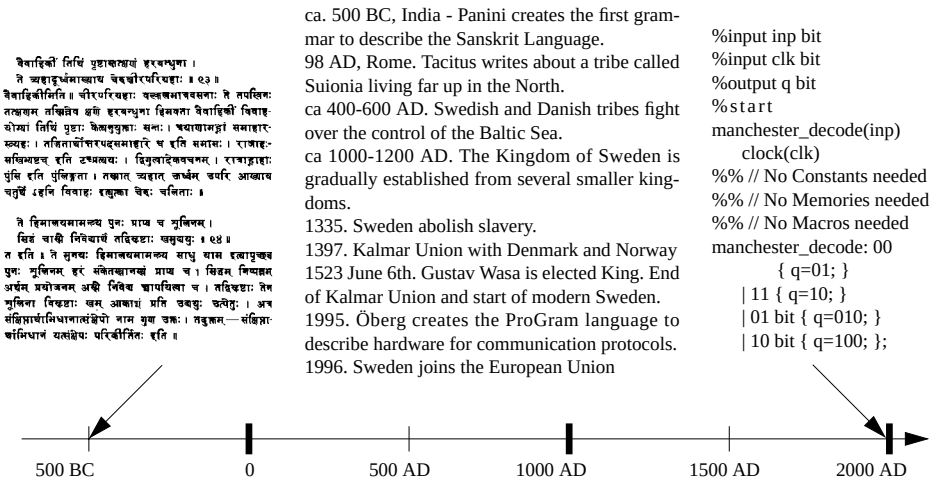
ISRN KTH/ESD/ AVH--99/3--SE
ISSN 1104-8697
TRITA-ESD-1999-03

Royal Institute of Technology

ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols

Johnny Öberg



Stockholm 1999

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

ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols

Johnny Öberg

Thesis submitted to the faculty of
Electrical Engineering in partial fulfilment
of the requirements for the degree of
Doctor of Technology

Stockholm 1999

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

ISRN KTH/ESD/ AVH--99/3--SE
ISSN 1104-8697
TRITA-ESD-1999-03

Abstract

A data communication protocol is an agreement between two or more communication parties about the exchange of messages in order to provide some service. The protocol specifies the language used to communicate over the interface between the involved systems. A natural way to specify an interface protocol is in terms of a grammar expressed in the BNF notation and annotated with actions. The vocabulary of messages used to implement the protocol correspond to the grammar rules. The encoding format of each message in the vocabulary corresponds to the token terminals of the grammar. The procedure rules guarding the consistency of message exchanges are also embedded in grammar rules. The service provided by the protocol corresponds to the actions in a grammar. Assumptions about the environment can be viewed as port width and throughput constraints posed to the synthesis process.

ProGram is a grammar-based specification language aimed for specification of protocols, interfaces and control dominated functionality. The language allows a designer to specify frame-based protocols in a port-size independent manner. Port sizes are used as constraints to derive the physical implementation.

The ProGram description is taken as an input to the ProGram Compiler. Input sequences are partitioned into a sequence of tokens, the size of the port width constraint for the input stream. Output sequences are partitioned into a sequence of output assignments, the size of the port width constraints for the output streams, and scheduled over the available input tokens.

The ProGram language and the ProGram compiler has established a working methodology for design space exploration of port sizes. The code of the language is compact and the results after hardware synthesis are comparable to hand-written designs.

Hardware synthesis from ProGram is a sort of High-level synthesis with different emphasis. High-level synthesis frees the designer from details like when an operation is scheduled, how many and what types of hardware resources are required, operation to hardware resource instance mapping. These absence of details are then exploited by the synthesis tool as degrees of freedom to explore design space. For instance, High-level synthesis can create a faster design by allocating more resource and scheduling operations in parallel or vice versa. ProGram extends this by making the specifications independent of communication details like port width. Synthesis tools can then create a faster design by using a wider port or a slower design by using narrower port. This feature is very useful in communication applications. For instance, UTOPIA, the standard ATM interface, specifies a 16 bit interface, whereas some ATM vendors implement 128 bit ports for a higher bandwidth. The ability to do such trade-offs without having to modify the specification is the principal contribution of this work.

To my late grandmother Therés, who
always wanted a doctor in the family

Preface

As usual when a thesis like this has been written, there are a lot of people without whom the thesis could never have been written. The people that has influenced me the most is without a doubt my parents and my brothers. The math quizzes that my parents gave me and my siblings in the car when we drove north during vacations together with watching the first lunar landings on TV with my parents, early awoke my curiosity for math and science.

Then of course, all the teachers that I have had in school throughout the years have also influenced me. There is one teacher in particular that has influenced me more than others: Ingvar Forsberg, my teacher in math and physics in 8th and 9th grade in “Grundskolan”. He had the notion that the pupils should write down any observations, ideas or inventions that they might get during labs, classes or at home and he encouraged me to do the same. I thought this was a marvellous idea, having a lot of ideas myself. I soon found the benefit of writing them down. When an idea is put on paper it forces you to focus on how it really works in order to make it understandable for others. Ideas are also much harder to forget once on paper. (It is also rather fun to see how I used to reason when I was fourteen years old.)

Another person that has influenced me is one of our summer neighbours, Gunnar Grangård. One summer in my late teens, I was trying to explain to him how my latest summer project (an audio sampler that I tried to use to let my CBM-64 understand spoken commands) worked, he told me “Johnny, I understood maybe 10% of what you said. It would be much more beneficial if you could find better ways of communication that would allow people to understand 90% of what you say”. I have thought on these words many times, and I find myself coming back to them time after time. I have realized that knowledge is like a picture kept imprisoned in once head and that teaching is to paint a copy of that picture inside the student’s head. Anyway, I learned the value of keeping a presentation on the level of the student. It has been a great help during classes when I have had to teach.

Of the people that work here at the department of Electronics I would like to thank Prof. Hannu Tenhunen for providing me with the opportunity to do research in an area which I find very interesting. Dr. Mehran Mokhtari for convincing me that Electronic System Design is what I find interesting and for all the funny discussions in the “Institute of Fuzzy Science”. My supervisor, Docent Ahmed Hemani for providing me with challenging tasks and problems to solve. Dr. Axel Jantsch for being a constant source of inspiration and support. Dipl. Ing. Peeter Ellervee for all the help, all the fuzzy discussions and for teasing me into doing the impossible. His price for help is always two beers, although the size of the beers is negotiable. I owe him two really large tankards by now!

I would also like to thank Prof. Anshul Kumar of the Indian Institute of Technology in New Delhi, India for his extremely valuable collaboration during various parts of the this work and Prof. Shashi Kumar, also he from IIT New Delhi, India for the interesting discussions we had during his stay here and for his suggestions of improvement on various parts of this thesis. I would also like to thank my opponent Prof. Forrest Brewer, for taking the time to read it.

Other people that deserves mentioning is Hans Berggren, Richard Andersson and Julio Mer-

cado in the system group, for keeping the computers in good shape, Aila Pohja for keeping an eye on things when she was here and Lena Beronius for keeping an eye on things around here now. I would also like to thank all colleagues at ESDLab, the high-speed group at FMI, Henk Martijn at IMC and all my other friends for their support and friendship.

Last but not least I would like to thank all my lovely wife Helena for her constant support and understanding, and my son Oscar for being the little beam of sunshine that he is. I hope that I and Helena will be able to offer him an as inspiring environment as my parents have offered me. Finally, I would like to dedicate this thesis to my late grandmother Therés. She always wanted to have a doctor in the family, although I am quite sure that my line of study was not the one that she had in mind.

Front Cover

The picture is an attempt to illustrate the enormous time-span of 2500 years that has passed since Panini made his first grammar. I have chosen to illustrate the time axis by selected events in about 500 year intervals from one of my favourite hobbies: the Swedish history. By the time Panini wrote his grammar, we had early Iron age up here in the North and no one had ever heard of us in the more civilized parts of the world. Many things have happened here since then.

The text on the left is part of an exercise from the book “Teach yourself Sanskrit” by Michael Coulson. It is an extract from the text Kumara-sambhava ‘The birth of Kumara’ by the poet Kalidasa. It is commented by the medieval scholar Mallinatha, who is quoting various sutras (rules) of the Paninian grammar. The text on the right is one of the simplest communication protocols that exist, a Manchester Decoder, written in the ProGram language.

Reading Advice

I have tried to write this thesis so that not only people familiar with the area, but also people from neighbouring and other areas will find it interesting and understandable. Unfortunately, I have not managed to do so throughout the whole thesis. Some parts are really dense, especially in the sections that describe the details of the ProGram language and the ProGram compiler. People not from my area should consider to skip the main body of these sections and only read the introductions.

Stockholm, April 1999
Johnny Öberg

Table of Contents

List of Papers included in the thesis

I **Grammar-based Hardware Synthesis of Data Communication Protocols**
J. Öberg, A. Kumar, A. Hemani, In Proc. of ISSS-96, pp. 14-19, (1996)

II **Comparing Conventional HLS with Grammar-Based Hardware Synthesis: A Case Study**
J. Öberg, P. Ellervee, A. Kumar, A. Hemani, In Proc. of NorChip-97, pp. 52-59, (1997)

III **Scheduling of Outputs in Grammar-based Hardware Synthesis of Data Communication Protocols**
J. Öberg, A. Kumar, A. Hemani, In Proc. of DATE-98, pp. 596-603, (1998).

IV **Specification of Exception Handling in Grammar-based Hardware Synthesis**
J. Öberg, A. Kumar, A. Hemani, (poster paper), In Proc. of EuroMicro'98, Vol. I, pp. 38-41, (1998).

V **Synthesis of Exception Handling in Grammar-based Hardware Synthesis**
J. Öberg, A. Kumar, A. Hemani, In Proc. of APCHDL-98, pp. 135-140, (1998).

VI **Specification and Synthesis of Exception Handling in Grammar-based Hardware Synthesis**
J. Öberg, A. Kumar, A. Hemani, S. Kumar, Accepted for publication in the Journal of Electrical Engineering and Information Science, Korea.

VII **Transition Graph representation of FSMs and its application to State Minimization in the ProGram Compiler**
J. Öberg, P. Ellervee, S. Kumar, TRITA-ESD-1998-01, parts of the paper will be rewritten and submitted either as a Transactions brief or to Electronic Letters.

VIII **Grammar-Based Hardware Synthesis from Port Size Independent Specifications**
J. Öberg, A. Kumar, A. Hemani, Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

Publications not included in the thesis

Related papers

[1] M. O'Nils, J. Öberg, A. Jantsch, "Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces", (poster paper), In Proc. of EuroMicro'98, Vol. I, pp. 55-58, Västerås, Sweden, Aug. 25-27, 1998.

[2] J. Öberg, A. Jantsch, A. Hemani, "Validation of Interface Protocols Using Grammar-based Models", In the Proc. of the IEEE International High Level Design Validation and Test Workshop (HLDVT'98), pp. 40-46, La Jolla, California, Nov. 12-14, 1998.

[3] J. Öberg, P. Ellervee, A. Hemani, "Grammar-based Modelling of Clock Protocols for Low-Power Implementation: A Case Study", In Proc. of NorChip-98, pp. 144-153, Lund, Sweden, Nov. 9-10, 1998.

[4] A. Jantsch, J. Öberg, A. Hemani, "Is there a Niche for a General Purpose Protocol Processor?", In Proc. of NorChip'98, pp. 93-100, Lund, Sweden, Nov. 9-10, 1998.

[5] A. Hemani, J. Öberg, A. Kumar Deb, D. Lindqvist, B. Fjellborg, "System Level Prototyping of DSP ASICs using grammar based approach", Accepted for inclusion in the Proc. of Rapid System Prototyping (RSP'99).

Papers about HLS and Hw/Sw Co-Design

[6] A. Jantsch, P. Ellervee, J. Öberg, A. Hemani, "A Case Study on Hardware/Software Partitioning", In Proc. of IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94), pp 226-231, April 11-13, 1994.

