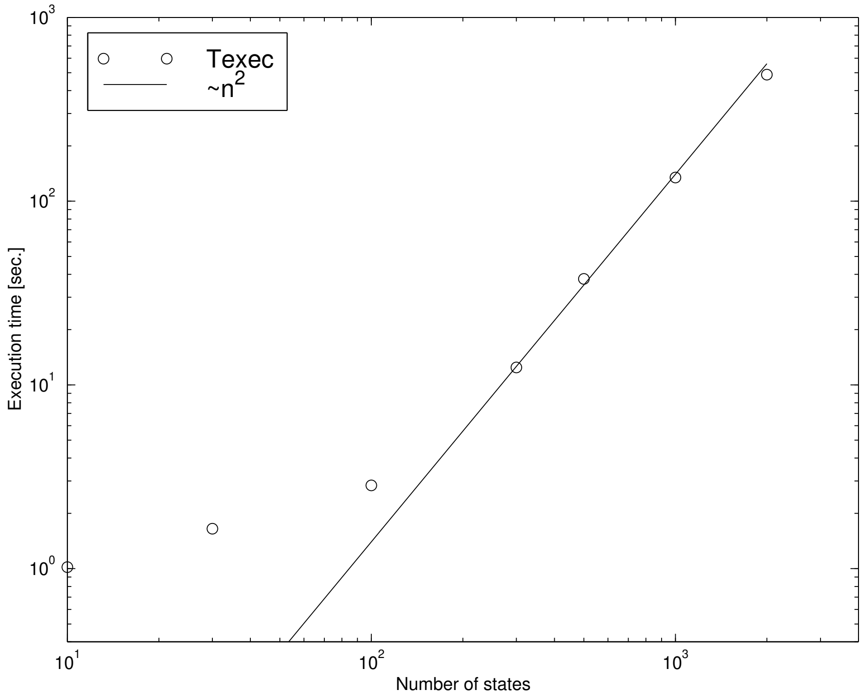


Figure 13. Synthesis times for random generated state machines. (On a Sparc Ultra 10, 230MHz)

show the result from these experiments and these verify that the complexity of the procedure is $O(n^2)$.

5. DMA controller synthesis

Synthesis of DMA controllers from parts of a device driver description is a complement to the software synthesis in section 4. The API for a device driver implemented in software and with a DMA controller is the same as for a pure software implementation, i.e. the only difference is the implementation of the behaviour. The DMA implementation has apart from the software code a generated DMA controller. The DMA controller consists of four blocks (figure 14): a master interface, a slave interface, a register file and a controller. The master and slave interfaces are library elements described in section 3.3. The controller and register file are both generated from the ProGram description. The register file contain pointers and variables used by the device driver. The controller is a transformed state machine generated from the ProGram description. Before the synthesis of the controller part, the designer has to select which of the device driver state machines should be implemented in hardware as a DMA controller. Both interrupt and access type state machines are candidates for DMA implementation. When this is done the synthesis procedure generates the DMA controller in three steps; FSM transformation for synchronization, FSM transformation for data accesses and generation of register file.

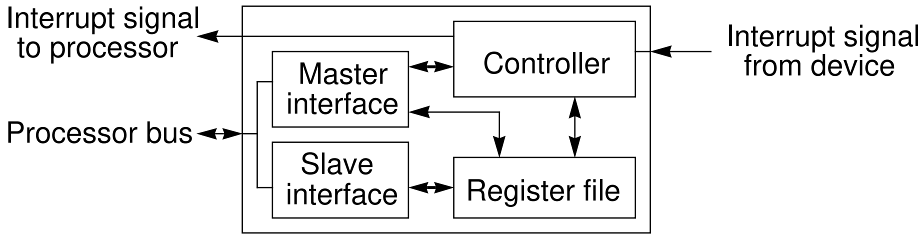


Figure 14. Architecture for the generated DMA controller.

FSM transformation for synchronization: There are two types of synchronization considered here; start and stop of the whole DMA state machine and synchronization within state machines. The transformation rules are formally described in table 4. Synchronization of access type state machines is done in four stages (see figure 15): (1) in the generated software part of the access function the parameters are downloaded to the register file of the DMA controller. (2) send start signal to the DMA controller, sensed by an inserted wait-state. (3) when the DMA controller state machine reach the exit transition, exit transition is redirected to the idle state, sends an interrupt to a generated added interrupt routine. (4) The generated interrupt routine sends an event to the waiting software part of the access function. Control of start and stop of interrupt type state machines are handled by inserting a wait-state be for entering the state machine. The wait-state senses the interrupt signal from the device and makes a transition to the original part of the state machine when the interrupt signal is asserted.

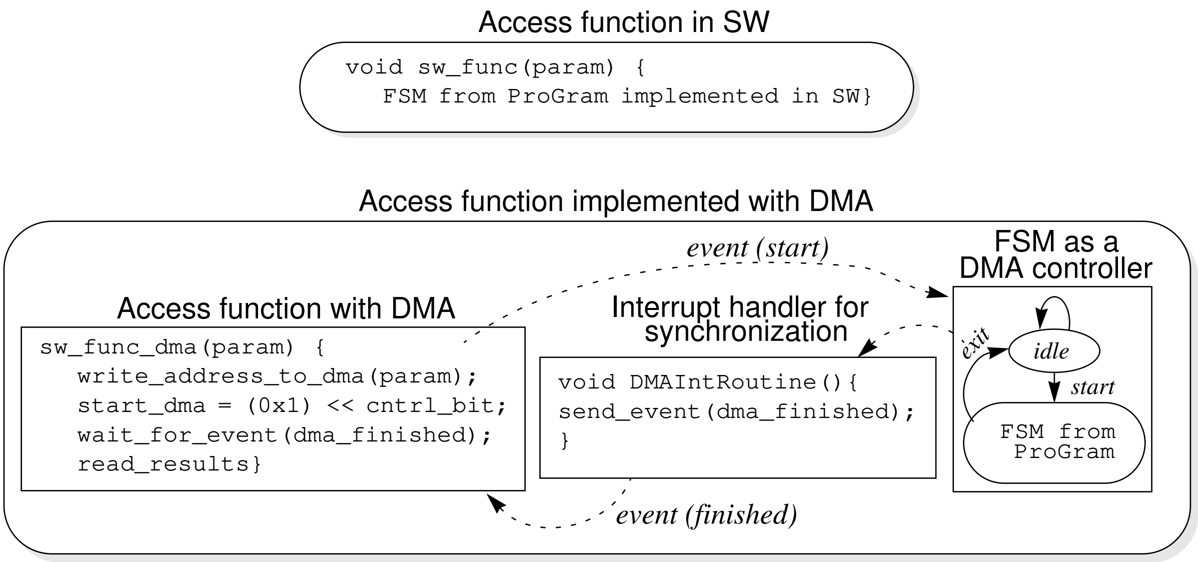


Figure 15. Access function implementation in software and with SW/DMA.

Implementation of synchronization points within and between state machines, i.e. what is called external synchronization and internal synchronization in section 4, is handled differently for state machines implemented as DMA controllers. External synchronization, i.e. synchronizing with an event via an interrupt routine, is implemented by sensing the interrupt signal from the device directly in the state machine. Two different cases of internal synchronization exist: the event sender is a DMA controller and the event receiver is a DMA controller. For the first case, instead of semaphore manipulation, the interrupt signals to the processor is asserted. The interrupt is received by an interrupt routine that sends the event to the receiver as a software event. For the second case, the receiver of the event will wait for a control bit to change instead of waiting for a semaphore event. The sender, i.e. a software function, of the event asserts the control bit.

Transformation rule	Constraints	Transformations
Access function control	• FSM of access type	• Generate SW function and interrupt handler • Add idle state before entry state of the state machine. • See figure 15.
Interrupt routine control	• FSM interrupt routine type.	• Insert an idle state before the entry state in state machine.
External synchronization	• same as in table 3	• Interrupt signal from device is used as synchronization signal.
Internal synchronization	• same as in table 3 • sender is a DMA controller	• Set interrupt signal to processor. • Add an interrupt routine that receives interrupt and sends the software event.
Internal synchronization	• same as in table 3 • receiver is a DMA controller	• Wait for control signal to be asserted • Assert control signal instead of send event

Table 4. Synchronization transformation rules used in DMA controller synthesis.

Transformation of data transfers: Data in memory and device registers have to be accessed through the processor bus. Thus, the state machine has to communicate with the master interface, described in section 3.3, to access the memory and device. This is achieved by inserting a state for reading or writing data, see figure 16.

Data read	• External value (memory location device register) used in expression.	• Insert read state (see figure 16a) before the state that fit the constraint.
Data write	• Data assigned to memory or device register.	• Insert write state (see figure 16b) after the state that fit the constraint.

Table 5. Data access transformation rules used in DMA controller synthesis.

For a data-read, data is read into a register by inserting a read state that reads the data via the master interface. The state is inserted before the state where the data is used. Assignment of variable/register is done by inserting a write state after the state where the data is written. Thus, data is written via the master interface These two transformation rules are formally described in table 5.

a.
```
when state_n1 =>
   req <= ‘1’; rw <= ‘1’; size <= “00”
   address <= pointer + offset;
   if (rdy = ‘1’) then
      req <= ‘0’;
      next_state <= state_n2;
      tmp <= data_in;
end if;
```

b.
```
when state_n1 =>
   req <= ‘1’; rw <= ‘0’; size <= “00”;
   address <= pointer + offset;
   data_out <= tmp;
   if (rdy = ‘1’) then
      req <= ‘0’;
      next_state <= state_n2;
   end if;
```

Figure 16. States for accessing external data.

Generating the register file: A register file for accessing the device registers and memory data is generated. The register file contains: device addresses (figure 17a), internal variables, external variables and vector addresses (figure 17b). A control register for monitoring the state of the DMA controller and for synchronization is also generated (figure 17c).

The device access registers are used to generate the device register addresses in the DMA controller during execution. There is one base register that stores the device base address and several offset registers (one for each device register). Address to a device register is the sum of the base register and the offset address for a specific device register. The base register has the same bit width as the processor address bus, and the register offset is set to $\lceil \log 2(\mathrm{size}(deviceRegFile)) \rceil$. Internal signals are either internal variables or pointers to vectors. The internal variables have the same bit width as specified in the protocol description and the pointers have the same width as the processor address bus. Addresses to vector elements are generated in the same way as device register addresses, but as a sum of the vector pointer and the index variable. The control register's least significant bit indicates if the base register address is written or not. If this address is not written, the controller can not serve any interrupts from the device since the controller does not have the address to the device. The remaining bits are used for synchronization between different parts of the device driver.

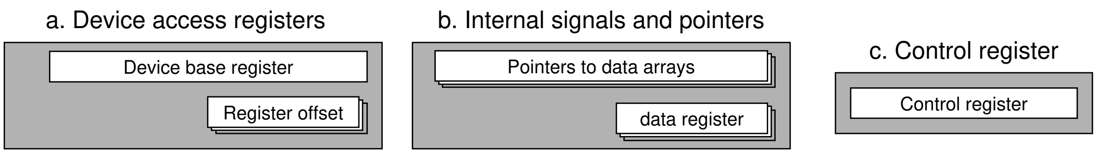


Figure 17. Register file for the DMA controller.

6. Examples

We present results from hardware/software interface protocols to five different components: three standard components (MAX197 [20] a 8 channel 12-bit ADC, MAX530 a 12-bit DAC [21] and TL16552 [29] a 2 channel UART) and two system design case studies, i.e. channel decoder in a D-AMPS base station [10] and a operation and maintenance unit (OAM) of ATM switch on the F4 level [8].

6.1. HW/SW communication of a IS-54B base station channel decoder

D-AMPS [10] (IS-54B) is a North American digital cellular standard developed in the late 80's by the Cellular Telecommunications Industry Association. The objective was to increase the performance of the existing analog AMPS standard using the same infrastructure and to support AMPS operation, hence a dual-mode cellular system. D-AMPS is a time division multiple access (TDMA) cellular standard. The standard has 832 channels per carrier in the 800 MHz range and a total of 1164 channels. Each radio channel has 30KHz bandwidth and is segmented in time into six slots using TDMA, whereas 2 slots make up one voice circuit. The data rate is 48.6 Kbits/s, with each slot transmitting 324 bits in 6.67 ms. The channel decoder functionality receives 272 bit frames that should be processed in 6.67 ms. The process consists of: de-interleaving, Viterbi decoding, CRC-sum calculation, bit error rate estimation, sorting of speech data, and masking of bad speech frames.

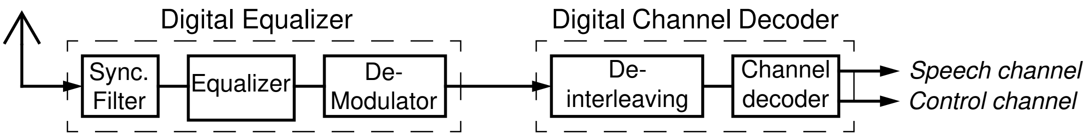


Figure 18. Receiver in IS-54B base station.

We have selected a hardware/software partition from an industrial implementation, i.e. a Viterbi decoder is implemented in hardware and the rest of the functionality is implemented in software. From a known partition we can extract the software interface (interface to the application code, figure 19a) and a hardware interface (device register file in figure 19b).

Figure 19 describes the interfaces of the protocol, the behaviour of the protocol is defined as follows. When the protocol is called from software the inputs from the application (*data*, *facch_polyn*, *uch_polyn*, *data_type*) is read. Then the data and polynomial signals are formatted according to what is indicated by the *data_type* signal and then down loaded onto the device registers. The device is given a command and starts executing while software waits for a ready event from the device. The device can be used on the data multiple times for each access depending on the *data_type* signal. When the ready event is received by the software, the decoded data (*decoded_data_mem*) is read and formatted to the right output signal (*decoded_facch, decoded_uch*) and

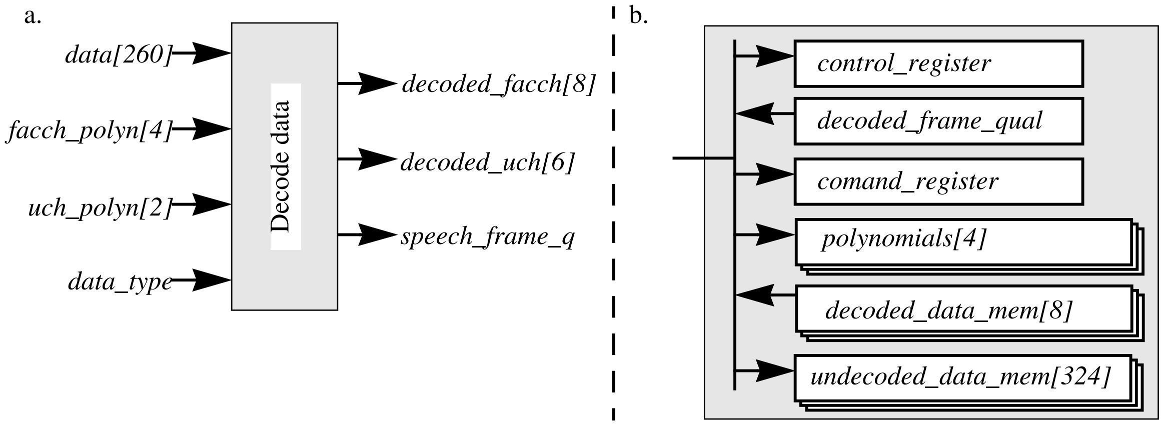


Figure 19. (a) Application (SW) interface for the decode data functionality (all signals are 16-bits wide). (b) Device (HW) interface to the decode circuit, registers and memory organized as 16-bit blocks.

the *decoded_frame_qual* register is copied to the *speech_frame_q* signal.