[7] A. Jantsch, J. Öberg, P. Ellervee, A. Hemani, "A software oriented approach to hardware-software co-design", (poster paper), In Proc. of the Poster session of the International Conference on Compiler Construction (CC-94), Edinburgh, Scotland, April 1994.

[8] J. Isoaho, J. Öberg, A. Hemani, H. Tenhunen, "High Level Synthesis in DSP ASIC Optimization", In Proc. of 7th IEEE ASIC Conference and Exhibit (ASIC'94), pp 75 - 78, Rochester, New York, Sept. 1994.

[9] P. Ellervee, A. Jantsch, J. Öberg, A. Hemani, H. Tenhunen, "Exploring ASIC Design Space At System Level with a Neural Network Estimator", In Proc. of 7th IEEE ASIC Conference and Exhibit (ASIC'94), pp 67 - 70, Rochester, New York, Sept. 1994.

[10] A. Jantsch, P. Ellervee, J. Öberg, A. Hemani, H. Tenhunen, "Hardware-Software Partitioning and Minimizing Memory Interface Traffic", In Proc. of EURO-DAC'94, pp 226 - 231, Grenoble, France, Sept. 1994.

[11] P. Ellervee, J. Öberg, A. Jantsch, A. Hemani, "Neural Network Based Estimator to Explore the Design Space at System Level", In Proc. of the 4th Biennial Baltic Electronic Conference, pp 391-396, Tallinn, Estonia, Oct. 1994.

[12] J. Isoaho, J. Öberg, A. Hemani, H. Tenhunen, "HLS based DSP Optimization with ASIC RTL Libraries", (poster paper), In VLSI Signal Processing VII, J. Rabaey, P.M. Chau and J. Eldon, editors, pp 218 - 225, IEEE Inc., New York, 1994.

[13] A. Hemani, B. Svantesson, P. Ellervee, A. Postula, J. Öberg, A. Jantsch, H. Tenhunen, "Trade-offs in High-level Synthesis of Telecommunication Circuits", In Proc. of the 5th Workshop on Synthesis And System Integration (SASIMI'95), pp 65-72, Nara, Japan, Aug. 25-26, 1995.

[14] A. Hemani, B. Svantesson, P. Ellervee, A. Postula, J. Öberg, A. Jantsch, H. Tenhunen, "High-level Synthesis of Control and Memory Intensive Communication Systems", In Proc. of the 8th Annual IEEE International ASIC Conference and Exhibit (ASIC'95), pp 185-191, Austin, Texas, Sept. 18-22, 1995.

[15] B. Svantesson, A. Hemani, P. Ellervee, A. Postula, J. Öberg, A. Jantsch, H. Tenhunen, "Modelling and Synthesis of Operational and Management System (OAM) of ATM Switch Fabrics", In Proc. of the 13th NORCHIP Conference (NORCHIP'95), pp 115-122, Copenhagen, Denmark, Nov. 7-8, 1995.

[16] B. Svantesson, P. Ellervee, A. Postula, J. Öberg, A. Hemani, "A Novel Allocation Strategy for Control and Memory Intensive Telecommunication Circuits", In Proc. of the VLSI Design'96 Conference, pp 23-28, Bangalore, India Jan. 3-6, 1996.

[17] J. Öberg, J. Isoaho, P. Ellervee, A. Jantsch, A. Hemani, "A Rule-Based Allocator for Improving Allocation

of Filter Structures in HLS", In Proc. of the VLSI Design'96 Conference, pp 133-139, Bangalore, India, Jan. 3-6, 1996.

[18] J. Öberg, P. Eles, A. Hemani, K. Kuchcinski, Z. Peng, "Specifying Local Timing Constraints for HLS of Digital Systems in VHDL". In Proc of the 3rd Asian Pacific Conference on Hardware Description Languages (APCHDL'96), pp 145-149, Bangalore, India, Jan. 8-11, 1996.

[19] P. Ellervee, A. Hemani, A. Kumar, B. Svantesson, J. Öberg, H. Tenhunen, "Controller Synthesis in Control and Memory Centric High-Level Synthesis System", In Proc. of the 5th Biennial Baltic Electronic Conference, pp 393-396, Tallinn, Estonia, Oct. 7-11, 1996.

[20] F. Bueno, J. Öberg, A. Kumar, M. Torkelsson, "High-Level Synthesis of a 1 Mbps Direct-Sequence Spread-Spectrum RAKE Receiver", In Proc. of the 6th Workshop on Synthesis And System Integration (SASIMI'96), pp 170-177, Fukuoka, Japan, Nov. 25-26, 1996.

[21] J. Öberg, A. Kumar, A. Jantsch, "An Object-Oriented Concept for Intelligent Library Functions", In Proc. of the VLSI Design'98 Conference, pp. 355-358, Chennai, India, Jan. 4-7, 1998.

[22] A. Jantsch, S. Kumar, I. Sander, B. Svantesson, J. Öberg, A. Hemani, "Comparison of Six Languages for System Level Descriptions of Telecom Systems", In Proc. of FDL'98, Vol. 2, pp. 139-148, Lausanne, Switzerland, Sept. 6-11, 1998.

[23] T. Meincke, A. Hemani, P. Ellervee, J. Öberg, S. Kumar, D. Lindqvist, H. Tenhunen, A. Postula, "Evaluating benefits of Globally Asynchronous Locally Synchronous VLSI Architecture.", In Proc, of NorChip'98, pp. 50-57, Lund, Sweden, Nov. 9-10, 1998.

[24] T. Meincke, A. Hemani, S. Kumar, P. Ellervee, J. Öberg, T. Olsson, P. Nilsson, D. Lindqvist, H. Tenhunen, "Globally Asynchronous Locally Synchronous VLSI Architecture for large high-performance ASICs", Accepted for publication in ISCAS'99.

[25] A. Hemani, T. Meincke, T. Olsson, P. Nilsson, J. Öberg, P. Ellervee, S. Kumar, D. Lindqvist, "Lowering Power Consumption in Clock by Using Globally Asynchronous, Locally Synchronous Design Style", Accepted for publication in DAC'99.

Other papers

[26] T. Lazraq, F. Östman, J. Öberg, H. Tenhunen, "ATM Switching System Performance Analysis via Modelling and Simulation", In Proc. of SIMS'94 Conference, pp 326-332, Stockholm, Aug. 17-19, 1994.

[27] T. Lazraq, J. Öberg, H. Tenhunen, "FPGA Basic ATM Traffic Shaper for Event-Building Networks", In Proc of the 8th Annual IEEE International ASIC Conference and Exhibit (ASIC'95), pp 177-180, Austin, Texas, Sept. 18-22, 1995.

[28] J. T.A. Waldemark, T. Lindblad, C. S. Lindsey, K. E. Waldemark, J. Öberg, M. Millberg, "Pulse Coupled Neural Network implementation in FPGA", In Proc. of SPIE AreoSense '98, the Conference on Applications and Science of Computational Intelligence, Orlando, Florida USA, 13-17 April 1998.

[29] M. Millberg, J. Öberg, J. Waldemark, "Generic VHDL Implementation of a PCNN with Loadable Coefficients", Presented at the VIDYNN'98 Workshop, Stockholm, Sweden, June, 1998.

[30] J. Öberg, P. Ellervee, "Revolver: A high-performance MIMD architecture for collision free computing", In Proc. of EuroMicro'98, pp. 301-308, Västerås, Sweden, Aug. 25-27, 1998.

Theses

[31] J. Öberg, "Tuning of JET Transmission Line/Antenna System During ICRH", Master's Thesis, Fusion Plasma Physics, Alfvén Laboratory, TRITA-ALF-1993-06.

[32] J. Öberg, "An Adaptable Environment for Improved High-Level Synthesis", Licentiate Thesis, Electronic Systems Design, TRITA-ESD-1996-14.

+ 4 international posters (page size <4), 3 national publications, 9 additional technical reports in the KTH TRITA-series, and 2 submitted papers

List of Abbreviations

AFAP	As Fast As Possible
ALAP	As Late As Possible
ASAP	As Soon As Possible
ATM	Asynchronous Transfer Mode
BNF	Backus-Naur Form
CDFG	Control and Data Flow Graph
DAG	Directed Acyclic Graph
DFA	Deterministic Finite Automaton
DMA	Direct Memory Access
FSM	Finite State Machine
FSMD	Finite State Machine with a Datapath
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
LALR	Lookahead LR
LBA	Linear Bounded Automaton
LHS	Left-Hand Side
LL	Left-Leftmost parsing
LR	Left-Rightmost parsing
NFA	Non-deterministic Finite Automaton
OAM	Operation And Maintenance
PCI	Personal Communication Interface
RHS	Right-Hand Side
RTL	Register Transfer Level
STG	State Transition Graph
SW	Software
THG	Transition Hyper Graph
TG	Transition Graph
TM	Turing Machine

1. Introduction

In this chapter the reader is introduced to Digital System design and Design Automation in general. Its history is reviewed and reasons for why there is a need for better methods as chip sizes increase towards 1 Billion transistors/chip is presented to the reader.

1.1. Introduction

Moore's Law [1] predicts a doubling of the number of transistors that can be made on a chip every 18 months. At present time, the process people are continuing to achieve these performances and the production lines have been able to fulfil the predictions. One problem is that the development of tools do not keep up with this development, which has led to a productivity gap. To be able to develop the everly increasingly bigger sized systems has been met with hiring more people. However, as chip sizes increase into the region of 1 Billion transistors per chip, new tools (and computer systems) that can handle this kind of complexity are needed.

The history of design automation begins around the time when the first computers come, but it was not until the end of the 70s that the computer systems were becoming advanced enough to handle bigger design problems. Before 1979, about 40% the effort in designing a 20 kGates circuit was spent in the physical design [2]. Around the year 1983, tools for performing placement and routing had appeared on the market, reducing the time spent for physical design from 70 Person-Months to 2. At the same time had improvements on the system design reduced from 40 to 30 Person-months by the introduction of simulation and from 60 to 50 Person-Months by the introduction of schematic capture. The introduction of hierarchy and hardware generators had by 1986 reduced the time spent in logic design to around 40 Person-Months. Between 1988 and 1992, the emergency of logic synthesis brought down this figure to 2 Person-Months, but the time spent in system design was not reduced at all. Around 1995, High-Level Synthesis (HLS) had been introduced, which reduced the design time to 10 Person-Months.

However good in performing computation intensive application like Digital Signal Processing (DSP) algorithms, High-Level Synthesis has proven inefficient for implementing communication intensive specifications. Several attempts to extend HLS to cope with the communication has been proposed. They will be covered more deeply in section 1.4.1. Although the proposed methods work, they all suffer from being generated from being specified in a procedural manner. procedural descriptions are good in capturing the computation behaviour of a design. They are not so good however in capturing the control flow of a design. The many levels of control constructs like IF-THEN-ELSE and CASE-SELECT etc. makes the descriptions unreadable and the communication behaviour difficult to grasp.

With the introduction of interface-based synthesis, these difficulties are overcome. In interface-based design descriptions, the communication behaviour comes first and the computation behaviour second. It is the author's belief that interface-based synthesis will reduce the design time for performing the system design of a 20 kGates design to the same order as for the phys-

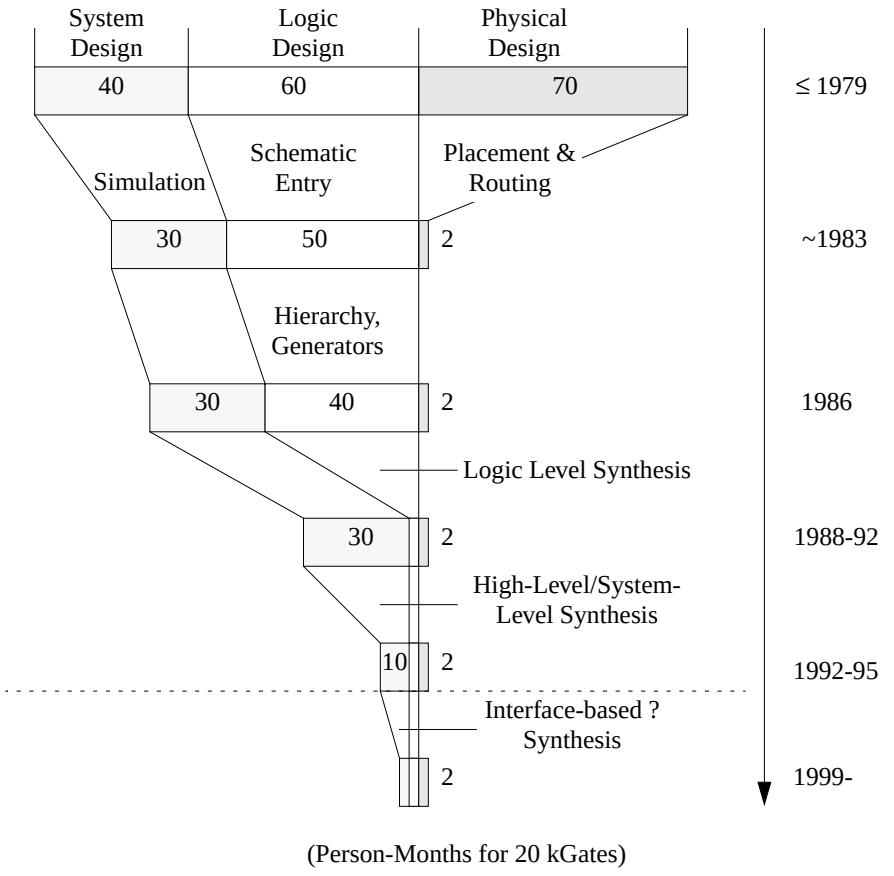


Figure 1.1. Effort spent in developing a 20kGates design at various design steps over the years. (Modified from [2])

ical and logic design phases.

The rest of this chapter will be organized as follows. In section 1.2., the different design paradigms in terms of levels of abstraction and domains will be presented. In section 1.4., High-Level Synthesis will be presented. Section 1.4.1. covers High-Level Synthesis for Control and Memory Intensive Applications for Telecommunications (CMIST). Finally, in section 1.5., some conclusions will be drawn and the rest of the thesis will be outlined.

1.2. Domains and levels of description

To be able to develop tools for design automation it is important to have conceptual models of the different domains and levels of descriptions if the designs we're trying to model. In 1983, Gajski and Kuhn introduced the Y-chart for describing the taxonomy of design automation in electronic systems [3, 4]. It has proven to be a very valuable reference point and is up to today the most commonly used method for taxonomy of electronic systems. The Y-chart is shown in

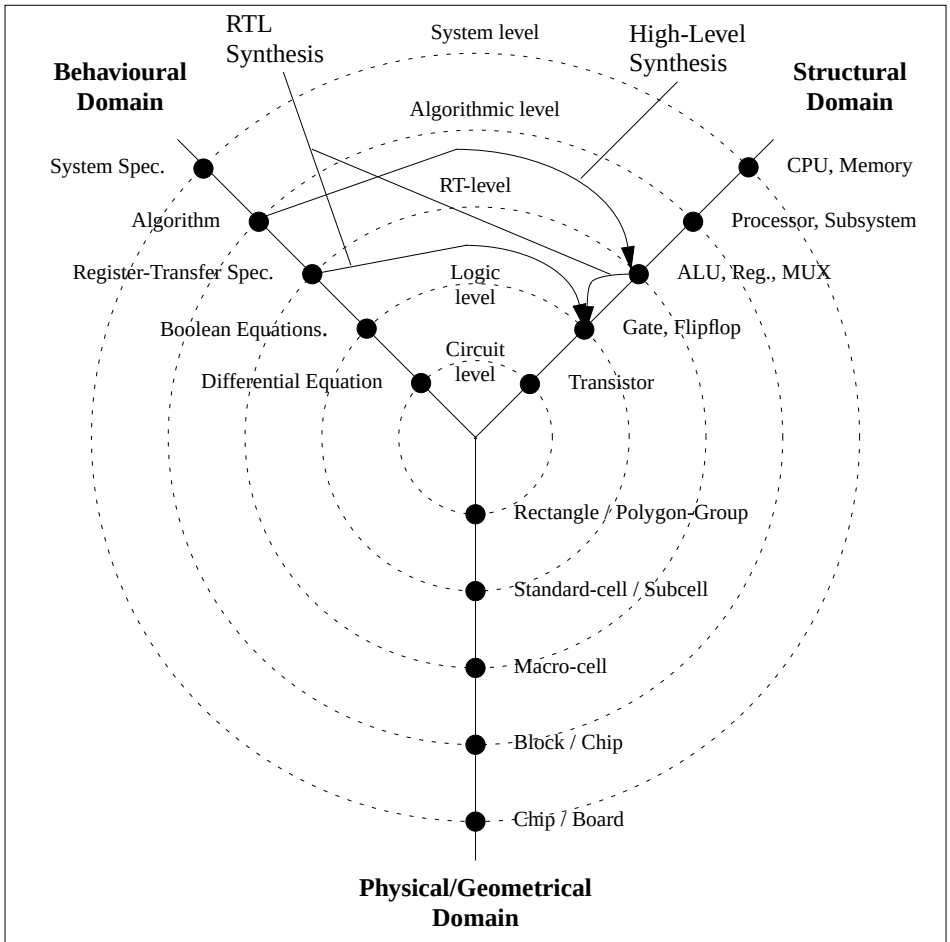


Figure 1.2. Domains and levels of description in the Gajski Y-chart

Figure 1.2. The Y-chart consists of three axis describing the domains, the *Behavioural Domain*, the *Structural Domain* and the *Physical Domain*. Centred around the origin of the axis are concentric circles, which are describing the levels of abstraction in the system. A synthesis task can then be viewed as a transformation from one axis to the other, and/or as a transformation from one level to another (lower) level. High-Level Synthesis can for instance be described as a transformation from the algorithm level in the behavioural domain to the ALU, Reg, Mux level in the Structural domain. Behavioural RTL synthesis is then a transformation from the Register Transfer specification in the behavioural domain to the Gate, Flip-flop level in the structural domain etc.

Other design representations have been presented over the years that expands the Y-chart or provides a different framework, but none has yet proven to be more advantageous than the other or over the Y-chart. The X-chart [6] introduce testing as the fourth axis. The Multi-level Cybernetic [7] can model design processes and design strategies. The Design Cube [8] is a framework for modelling design activities in the VHDL-environment and, unlike the other frameworks, can also model time and data at different levels of abstractions.

The Rugby model [9], finally, is slightly different from the other frameworks. It models the

design process as it expands from an idea to a physical system over time. It covers and relates all design phases from requirements to implementation in four dimensions: Computation, Communication, Data and Time. It also allows studying of the HW/SW Codesign process when each of the dimensions split into two, one dimension for modelling the hardware aspects and the other for modelling the software aspects, as the dimension goes toward lower abstraction levels. An interesting aspect of the Rugby model is a fifth dimension, orthogonal to the other four, called Design Manipulation. It represents different abstraction levels in the tools used, with a low abstraction level assigned to design entry editors, a medium abstraction level for synthesis algorithms and the highest abstraction level for design methodologies.

In contrast to the Y-chart, in which a synthesis task is described in a very intuitive and distinct way as a transformation or refinement along or between an axis/axes, classification of different synthesis tasks in the Rugby-model is less intuitive because of the complex description of a design. In the Rugby-model, a design is always represented at some abstraction level in all four design dimensions. A synthesis task must then be described as refinements and transformations in all four design dimensions at the same time.

Another thing that is unclear is the Communication dimension. It does not provide any abstractions when it comes to message exchanges as is the case for communication protocols. Instead it provides abstractions for the interconnect wiring, i.e. the interface. The dimension for FSM abstraction is instead the Computation dimension. Thus, the model does not differ between communication and behaviour and provides only the same type of abstractions as the Y-chart model for describing the abstraction of the message exchange behaviour when data communication protocols are concerned.

1.3. Design Space Exploration

The reason to move on to higher and higher abstraction levels is that at a higher level of abstraction, less details about the actual implementation is needed to specify. Instead, the designer controls the details of the implementations by posing constraints and design goals to a synthesis tool. The synthesis tools then derive an implementation from the constraints and design goals posed by the designer. The advantage of this method is that the design is decoupled from the lower levels details of the implementation, like the used technology etc. It would improve the designer's productivity by having to specify less and the design is also easy to move across different technology platforms. It is enough to resynthesize the design using another target technology.

Another advantage is that the solution space that can be reached is larger. At every design level, the number of possible solutions that can be reached is drastically reduced, as soon as a decision is made about the implementation. Consider the example in Figure 1.3 [5]. The system specification is a sorting algorithm of some kind that should be run at some speed. As soon as the Bubble Sort algorithm is selected, all possible implementations that are based on another algorithm cannot be reached any more. The designer chooses algorithm from experience and his/her knowledge about the problem, the technology and the given constraints. At the next levels, several possible RTL implementations of the algorithm are possible. The designer then chooses the RTL design that will be cheapest to implement, given some important metric. Finally, a gate-level netlist is produced by the synthesis tool after applying some optimization directives. Depending on what directives that are given, different netlists that will appear.

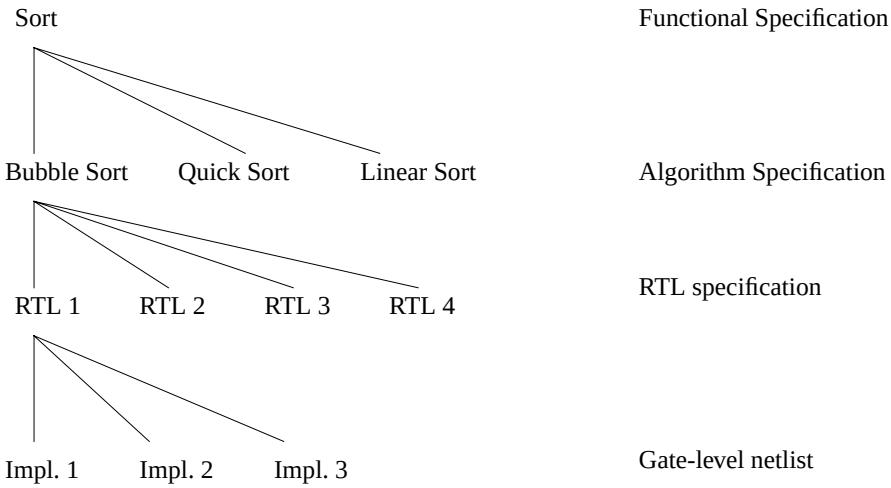


Figure 1.3. Example Design Space of a Sorting Algorithm

To make a good decision at an early stage of the design process, good estimation algorithms are needed. The advantage of doing the decision at an higher level of abstraction is not only that a larger solution space can be reached, but also to speed up the design process. After performing design space exploration, the alternative designs are evaluated. If the design does not meet the performance requirements the specification is changed and another design space exploration is performed. This process is iterated until an acceptable solution is found. Such a methodology is termed specify-explore-refine methodology [15]. Design space exploration is a complex task in general, but can be more manageable in a specific application domain

1.4. High-Level Synthesis

High-Level Synthesis [10, 11, 12] is the task of translating a description of a design at the algorithmic level and translate it into a Register Transfer (RT) level description or an RT-level netlist. Since most of the tool designers had been working with implementing DSP-algorithms before the research started with HLS in the middle of the 1980's, DSP-algorithms were the obvious choice from which the synthesis would proceed. Therefore, current high-level synthesis is mostly targeted for computation intensive applications like the FFT, FIR- and IIR-filters, etc. The principles of High-Level Synthesis are shown in Figure 1.4.