6.2. HW/SW communication of an ATM (F4) OAM block

Our second example implements the operation and maintenance (OAM) functions corresponding to the F4 level of the Asynchronous Transfer Mode (ATM) protocol layer [8]. This level handles OAM functionality concerning fault management, performance monitoring, fault localization, and activation/deactivation of functions. ATM is based on a fixed-size virtual circuit-oriented packet switching methodology. All ATM traffic is broken into a succession of cells. A cell consists of five bytes of header information and a 48 byte information field. The header field contains control information of the cell (identification, cell loss priority, routing and switching information).

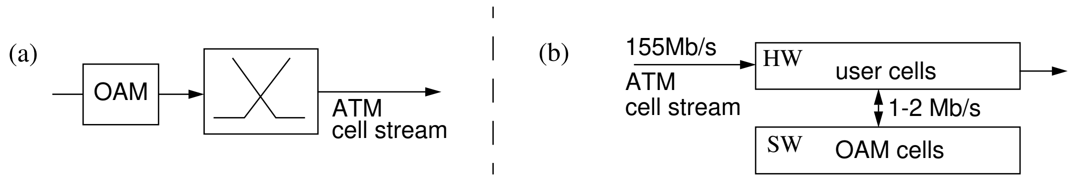


Figure 20. OAM block in an ATM network (a) and selected HW/SW partitioning (b).

The F4 level handles the OAM functionality concerning:

- Fault management: when the appearance of a fault is reported to the F4 block, special OAM cells will be generated and sent on all affected connections, if the fault persists the management system (MS) is notified.
- Performance monitoring: normal operation of the network is monitored by continuous or periodic checking of cell transmission.

- Fault localization: when a fault occurs it is necessary to localize it; for this purpose special loop back OAM cells are used.
- Activation/Deactivation: a special convention for activation and deactivation of OAM functions that require active participation of several F4 blocks, e.g. performance monitoring, has to be implemented.

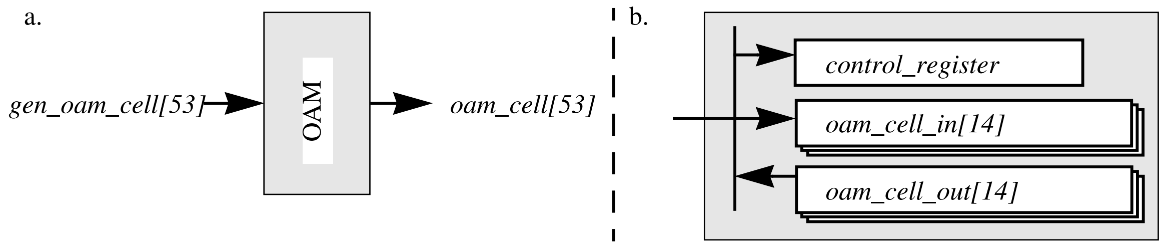


Figure 21. (a) Application (SW) interface for OAM unit (signals are 8 bits wide), (b) Device (HW) interface for the OAM unit (registers are 32 bit wide).

Figure 21 describes the interfaces of the protocol, the behaviour of the protocol is defined as follows. The protocol consist of two parts; one for reading OAM cells and one writing OAM cells from/to the OAM unit. The read part inputs a *gen_oam_cell* from software. Then it waits for the device to be ready to receive the frame. The data from software is then reformatted and written to the *oam_cell_in* device memory of the OAM unit. The write part reads the *oam_cell_out* device memory and reformats it into the *oam_cell* which is delivered to the software.

6.3. Results from software synthesis

The examples described above have been used to evaluate the system by comparing results generated in a traditional way writing C code and by using our approach starting from ProGram. Several experiments have been conducted with different combinations of the examples, real-time kernels and embedded processors. Ta-

<table>
<tr><th rowspan="2">#</th><th rowspan="2">behaviour</th><th rowspan="2">kernel</th><th rowspan="2">processor</th><th colspan="2">Modelling [LoC]</th><th colspan="3">Code generation time [sec.]</th></tr>
<tr><th>C</th><th>ProGram</th><th>ProGram to FSM</th><th>FSM to C</th><th>Total</th></tr>
<tr><td>1</td><td>MAX197</td><td>µCOS</td><td>ARM</td><td>75</td><td>20</td><td>0.09</td><td>0.90</td><td>2.18</td></tr>
<tr><td>2</td><td>MAX 530</td><td>µCOS</td><td>ARM</td><td>41</td><td>9</td><td>0.06</td><td>0.83</td><td>0.89</td></tr>
<tr><td>3</td><td>TL16552</td><td>µCOS</td><td>ARM</td><td>180</td><td>118</td><td>0.13</td><td>1.05</td><td>2.08</td></tr>
<tr><td>4</td><td>OAM unit</td><td>CREX</td><td>MC68000</td><td rowspan="2">75</td><td rowspan="2">45</td><td rowspan="2">0.12</td><td rowspan="2">1.00</td><td rowspan="2">1.12</td></tr>
<tr><td>5</td><td>OAM unit</td><td>µCOS</td><td>ARM</td></tr>
<tr><td>6</td><td>Channel decoder</td><td>CREX</td><td>MC68000</td><td rowspan="2">254</td><td rowspan="2">75</td><td rowspan="2">0.11</td><td rowspan="2">1.02</td><td rowspan="2">2.42</td></tr>
<tr><td>7</td><td>Channel decoder</td><td>µCOS</td><td>ARM</td></tr>
</table>

Table 6. Code size and code generation times for implementation examples used in figure 22 and 23.

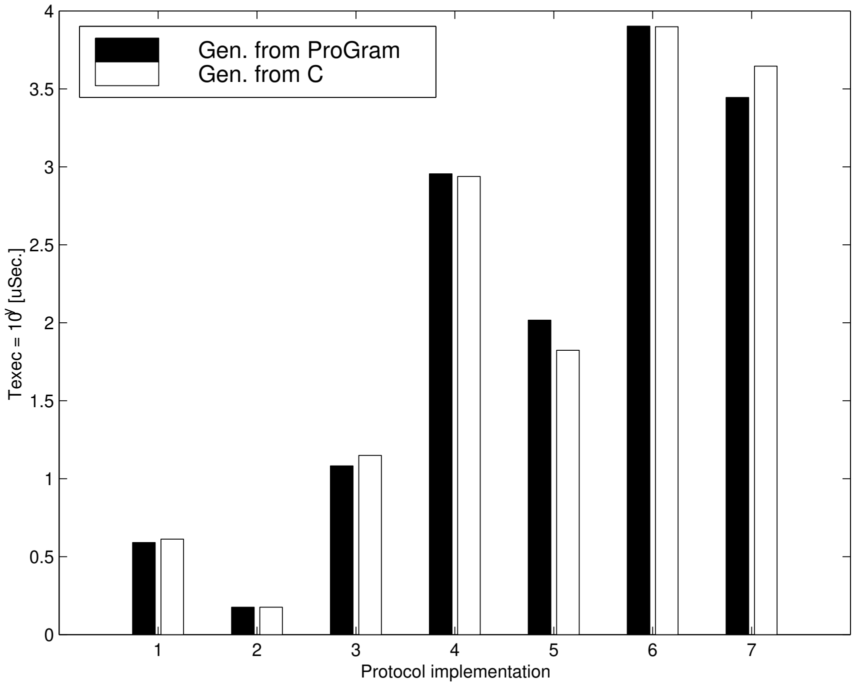


Figure 22. Performance for the different implementations described in table 6.

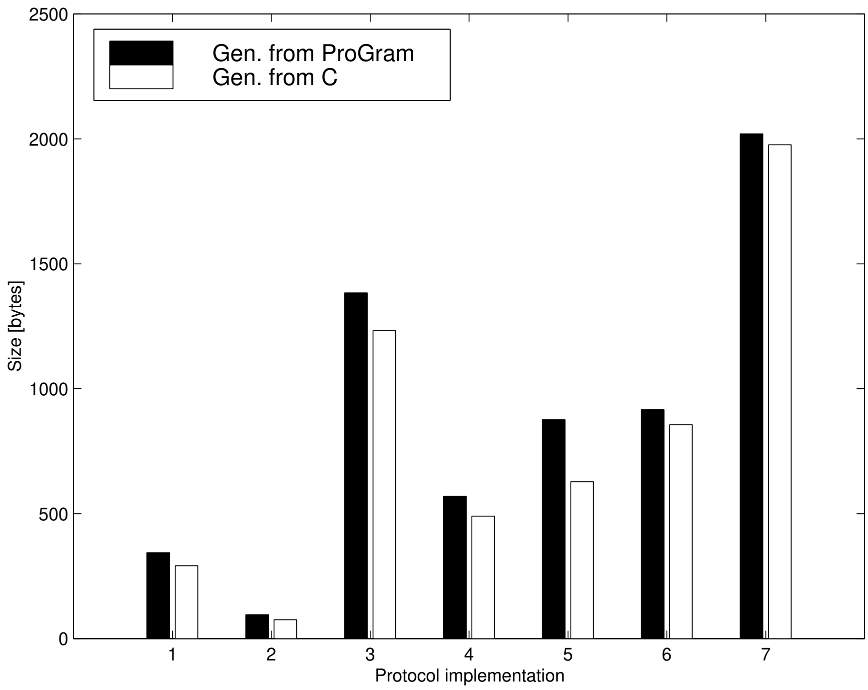


Figure 23. Code size in bytes for the different implementations described in table 6.

ble 6 lists the seven selected configurations. They contain two processors, MC68000 [22] and ARM7 [13], and two real-time kernels, μCOS [19] and CREX [15].

First column in table 6 indicates the implementation number used in figures 22 and 23. Second through fourth column indicate the configuration. A comparison of the lines of code (LoC) needed to model the behaviour in C and ProGram code is shown in column 5 and 6. Finally, the last three columns show the time in seconds to generate C code from ProGram on an HP 735 workstation. The code generation times show that implementations for different architectures can be generated in less than three seconds, thus enabling fast exploration of different architectures.

Figures 22 and 23 show a comparison of the performance and code size between implementations generated from hand written C code and from ProGram. These values are captured using the configurations described in table 7. Figures 22 and 23 show that code size and performance of device drivers generated from

Processor	Speed	Memory access time	Compiler/Simulator	Compiler directive
ARM	33 MHz	30 ns	ARM [1]	Optimize for speed
MC68000	16 MHz	125 ns	INTROL/CODE [15]	Optimize

Table 7. Compiler and simulator settings used to generate results.

ProGram is similar to those manually written in C. The code size numbers are generated by the cross-compilers used to generate the executable [1,15]. The code performance numbers are generated by taking a mean value from the execution of the device driver code (all functions except the initialization functions) with varying inputs. The presented performance values do not include the execution time for the real-time kernel.

6.4. Results from DMA synthesis

The components described above have been used to compose a set of protocols to evaluate the DMA controller synthesis by comparing results from the protocols modelled in hand written C code and ProGram. From the ProGram specification the tool generates C code and a DMA controller. The C code is implemented in software. The DMA controller is generated for an Z80 [32] micro processor.

Table 8 shows the synthesis results for the generated DMA controllers. The first column contains the name of the interfaced device; the second column describes the behaviour of the communication protocols. These protocol behaviours are composed of an access function and an interrupt routine. Tables 8 and 9 present results regarding the interrupt routine part of the protocol, i.e. each DMA controller corresponds to an interrupt routine. The examples are all of the type read or write blocks of data, either of blocking type (i.e. wait for data transfer to finish) or non-blocking type (no wait for the transfer to finish). Columns three through five present the synthesis results from a RTL synthesis tool for the LSI 10k technology. The third column presents the critical path for the generated controller. Columns four and five give the gate count for the generated part of the DMA controller and the whole DMA controller (i.e. the master interface from the library + generated controller), respectively.

Device	Synthesized behaviour	Critical path	Controller gate count	Total gate count
MAX197	1. Read buffer (x values at channel ch)[1]	10 ns	1141	2100
	2. Read buffer (x values at channel ch)[2]	10 ns	1156	2120
	3. Read all channels and store in buffer[1]	10 ns	999	1970
	4. Read all channels and store in buffer[2]	8 ns	1018	1980
MAX530	5. Write x values from a buffer[1]	12 ns	1086	2150
	6. Write x values from a buffer[2]	12 ns	1101	2170
16552	7. Ring buffers (Rx buffer & Tx buffer)	14 ns	1997	2960

Table 8. Synthesis results for the generated DMA controller.

Table 9 compares the performance and bus usage between a pure software solution and the hardware solution with an application specific DMA controller. The first column contains the name of the interfaced device; the second column gives a reference to the behaviour description in table 8. Columns four and six give the performance (calculated as $performance = 1/t_{exec}$) numbers for two different configurations, software and DMA controller, respectively. Columns three and five present the numbers for system bus usage normalized to the all-software solution, this is calculated as $usage_{HW} = mem_{access}(HW)/mem_{access}(SW)$. The sixth column shows the speed-up for a hardware solution compared to a software solution, calculated as $Speedup = t_{exec}(SW)/t_{exec}(HW)$. This illustrates that the hardware solution results in 8 to 20 times higher performance. But in many cases more important is the reduced bus utilization by a factor between 10 to 20, because the processor bus is very often a scarce and highly utilized resource.

From these results we can see that the controller is quite fast (above 70 MHz) even for such an old technology as LSI 10k. The most likely application area for this is a system on a chip (SoC), but the DMA controller will work as good as a separate device on a board. Gate count for a system on a chip can be > 1M gates. Thus with our approach, we achieve a 20 times speed-up of the communication performance for a fraction of the cost of the total system. This makes the application specific DMA controller solution an appropriate choice if communication performance is a bottleneck.

7. Conclusion

We have presented an approach to hardware/software communication synthesis. Synthesis is done from an

Device	Behaviour	Pure software		With generated DMA controller		Speedup
		Bus usage[3]	Performance[4]	Bus usage[3]	Performance[4]	
MAX197	1	100 %	$2.9{\cdot}10^4$ s^{-1}	4.7 %	$5.6{\cdot}10^5$ s^{-1}	19.4
	2	100 %	$2.9{\cdot}10^4$ s^{-1}	4.9 %	$5.3{\cdot}10^5$ s^{-1}	18.4
	3	100 %	$3.0{\cdot}10^4$ s^{-1}	4.9 %	$5.6{\cdot}10^5$ s^{-1}	18.7
	4	100 %	$2.9{\cdot}10^4$ s^{-1}	8.5 %	$2.5{\cdot}10^5$ s^{-1}	8.6
MAX530	5	100 %	$4.2{\cdot}10^4$ s^{-1}	5.1 %	$6.7{\cdot}10^5$ s^{-1}	16.0
	6	100 %	$4.2{\cdot}10^4$ s^{-1}	5.4 %	$6.1{\cdot}10^5$ s^{-1}	14.5
TL16552	7	100 %	$2.9{\cdot}10^3$ s^{-1}	8.6 %	$4.4{\cdot}10^4$ s^{-1}	15.2

Table 9. Performance comparison between pure software implementation and an implementation with software generated from ProGram combined with a DMA controller generated from ProGram.

architectural and implementation independent description of a device access protocol. The protocol description can be mapped, according to a designer decision, to a pure software device driver implementation or a mixed hardware/software device driver using a DMA controller to speed-up the data transfer. The synthesis procedure maps the generated code to fit the selected real-time kernel and processor. For parts selected to be implemented as a DMA controller the synthesis procedure generates RTL VHDL code. The generated DMA controller is connected to the processor bus via a library interface. The synthesis procedures have been tested with several realistic examples.