In High-Level Synthesis, a description of an algorithm in a high-level language like VHDL is translated into a Control and Data Flow Graph (CDFG). The operations in the graph are examined from the ranges that they will have on their inputs and the necessary bit widths are calculated. Then, functional units (FUs) are allocated that can perform the operation and have enough bits in their datapath are allocated. The operations are then scheduled over a given number of control steps, the number of control steps that the sample period has been divided into. After the operations has been scheduled, the number of FUs that will be needed to be able to perform the functionality in the given number of periods are determined. The operations are then bound to functional units. An operation is bound to a FU that is not used for other things

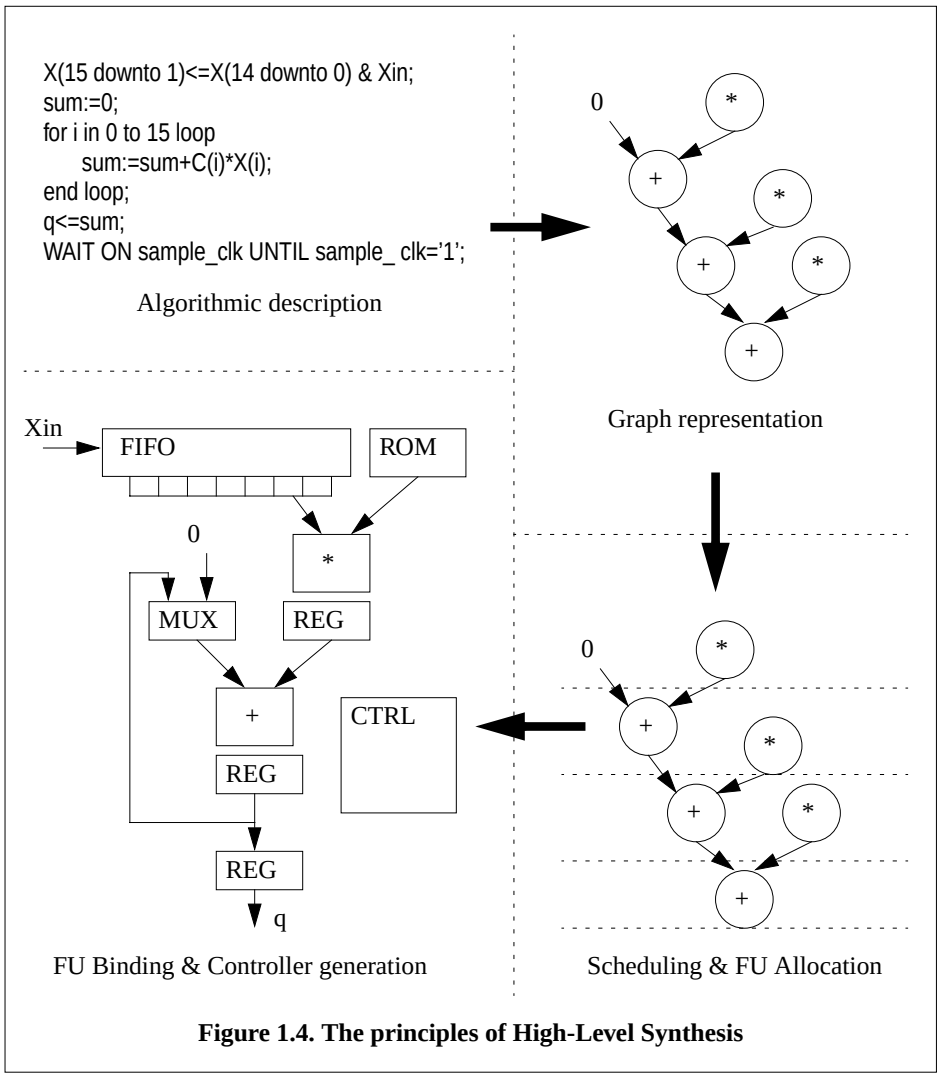


Figure 1.4. The principles of High-Level Synthesis

difficult to solve and heuristics are needed in order to get a solution in reasonable time. Finally, the controller that control the muxes in the system is generated.

1.4.1. HLS for CMIST

HLS was originally developed to synthesize DSP algorithms and it has proven to be inefficient for implementing communication intensive specifications. Ordinary HLS is primarily concerned with reuse and optimization of arithmetic functional units. In telecommunication systems, targeted for routing, switching etc., the functionality is dominated by control decision and interactions with memory, with little or no arithmetic operations. There has been several suggestions on how to extend HLS to cope with control dominated functionality [17, 18, 19, 20, 21, 22]. While the work by W. Wolf [17] presents a model for performing behavioural synthesis, the other papers address the problem of global scheduling of the of the datapath operations into control states.

This design domain has been termed CMIST, from Control and Memory Intensive Telecommunication Circuits [16]. In HLS for CMIST designs, the focus is on synthesizing a controller, without concern for the functionality that will be handled by a (small) datapath. In traditional HLS, the focus is on synthesizing the datapath, and the generation of controller is a secondary issue that only reflect the results of datapath synthesis.

When communication is concerned, the actual waveform of the interface, i.e. the actual implementation of the control path is the crucial design problem. Although the suggested approaches work for performing communication synthesis, they all suffer from being generated from being specified in a procedural manner. procedural descriptions are good in capturing the computation behaviour of a design. They are not so good however in capturing the control flow of the signalling performed over an interface. The many levels of control constructs like IF-THEN-ELSE and CASE-SELECT etc. makes the descriptions unreadable and the communication behaviour difficult to grasp.

1.4.2. HLS for data communication protocols

A data communication protocol is an agreement between two or more communication parties about the exchange of messages in order to provide some service. The five elements of a protocol that are needed for a complete specification [25] are listed in Figure 1.5.

1) The service to be provided by the protocol.
2) The assumptions about the environment in which the protocol is executed.
3) The vocabulary of messages used to implement the protocol.
4) The encoding format of each message in the vocabulary.
5) The procedure rules guarding the consistency of message exchanges.

Figure 1.5. Protocol Specification Requirements

A natural way to specify the elements 1, 3, 4 and 5 is in terms of a grammar expressed in the BNF (for Backus-Naur Form) notation [56] and annotated with actions. The vocabulary of messages used to implement the protocol correspond to the grammar rules. The encoding format of each message in the vocabulary corresponds to the token terminals of the grammar. The procedure rules guarding the consistency of message exchanges are also embedded in grammar rules. The service provided by the protocol corresponds to the actions in a grammar. We view the element 2, i.e. the assumptions about the environment, as port width and throughput constraints posed to the synthesis process.

ProGram (for Protocol Grammar) is a grammar based specification language aimed for specification of protocols, interfaces and control dominated functionality. Synthesis from grammar-based descriptions result in efficient hardware. In addition, the ProGram methodology allows to perform design space exploration of the port-sizes by letting the designer to pose the input and output port-sizes as constraints to the tools. The input port constraints are used to partition the message specified by the grammar rules into chunks of appropriate size. The output port constraints are used in the same way to partition the output assignments into chunks of appropriate size. The partitioned output assignments are then scheduled over the available stages of the input message.

The ProGram language uses a BNF-like notation to code both input and output sequences. Incoming and outgoing sequences/messages are coded as bit-streams. Bit-streams have no natural “white space”-characters because the width of the bit-stream and the interpretation of the

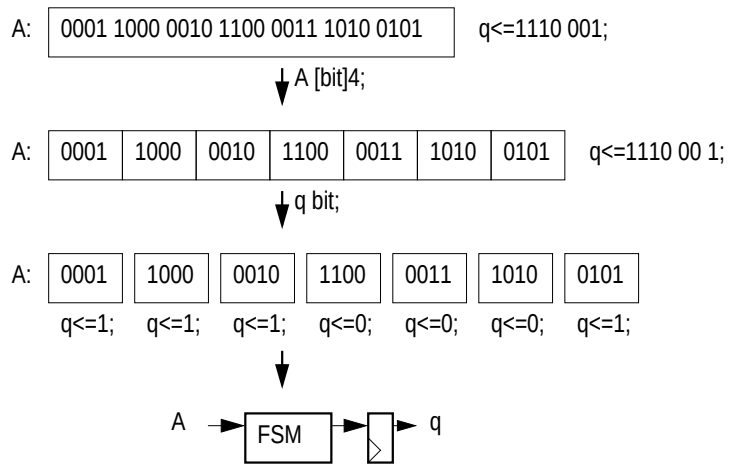


Figure 1.6. Overview over the Message Scheduling performed by the ProGram compiler

bit-patterns varies from application to application. The input production rules specifies in an hierarchical way all valid input sequences that the protocol should recognise and the output production rules specifies the output sequences that are associated with each input. The input sequences are partitioned into tokens, the size of the input stream and reduced during the synthesis steps to remove non-determinism from the internal representation, the grammar DAG. The output sequences are partitioned into tokens, the size of the output streams, and scheduled over the available input tokens, see Figure 1.6.

The ProGram language has been extended to include exception handling. The inclusion of exception handling constructs results in constraints which make the previous output scheduling algorithms produce inefficient or, in some cases, wrong results.The synthesis algorithms have been extended to handle the new exception constructs, but the extended scheduling algorithms still have a few limitations.

1.5. Conclusion and Outline of the thesis

Research in High Level Synthesis has made it possible to describe designs at behavioural level in languages like VHDL and to synthesize detailed circuits automatically. However, at the level of description supported by the present day HLS tools, a designer still cannot work entirely independent of the implementation. The need for languages and synthesis tools to support higher levels of design specification is evident from the recent research in system level synthesis[13][14].

Another problem with HLS research is that it has been focused on the high level specification and synthesis of DSP algorithms that are sampled on a single clock. This is not a common situation. Many systems have multi-rate components. Also, when a system is being designed, the hardware usually has to communicate with other units in the system. Thus, the situation displayed in Figure 1.7 is more close to reality. If, in addition, the communication is conducted over a bus with a complex data communication protocols, the ability to describe complex state machine in an easy way is crucial and the languages that are currently used do not capture this

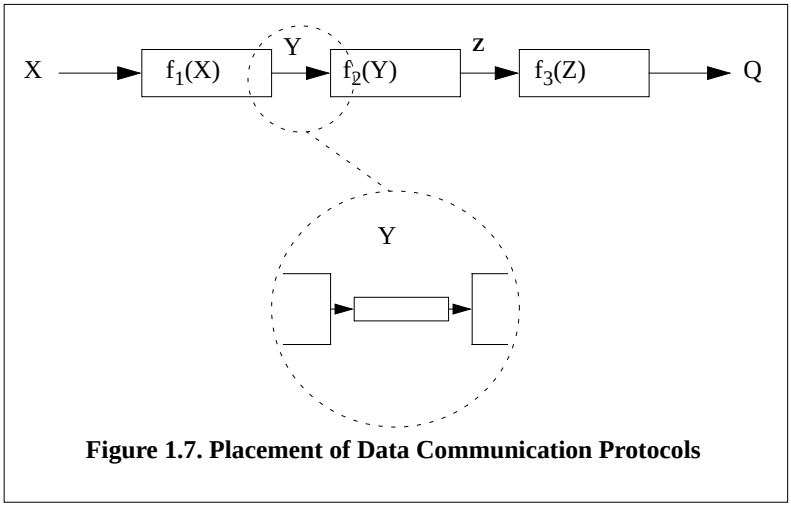


Figure 1.7. Placement of Data Communication Protocols

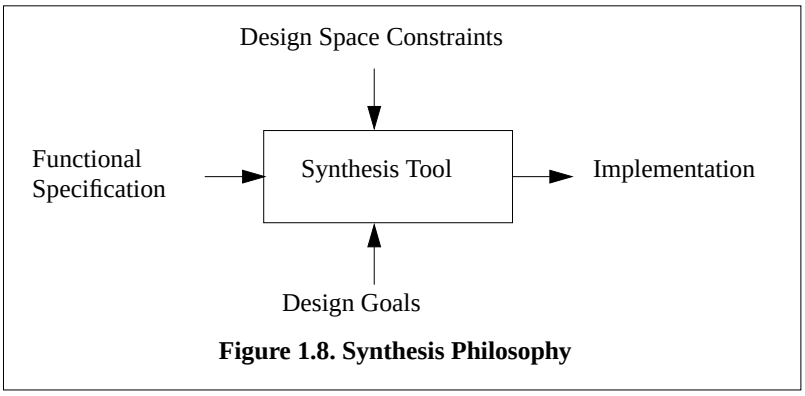


Figure 1.8. Synthesis Philosophy

communication behaviour in an easily understandable manner.

Design space exploration at high levels of abstraction is also an important task. With this methodology, we are able to make good decisions at an early stage of the design process.Then, a larger solution space can be reached, and in turn also speed up the design process. Also, if the decisions are made on a higher level of abstraction, and a synthesis tool is used to derive the implementation, the design can be retargeted to different constraint needs, environments and technologies.

We would therefore like to have a synthesis tool that supports a description style that is as implementation independent as possible. The input to such a tool is shown in Figure 1.8. The actual implementation is derived by the synthesis tool from the constraints given by the designer. The designer may also specify design goals. These are used by the synthesis tool to achieve goals like minimal area, fastest design etc.

High Level Synthesis has made it possible to describe designs at behavioural level in languages like VHDL and to synthesize detailed circuits automatically. Recently, new ways of describing and synthesizing control-dominated communicating hardware like interfaces and protocols has sailed up to challenge HLS as the method for the future: hardware synthesis from grammar-based specifications. Grammar-based specifications allows the designer to specify the behaviour of control-dominated designs at a high-level in terms of sequences of

incoming symbols that together forms a valid input sentence. When a meaningful sequence has been detected the associated action is performed.

In this thesis, a grammar-based specification language and a synthesis methodology are presented that aims at supporting high level specifications in an implementation independent manner and to perform design space exploration of port sizes for a specific domain: data communication protocols.

Hardware synthesis from ProGram is a sort of High-level synthesis with different emphasis. High-level synthesis frees the designer from details like when an operation is scheduled, how many and what types of hardware resources are required, operation to hardware resource instance mapping. These absence of details are then exploited by the synthesis tool as degrees of freedom to explore design space. For instance, High-level synthesis can create a faster design by allocating more resource and scheduling operations in parallel or vice versa. ProGram extends this by making the specifications independent of communication details like port width. Synthesis tools can then create a faster design by using a wider port or a slower design by using narrower port. This feature is very useful in communication applications. For instance, UTOPIA [71] the standard ATM interface specifies 16 bit interface, whereas some ATM vendors [72] implement 128 bit port for a higher bandwidth. The ability to do such trade-offs without having to modify the specification is the principle contribution of this work.

The thesis is organized as follows. In Chapter 2., the concepts of Interface and Protocol Synthesis are introduced. In Chapter 3., Grammar-based Interface and Protocol Synthesis is introduced. Chapter 4., the ProGram language is introduced. The ProGram Compiler is presented in Chapter 5. Chapter 6. presents some modelling and synthesis results from applying the ProGram Methodology onto several examples and finally, in Chapter 7., some conclusions will be drawn and directions for future work will be discussed.

2. Interface/Protocol Synthesis

The purpose of this chapter is to introduce the reader to related work performed in the area of Interface Synthesis, and briefly describe the State of the Art in this area. Grammar-based description styles of the same area are reviewed in the next chapter.

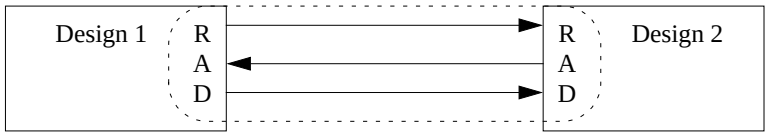
2.1. Introduction

System descriptions are composed of three elements: Data structuring, Computation and Communication. Languages like VHDL, C++ etc. have evolved to support computation intensive applications. Communication in such languages are weakly supported using data structuring elements to specify the interface and computation elements to specify the implementation of the protocol used for communication between systems.

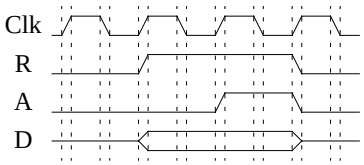
An interface describes the ports and their associated legal signal values. An interface model may also describe timing relations between and combinations of signal values present at the ports. A protocol defines how systems communicate. In general, the protocol specifies the order in which signals should occur and be presented at the interface. A data communication protocol is an agreement between two or more communication parties about the exchange of messages in order to provide some service. One can say that the protocol specifies the language used to communicate over the interface between the involved systems. This is shown in Figure 2.1 a) and b). a) describes the channel interface as a collection of wires and b) shows the channel protocol, i.e. the sequencing of the signals that pass over the interface.

Another property that an interface protocol has is that it is possible to compose new protocols by merging two existing ones and decompose a protocol into two new protocols by splitting an existing one.This is possible because a protocol always has two sides. One for the incoming and one for the outgoing. In a sense, a protocol behaves just like a magnet. Cut it in two halves and you get two new magnets, with a new south and north pole in between. Put two magnets together and you get a new larger magnet. A source and sink node, i.e. a design that only communicates over a single protocol is then the equivalent of a magnetic monopole. This situation is displayed in Figure 2.1 c).

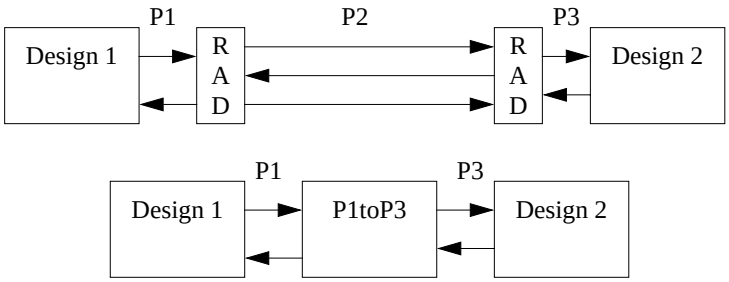
When two protocols are connected together over an interface, the most important thing is that the two protocols are compatible, i.e. that the two communication parties are synchronized with each other. The communication over an interface can be drawn in an abstract way using symbols as shown in Figure 2.1 d). Each symbol must match the symbol on the other side for the two protocols to be compatible. In a sense, the way the communication is put together mimics how the DNA string is put together in our genes: a genome matches only its complementary genome on the other side, and the sequence determines the functionality, in our case the sequence of interactions, messages sent and messages received, over the interface over time, i.e. the communication language of the interface protocol. And, just as with the DNA string, once you know one side of the protocol, it is easy to derive and synthesize the remote side.



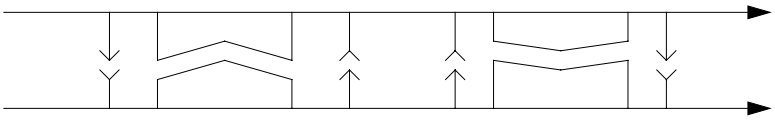
a) Channel Interface



b) Channel Protocol



c) Composition and Decomposition of Protocols



d) The DNA Protocol Communication Model

Figure 2.1. Interfaces and Protocols. a) Channel Interface b) Channel Protocol c) Composition and Decomposition of Protocols d) The DNA Protocol Communication Model

Communication protocols and interface synthesis has been a research topic for the past decade. It is a broad area. An attempt to cover all work done in this area would be out of scope of this thesis. Instead, the rest of the chapter will be dedicated to present some of the research that has some relevance to grammar-based interface synthesis. In section 2.2., some formal description methods are reviewed. Section 2.3., gives a review of the research done in incompatible interface design. Section 2.4. describes research on how interfacing between HW/SW can be automated and finally, in section 2.5., some conclusions are drawn.

2.2. Formal descriptions: Extended Timing Diagrams

Original work by Borrielo [97, 98, 99] introduced the concept of event graphs for establishing the correct synchronization and data sequencing between two protocols. In 1987, Borrielo and Katz introduced the concept of *transducer* synthesis [97]. A transducer is defined as the glue logic that connects two circuit blocks with incompatible interfaces. As an input description they used *timing diagrams* of the two interfaces. Timing diagrams with imposed timing constraints are very common in data sheets for describing the interfaces in commercial component data books. There is also an extension to timing diagrams called Extended Timing Diagrams (ETD). In an ETD, the events specified in the timing diagram can be annotated with actions, describing what action should be performed once the event has taken place.

In [42], Ravn and Staunstrup describe a formalism for modelling the behaviour of interfaces for systems comprised of modules. They define properties of the interfacing protocol like definedness using Duration Calculus. When two modules are connected together, these properties are checked for feasibility, i.e. if the two interface protocols are compatible.

Tiedemann [44] use timing diagrams as input to multi-paradigm controller synthesis. The timing diagrams are then transformed using a process calculus [100],which is used to derive the target circuit architecture. Multi-paradigm synthesis is not restricted to a single target architecture. Tiedemann shows how a description can be implemented both in an asynchronous style (interactively) as well as in a synchronous style (automatically).

Fujita and Fujisawa [101] also use timing diagrams as a high-level specification. The diagram is then mapped into propositional temporal logic formulas using a rule-base. The formulas are then used as formalism for translation into finite state machines [102].

In more recent research by Grass et. al. [45, 46], extended timing diagrams are used as a formal specification of a hardware interface. The timing diagram input is specified using a Graphical User Interface (GUI). The synthesis of the protocol implementing the specification is done formally by deriving a correct implementation of the description.

2.3. Incompatible Interface Design

In Incompatible Interface Design, the problem formulation is to design an interface between two incompatible interface protocols. When two incompatible protocols must be interfaced, a state machine must be synthesized that will convert one protocol into the other [99, 103]. This research is driven by the increasing complexity of electronic systems. A way to meet the complexity growth is by introducing design reuse and Intelligent Property (IP). Another thing that these approaches have in common is the attempt to hide the interface protocols from the designer. Instead a proper interface between two incompatible interfaces is either a) selected from a library or b) generated from the descriptions of the interfaces of the two IP blocks.

In [36], Sun and Brodersen presents a method for designing system interface modules. The behaviour of the interface protocol is described in a high-level specification language called IDL. The description is then passed to the ALOHA interface generator which generates an event graph. The event graph is stored in a library as an interface module. The ALOHA interface generator is part of the SIERA tool set [104].

In [37], Lin and Vercauteren present a high-level compiler called Integral for designing asyn-

chronous concurrent system interface modules. They take a high-level description of the communication and insert the lower-level details from a library into their internal Petri-Net representation. Their approach is based on a Petri-net algebra, described in [106].

Narayan and Gajski [38], presents a method for interfacing incompatible protocols. They reduce the protocol specification to five atomic operations. These atomic operations are used to capture the protocol description. They take two incompatible protocols, analyse the protocol sequences and partition the protocols into blocks, called relations. The relations of the two protocols are ordered into groups which are used to create an interface process. The interface process converts the signals of the two interface protocols into each other using the duals of the derived relations. The method uses a procedural specification style which makes it difficult to change data sequencing. For complex data communication protocols, the ability to describe complex state machines in an easy way is crucial.

Another way to solve the same problem is presented in [41]. Instead of deriving the relations from a procedural language, the two incompatible protocols are specified using a regular expression based protocol description language, a method that will be covered in more detail in the next chapter. The two protocols are first converted into two finite automatons. The product machine is then constructed recursively. If a product branch leads to a conflict with the original two machines, that branch is deleted from the final product machine. Their conversion algorithm has been implemented in a tool called PIG.

A third way of automatically generating an interface protocol is presented in [40]. Instead of converting one interface protocol into another in a point-to-point connection as the two previous methods, it converts all protocols into a common bus-based interface architecture. The advantage with this method is that any number of IP block protocols can be served on the same bus. A bus-arbiter is used to select the current IP to send in a round-robin manner.