Our approach enables a designer to *plug-and-play* with IP components without having to go through the tedious work of writing the interface code for the different components. Instead, the designer just lets the tool generate the interface code for the selected architecture and configuration, which takes less than 3 seconds for the presented examples. The components are accessed through the generated API. This way the designer can evaluate different architectures and configurations, without spending time on rewriting the interface code. Another major benefit is the ease of design maintenance, i.e. the designer does not need to redo the interface parts for new design generations. All implementations in our approach are generated from one protocol description, regardless of kernel, processor and DMA controller. Thus, the probability of design errors are decreased since a protocol is only captured once as opposed to other approaches where the same protocol needs to be captured several times. Hence, implementations for different architectures will be consistent with each other.

Acknowledgements

This work has been supported by Mid Sweden University, the Swedish National Board for Industrial and Technical Development - NUTEK, and the Swedish Foundation for Strategic Research SSF.

References

1. ARM Software Development Toolkit - Reference Manual, Version 2.0, Advanced RISC Machines Ltd., 1995.
2. F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A, Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara, Hardware-Software Co-Design of Embedded Systems: The Polis Approach., Kluwer Academic Press, 1997.
3. B. Bohem, B. Clark. E. Horowitz, C. Westland, R. Madachy, R. Selby, "Cost models for future software life cycle processes: COCOMO 2.0", Annals of Software Engineering, Vol. 1, pp. 57-94, 1995.
4. I. Bolsens, H. J. De Man, B. Linn, K. van Rompaey, S. Vercauteren, D. Verkest, "Hardware/Software Co-Design of Digital Telecommunication Systems", Proceedings of the IEEE, Vol. 85, No. 3, pp. 391-418, 1997.
5. P. H. Chou, R. B. Ortega, G. Borriello, "The Chinook Hardware/Software Co-Synthesis System", Proceedings of the International Symposium on System Synthesis, 1995.
6. P. H. Chou, R. B. Ortega, G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems", Proceedings of the International Conference on Computer Aided Design, pp. 488-495, 1992.
7. J-M Daveau, G. F. Marchioro, T. Ben-Ismail, A. A. Jerraya, "Protocol selection and interface generation for hw-sw codesign", IEEE Transaction on Very Large Scale Integration, Vol. 5, No. 1, pp. 136-144, 1997.
8. M. De Prycker, Asynchronous Transfer Mode, Prentice Hall, 1995.
9. J. C. Días, J. Riesco, P. Plaza, "Design of an ARM based System-on-a-Chip for Pay Phones", Proceedings of the International Workshop on IP Based Synthesis and System Design, pp. 101-105, 1998.
10. EIA/TIA Interim Standard, Cellular System Dual-Mode Mobile Station - Base Station Compatibility Standard, IS-54-B, April, 1992.
11. M. Eisenring, J. Teich, "Domain-Specific Interface Generation from Dataflow Specifications", Proceedings of the 6th International Workshop on Hardware/Software Codesign, pp. 43-47, 1998.
12. R. Ernst, Th. Benner, "Communication, Constraints and User Directives in COSYMA", Technical Report CY-94-2, Technische Universität Braunschweig, June 1994
13. S. Furber, ARM System Architecture, Addison Wesley Longman, ISBN 0-201-40352-8, 1996.
14. R. Grehan, "Driver Assistance", Computer design, vol. 36, no. 1, pp. 75-80, 1997.
15. INTROL/CODE - Reference Manual, Introl Inc., Milwaukee, WI 53202 USA, 1999.
16. D. C. R. Jensen, J. Madsen, S. Pedersen, "The Importance of Interfaces: A HW/SW Codesign Case Study", Proceedings of 5th International Workshop on Codesign, 1997.
17. A. A. Jerraya, K. O'Brien, "SOLAR: An Intermediate Format for System-Level Modeling and Synthesis", Computer Aided Software/Hardware Engineering, Ed. J. Rozenblit, IEEE Publisher, chap. 10, 1994.
18. S. C. Johnsson, Yet another compiler compiler, Computing Science Tech. Rep. 32, AT&T Bell Lab. Murray Hill, 1975.
19. J. J. Labrosse, mC/OS - The Real-Time Kernel, R&D Publications, Lawrence, Kansas 66046, 1992.
20. MAX 197 - Multi-Range 12-bit ADC, Data sheet, Maxim Integrated Products, CA.
21. MAX 530 - Multi-Range 12-bit DAC, Data sheet, Maxim Integrated Products, CA.
22. MC68000 Family Reference Manual. Motorola Inc., 1990.
23. R. Niemann, P. Marwedel, "Synthesis of Communicating Controllers for Concurrent Hardware/Software Systems", Proceedings of Design automation and Test in Europe, pp. 912-913, 1998.
24. M. O'Nils, J. Öberg, A. Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", Proceedings of Euromicro Conference, pp. 55-58, 1998.
25. M. O'Nils, A. Jantsch, "Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models", Proceedings of IEEE Workshop on High Level Design, Validation and Test, pp. 32-40, 1998.
26. R. B. Ortega, G. Borriello, "Communication Synthesis for Distributed Embedded Systems", Proceedings of the International Conference on Computer Aided Design, 1998.
27. R. B. Ortega, L. Lavagno, G. Borriello, "Models and methods for hw/sw intellectual property interfacing", NATO ASI on System-level Synthesis, 1998.
28. A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, J. Buck, "A System for Compiling and Debugging Structured Data Processing Controllers", Proceedings of the European Design Automation Conference, 1996.
29. TL16552 - Dual Asynchronous Communications Element with FIFO, Data sheet, Texas Instruments Inc., 1996.
30. E. Tuggle, "Writing Device Drivers", Embedded Systems Programming, Jan. 1993, pp. 42-65.
31. F. Vahid, L. Tauro, "An Object-Oriented Communication Library for Hardware-Software CoDesign", Proceed-

ings of 5th International Workshop on Hardware/Software Codesign, 1997.
32. Z80 - Microprocessor Family User's Manual, Zilog Inc., 1994.
33. J. Öberg, A. Kumar, A. Hemani, "Grammar-based Hardware Synthesis of Data Communication Protocols", Proceedings of the 9th International Symposium on System Synthesis, pp. 14-19, 1996.

Footnotes

1. Non-blocking communication.
2. Blocking communication.
3. Bus usage normalized the pure software implementation.
4. Maximum number of interrupts served per second.

Paper 6

Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models

6

Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models

Mattias O'Nils, Axel Jantsch

Royal Institute of Technology
Department of Electronics, Electronic System Design Lab.
Electrum 229, S-164 40 Kista, Sweden
Email: {mattias,axel}@ele.kth.se

Abstract

Hardware/Software interfaces are a critical point for system simulation because the hardware and software parts are developed by separate development activities. Our multi-phase validation approach facilitates the simulation of the interfaces at several steps during development. We keep all the simulation models consistent with both the specification and the implementation by generating the models with a technique derived from interface synthesis, that enables the interface specification, simulation models, and implementation to be consistent with each other. We justify validation in several phases by (a) the up to four orders of magnitude faster simulation of early phases compared to late phases, and by (b) allowing both HW designer and SW developers to work in their familiar tool environment as long as possible.

1 Introduction

Heterogeneous hardware/software systems are often verified at several different levels. Figure 1 identifies four levels of HW/SW cosimulation, which differ in abstraction, simulation performance, and accuracy. Simulation of functional system descriptions (figure 1a) captures only the system functionality without any implementation details, e.g. SDL [12]. While cosimulation of the system for the target processor (figure 1d) captures most of the system functionality (i.e. very consistent with the implementation) but with a decrease of simulation performance. Cosimulation tools can also be used to connect the software part of the system executing on the host system (figure 1c) to a hardware simulator. This increases the performance by decreasing the level of details. Another way to further increase the simulation performance is to remove the hardware/software communication [8], as illustrated in figure 1b. For software development, simulation and debugging we do not need to simulate the entire connected hardware. What we need is a model of the hardware device, accurate enough so that the software does not notice the difference. Similarly, for simulating the hardware we do not need to include the entire processor and software into the simulation. But we need a model of the software device drivers, which is accurate enough so that the hardware does not notice the difference. This is accomplished for both sides in the HW/SW stub simulation (figure 1b). In fact, it includes two separate simulations, each containing a stub-interface hiding the non-existing second part. This level is suited for perhaps 90% of the separate hardware and software development, which does not rely on the full and exact behaviour of the other part. But short simulation time is critical for fast code-simulate-debug cycles for both hardware and software development.

A main problem throughout the validation flow in figure 1 is to keep the hardware/software communication consistent with the implementation, because only a high consistency of the simulation models with the final implementation gives value to the validation steps. Traditionally, the hardware and software stub models, device drivers for host code cosimulation, and device drivers for the target system are all independently derived from the specification.

We address this consistency problem by extending our approach to device driver synthesis [9] to also handle the generation of hardware and software stubs for HW/SW stub simulation and device drivers for host code simulation. That is, the hardware/software communication in all three validation levels (b to d in figure 1), will be generated from the same formal specification and thus, the simulations behaviour will be consistent with both the implementation and the specification. This is illustrated in figure 2, where the dashed line represents traditional approaches, if simulations at these intermediate steps are done at all. The bold line represents our approach. In essence, at all three

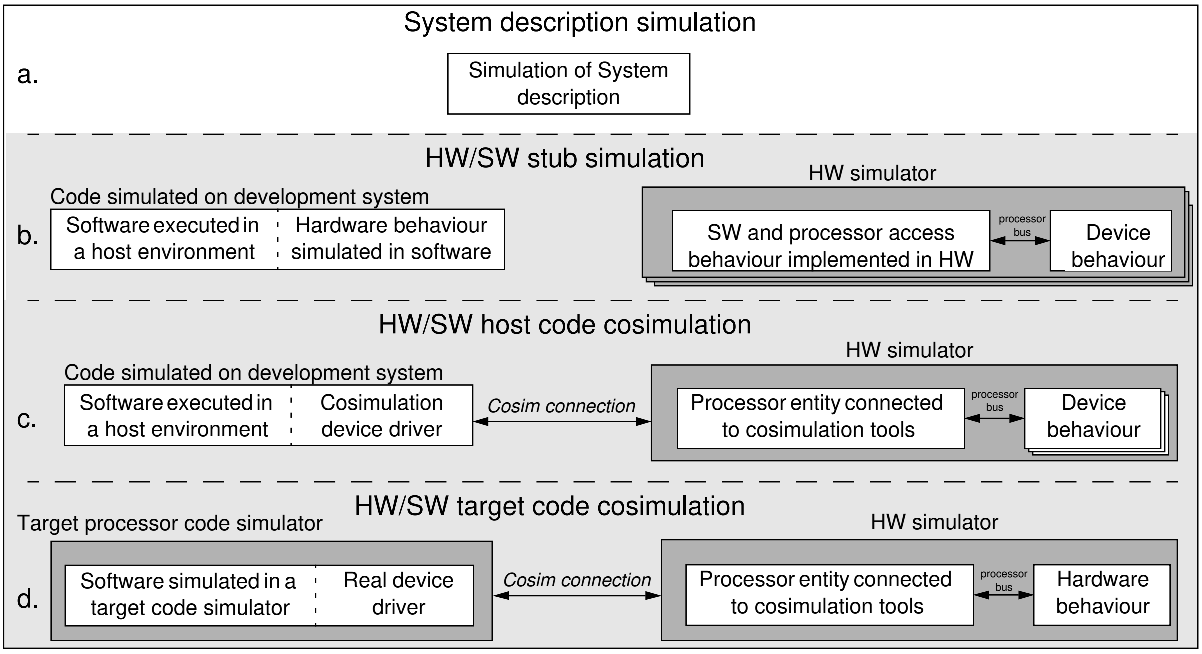


FIGURE 1. Simulation levels in embedded hardware/software systems development. (a) System description simulation, (b) HW/SW stub simulation, (c) HW/SW host code cosimulation, and (d) target code cosimulation.

discussed validation phases can we increase model consistency significantly without sacrificing simulation speed. The layered validation technique is in it self not unique, on the contrary, these techniques are well established [3,8,13] among designers. The contribution is the generation of hardware/software simulation models, thus decrease the development effort and increase the accuracy of the validation phase.

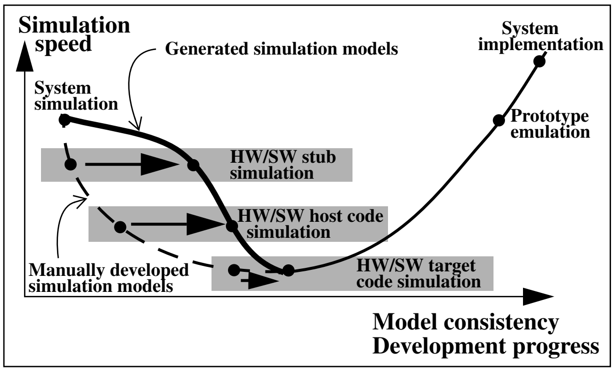


FIGURE 2. Simulation speed/model consistency trade-off in cosimulation

The next section discusses the related work. Then we present the proposed design and validation flow for device drivers. Sections four through six present the techniques for generation of the different models. In the seventh section we illustrate the different techniques with an example.

2 Related work

The latest commercial advances in validation of hardware/software systems are cosimulation tools, e.g., Seamless [13] and Eagle tools [3]. These tools connect software simulators or programs with a HDL or gate-level simulator. But these tools are not enough for validation of the hardware/software interfaces, due to the bad performance. Higher performance is achieved with these tools if the host code cosimulation option is used, but this level need a different interface then the implementation, which decrease the accuracy.

CoWare [15] and Polis [1] target design of hardware/software systems but which concentrate on the case where the whole design functionality is captured within their environment. During the system synthesis the communication is refined, i.e. the device drivers are generated together with the custom HW and operating system. But if the user wants to use IP blocks and an off-the-shelf RTOS, he/she will face the same troubles as for manual design [4,14]. CoWare offers cosimulation capabilities similar to Seamless [13] and Eagle [3]. For both CoWare and Polis the designer has to write their own verification models for levels (b) and (c) in figure 1. Thus, the simulated interfaces are not the same as for implementation.