Another method that can be used with advantage together with IP is presented in [29]. They present a method for channel synthesis based on an algebra that manipulates an abstract communication behaviour between two units, termed client and server, modelled as Protocol Flow Graphs. The advantage of their method is that it allows the client, the server as well as the channel to have a fixed interface. They apply transformations that allow both for data segmentation, i.e. splitting data over several cycles if the width of the channel is too narrow, and data combination, i.e. merging data into larger words if the width of the channel can accommodate simultaneous transfer of several data. The data sequencing over the channel is implemented as an FSM on the client side.

2.4. HW/SW Interface Synthesis

Hardware/Software Interface Synthesis must be part of any system design methodology. In system design, the task is to take a system description of a design and partition it into concurrent executing tasks, each task implementing a part of the system functionality. The tasks can be implemented either in hardware or in software. Interface synthesis must be performed between the partitioned tasks to enable them to communicate. Three types of communication is needed: Hw-Hw communication, Hw-Sw communication, and Sw-Sw communication.

COSMOS, presented by Ismail et. al. in [28], is a method for modelling and synthesis of complex communicating systems. It takes a system description written in SDL and converts it to their internal representation, SOLAR. The system is then partitioned into DesignUnits that

implements the behaviour and ChannelUnits that implements the communication. The channel behaviour is selected from a library of protocols. The selection of the communication protocol is treated as an allocation problem by Daveau et. al. [30, 27]. Function calls representing communicating primitives of the selected protocol is then inserted into the DesignUnits. The advantage with the method is its clean separation of communication from behaviour.

A method arguing against the use of protocol libraries for selecting of an appropriate protocol is presented by Eisenring and Teich in [34, 35]. The multitude of different communication styles makes it inefficient to store all combinations of communication types in a library. Instead, they propose to store interfaces as classes in an object-oriented library and having class methods describe how to generate the code for the interface. The approach has the advantage of producing code that is good for efficient implementation by the backend tool. A similar approach is presented by Vahid and Tauro in [33]. In contrast with [34], they use classes of objects to encapsulate the code needed to describe the communication for different target implementations instead of generating it.

Chou et. al. present a set of techniques for performing hardware/software interface synthesis [31]. Software drivers and glue logic are generated from Control Flow Graph (CFG) descriptions of the communication between an embedded processor and its connected peripheral devices, hardware co-processors, or communication interfaces while meeting bandwidth and performance requirements.

In [32], Lin et. al. present a synthesis system called Symphony. It provides the designer with tools to implement application-specific heterogeneous architectures and hardware/software communication layers. More specifically, they present useful methods for adapting two protocols that run on different clock speeds. Their approach is channel-centric and is based on the synchronous wait protocol. They also show how the synchronous wait protocol can be used to build channel adapters for other types of synchronous protocols and for the asynchronous four-phase protocol.

2.5. Conclusion

Many methods dealing with Interface/Protocol synthesis have been presented over the last decade since Borrielo presented his transducer synthesis technique in 1987. Most of the methods are based on procedural HDLs or a control flow graphs derived from a procedural HDL. However, describing a communication protocol in a procedural style is not always convenient although calling an abstract send/receive function may be. Protocols have long been modelled using grammars in the software world. Recently, grammar-based HDLs for describing data communication protocols for hardware synthesis has appeared. A grammar-based approach has several advantages over procedural specification styles when it comes to modelling and verification of communication centric designs and it has also shown some promising results when it comes to hardware synthesis. Grammar-based Interface/Protocol description styles will be covered in the next chapter.

3. Grammar-based Interface/Protocol Synthesis

This chapter gives the reader a brief overview over grammars and their use in the area of Interface and Protocol Synthesis. The rest of the chapter is dedicated to related work in grammar-based Interface and Protocol Synthesis.

3.1.Introduction

Grammars define the syntax of languages, both natural languages and those used by computers. The first known grammar was the one invented by Panini to describe the structure of the Sanskrit language in ca 500 BC [47]. In many respects, it had many similarities with the modern grammars used to describe parsers of computer languages.

Grammars define allowed patterns and are as such not restricted to defining languages where it specifies legally allowed patterns of words. In Mechanical CAD, they are used as a formal design method to define patterns in geometry, for instance, to describe the language of paths and, using boundary solid grammars, to describe topology graphs [48].

In CAD for ASIC design, it is not only used to describe communication protocols. It has also been used in the AGENDA Attribute Grammar tool environment as a specification language to add user interaction by letting the user add and/or modify parse rules to accept operator attributes and implement different scheduling algorithms. These operator attributes are then taken into account during the High-level synthesis tasks, for instance to let the user influence the results achieved by ALAP and ASAP scheduling [65, 66].

Though grammar is a general tool for specification of patterns, this chapter will focus on the usage of a grammar-based methods for protocol specification and hardware synthesis. A review of related work in this field is presented in section 3.2. Next in section 3.3., related work in Grammar-based Hardware/Software Interface/Protocol Synthesis is reviewed and finally, in section 3.4., conclusions are drawn.

3.2.Grammar-based Hardware Synthesis

A protocol defines how systems communicate. A data communication protocol is an agreement between two or more communication parties about the exchange of messages in order to provide some service. There are two ways of specifying the protocol - the language of communication. One way is to specify the automaton that recognizes the language, this is the approach taken by SDL etc. A more natural way to specify the communication protocol is in terms of a grammar expressed in the BNF notation [56] and annotated with actions. The actions specifies what should happen once an input has been recognized, i.e. the data that should be output. A compiler then synthesizes the automaton.

Automatic generation of language recognizers from grammar specifications has been extensively used in the software area for a long time [56]. A few years ago, synthesis of hardware from such specifications was reported by Seawright et. al. [59, 60, 61, 62]. Directly specifying the automaton makes the specification implementation dependent in two ways: the time behaviour is specified in term of states and the width of the inputs and outputs is fixed. By specifying the grammar, the specification is potentially independent of both these implementation details and allows for design space exploration in these dimensions.

3.2.1. Clairvoyant

In [59, 60, 61, 62] Seawright et. al. describe a system, called Clairvoyant. Clairvoyant takes a grammar-based specification and outputs an FSM described in VHDL that is synthesizable by logic synthesis tools. The system is targeted for detailed specification of communication protocols and interfaces, and other control-dominated circuits including Communication Protocols. In Clairvoyant, a design entity with a single process and a well defined boundary and interface is specified. All interactions with inputs and outputs are described at clock cycle level. Actions are specified in VHDL-code, similar to the software approaches used by, for instance, YACC [54, 56].

3.2.2. The Synopsys Protocol Compiler

The Clairvoyant approach has been developed further into a commercial tool, the Synopsys Protocol Compiler (PC) [63, 64]. It has a graphical user interface for entering the production rules in a hierarchical manner together with their actions. As with Clairvoyant, PC supports actions in a “host” language (VHDL or Verilog). In addition, Clairvoyant/PC has some inbuilt simple data manipulation and communication primitives. They also have explicit operators for implementing exception handling. PC also have a special conditional construct that terminates the parsing of the current input if the condition is true.

The main difference between Clairvoyant/PC and ProGram is ProGram’s ability to specify input and output sequences independent of the size of the ports. Clairvoyant/PC facilitates a clock true description, where every input and output sequence needs to be specified according to the port width constraints. ProGram treats the grammar specification as a high-level specification, independent of the port-widths, and uses constraints to specify how the final clock-true implementation should be built.

3.2.3. ProGrIL

Bloks [58], presents the Protocol Grammar Interface Language (ProGrIL), based on a context free LL(1) grammar extended with attributes. The work is a theoretical study on how a context-free grammar can be used to automate the implementation of data communication protocols in hardware. ProGrIL is implemented as a push down automaton (PDA). The approach has two drawbacks: (1) the code is based on a mathematical notation which makes it difficult to read. (2) the synthesis engine generates microcode for a predefined grammar processor that emulates the behaviour of a software compiler. The problem with this approach is that sifting through data together with reading and writing values and tokens onto the stack is rather inefficient from a hardware perspective. This is in contrast to our and Clairvoyant/PC strategy that generates a custom hardwired solution.

ProGrIL can handle three types of exceptions: the standard event, the reset event and the input event. The standard event interrupts the execution stream after completion of the current instruction. The reset event is used to implement a restart and a clean start-up of the protocol

engine. The input event checks for a particular value on an input and is automatically invoked if that value appears on the input during parsing of a rule where input events on this input is allowed.

3.2.4. Interface-based Design

In [39], Rowson and Sangiovanni-Vincentelli present a new system design methodology, named Interface-based Design, that separates communication from the behaviour. Their methodology has been implemented in a simulator named Cheetah and the potential for improving verification, modelling and synthesis is explored.

Their work was extended in [41] by introducing a language for protocol specification when it was found that the above approach was good for fast simulation but not efficient for implementation. They address the problem of automatic synthesis of interfaces between incompatible protocols. The language is based on a regular expression based notation. Hierarchy is expressed using a regular grammar. This is the same approach as that taken by Clairvoyant. The way in which the derivatives are calculated in their approach resembles the way that *Control-Flow Expressions* are calculated as introduced by Coelho and DeMicheli in [67].

3.2.5. ProGram - Port Size Independent Specifications

Our approach [paper I - VIII] is similar to the Clairvoyant/PC and the Interface-based Design approaches to the extent that the input is a production based specification and the output is in RT-level VHDL. However, they differ in notation and interpretation: 1) Clairvoyant/PC has an inbuilt concurrent-operator that is missing in ProGram. In ProGram, parallelism must be stated explicitly using concurrent start rules. 2) The Clairvoyant/PC makes use of the Kleene Closure operator for recursion, while recursion is specified using grammar rules in ProGram. 3) Clairvoyant/PC associates an input transaction with an output assignment in a clock-cycle true manner, while ProGram specifies the whole sequence that is associated with the input.

ProGram is targeted for specification and synthesis of large systems and differs from the Clairvoyant approach in being at a higher level of abstraction. The synthesizer allows fast exploration of the lower level design alternatives and frees the designer of clock cycle level details and exact input-output description. Furthermore, concurrent processes can be specified in ProGram and reading and writing of the multiple input and output streams can take place independently. This is very useful for structuring large designs. In addition, the ProGram descriptions are independent of the width of I/O. The I/O sizes are derived to satisfy the throughput constraints posed in the grammar description. This entails splitting and scheduling the logic for message recognition from input stream and assignment to output stream.

ProGram's actions are specified as lists of assignments. This helps in analysing the specifications for design space exploration, as well as for design verification. ProGram is also inspired by YACC in terms of its notations, but has been designed with the aim of hardware synthesis. In spirit, it has some similarity with LOTOS [24] in the sense that like LOTOS, specifications in ProGram deal with sequences of allowed events rather than states and state transitions as is the case with Estelle [24], SDL [24] and Promela [25].

Synthesising ProGram specification essentially involves synthesising state machine(s) that parses the input data stream(s) according to the specified Grammar and produces output according to the annotated actions.

The grammar, which is specified as a set of hierarchical production rules, is implementation

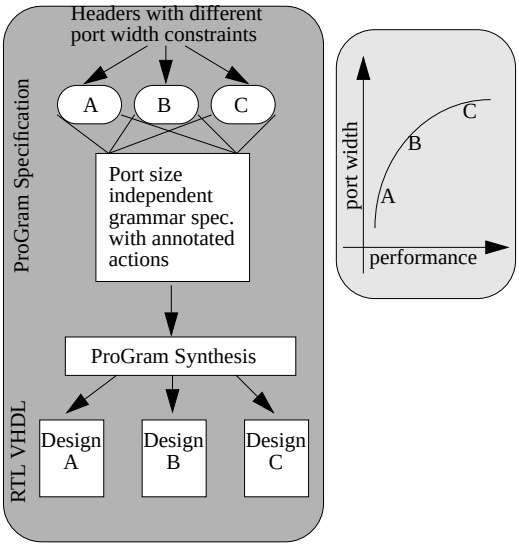


Figure 3.1. Synthesis framework and design space exploration

independent in two respects: 1) the production rules are independent of the port width. Synthesis spreads the recognition of a production over more cycles if the port width is narrower and merge them into less cycles if the port width is wider. This is similar in spirit with the data sequencing and data combination presented by Madsen and Hald in [29]; 2) Actions are also independent of the output port width. Once again, to comply with the port width constraints, synthesis will spread the actions over several clock cycles if necessary. As an implication, neither productions nor the annotated actions are then interpreted on a cycle true basis, this is in contrast to the approach taken by Protocol Compiler which interprets both productions and actions on a cycle true basis.