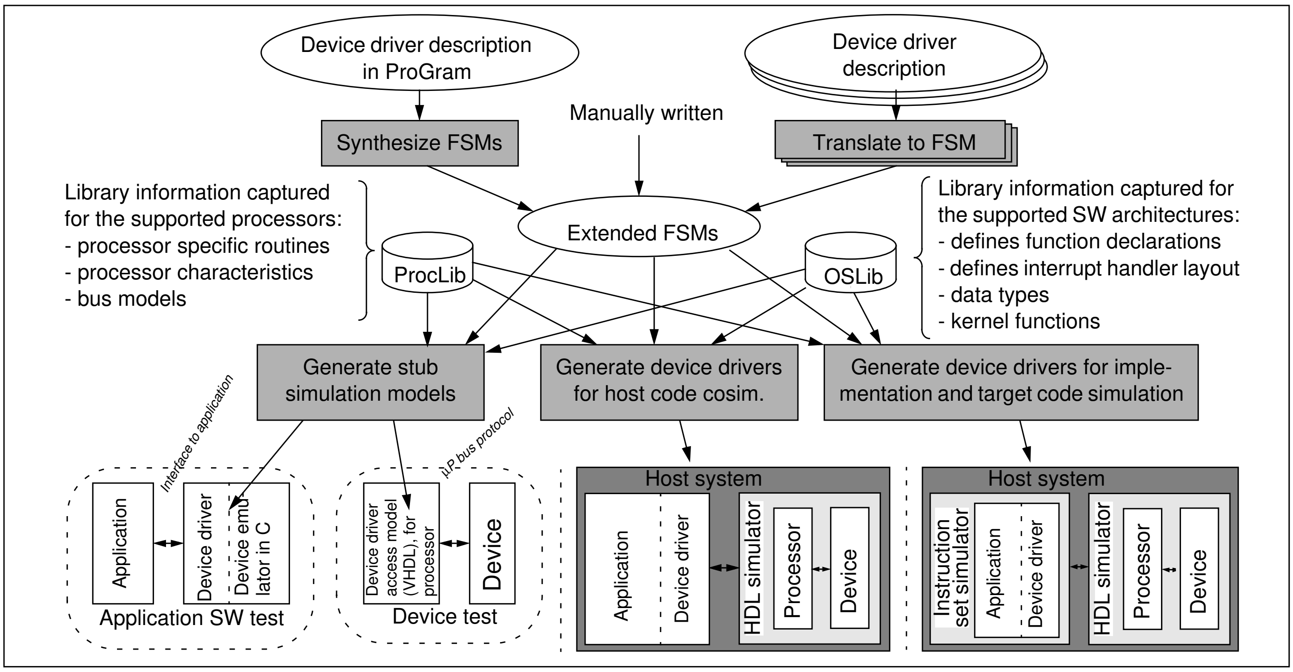


FIGURE 3. Generation HW/SW communication modules.

3 ComSyn - HW/SW interface synthesis

As described in [9], we model device drivers independently of the architecture using ProGram [16]. ProGram is a grammar based notation for protocol applications. Specifications in ProGram deal with sequences of allowed events as opposed to states and state transitions in an FSM model.

The ProGram description is synthesised into an untimed extended state machine (described in [9,16]), which in turn is the input to the architecture mapping procedure described in section 5. The mapping procedure uses data from two libraries to generate the architecture specific code. The first of these libraries is the library that captures the information on the OS architecture, the second captures the processor specific characteristics, see figure 3.

4 Target code cosimulation

In target code cosimulation the whole system is modelled in the hardware simulator with the processor entity, which is connected to a processor simulator. Thus, the device driver used in target code cosimulation is exactly the same as the one used for implementation, with respect to timing and bus behaviour. So the device driver generated for implementation is also used for target code cosimulation.

Device drivers for implementation and target code cosimulation is generated from FSMs (generated from ProGram) that capture the behaviour and two libraries that model the OS kernel (OSLib) and processor (ProcLib), see figure 3.

For all variables
    Map bit vector types onto data types in OSLib
For all *FSMs*
Begin
    Apply transformation rules to all states in *FSM*
    Map OS kernel specific parts of *FSM*
    Map processor specific parts of *FSM*
    case (type of *FSM*)
        ACCESS_FUNCTION : generate_function(*FSM*)
        INT_HANDLER : generate_int_handler(*FSM*)
        TASK : generate_task_function(*FSM*)
End

FIGURE 4. Synthesis procedure of device drivers.

The synthesis procedure starts with mapping of bit vector types to data types used by the OS kernel (defined in OSLib). Then the state machine is transformed to software implementation by applying a set of transformation rules on each state. The architecture dependent parts of the transformed FSMs are then mapped onto the selected architecture using the libraries OSLib and ProcLib. Depending on the type of FSM description different code structures are generated, see figure 4.

5 Host code cosimulation

Device drivers for host code simulation have to communicate with the hardware via a hardware simulator instead of directly accessing the hardware with pointer accesses. This is achieved by replacing the access to registers with cosimulation access functions.

There are two possible cases of host code cosimulation: (1) single process execution and (2) kernel emulation. For the first case, the software is implemented using a single process on a workstation, i.e. all parallel constructs have to be serialized. For the second case, software is executed with the target OS kernel that is emulated on the host (workstation), i.e. the interface to the application and OS kernel will be the same as for implementation but the interface to the device will be different. In this paper we only consider the second case, emulation of the OS kernel. Host code cosimulation will have higher performance than host code simulation, but at the cost of software timing and lower bus behaviour accuracy.

Code generation for host code cosimulation follows the same procedure as for target code cosimulation (implementation) in figure 4, except for the behaviour of the mapping of variables and generation of interrupt routines.

5.1 Mapping of variables

In the synthesis for implementation the mapping of types for ports to hardware, ports to software and internal variables is treated in the same way. For host code cosimulation the mapping of types for the ports to the device has to be treated differently, since these have to be replaced with accesses to the hardware simulator.

Implementation

```
UINT8 *TRX;*LCR;
...
setup = *LCR;
*TRX = data;
```

Simulation

```
HCS8 TRX=0x200;
HCS8 LCR=0x201;
...
setup = *LCR;
*TRX = data;
```

FIGURE 5. Implementation of device register access in host code cosimulation.

Supplying a host code cosimulation access class written in C++ similar to the one use in [13] solve this. In this class there will an 8,16 and 32 bit type for accessing the memory space of the embedded system. Variables declared with this type are accessed with the star and type assignment operators in the same way as in the implementation, as seen in figure 5. Thus, the code generation for host code cosimulation can be the same as for implementation except for the declaration of the ports. The ports are instead declared with the host code cosimulation class, and the port address is assigned in the declaration. The body code for the operators for the class has to be supplied for each simulator.

5.2 Generation of interrupt handler

Since in a host system interrupts cannot be generated from the hardware simulator to the workstation, interrupts have to be treated as a special case in host code cosimulation.

The way we have solved this is to generate a separate task that receives the interrupt events from the hardware simulator. The task is generated as an infinite loop with a wait on interrupt library instruction (unique for each simulator) that also identifies what interrupt was asserted so the appropriate interrupt routine can be called.

6 Stub simulation

For stub simulation two stubs (models) are generated, one for software simulation and another for hardware simulation. As for host code cosimulation, stub simulation can be done as (1) a single process or (2) with the OS kernel emulated in the host system. In this paper we only consider the second case.

The stub for software simulation emulates the hardware behaviour by replacing accesses to read-only register with function calls, which capture the register behaviour. The designer selects the register behaviour from a set of functions (static, random or reading values from a file) or writes the function manually. Write and read/write registers are replaced by memory accesses to a memory location in the host system

The stub for hardware simulation emulates the software behaviour by generating the device driver in VHDL. This device driver is connected to a bus model of the selected processor, i.e. the generated code together with the bus model forms a testbench for the device under test.

Stub simulation lacks the synchronization between hardware and software behaviour, thus decreasing the simulation accuracy since the behaviour is emulated in either hardware or software. But increasing the simulation performance (for software simulation) with maintained interfaces between device driver and application/kernel for software simulation and between device and device driver for hardware simulation.

6.1 Generation of stubs modelling the HW behaviour for SW simulation

Code generation of stubs for software simulation follows the same procedure as for target code cosimulation (implementation) in figure 4, except for the parts for mapping of variables and generation of interrupt routines.

6.1.1 Handling of device registers

From the device driver perspective the device is just a set of registers and vectors. There are three types of registers: read only registers (ROR), write only registers (WOR) and read/write registers (RWR). The WOR and RWR registers have the same behaviour as a memory, i.e. it has the ability to store data and recall stored data. Since registers declared as WOR will not be read and RWR will contain same data as previously written. If a register at a specified address can be written and read, but the read information differs from the one written, then it should be declared as one ROR register and one WOR register with the same address. Hence, from the device driver perspective the device behaviour is captured by the ROR registers, see figure 6.

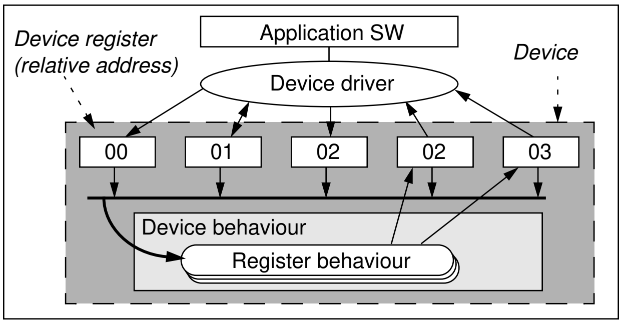


FIGURE 6. Simulation of device behaviour for software execution.

RWR and WOR registers are implemented with the generic C++ class `RWR`. Objects declared with the class `RWR` are accessed the same way as the device registers are accessed in the implementation, similar to host code cosimulation class in figure 5. The `RWR` class's only behaviour is to store and recall values of a desired type, i.e. a `*` and = operators.

The device behaviour is divided into several behaviours each handling a ROR register. There are several ways to capture the behaviour of a device. To capture the full behaviour, a function for each ROR register should be supplied. If the exact behaviour of a device register is not required, there are three ways to generate the read values: constant value, random values, or a series of values read from a file. ROR registers are implemented as objects of a generic C++ class `ROR`. This class is a library module that captures all the behaviours described above: static, random, file, and user defined. Values are read in the same way as a ROR register is read in the implementation, see figure 5. The behaviour for a register object declared with the `ROR` class is interactively selected by using different constructors. For user defined behaviour, a function defining the register behaviour has to be written.

6.1.2 Handling of interrupt routines

An interrupt routine executes asynchronously with respect to the access functions of a device driver and is triggered by hardware generated events. Since there are no events generated by hardware, the events are emulated with delays. The interrupt routines are activated within a task. Within the task there is an infinite loop, which has a wait-for-time in it. The type of delay is interactively selected as constant delay, random delay, or a series of delays read from a file. After each delay, there is a function call to the interrupt routine, i.e., same interrupt routine as generated for implementation.

6.2 Generation of stubs modelling the SW access behaviour for HW simulation

The libraries used for device driver synthesis presented in figure 3, capture the characteristics of the processor and the kernel. In line with this we can define the generation of the stub for hardware simulation as an implementation target, i.e. describe the target by means of the libraries.

6.2.1 Library modules

The processor library (ProcLib in figure 3) captures the characteristics of a processor. But it does not capture all required information for the generation of device testbenches. To this end a new entry in the ProcLib library describes the generation of a VHDL entity, in the following called the *master interface*. On one hand it implements the processor specific bus interface, and on the other hand it implements the master interface protocol to communicate with the generated VHDL stub models of the device drivers, i.e. code generation is done for a fixed protocol.

The master interface components use a 2-phase request/acknowledge protocol. The protocol is synchronized with the bus clock of the processor bus. The device is accessed through two procedures, `read_data` and `write_data`. These read/write data from/to the device at a specified address by interfacing the master interface module. The address and data widths are defined as generics in the generated model.

Since the testbench is implemented in VHDL, the OS-Lib (figure 3) library structure is not suited for capturing the necessary information about the VHDL target. But the only behaviour of a kernel that needs to be emulated with VHDL is the handling of semaphores. For this we use

```
entity sim_comp is
   generic (ADDR_BUS_WIDTH,DATA_BUS_WIDTH);
   port(Master interface + Interrupt signals);
end sim_comp;

architecture user of sim_comp is
   procedure read_data(in adr,out data);
   procedure write_data(in adr,out data);
   procedure set_semaphore(inout semaphore);
   procedure pending_semaphore(inout semaphore);
   -- Declaration of semaphores as pair of boolean signals
   -- Declaration of address pointers to device registers as shared variables [5]
   -- Declaration of generated procedures for accessing the device.
   -- Declaration of interrupt routine procedures
begin
   SW_EXECUTION: process
     -- Declaration of user variables
   begin
     -- User defined simulation behaviour using the generated
     -- procedures or for accessing the device.
   end process SW_EXECUTION;
   -- Interrupt routines procedures instantiated
   end architecture user;
```

Fixed code

Generated code

Code written by user

FIGURE 7. Structure of the generated VHDL simulation model.

shared variable capability of VHDL-93 [5] to implement binary semaphore. If there would be a need for counting semaphores, techniques for shared variables presented by Madsen et al. [7] have to be used.

6.2.2 Structure of the generated VHDL testbench

Since the VHDL stub for hardware simulation should work as the device driver generated for the real systems, each state machine is translated into a procedure. Access and initialisation functions are to be used in a sequential process. Interrupt routines execute in parallel to the sequential process, see figure 7. The entity of the stub for hardware simulation is the interface defined by the master interface together with generics that define the address and data width. In the declaration part of the architecture the library procedures for accessing the device through the master interface and procedures for handling of the semaphores are declared. After this follows the declaration of the generated parts of the code, i.e. declaration of semaphores and address pointers to device registers (address pointers are implemented as shared variables [5]) and declaration of the generated procedures for access/initialization and interrupt routines. The active part of the architecture has one process where the user defines the software behaviour, i.e. how the initialization and access procedures should be executed. The interrupt routines are instantiated as concurrent procedures to the process executing the user-defined behaviour, see figure 12.

6.2.3 Generating code for the entries

The generation of the procedures is similar to the generation of C functions for implementation as device drivers. Signals declared as internal signals can be accessed both by interrupt routines and by the access functions but with only one driver. To ensure the consistency with software behaviour, i.e. each signal assignment should be valid at the next statement, the internal signals have to be treated as variables within the software execution process and the interrupt handler procedure. This is achieved by using shared variable capability of VHDL-93 [5]. Code for the state machine used in the VHDL procedures in figure 7 are generated in a similar way as for implementation. All read and write operations of device registers are replaced with calls to the `read_data` and `write_data` procedures.

7 Example to illustrate the approach

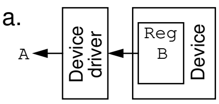
To further explain the generation of the different models we provide a schematic example of a simple read function, where a value is read from a register (`B`) and presented as a out value (`A`) to the application. Figure 8 (a) shows the information flow and (b) shows the corresponding ProGram code.

Interfaced device	Host code cosimulation	Generated stubs for hardware/software simulation			
		HW	SW constant	SW random	SW file
MAX197	479	306	$2.2 \cdot 10^6$	$1.5 \cdot 10^6$	$1.3 \cdot 10^6$
MAX530	1039	476	$13.8 \cdot 10^6$	$13.8 \cdot 10^6$	$13.8 \cdot 10^6$
UART, 16552	280	164	$2.2 \cdot 10^6$	$2.2 \cdot 10^6$	$2.2 \cdot 10^6$
CD interface	630	302	$4.9 \cdot 10^6$	$3.6 \cdot 10^6$	$1.8 \cdot 10^6$
DS12887	647	408	$5.0 \cdot 10^6$	$3.4 \cdot 10^6$	$3.3 \cdot 10^6$
ICM7218	654	150	$4.0 \cdot 10^6$	$4.0 \cdot 10^6$	$4.0 \cdot 10^6$

TABLE 1. Number of device driver accesses per second on a Sparc Ultra 1, 167 MHz.

Figure 8 (c-e) shows the generated C code for the different levels: (c) target code cosimulation, (d) host code cosimulation and (e) stub for software simulation. For these, the access function (`read`) looks exactly the same for all three levels, the only thing that differ is declaration and address initialization of the device register. Figure 8f shows the access function (stub) generated for hardware simulation. In figure 8f within the read function we see that data is read from the device using the library function `read_data`. `N` in figure 8 f is the number of address bits for the selected processor.

a.



b.

```
%output A [bit]8 sw
%input B [bit]8 hw 01
%start read
read: {A=B;};
```

c.

```
byte8 *B;
void i(void *base) {
    B = base + 1;}
void read(byte8 *A){
    *A = *B;}
```

d.

```
HCS8 B;
void i(void *base) {
  B = (int) base + 1;}
void read(byte8 *A) {
    *A = *B;}
```

e.

```
ROR B("b.dat");
void read(byte8 *A){
    *A = *B;}
```

f.

```
shared variable B:unsigned(N-1 downto 0);
procedure i(variable base:in unsigned(N-1 downto 0) is
begin   B = base + 1; end procedure init;
procedure read(variable A:out unsigned(7 downto 0) is
    variable B_tmp : unsigned(7 downto 0);
begin
    read_data(B,B_tmp); A := B_tmp;
end procedure read;
```

FIGURE 8. Example of code generation.