The synthesis framework is shown in Figure 3.1. The design space constraints are the port width constraints specified in the interface section of a ProGram specification, shown as headers A, B and C in Figure 3.1. Design space exploration is achieved by varying this part of the specification. The synthesis will generate corresponding designs with varying performance according to the port width, this is symbolically shown as the banana curve on right in Figure 3.1.

The generated state machine can be potentially large and is not minimal. Though the logic synthesis tools can do state minimisation they typically fail to handle the size of state machines that are synthesised from ProGram. As a remedy, state minimisation is performed before handing over the design for logic synthesis.

3.3.Grammar-based HW/SW Interface Synthesis

O'Nils and Jantsch [68, 69, 70] have extended the scope of using ProGram for Interface Synthesis to include the Hardware and Software Interface and Device Driver Synthesis. The ProGram language was extended with some constructs necessary for managing device driver

synthesis [68].

In [69], their method was used to generate device drivers adapted for particular operating systems. They show that their method reduces the development time of a device driver by 50% the first time it is written, and up to 98% for each time the device driver should be adapted to a new operating system or to a new processor.

In [70], the generated device drivers descriptions was used as input to generate application specific DMA controllers as a way to boost performance by implementing parts of the interrupt routines in hardware and as a way for the designer to explore the design space of the communication protocol by trading off between performance and cost.

3.4.Conclusion

A protocol defines how systems communicate. A data communication protocol is an agreement between two or more communication parties about the exchange of messages in order to provide some service. There are two ways of specifying the protocol - the language of communication. One way is to specify the automaton that recognizes the language, this is the approach taken by SDL etc. A more natural way to specify the communication protocol is in terms of a grammar expressed in the BNF notation and annotated with actions. The actions specifies what should happen once an input has been recognized, i.e. the data that should be output. A compiler then synthesizes the automaton. Directly specifying the automaton makes the specification implementation dependent in two ways: the time behaviour is specified in term of states and the width of the inputs and outputs is fixed. By specifying the grammar the specification is potentially independent of both these implementation details and allows design space exploration in these dimensions.

Grammars are excellent for describing the structure of languages, both natural languages and those used by computers. Automatic generation of language recognizers from grammar specifications has been extensively used in the software area for a long time. Hardware synthesis from such specifications is currently a hot research topic.

Our approach to grammar-based hardware synthesis of interface protocols are presented in Chapters 4 and 5. The advantage with the approach is that the protocol description is independent of the port sizes and allows for design space exploration in these terms. Chapter 4 describes the ProGram language. Synthesis from the language is presented in Chapter 5.

4. ProGram - A Protocol Grammar

This chapter presents an overview of the ProGram language. An introduction to grammar terminology, concepts and principles is given to the non-initiated reader. The chapter is based on papers I, IV and VI.

4.1.Introduction

Computers were originally developed to crunch numbers and languages like FORTRAN that were developed early on to program these machines reflect this need. Half a century later, number crunching is no longer the principal application; “Beyond Calculation” [23] a book published by ACM to mark 50 years of development in computers, echoes this sentiment.

However, modern programming languages like C++ and Java and the Hardware Description Languages like VHDL and Verilog still reflect the number crunching past; they are modern because they support a rich repertoire of data structuring constructs and addresses other organisational issues. However, communication in these languages is viewed as a means to partition complex behaviour, it is not meant to express functionality where communication is the end goal. With the emergence of networking and the ongoing fusion of computer and communication, communication related applications have taken centre stage. These applications are exemplified by protocol processing functionality that parses and interprets data streams, just as sorting was representative of the number crunching age.

Protocols specify the language used for communication among systems. At present protocols are implemented by automatons that parses the input data streams to make sure that they do not violate the grammar of the language specified by the protocol. When the protocol is complex there can be many states and the state machine becomes unwieldy. Alternatively, the automaton can be expressed algorithmically using nested IF-THEN-ELSE or equivalent constructs which is also hard to understand and debug. A more abstract and effective way is to specify the grammar of the language specified by the protocol and let a tool automatically build the automaton. In the software domain tools such as YACC & Lex use BNF like notation to specify the grammar of programming languages and automatically build parsers for them. We propose extending this method to implementing hardware parsers for communication protocols. The essential extension to the YACC & Lex approach is our ability to parse data streams from multiple inputs and generate concurrent state machines that can potentially run at different clock speeds. The meta language we have defined to specify the protocols is called *ProGram* - abbreviated from Protocol Grammar. Estelle, Lotos and SDL are formal languages also used for specifying protocols. Their primary purpose is to verify and validate the desired properties of protocols and obtain their software implementation. Moreover, SDL and Estelle directly specify the automaton to parse the input but Lotos is similar to ProGram in spirit. In contrast, ProGram is explicitly developed for hardware implementation to gain performance, as increasingly, protocol processing is proving to be the bottleneck in the network performance [26].

4.2.Some Grammar Terminology, Concepts and Principles

A grammar is used to describe the syntax of a language, that is, all legal combinations of how a sentence in the language can be written in terms of the words that make up that language. More formally, the grammar G describes all allowed legal sequences of strings. This is called the language L(G) of the grammar G.

A string is then a sequence of letters that make up a word. For instance, the word “hello” is composed of the letters ‘e’,’h’,’l’, and ‘o’. To be able to recognize words, some special constructs are needed. This is especially the case if numbers are to be recognized. These special constructs make up the language that can be recognized by *regular expressions* [56]. Alternatives are specified using the symbol ‘|' or by enclosing letters in squared brackets, ‘[‘ and ‘]'. Zero or more letters of the same kind are specified using the ‘*' symbol. This symbol is called the Kleene Closure operator. One or more letters of the same kind are specified using ‘+' symbol. Ranges are specified using a ‘-'. Thus, the regular expression [a-zA-Z] recognizes a single letter in the range all capital and lower case letters and the regular expression [1-9][0-9]* recognizes all positive integers.

Sometimes the regular expression becomes too complicated to be easy to write. Then it is common to introduce hierarchy. The hierarchy is written as production rules, using a grammar. Once something has been recognized, the input string is accepted and actions can be taken. Input strings are analysed and coded into keywords by a lexical analyser. LEX [55], is a lexical analyser generator. The user specifies a set of regular expressions that form the keywords within a language. Once it finds a string that matches one of its regular expression, a value can be returned. A matched string is called a token. Inside the grammar description, keywords are referred to as terminals or terminal tokens to differ them from non-terminal rule references. Non-terminals are used to describe hierarchy inside the grammar. LEX is usually used together with YACC [54]. YACC is a compiler compiler. The user specifies the order in which sequences of words recognized by a lexical analyser, for instance LEX, are allowed using a grammar.

4.2.1. Parsing techniques

LEX scans the input from left to right and returns a token. Reading in tokens and evaluating if the order in which they appear are consistent with the grammar is called parsing. However, parsing is sometimes ambiguous, the result may depend on the priority between tokens and the order in which the production rules are evaluated. A leftmost derivation expands the grammar productions from left to right. A rightmost derivation expands grammar productions from right to left. For instance, parsing the sentence “a+b*c” may be interpreted in two ways: either as “(a+b)*c”, which is obviously wrong, or as “a+(b*c)”, which is the correct interpretation. The grammar that parses this string can be written as “E: E + E | E * E | (E) | **id**”, where E denotes a (recursive) production rule and **id** denotes a token made up of letters. This style of writing grammars is called Backus-Naur Form (BNF) notation. During parsing, the grammar is expanded until it matches the input string. Without precedence between the alternatives, the rightmost derivation results in an erroneous parse-tree, see Figure 4.1. With precedence between the alternatives from right to left, both produce correct result. Using the precedence and rightmost derivation for the example, the grammar is expanded as:

> “E -> E+E-> E+E*E -> E+E***id**” -> E + **id***id -> **id** + **id***id”.

The order in which a grammar is evaluated is used to classify the parser. A parser that scans the string of tokens from left to right is denoted L while a parser that parses the string of

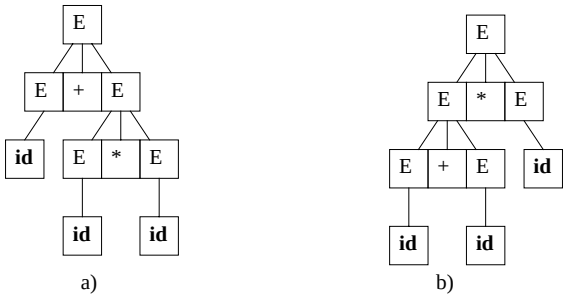


Figure 4.1. Resulting parse tree from a) Leftmost and b) Rightmost derivation

tokens from right to left is denoted R. A second letter is used to denote if leftmost (L) or rightmost (R) derivation is used. During the derivation process, the parser needs to see some tokens in advance. The number of lookahead-tokens that is used is indicated within parenthesis. Thus, an LL(1)-parser parses the string of tokens from left to right using leftmost derivation and one lookahead token. An LR(1)-parser parses the string of tokens from left to right using rightmost derivation with a single lookahead token etc. LR-parsing is the most common technique used for parsing programming languages. However, implementing full LR-parsing is somewhat difficult and efficient. In practice a heuristic technique called lookahead LR-parsing, denoted as LALR, is used. It takes smaller space and terminates quicker, but is not as powerful. LALR(1) is the technique employed by YACC [56, 57].

4.2.2. Classification of Grammars

Grammars are classified into four classes, called the Chomsky hierarchy [49, 58]. Type 0 grammars are the Unrestricted Grammars. They accept productions of the form:

v -> *w*

where *v* is any non-empty string and *w* is any string. The class of languages generated from the unrestricted grammars are exactly the same class of languages that are accepted by Turing Machines (TM).

Type 1 grammars are the Context-Sensitive Grammars. They accept productions of the form:

v -> *w*

where *v* and *w* are arbitrary strings with the restriction that the cardinality of *v* must be less than the cardinality of *w*, i.e. the number of symbols on the Left-Hand Side (LHS) must be less than the number appearing on the RHS. Thus, the empty string ε cannot be a part of the RHS. A typical example of a context-sensitive language is $L(G)=\{a^n b^n c^n\}$. Context-sensitive grammars can be recognized by Linear Bounded Automaton (LBA).

Type 2 grammars are the Context-Free Grammars. They accept productions of the form:

A -> *w*

where A is an arbitrary non-terminal and *w* is an arbitrary string of grammar symbols. A context-free grammar allows recursive symbols anywhere on the RHS of a production. A typical

language recognized by a type 2 grammar is $L(G)=\{a^n b^n\}$, i.e. that language of all palindromes. Context-sensitive grammars can be recognized by Push Down Automatons.

Type 3 is the simplest grammar, the class of Regular Grammars. A regular grammar accepts productions of the following form:

A-> ε ;
A -> *w* ;
A -> B ;
A -> *w* B ;

where A and B are arbitrary non-terminals and *w* is a non-empty string of terminals. ε denotes the empty string. A regular grammar allows recursive symbols only as the first symbol of the right-hand side (RHS) of a production, called left recursion, or as the last symbol of the RHS. Regular grammars describe exactly the same language as regular expressions. A typical language that can be parsed by regular grammars is $L(G)=\{a^n b^m\}$. Regular grammars can be recognized by a Deterministic Finite Automaton (DFA) or by a Non-deterministic Finite Automaton (NFA).