8 Results

A number of representative device driver examples have been used to evaluate the synthesis system [9]. The same examples have been used to test and evaluate the generation methods presented in this paper. Note that the size of the overall designs, in terms of the number of lines of code, is not an important factor. The feasibility of implementations and the possibility to generate simulation models for different classes of devices are more important considerations. Table 1 compares the simulation performance for 5 different ways to model and simulate hardware/ software communication. In the first column is the name of the interfaced device; the second column gives the performance for host code simulation with Seamless [13] (device is simulated in an HDL simulator [11]). In the columns three through six the performance numbers for different configurations of hardware/software simulation with stubs are shown. The third column presents results for the approach of section 6.2, i.e. the device driver is implemented in VHDL and is connected to the hardware via a processor bus protocol interface, the master interface. Columns four to six presents the performance numbers for the software simulation with the device behaviour as a stub simulated in software, see section 6.1. Register behaviour is modelled as a constant value in column 4, as a random value in column 5, and by reading values from a file in column 6.

The reason why software stub-simulation of MAX530, ICM7218 and UART 16552 results in the same values for all three configurations (column 4-6) is, that they do not have ROR registers in the measured functions. The difference in performance between the host code cosimulation and hardware stub simulation is due to the greater number of events generated to simulate the hardware stub. This difference will decrease for larger devices since the simulation of the stub will be a smaller fraction of the whole simulation.

9 Conclusion

We present an approach for generation of both imple-

Characteristic	Hand written HW & SW stub sim.		Generated HW/SW stub sim.		Hand written host code cosimulation	Generated host code cosimulation	Target code cosimulation	Prototype
	SW	HW	SW	HW				
Need to simulate the full design	no		no		no	no	yes	yes
Effort to setup a simulation[a]	20		1		20	1	200	>600
Simulation performance	1	10^{-4}	1	10^{-4}	10^{-4}	10^{-4}	$<10^{-4}$	1
Interface to SW/HW consistent with implementation	no		yes		no	yes	yes	yes
The simulated system consistent with implementation	no		no		no	yes	yes	yes

TABLE 2. Comparison of the capability for different simulation methods.

a. Estimated values for a medium sized device driver (~100 LoC) based on results from table 1 and programmer efficiency models [2], normalized to the generated stub simulation.

mentation, and simulation models from an architectural and implementation independent description. This approach guarantees that the implementation and simulation will be consistent with the specification. It also reduces the design and validation effort for the hardware/software communication, see table 2.

For generated stub simulation we achieve up to four orders of magnitude faster simulation of the software compared to host code cosimulation, see table 1. An important benefit is that software designers need only to handle software tools and hardware designers need only to handle hardware tools. Hence, these designers are not forced to control the other worlds design tools and do not need to handle and set-up the cosimulation software. This is a tremendous advantage with today's complex design tools and the even more complex task of interfacing tools.

10 Reference

[1] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press , June 1997.

[2] Boehm B. et al., "Cost models for future software life cycle processes: COCOMO 2.0", *Annals-of-Software-Engineering*, vol. 1, 1995, pp.57-94.

[3] Eagle Tools, Synopsys, http://www.viewlogic.com/products/eagletools.html.

[4] Shaul Gal-Oz, Avi Cohen, "The Hazards of Device Driver Design", *Embedded Systems Programming*, May 1997, pp. 34-46.

[5] *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076-1993.

[6] S.C. Johnsson, *Yet another compiler compiler*, Computing Science Tech. Rep. 32, AT&T Bell Lab., Murray Hill, 1975.

[7] Jan Madsen, Jens P. Brage, "Modeling Shared Variables in VHDL", In *Proceedings of the European Design Automation Conference with Euro-VHDL'94*.

[8] Larry Mittag, "Device Drivers for Nonexistent Devices", *Embedded System Programming*, Aug. 01 1996, vol. 9, no. 8, pp. 30-32,34,36,38,40.

[9] Mattias O'Nils, Johnny Öberg, Axel Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", in the *Proceedings of the Euromicro Workshop on Digital System Design*, 1998.

[10] M. De Prycker, *Asynchronous Transfer Mode*, Prentice Hall, 1995.

[11] *QuickHDL User's and Reference Manual*, Ver. 8.5_4, Mentor Graphics Inc., 1997.

[12] *SDT User's Guide*, TeleLOGIC AB, Malmö, Sweden, 1993.

[13] *Seamless Co-Verification Environment User's and Reference Manual*, Ver. 2.3, Mentor Graphics Inc., 1998.

[14] E. Tuggle, "Writing Device Drivers", *Embedded Systems Programming*, Jan. 1993, pp. 42-65.

[15] S. Vercauuteren, B. Lin, "Hardware/Software Communication and System Integration for Embedded Architectures", in *Design Automation for Embedded Systems*, Vol 2, No. 3/4, May 1997.

[16] J. Öberg, A. Kumar, A. Hemani, "Grammar-based Hardware Synthesis of Data Communication Protocols", In *Proc. of the 9th ISSS*, Nov. 1996, pp. 14-19.

Paper 7

HW/SW Interface Validation in IP based System Design

7

HW/SW Interface Validation in IP based System Design

Mattias O'Nils, Axel Jantsch

Royal Institute of Technology
Department of Electronics, Electronic System Design Lab.
Electrum 229, S-164 40 Kista, Sweden
Email: {mattias,axel}@ele.kth.se

Abstract

Hardware/Software interfaces are a critical point for system simulation because the hardware and software parts are developed by separate development activities. Our multi-phase validation approach facilitates the simulation of the interfaces at several steps during development. We keep all the simulation models consistent with both the specification and the implementation by generating the models with a technique derived from interface synthesis, which enables the interface specification, simulation models, and implementation to be consistent with each other. We justify validation in several phases by (a) the up to four orders of magnitude faster simulation of early phases compared to late phases, and by (b) enable efficient and accurate validation and reuse of interfaces to IP components in embedded HW/SW systems.

1 Introduction

Traditionally, reuse of IP components have been to use HW components within embedded systems. Even though this is a well established technique, there have not existed any tools to support this [12]. Recent years, tools like Polis [1], CoWare [13] have emerged from HW/SW codesign research efforts and MakeApp [5] from embedded software developers. Both CoWare and Polis target design of hardware/software systems where the whole design functionality is captured within their environment. During the system synthesis the communication is refined, i.e. the device drivers are generated together with the custom HW and operating system. But if the user wants to use IP blocks and an off-the-shelf RTOS, he/she will face the same troubles as for manual design [4,12]. CoWare offers cosimulation capabilities similar to Seamless [11] and Eagle [3]. For both CoWare and Polis the designer has to write their own verification models for levels (b) and (c) in figure 1. Thus, the simulated interfaces are not the same as for implementation.

The latest commercial advances in validation of hardware/software systems are cosimulation tools, e.g., Seamless [11] and Eagle tools [3]. These tools connect software simulators or programs with a HDL or gate-level simulator. But these tools are not enough for validation of the hardware/software interfaces, due to the bad performance. Higher performance is achieved with these tools if the host code cosimulation option is used, but this level needs a different interface than the implementation, which decrease the accuracy.

Heterogeneous hardware/software systems are often verified at several different levels. Figure 1 identifies four levels of HW/SW cosimulation, which differ in abstraction, simulation performance, and accuracy. Simulation of functional system descriptions (figure 1a) captures only the system functionality without any implementation details, e.g. SDL [10]. While cosimulation of the system for the target processor (figure 1d) captures most of the system functionality (i.e. very consistent with the implementation) but with a decrease of simulation performance. Cosimulation tools can also be used to connect the software part of the system executing on the host system (figure 1c) to a hardware simulator. This increases the performance by decreasing the level of details. Another way to further increase the simulation performance is to remove the hardware/software communication [6], as illustrated in figure 1b. For software development, simulation and debugging we do not need to simulate the entire connected hardware. What we need is a model of the hardware device, accurate enough so that the software does not notice the difference. Similarly, for simulating the hardware we do not need to include the entire processor and software into the simulation. This is accomplished for both sides in the HW/SW stub simulation (figure 1b). In fact, it includes two separate simulations, each containing a stub-interface hiding the non-existing second part. This level is suited for perhaps 90% of the separate hardware and software development, which does not rely on the full and exact behaviour of the

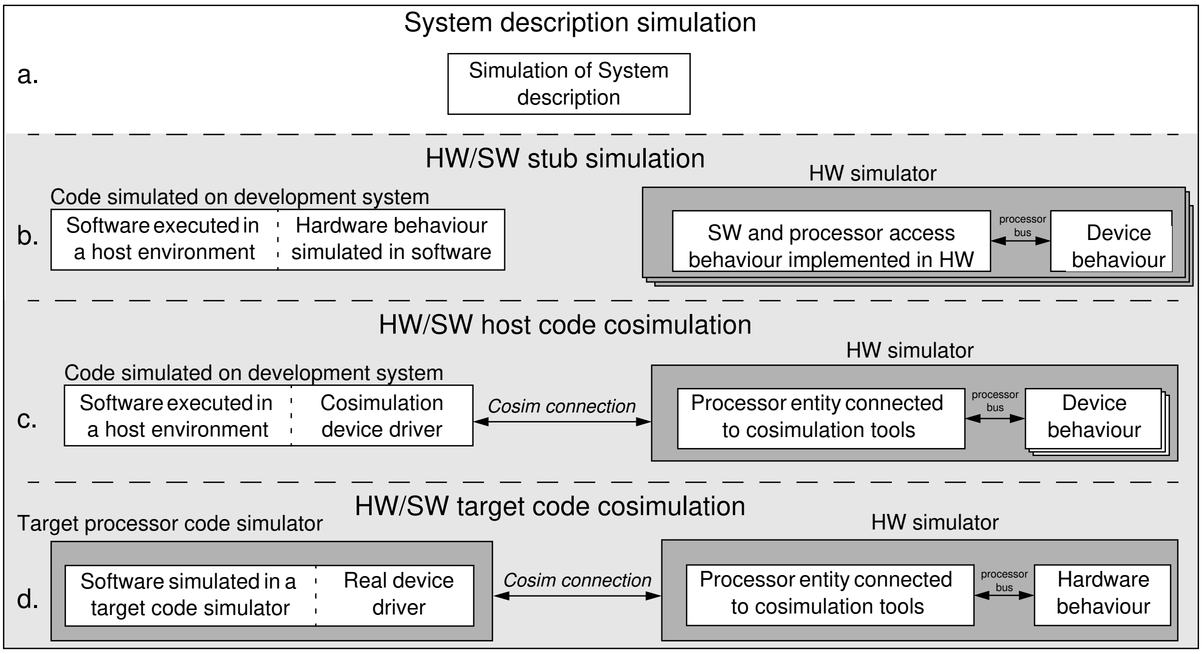


FIGURE 1. Simulation levels in embedded hardware/software systems development. (a) System description simulation, (b) HW/SW stub simulation, (c) HW/SW host code cosimulation, and (d) target code cosimulation.

other part. But short simulation time is critical for fast code-simulate-debug cycles for both hardware and software development.

A main problem throughout the validation flow in figure 1 is to keep the hardware/software communication consistent with the implementation, because only a high consistency of the simulation models with the final implementation gives value to the validation steps. Traditionally, the hardware and software stub models, device drivers for host code cosimulation, and device drivers for the target system are all independently derived from the specification.

We address this consistency problem by extending our approach to device driver synthesis [7] to also handle the generation of hardware and software stubs for HW/SW stub simulation and device drivers for host code simulation. That is, the hardware/software communication in all three validation levels (b to d in figure 1), will be generated

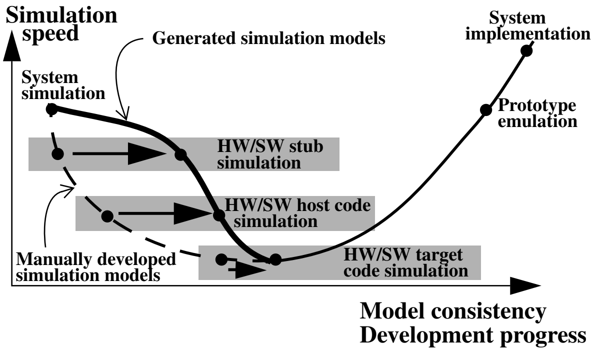


FIGURE 2. Simulation speed/model consistency trade-off in cosimulation

from the same formal specification and thus, the simulations behaviour will be consistent with both the implementation and the specification. This is illustrated in figure 2, where the dashed line represents traditional approaches, if simulations at these intermediate steps are done at all. The bold line represents our approach. In essence, at all three discussed validation phases can we increase model consistency significantly without sacrificing simulation speed. The layered validation technique is in it self not unique, on the contrary, these techniques are well established [3,6,11] among designers. The contribution is the generation of hardware/software simulation models, thus decrease the development effort and increase the accuracy of the validation phase.

The next section presents the proposed design and validation flow for device drivers. Sections three through five present the techniques for generation of the different models. In the sixth section we illustrate the different techniques with an example.

2 ComSyn - HW/SW interface synthesis

As described in [7], we model device drivers independently of the architecture using ProGram [14]. ProGram is a grammar based notation for protocol applications. Specifications in ProGram deal with sequences of allowed events as opposed to states and state transitions in an FSM model.

The ProGram description is synthesized into an untimed extended state machine (described in [7,14]), which in turn is the input to the architecture mapping procedure. The mapping procedure uses data from two libraries to generate the architecture specific code. The first of these li-

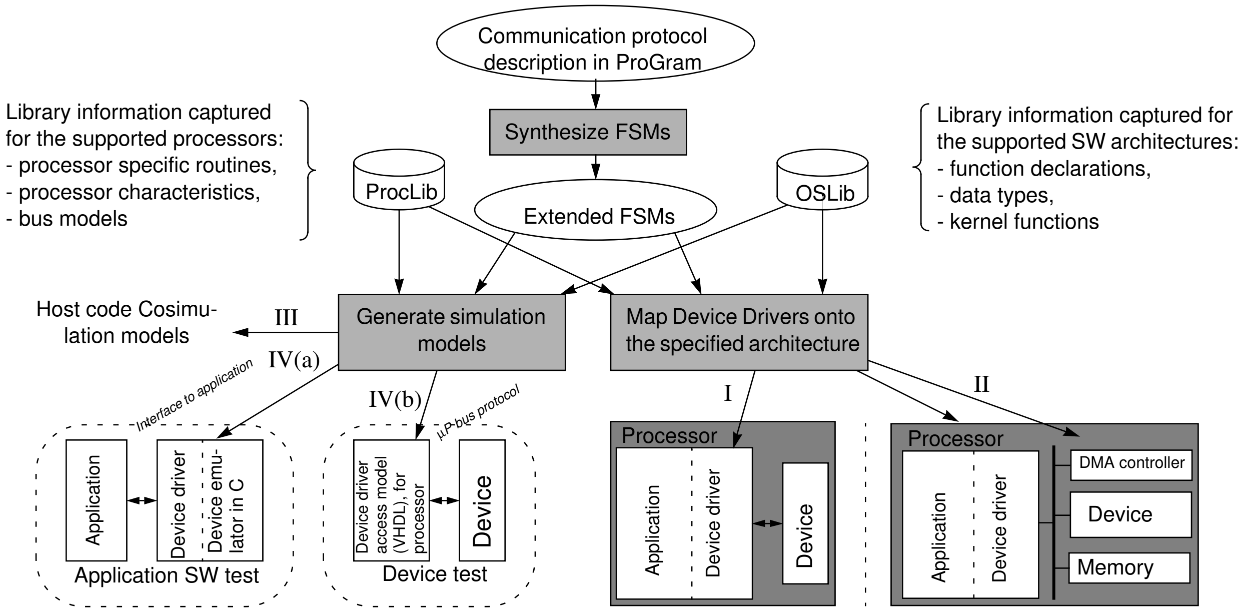


FIGURE 3. Overview of the ComSyn synthesis system.

braries is the library that captures the information on the OS architecture, the second captures the processor specific characteristics, see figure 3.

From this information the tool generates code for: (I) implementation for pure software (and target code cosimulation), (II) code for mixed hardware/software solutions with a DMA controller, (III) device drivers for host code cosimulation, (IVa) stub code for software simulation, and (IVb) stub code for hardware simulation.

3 Target code cosimulation

In target code cosimulation the whole system is modelled in the hardware simulator with the processor entity, which is connected to a processor simulator. Thus, the device driver used in target code cosimulation is exactly the same as the one used for implementation, with respect to timing and bus behaviour. So the device driver generated for implementation is also used for target code cosimulation.

Device drivers for implementation and target code cosimulation is generated from FSMs (generated from ProGram) and the two libraries OSLib and ProcLib, see figure 3.

The synthesis procedure starts with mapping of bit vector types to data types used by the OS kernel (defined in OSLib). Then the state machine is transformed to software implementation by applying a set of transformation rules on each state. The architecture dependent parts of the transformed FSMs are then mapped onto the selected architecture using the libraries OSLib and ProcLib. Depending on the type of FSM description different declarations are generated, see figure 4.

```
for all Signals
    signal.map_bit_vector_types(OSLib)
end for
for all FSMs
    for all FSM.States
        state.apply_transformation_rules(OSLib, ProcLib)
    end for
    FSM.minimize_no_states()
    case (type of FSM)
        ACCESS : FSM.gen_function(OSLib)
        INTERRUPT : FSM.gen_int_handler(OSLib)
        TASK : FSM.gen_task_function(OSLib)
    end case
    FSM.generate_body_code()
```

FIGURE 4. Synthesis and mapping procedure of device drivers.

4 Host code cosimulation

Device drivers for host code simulation have to communicate with the hardware via a hardware simulator instead of directly accessing the hardware with pointer accesses. This is achieved by replacing the access to registers with cosimulation access functions.

There are two possible cases of host code cosimulation: (1) single process execution and (2) kernel emulation. For the first case, the software is implemented using a single process on a workstation, i.e. all parallel constructs have to be serialized. For the second case, software is executed with the target OS kernel that is emulated on the host (workstation), i.e. the interface to the application and OS kernel will be the same as for implementation but the interface to the device will be different. In this paper we only consider the second case, emulation of the OS kernel. Host code cosimulation will have higher performance than host code simulation, but at the cost of software timing and lower bus behaviour accuracy.

Code generation for host code cosimulation follows the same procedure as for target code cosimulation (implementation) in figure 4, except for the behaviour of the mapping of variables and generation of interrupt routines.

4.1 Mapping of variables

In the synthesis for implementation the mapping of types for ports to hardware, ports to software and internal variables is treated in the same way. For host code cosimulation the mapping of types for the ports to the device has to be treated differently, since these have to be replaced with accesses to the hardware simulator.

Implementation	Simulation

`UINT8 *TRX;*LCR;`
`...`
`setup = *LCR;`
`*TRX = data;`
	`HCS8 TRX=0x200;`
	`HCS8 LCR=0x201;`
	`...`
	`setup = *LCR;`
	`*TRX = data;`

FIGURE 5. Implementation of device register access in host code cosimulation.

Supplying a host code cosimulation access class written in C++ similar to the one use in [11] solve this. In this class there will an 8,16 and 32 bit type for accessing the memory space of the embedded system. Variables declared with this type are accessed with the star and type assignment operators in the same way as in the implementation, as seen in figure 5. Thus, the code generation for host code cosimulation can be the same as for implementation except for the declaration of the ports. The ports are instead declared with the host code cosimulation class, and the port address is assigned in the declaration, see figure 5. The body code for the operators for the class has to be supplied for each simulator.

4.2 Generation of interrupt handler

Since in a host system interrupts cannot be generated from the hardware simulator to the workstation, interrupts have to be treated as a special case in host code cosimulation.

The way we have solved this is to generate a separate task that receives the interrupt events from the hardware simulator. The task is generated as an infinite loop with a wait on interrupt library instruction (unique for each simulator) that also identifies what interrupt was asserted so the appropriate interrupt routine can be called.

5 Stub simulation

For stub simulation two stubs (models) are generated, one for software simulation and another for hardware simulation. As for host code cosimulation, stub simulation can be done as (1) a single process or (2) with the OS kernel emulated in the host system. In this paper we only consider the second case.

The stub for software simulation emulates the hardware behaviour by replacing accesses to read-only register with function calls, which capture the register behaviour. The designer selects the register behaviour from a set of functions (static, random or reading values from a file) or writes the function manually. Write and read/write registers are replaced by memory accesses to a memory location in the host system

The stub for hardware simulation emulates the software behaviour by generating the device driver in VHDL. This device driver is connected to a bus model of the selected processor, i.e. the generated code together with the bus model forms a testbench for the device under test.

Stub simulation lacks the synchronization between hardware and software behaviour, thus decreasing the simulation accuracy since the behaviour is emulated in either hardware or software. But with increased simulation performance (for software simulation) with maintained interfaces between device driver and application/kernel.

5.1 Generation of stubs for SW simulation

Code generation of stubs for software simulation follows the same procedure as for target code cosimulation (implementation) in figure 4, except for the parts for mapping of variables and generation of interrupt routines.

5.1.1 Handling of device registers

From the device driver perspective the device is just a set of registers and vectors. There are three types of registers: read only registers (ROR), write only registers (WOR) and read/write registers (RWR). The WOR and RWR registers have the same behaviour as a memory, i.e. they have the ability to store data and recall stored data. Since registers declared as WOR will not be read and RWR will contain same data as previously written. If a register at a specified address can be written and read, but the read information differs from the one written, then it should be declared as one ROR register and one WOR register with the same address. Hence, from the device driver perspective the device behaviour is captured by the ROR registers, see figure 6.

RWR and WOR registers are implemented with the generic C++ class `RWR`. Objects declared with the class `RWR` are accessed the same way as the device registers are accessed in the implementation, similar to host code cosimulation class in figure 5. The `RWR` class's only behaviour is to store and recall values of a desired type, i.e. a `*` and = operators.

The device behaviour is divided into several behaviours each handling a ROR register. There are several ways to capture the behaviour of a device. To capture the full behaviour, a function for each ROR register should be sup-

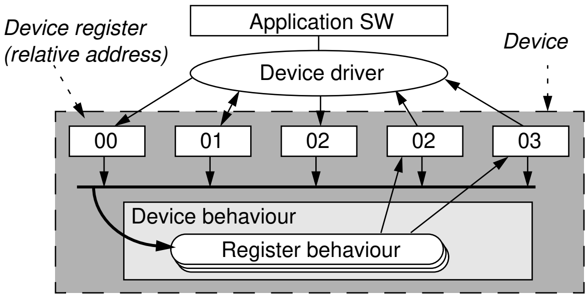


FIGURE 6. Simulation of device behaviour for software execution.

plied. If the exact behaviour of a device register is not required, there are three ways to generate the read values: constant value, random values, or a series of values read from a file. ROR registers are implemented as objects of a generic C++ class `ROR`. This class is a library module that captures all the behaviours described above: static, random, file, and user defined. Values are read in the same way as a ROR register is read in the implementation, see figure 5. The behaviour for a register object declared with the `ROR` class is interactively selected by using different constructors. For user defined behaviour, a function defining the register behaviour has to be written.

5.1.2 Handling of interrupt routines

An interrupt routine executes asynchronously with respect to the access functions of a device driver and is triggered by hardware generated events. Since there are no events generated by hardware, the events are emulated with delays. The interrupt routines are activated within a task. Within the task there is an infinite loop, which has a wait-for-time in it. The type of delay is interactively selected as constant delay, random delay, or a series of delays read from a file. After each delay, there is a function call to the interrupt routine, i.e., same interrupt routine as generated for implementation.

5.2 Generation of stubs for HW simulation

The libraries used for device driver synthesis presented in figure 3, capture the characteristics of the processor and the kernel. In line with this we can define the generation of the stub for hardware simulation as an implementation target, i.e. describe the target by means of the libraries. This feature is further described in [8] and not included here since this feature is not needed for design with IP components.

6 Example to illustrate the approach

To further explain the generation of the different models we provide a schematic example of a simple read function, where a value is read from a register (`B`) and presented as a out value (`A`) to the application. Figure 7 (a) shows the

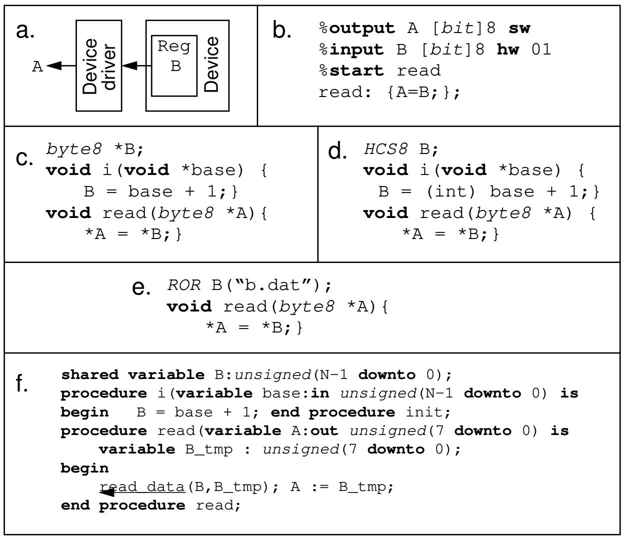


FIGURE 7. Example of code generation.

information flow and (b) shows the corresponding ProGram code.

Figure 7 (c-e) shows the generated C code for the different levels: (c) target code cosimulation, (d) host code cosimulation and (e) stub for software simulation. For these, the access function (`read`) looks exactly the same for all three levels, the only thing that differ is declaration and address initialization of the device register. Figure 7f shows the access function (stub) generated for hardware simulation. In figure 7f within the read function we see that data is read from the device using the library function `read_data`. `N` in figure 7 f is the number of address bits for the selected processor.

7 Results

A number of representative device driver examples have been used to evaluate the synthesis system [7]. The same examples have been used to test and evaluate the generation methods presented in this paper. Note that the size of the overall designs, in terms of the number of lines of code, is not an important factor. The feasibility of implementations and the possibility to generate simulation models for different classes of devices are more important considerations. Table 1 compares the simulation performance for 5 different ways to model and simulate hardware/software communication. In the first column is the name of the interfaced device; the second column gives the performance for host code simulation with Seamless [11] (device is simulated in an HDL simulator [9]). In the columns three through six the performance numbers for different configurations of hardware/software simulation with stubs are shown. The third column presents results for the approach of section 5.2, i.e. the device driver is implemented in VHDL and is connected to the hardware via a processor bus protocol interface, the master interface. Columns four to six presents the performance numbers for the software simulation with the device behaviour as a stub simulated in software, see section 5.1. Read-only-register behaviour is modelled as a constant value in column 4, as a random value in column 5, and by reading values from a file in column 6.

Interfaced device	Host code cosimulation	Generated stubs for hardware/software simulation			
		HW	SW constant	SW random	SW file
MAX197	479	306	$2.2 \cdot 10^6$	$1.5 \cdot 10^6$	$1.3 \cdot 10^6$
MAX530	1039	476	$14 \cdot 10^6$	$14 \cdot 10^6$	$14 \cdot 10^6$
UART, 16552	280	164	$2.2 \cdot 10^6$	$2.2 \cdot 10^6$	$2.2 \cdot 10^6$
CD interface	630	302	$4.9 \cdot 10^6$	$3.6 \cdot 10^6$	$1.8 \cdot 10^6$
DS12887	647	408	$5.0 \cdot 10^6$	$3.4 \cdot 10^6$	$3.3 \cdot 10^6$
ICM7218	654	150	$4.0 \cdot 10^6$	$4.0 \cdot 10^6$	$4.0 \cdot 10^6$

TABLE 1. Number of device driver accesses per second on a Sparc Ultra 1, 167 MHz.

The reason why software stub-simulation of MAX530, ICM7218 and UART 16552 results in the same values for all three configurations (column 4-6) is, that they do not have ROR registers in the measured functions. The difference in performance between the host code cosimulation and hardware stub simulation is due to the greater number of events generated to simulate the hardware stub. This difference will decrease for larger devices since the simulation of the stub will be a smaller fraction of the whole simulation.

8 Conclusion

We present an approach for generation of both implementation, and simulation models from an architectural and implementation independent description. This approach guarantees that the implementation and simulation will be consistent with the specification. It also reduces the design and validation effort for the hardware/software communication, see table 2. Since the protocol can be used several times. Values of the second row is estimated for a medium sized device driver (~100 LoC) based on results from table 1 and programmer efficiency models [2], normalized to the generated stub simulation.

For generated stub simulation we achieve up to four orders of magnitude faster simulation of the software compared to host code cosimulation, see table 1.

An important benefit is that software designers need only to handle software tools and hardware designers need only to handle hardware tools. Hence, these designers are not forced to control the other worlds design tools and do not need to handle and set-up the cosimulation software. This is a tremendous advantage with today's complex design tools and the even more complex task of interfacing tools.

Characteristic	Hand written HW & SW stub sim.	Generated HW/SW stub sim.	Hand written host code cosimulation	Generated host code cosimulation	Target code cosimulation	Prototype
	SW/HW	SW/HW				
Full design simulated	no	no	no	no	yes	yes
Effort to setup	20	1	20	1	200	>600
Sim. performance	$1/10^{-4}$	$1/10^{-4}$	10^{-4}	10^{-4}	$<10^{-4}$	1
HW/SW interface consistent with impl.	no	yes	no	yes	yes	yes
Simulated system consistent with impl.	no	no	no	yes	yes	yes

TABLE 2. Comparison of the capability for different simulation methods.

9 Reference

[1] Balarin F. et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, June 1997.

[2] Boehm B. et al., "Cost models for future software life cycle processes: COCOMO 2.0", *Annals-of-Software-Engineering*, vol. 1, 1995, pp.57-94.

[3] Eagle Tools, Synopsys, http://www.viewlogic.com/products/eagletools.html.

[4] Gal-Oz S., Cohen A., "The Hazards of Device Driver Design", *Embedded Systems Programming*, May 1997, pp. 34-46.

[5] Grehan R., "Driver assistance", *Embedded Systems Programming*, Nov. 1997, pp. 75-80.

[6] Mittag L., "Device Drivers for Nonexistent Devices", *Embedded System Programming*, Aug. 01 1996, vol. 9, no. 8, pp. 30-32,34,36,38,40.

[7] O'Nils M., Öberg J., Jantsch A., "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", *Proc. of the Euromicro Workshop on Digital System Design*, 1998.

[8] O'Nils M., Jantsch A., "Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models", *Proc. of the High Level Design, Verification and Test Workshop*, 1998.

[9] *QuickHDL User's and Reference Manual*, Ver. 8.5_4, Mentor Graphics Inc., 1997.

[10] *SDT User's Guide*, TeleLOGIC AB, Malmö, Sweden, 1993.

[11] *Seamless Co-Verification Environment User's and Reference Manual*, Ver. 2.3, Mentor Graphics Inc., 1998.

[12] Tuggle E., "Writing Device Drivers", *Embedded Systems Programming*, Jan. 1993, pp. 42-65.

[13] Vercauuteren S., Lin B., "Hardware/Software Communication and System Integration for Embedded Architectures", in *Design Automation for Embedded Systems*, Vol. 2, No. 3/4, May 1997.

[14] Öberg J., Kumar A., Hemani A., "Grammar-based Hardware Synthesis of Data Communication Protocols", *Proc. of the 9th ISSS*, Nov. 1996, pp. 14-19.

Paper 8

Refinement of HW/SW Communication Channels: Case Study and Comparison

8

Refinement of HW/SW Communication Channels: Case Study and Comparison

Mattias O'Nils, Axel Jantsch
Royal Institute of Technology, Department of Electronics, ESD Lab
Electrum 229, S-164 40 Kista, Sweden
Voice(recept): +46 8 752 14 00, Fax: +46 8 752 12 70
Email: {mattias,axel}@ele.kth.se

Abstract

We compare four different approaches to specification and refinement of the software parts in HW/SW communication channels with respect to design productivity. The four approaches are CoWare [13], MakeApp [7], ComSyn [8], and manual refinement. Manual refinement is still predominant in practice for most embedded systems. With this case study we hope to show its deficiencies and how it can be improved with tool support available today. The comparison is done by analysing the impact of different techniques on the design process. The analysis is based on two designs, a channel decoder of a transceiver in a D-AMPS base station and an operation and maintenance block of an ATM network. We compare the design productivity with respect to development effort, validation effort, and library maintenance effort.

1 Introduction

Communication and interface design in a digital embedded hardware/software system is an error prone task. The productivity per line of code is significantly lower for communication modules than for other system parts. Thus, even though the communication modules seem to be small in size compared to the rest of the system, they often dominate the validation effort because critical problems cannot be detected during module simulation but surface only during the system validation phase, if not later. In this phase the cost of correcting errors is substantially higher than in earlier phases and can hurt a project schedule decisively. This is the motivation to research effective ways to model and refine communication within an embedded system and to transform these models into an implementation for some specific architecture.

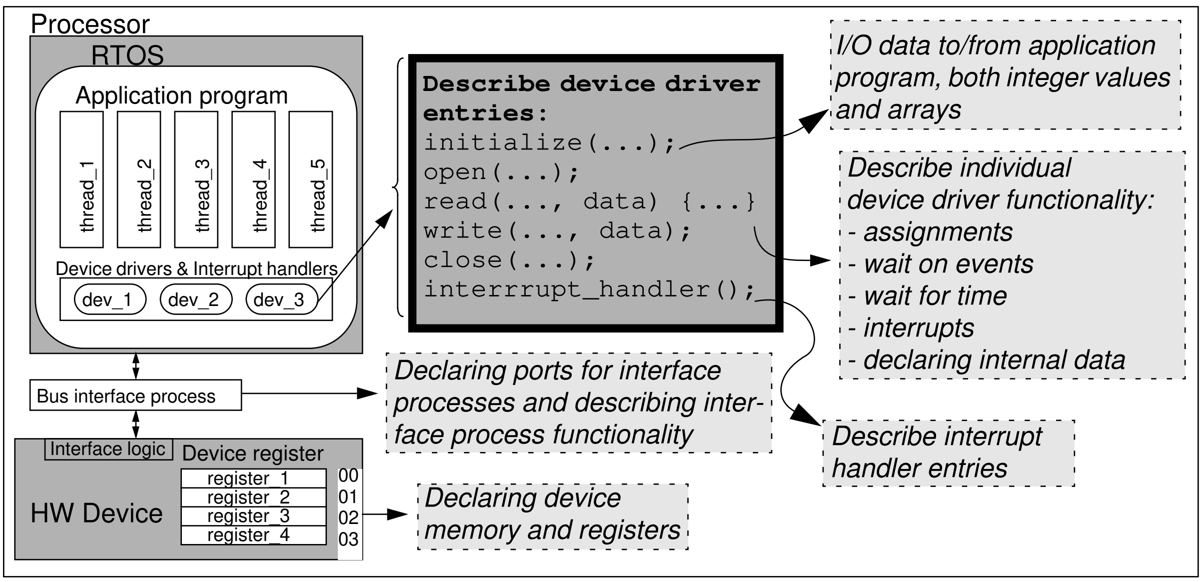


FIGURE 1. Identifying functionality in HW/SW communication.

From the software point of view hardware/software communication is only reading/writing hardware registers from software. But it is still one of the major causes of errors during design of hardware/software embedded system, due to the lack of tools for design and simulation.

Hardware/software communication channels consist of three parts (figure 1): (1) a software part, (2) a hardware part and (3) a bus interface which translates the device and processor bus protocols. The software part, i.e. device drivers, consists of interrupt, initialization and access functions. The hardware part consists of registers and memories (RAM, FIFO). How these registers are accessed, is determined by a control flow. The control flow is implemented in the device driver routines or in a DMA controller.

There are several research approaches addressing the modelling and synthesis of bus protocol interfaces [1,13]. A few research groups address the problem of selection of register and memory architectures [2]. For the software side, there exist commercial tools like MakeApp [7] that handle the configuration and code generation for kernel less software architectures, letting the designer concentrate on the behaviour instead of understanding the architecture and interfacing details. Other approaches provide communication libraries for the hardware software interfaces [5,13,14]. Common to all these approaches is that they only support one or a few software architectures. ComSyn [8,9,10] maps an architecture and implementation independent description of the software part of the hardware/software communication onto different architectures (combinations of kernels and processors), which is captured in separate libraries.

In this paper we compare four approaches to refinement of hardware/software communication: CoWare [14,13], MakeApp [7], manual refinement and ComSyn [8,9,10]. We concentrate on the software part of the hardware/software communication channels.

The next section introduces the different approaches. Section 3 presents the design examples used for evaluation. Section 4 discusses the comparison between the approaches.

2 Approaches to refinement of HW/SW communication

2.1 ComSyn

The unique feature of ComSyn is the clear separation of (1) communication specification (ProGram), (2) processor and bus interface description (ProcLib), (3) OS and SW architecture description (OSLib), see figure 2.

We model device drivers independently of the architecture using ProGram [8]. ProGram is a grammar based notation for protocol applications. Specifications in ProGram deal with sequences of allowed events instead of states and state transitions as it would in an FSM model.

The ProGram description is synthesized into an extended state machine [8,15]. This extended FSM could also be entered manually or generated from some other description language. The extended FSMs are fed to the architecture mapping procedure [9] (I in figure 2), which map the FSM onto a specific processor and OS kernel.

The descriptions can also be mapped to hardware, i.e., implemented as an application specific DMA controller [10] (II in figure 2). Several simulation models for different levels of abstractions are also derived from the protocol description [11]. That is: stub simulation (III(a) and III(b) in figure 2), host code cosimulation (IV) and target code cosimulation (I and II in figure 2).

The procedures for code generation of implementation and simulation models uses data from two libraries to generate the architecture specific code. The first of these libraries is the library that captures the information on the OS architecture, the second captures the processor specific characteristics.

2.2 CoWare

CoWare [14,13] is a design environment for heterogeneous hardware/software systems on a chip. CoWare accepts heterogeneous communicating processes, which are specified in: DFL, VHDL or C. CoWare's main focus is system integration and handling of communication in embedded systems. The heterogeneous specification is mapped onto different processors (DSP, micro controller or hardware). Intra and inter process communication are done with point-to-point

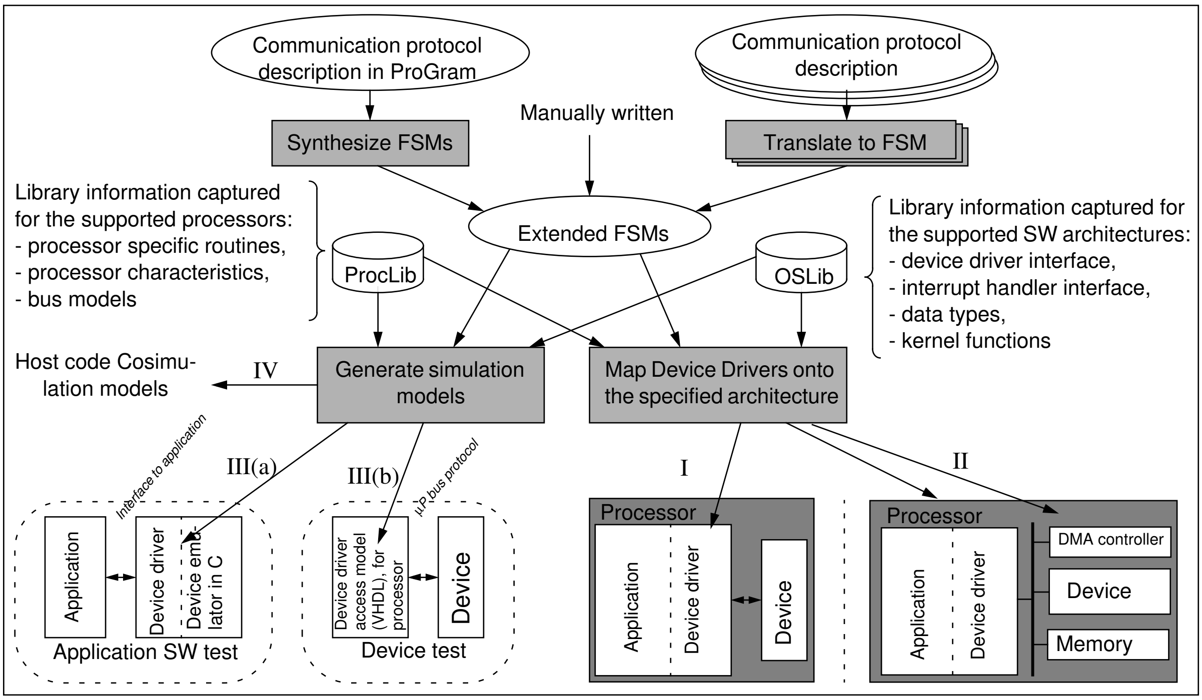


FIGURE 2. Overview of the ComSyn synthesis system.

communication channels. The communication semantics are based on the concept of rendezvous channel communication via send/receive operations. Hardware/software communication channels are mapped onto a fixed architecture. This architecture is based on several library models. For software, the communication procedures are captured as parameterised C functions that is mapped onto a software model, i.e. adopt to processor specific I/O handling, interrupt handling etc. For hardware, a hardware interface cell is generated to connect with a handshake protocol to an I/O control unit. This I/O control unit is a link between the processor and the handshake protocol.

Their approach is focused on application specific embedded systems on a chip, i.e. the whole product is designed from scratch. This means that the support for IP (hardware cells, real-time kernels) property is not good since these will not well adopt to their fixed architecture. For simulation they generate code for their own cosimulation approach.

CoWare is a broad and general design system. In this case study we only consider a part of it, i.e. the SW interface synthesis, for comparison with the other approaches.

2.3 MakeApp

MakeApp [7] is a commercial tool for generation of device drivers for selected combinations of devices and micro controllers. The user configures the device to a desired behaviour and initialization valid for a specific system. From this information and library information the tool generates the C code for the selected device and processor.

The code generation in MakeApp only supports kernel-less systems, i.e. the designer has to add the kernel specific parts. This can lead to inconsistency problems between the implementation and the desired functionality since the kernel related code is written manually. Simulation models have to be derived by the user.

2.4 Manually refinement

Today the device drivers for most embedded systems are still written manually. The designer refines the hardware/software communication from some formal or informal specification to simulation and implementation models. This is a difficult task [6] that can cause many errors and the number of models necessary to write can get out of hand [12] when several processors, OS kernels and behaviours must be supported.

3 Design example

We use two designs for the case study, a channel decoder of a transceiver in a D-AMPS base station (D-AMPS) and an operation and maintenance block (OAM) of an ATM network.

3.1 D-AMPS

D-AMPS [3] (IS-54B) is a North American digital cellular standard developed in the late 80's by the Cellular Telecommunications Industry Association. The objective was to increase the performance of the existing analog AMPS standard using the same infrastructure and to support AMPS operation, hence a dual-mode cellular system. D-AMPS is a time division multiple access (TDMA) cellular standard.

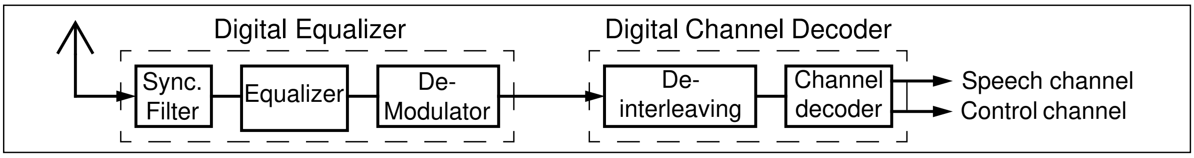


FIGURE 3. Receiver in IS-54B base station

The standard has 832 channels per carrier in the 800 MHz range and a total of 1164 channels. Each radio channel has 30KHz bandwidth and is segmented in time into six slots using TDMA, whereas 2 slots make up one voice circuit. The data rate is 48.6 Kbits/s, with each slot transmitting 324 bits in 6.67 ms.

The channel decoder functionality receives 272 bit frames that should be processed in 6.67ms. The process consists of: de-interleaving, Viterbi decoding, CRC-sum calculation, bit error rate estimation, sorting of speech data, and masking of bad speech frames.

We partitioned the design into HW and SW parts. The Viterbi decoder is implemented in hardware and the rest of the functionality is implemented in software. Figure 4 illustrates the hardware/software communication in the channel decoder design.

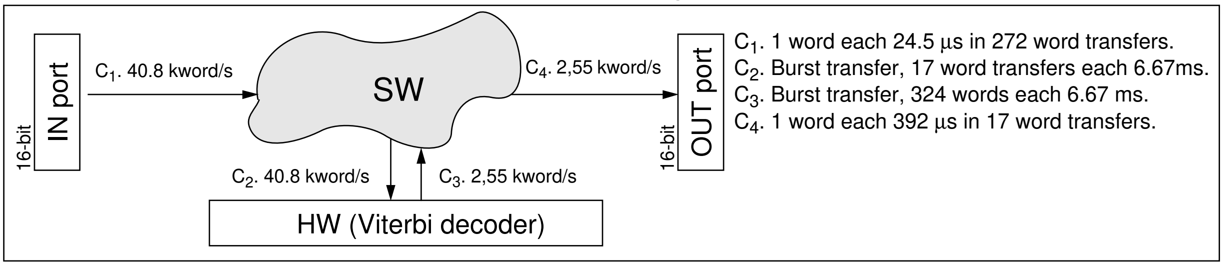


FIGURE 4. HW/SW communication in the channel decoder

3.2 ATM Operation and Maintenance Block

The Operation And Maintenance (OAM) functionality [4] in an ATM network is situated in

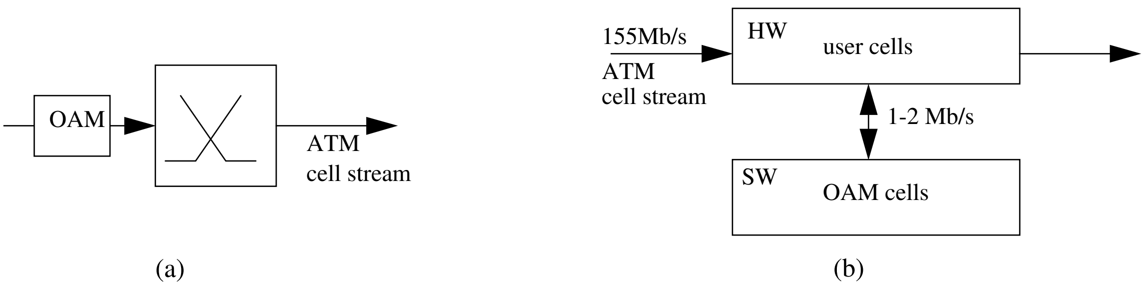


FIGURE 5. OAM block in an ATM network (a) and a reasonable HW/SW partitioning (b).

the neighbourhood of a switch. Specific ATM cells, OAM cells, are used to identify network problems, measure network performance, and communicate the state of the network between network nodes. OAM functions analyse, transform, and generate OAM cells. Figure 5b shows a typical HW/SW distribution, where ATM user cells are received and re-transmitted in custom hardware, while OAM cells are analysed and transformed in software. OAM cells typically constitute about

1% of the total traffic.

4 Comparison

On the basis of these two realistic example we compare the four approaches with respect to the following criteria:

- **Modelling effort**: How many models have to be developed for a new system and how many models must be maintained to support different protocols, processors, and operating systems?
- **Specification**: How appropriate is the formalism for protocol specifications?
- **Implementation**: Which target architectures are supported?
- **Validation**: How is the validation supported?

4.1 Modelling Effort

CoWare target systems where the complete design is synthesised from a specification into an implementation. That is, for the design examples in the section 3, the whole behaviour would be specified in the specification languages allowed by CoWare (C for software and VHDL/DFL for hardware) and then synthesized into a hardware/software solution. For this situation, CoWare offers an automated refinement of the communication down to implementation for their specific hardware and software architecture, i.e. the operating system is also synthesized. If a third party RTOS is used, the communication protocol has to be specifically modeled for this RTOS. Thus, CoWare does not ideally support IP based design, where off-the-shelf blocks including the operating system are integrated in the product.

MakeApp is the opposite of CoWare. It targets design with IP components. MakeApp produces code for communication between the application and standard devices (i.e. device drivers) and adopt the code to the selected standard processor. Generated code has to be adopted to the selected software architecture.

Approach	Number the of models
CoWare	$Entries = s \cdot k + p$
MakeApp	$Entries = s \cdot k + p$
ComSyn	$Entries = s + k + p$
Manual	$Entries = s \cdot k \cdot p$

TABLE 1. Number of models to be developed and maintained to support s protocols, k OS kernels, and p processors.

ComSyn targets both these application areas. Since ComSyn uses an implementation and architecture independent description of the communication behaviour and an efficient library structure [9], which leads to a linear growth of the number of models. This is a clear advantage if we compare to CoWare and MakeApp that have a square growth with respect to processors, RTOS', and protocols. For manual refinement, the library growth is cubic, see table 1.

4.2 Specification

The control behaviour of how information is read and written in hardware/software communication is captured in software or in hardware by a DMA controller. In CoWare the communication protocol is captured as parameterised C functions. This is similar to the MakeApp and a manual approach. In ComSyn the protocol is described in ProGram, as described in section 2.1. Table 2 shows a comparison between modelling the hardware/software communication for the channel de-

Approach	C_1[a]	$C_2 + C_3$[a] memory based	$C_2 + C_3$[a] FIFO based	C_4[a]	OAM HW/SW channel memory based	OAM HW/SW channel FIFO based
C	56	105	101	62	81	76
ProGram	27	58	56	25	43	39

TABLE 2. Comparison of different techniques to specify HW/SW communication in lines of source code.

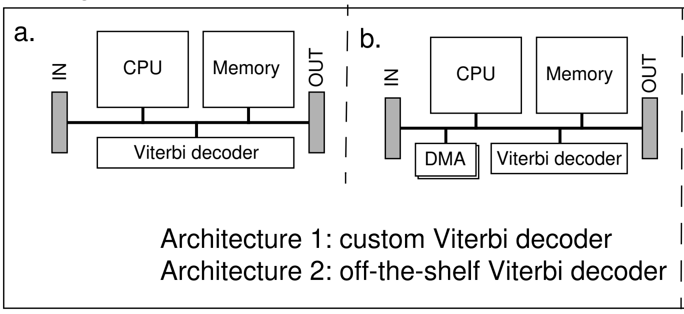
a. Refers to the HW/SW communication channels in figure 4.

coder in C (CoWare, MakeApp, manual) and in ProGram. ProGram specifications are on average

50.5% of the C models. This is mostly due to the fact that the ProGram code does not have any implementation specific code.

4.3 Implementation

CoWare, MakeApp and ComSyn, all produce C code for the hardware device access functionality. The CoWare approach is based on the assumption that the hardware and the software are or have been designed with the CoWare tool. MakeApp has the opposite assumption, that is, the hardware is a standard component. This affects considerably the refinement of the hardware/software communication. E.g. in the D-AMPS design the Viterbi decoder is implemented in hardware. This gives us two possible scenarios: (1) the Viterbi decoder is designed in-house or (2) it is bought. Two possible ways to implement the hardware/software communication: (a) with access functions or (b) direct memory access (DMA), see figure 6. As described in [13], CoWare supports the combination (1a), i.e. the device access from software and the hardware is designed in-house. MakeApp support combination (2a), i.e. a standard component with access functions in software. You could state that both these tools could support architecture (b) but for both tools this would result in twice as large libraries, since each architecture (a and b) must have a separate library entry for the same behaviour. Hence, several specifications of the same behaviour would be required, which is a potential source of inconsistencies. In ComSyn there is only one behaviour specification regardless of what kernel or what architecture (a or b) is targeted (figure 6).



	Manual	CoWare	MakeApp	ComSyn
arch. 1a	poss.	supp.	poss.	supp.
arch. 1b	poss.	poss.	poss.	supp.
arch. 2a	poss.	poss.	supp.	supp.
arch. 2b	poss.	poss.	poss.	supp.

FIGURE 6. Architectures for implementation of HW/SW communication.

4.4 Validation

Figure 7 identifies four levels of HW/SW cosimulation, which differ in abstraction, simulation performance, and accuracy. Cosimulation of the system for the target processor (figure 7c) captures most of the system functionality (i.e. very consistent with the implementation) but with a decrease of simulation performance. Cosimulation tools can also be used to connect the software part of the system executing on the host system (figure 7b) to a hardware simulator. This increases the performance by decreasing the level of details. Another way to further increase the simulation performance is to remove the hardware/software communication, as illustrated in figure 7a.

The only simulation support provided by MakeApp is the generation of device drivers which can be used for target code cosimulation (figure 7c). CoWare has its own approach to target code cosimulation (could be extended to host code cosimulation) instead of using commercial cosimulation tools. They do this be generating the code that performs the hardware/software communication by using the UNIX environment to communicate between a software simulator and set of hardware simulators. ComSyn generates simulation code for commercial cosimulation tools and also supports stub simulation level (figure 7a), by generating simulation modules for both software and hardware [11]. This is summarized in table 3.

Figure 8 compares the number of models to be verified in the different approaches, assuming the same level of confidence is targeted, i.e. three validation levels.

5 Conclusion

We have compared different approaches to refinement of hardware/software communication with respect to design effort and productivity. The basic assumption is, that the productivity is higher when less code is written. The increase comes directly from reduced coding time and, per-

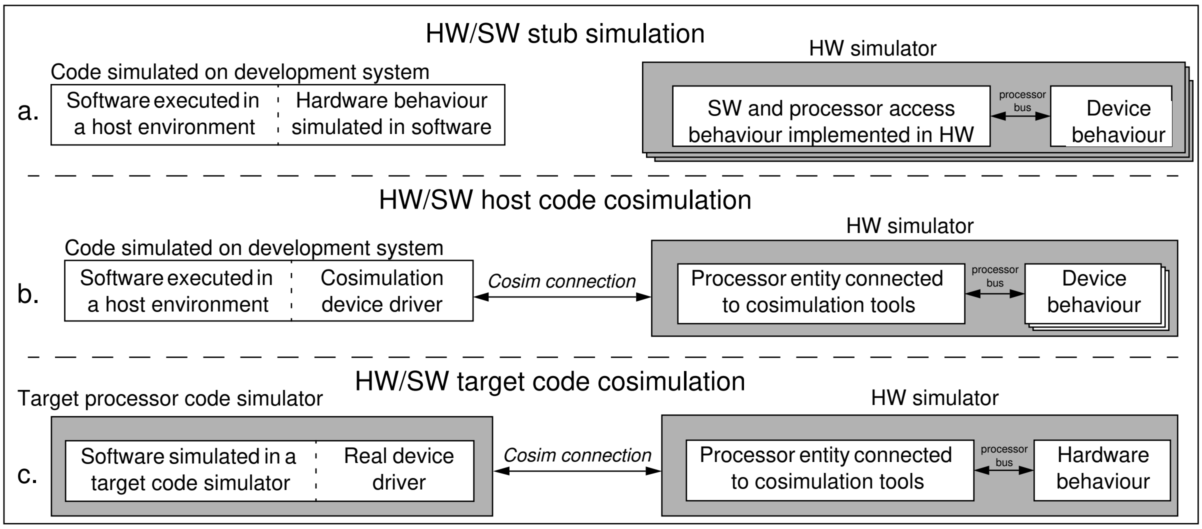


FIGURE 7. Simulation levels in embedded hardware/software systems development. (a) System description simulation, (b) HW/SW stub simulation, (c) HW/SW host code cosimulation, and (d) target code cosimulation.

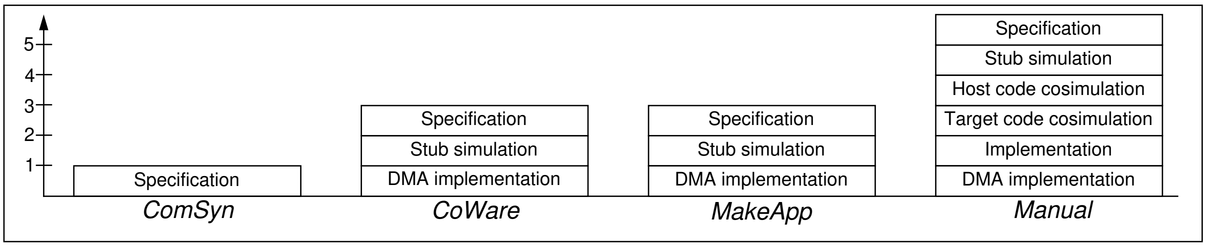


FIGURE 8. Comparison of the number of parts needed to be verified.

	Manual	CoWare	MakeApp	ComSyn
stub simulation	not supported	not supported	not supported	supported
host code cosimulation	not supported	extendable	not supported	supported
target code cosimulation	supported	supported	supported	supported

TABLE 3. Validation support.

haps more significant, from reduced validation time, assuming the number of errors is a linear function of the amount of code written. This is of particular significance for HW/SW communication blocks because they are the among the most error prone areas in embedded systems.

Figure 9 illustrates the grows of models and library elements to be developed and supported in the different approaches, assuming three different levels of simulation as indicated in figure 7. The size of each model is typically between several dozens and several hundred lines. So if we multiply the Y axis in figure 9 by 100 we get the approximate number of lines of code. If HW and SW IP blocks become as pervasive as many people argue now, we must strive for a linear growth curve similar to the ComSyn line in figure 9 rather than a square or cubic curve. This will only be possible if the description of the different blocks (processor, protocol, RTOS) required for communication are decoupled from each other.

6 Reference

1. P. Chou, R. Ortega, G. Borriello, “Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems”, In *Proc. of the IEEE/ACM ICCAD*, pp. 488-495, 1992.
2. J-M. Deveau, G. F. Merchioro, T. Ben Ismail, A. Jerraya, “Protocol Selection and Interface

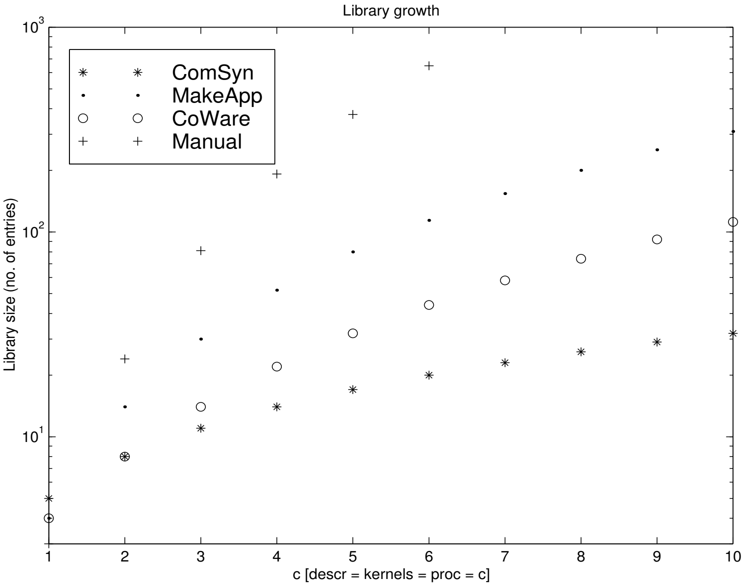


FIGURE 9. Comparison of number of library elements for three simulation levels.

Generation for HW-SW Codesign", In *IEEE Transactions on VLSI Systems*, March 1997.
3. EIA/TIA Interim Standard, Cellular System Dual-Mode Mobile Station - Base Station Compatibility Standard, *IS-54-B*, April, 1992.
4. Martin De Prycker, *Asynchronous Transfer Mode*, Prentice Hall, 1995.
5. M. Eisenring, J. Teich, "Domain-Specific Interface Generation from Dataflow Specifications", In *Proc. of the International Workshop on Codesign*, March1998.
6. S. Gal-Oz, A. Cohen, "The Hazards of Device Driver Design", *Embedded Systems Programming*, May 1997, pp. 34-46.
7. R. Grehan, "Driver assistance", *Embedded Systems Programming*, Nov. 1997, pp. 75-80.
8. M. O'Nils, J. Öberg, A. Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", *Proc. of the EuroMicro Workshop on Digital System Design*, 1998.
9. M. O'Nils, A. Jantsch, "Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification", submitted to *Design Automation and Test in Europe Conference (DATE)*, 1999.
10. M. O'Nils, A.Jantsch, "Synthesis of DMA Controllers from Architecture Independent Descriptions of HW/SW Communication Protocols", submitted to *Int. Conf. on VLSI Design*, 1999.
11. M. O'Nils, A. Jantsch, "Multi-phase Validation of Hardware/Software Interfaces based on Generated Simulation Models", submitted to *IEEE International High Level Design Validation and Test Workshop*, 1999.
12. Tuggle E., "Writing Device Drivers", *Embedded Systems Programming*, Jan. 1993, pp. 42-65.
13. S. Vercauteren, B. Lin, "Hardware/Software Communication and System Integration for Embedded Architectures", *Design Automation for Embedded Systems*, Vol. 2, No. 3/4, Kluwer Academic Publisher, 1996, pp. 359-382.
14. D. Verkest, K. Van Rompaey, I. Bolsens, H. De Man, "CoWare - A Design Environment for Heterogenous Hardware/Software Systems", *Design Automation for Embedded Systems*, Vol. 1, No. 4, Kluwer Academic Publisher, 1996, pp. 357-386.
15. J. Öberg, A. Kumar, A. Hemani, "Grammar-based Hardware Synthesis of Data Communication Protocols", In *Proc. of the 9th ISSS*, Nov. 1996, pp. 14-19.