Regular grammars and regular expressions are important since they can be used to describe Finite State Machines. Kleene [50] showed in 1956 that the class of regular languages is exactly the same as the class of languages accepted by DFAs. Another important result was presented by Rabin and Scott in 1959 [51]. They showed that the class of regular languages is exactly the same as the class of languages accepted by NFAs and thus that DFAs and NFAs are equivalent. Then there is the Myhill-Nerode theorem from 1958 [49, 52, 53] that states that every regular expression has a unique minimum-state DFA. This means that given a language described by a regular grammar, there exists a minimal state machine that parses it.

4.3.The ProGram Language

The ProGram language is based on a regular LL(1) grammar [56, 57] and uses a BNF-like notation to code both input and output sequences. Since ProGram is based on BNF, it can express context free grammars, but the synthesis strategy handles only regular grammars extended with attributes and actions. The matching of patterns from the stream and the parsing of tokens (symbols) has been merged in ProGram to form a unified basis from which sequences of bit-patterns in a stream can be recognized, parsed and appropriate actions taken accordingly.

Incoming and outgoing messages are coded as bit-streams. Bit-streams have no natural “white space”-characters because the width of the bit-stream and the interpretation of the bit-patterns varies from application to application. The input production rules specifies in an hierarchical way all valid input sequences that the protocol should recognise and the output production rules specifies the output sequences that are associated with each input. The input sequences are partitioned into tokens, the size of the input stream and reduced during the synthesis steps to remove non-determinism from the internal representation, the grammar DAG. The output sequences are partitioned into tokens, the size of the output streams, and scheduled over the available input tokens.

4.3.1. The ProGram specification

The ProGram specification is divided into five sections: *Interface Declarations, Terminal Token Definitions, Memory Layout Productions, Action Macro Productions* and *Protocol Grammar Productions*.This division is similar to YACC & LEX approach with the essential difference that we do not have a separate tokenizer like LEX. In our case tokenisation is implied as the word arriving at input ports and sampled by the specified clock. Other forms of tokenisation like using handshakes or tokens composed of multiple words can be expressed as grammar rules.

In the Interface Declaration Section, interfaces to the external world and between internal sub-protocols are specified in the form of port width and throughput requirements. The rest of the ProGram specification is independent of these constraints but is synthesised to comply with them. ProGram allows global variables in the form of scalars or structured data, referred as signals and memory declarations. The actual layout of memory elements is declared in the memory layout production section.

ProGram allows concurrent parsing of independent data streams. Each data stream can be viewed as having a separate protocol. These are referred to as sub-protocols. The top-level of the rule hierarchy for each sub-protocol is specified as a start rule. A start rule defines the top-level of the rule hierarchy, that makes up the input Frames of that sub-protocol.

In the Terminal Tokens Definition section, constant tokens can be declared to increase readability. Unlike BNF-like languages for software parsing, the parsing of tokens is embedded in the grammar since it is hard to tell if a sequence of bits belong to one token or another after partitioning.

In the Memory Layout Productions section, the layout of the records that make up a single record datum in the memory is specified. It is a hierarchical definition in terms of production rules for fields that together make up the memory record.

In the Action Macro Productions section, functions and other useful macros are defined. The action macros are also hierarchically defined, besides having arguments which allows them to be reused in different contexts.

The Protocol Grammar Productions section defines the language as a hierarchical list of production rules. Each production defines a legal sentence or part of a legal sentence. A production consists of a list of production alternatives. Each alternative is a list of production items. Each production item can be annotated with an action. An action is a list of assignments to outputs and internals of the specified language.

A. Interface Declaration Section

In this section interfaces to the external world is specified in the form of port width and throughput requirement. The rest of the ProGram specification is independent of these constraints but is synthesised to comply with them.

ProGram has three different kind of port declarations: the *input*, the *output* and the *internal* ports. These ports are accessed by name in the Action and Grammar Rules section to read data from and write data to data streams. Input ports are used to read from the external world. Outputs are used to write data to the external world. The internal port is used as a means of communication between concurrent sub-protocols. The only difference is that the port is not

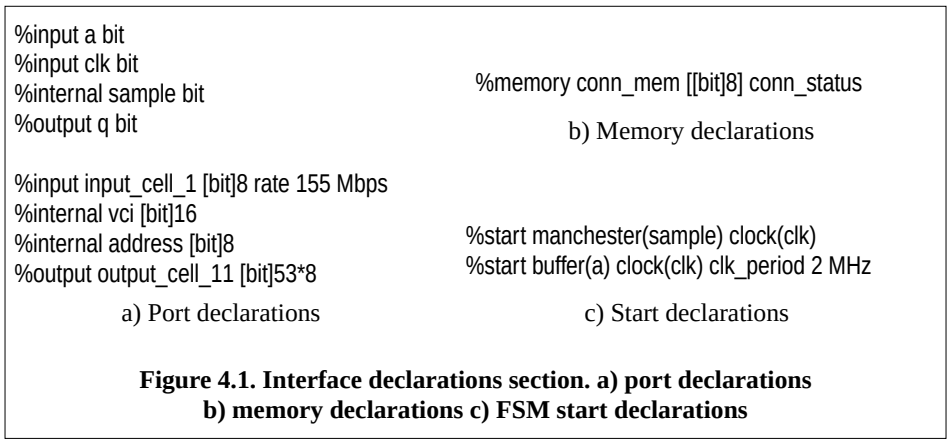


Figure 4.1. Interface declarations section. a) port declarations b) memory declarations c) FSM start declarations

visible from the outside. The internal and the output declarations have an optional parameter for specification of the type of reset to use on the port. It may be of the type asynchronous or synchronous. The synchronous reset is default. Examples of port declarations are shown in Figure 4.1 a).

Port declarations may have an optional rate constraint, which is currently not used by the ProGram Compiler. The purpose of this constraint is to be able to specify the input/output sample rates. It is currently used only as a help for the designer, but it could in theory be used as input for the Compiler to automatically adapt the data communication protocol for differences in sample speed of inputs read and outputs written by the same start symbol. This possibility has not been implemented yet. The issue is further discussed in chapter 7., Thesis Summary & Directions for Future Work.

ProGram allows global variables in the form of scalars or structured data, referred as signals and memory declarations. The actual layout of memory elements is declared in memory layout production section. An example of a memory declaration is shown in Figure 4.1 b). The size of the memory is given by the number in brackets as the number of address lines connected. Then the rule that specify the memory field layout is given.

The notion of inferring the memory size from the number of address lines connected is rather inefficient. In many applications less memory is needed, so it would be much better to specify the actual size of the memory and then infer the number of address lines. Currently, the data port width of the memory is directly inferred from the memory field layout. However, it is possible to consider this also as a target for design space exploration.

ProGram allows concurrent parsing of independent data streams. Each data stream can be viewed as having a separate protocol. These are referred to as sub-protocols. The top-level of the rule hierarchy for each sub-protocol is specified as a start rule that defines the top-level of the rule hierarchy that makes up the input Frames of each sub-protocol. Examples of start declarations are shown in Figure 4.1 c). The start declarations may have an optional clock period specification. It is currently not used by the ProGram Compiler, but is rather used to document for the designer the clock period constraint for the following logic synthesis.

B. Constant Token Definitions

In the Terminal Tokens Definition section, constant tokens can be declared to increase readability. Tokenisation is merged with the grammar specification for two reasons. First, since

there is no natural “white-space” character sent on a bit-stream, there is no natural way to delineate tokens from each other. Second, since the port width is passed as a constraint to the compiler, the tokens themselves must be split and merged according to the port-size depending on the place in the bit stream they were found.

A constant is written with upper-case letters to be able to distinguish them from grammar rules and action macros. Multiplicity of a pattern is specified by enclosing the value with squared brackets. To ease the specification, it is possible to specify constant values in hexadecimal notation. Example of constant declarations are shown in Figure 4.2.

```
VCI_SEGMENT        0000 0000 0000 0011
VCI_CONNECTION     H'0004
VCI_IDLE           0000 0000 0000 0000
SIXA42             [0110 1010]42
```

Figure 4.2. Examples of Constant declarations

C. Memory Layout Section

In the Memory Layout Productions section, layout of the records that make up a single record datum in the memory is specified. It is a hierarchical definition using BNF notation; each BNF production defines the fields at a particular level of hierarchy. Example of a memory layout is shown in

```
conn_mem: atm_header atm_data;
atm_header: gfc vpi vci hec;
vci: [bit]16;
vpi: [bit]8;
```

Figure 4.3. Example of a memory layout

D. Action Macro Section

In the Action Macro Productions section, functions and other useful macros are defined. The action macros are also hierarchically defined using BNF like notation, besides having arguments which allows them to be reused in different contexts.

Actions in the grammar specify assignment of values to signals. To ease the task of specification, actions that are frequently used can be specified as a macro which can be referenced later in the action part of the production rules. Expressions that compute values may be put directly in the assignments by enclosing the computation in parentheses with a bit size specification. Concatenation and conditionals in addition to the usual arithmetic and logic operations are also allowed in the action macro. The operands can be constants, signals, other action value symbols or bit patterns recognized by grammar symbols. Hierarchical descriptions are supported. The assignments can then simply refer to the name of the macro. A symbol prefixed with $ refers to the bit pattern recognized by that input symbol, i.e parsed by the corresponding grammar rule. The scope of such a reference is limited to the symbols already recognized in the current alternative. An example of an action macro declaration is shown in Figure 4.4.

E. Grammar Rules

The Protocol Grammar Productions section defines the language as a hierarchical list of pro-

```
q = $a next CPM16 ($trcc01+1)16;

next = if (inp=00) then 1001 else 0110 end if;
```

Figure 4.4. Example of an action value specification.

duction rules. Each production defines a legal sentence or part of a legal sentence. A production consists of a list of production alternatives. Each alternative is a list of production items. Each production item can be annotated with an action. An action is a list of assignments to outputs and internals of the specified language. A production put between squared brackets followed by a number *N* indicates a repetition of that production N times.

A grammar rule consists of a grammar symbol which serves as a rule identifier and a list of alternatives. Each alternative is a sequence of non-terminal symbols and/or terminals followed by actions enclosed in curly brackets as shown in Figure 4.5. A redirection of the input stream is done by passing the new signal stream as a parameter to the subtree of productions.

```
input_cell: gfc vpi VCI_SEGMENT
        { vci=VCI_SEGMENT; } pti clp hec { address = $vpi; } oam_segment_types
    | gfc vpi vci_user {vci=$vci_user;} pti clp hec { address = $vpi; } user_cell_body
      special_user_cell(connection_memory[address])
        { output_cell_11 = special_user_cell_action; priority=PRIORITY_1; };
```

Figure 4.5. Partial grammar specification for the F4 OAM example.

In the signal assignment in Figure 4.5., the signal address is assigned the value of the rule production vpi. The symbol *special_user_cell* parses the full record obtained from the memory *connection_memory* at address *address*.

Symbols that mark keywords, i.e. the leaf symbols of the grammar, are called Terminals. A terminal can be any of the three following things: a token, a bit string or an exception handling construct. Exception handling constructs, including the redirection construct, are covered in the next section.

4.4.Exception Handling

Exception handling is necessary to be able to describe hardware and real-time systems in an efficient way and is a key aspect of any system functionality, where it takes the form of reset sequences, interrupt service routines and error handling. The key characteristic of exception handling is its *global* nature. For instance, a reset would bring the state machine to a known state from *any* state and clear the storage elements. Reset, though a common hardware construct, does not have any explicit support in VHDL, in which any *exception of global nature* is treated as any other condition. This makes it very cumbersome to specify exception handling functionality. Other languages, like SpecChart [13, 14], have explicit support for specification of exception handling functionality. Interrupts are of a more local nature than a reset, and specify how the protocol should react if a high-priority situation occurs, caused by some hardware failure, some illegal operation performed elsewhere in the system, an interrupt request or some other external event. Error handling is crucial since it makes it possible to detect and recover from error situations during run-time execution.

4.4.1. Exception Handling Constructs

To allow specification of exception handling functionality, a construct for specifying asynchronous or synchronous reset behaviour of internal registers and output ports can be specified in the Interface section. The default is synchronous behaviour.

The rest of the exception handling constructs are specified in the grammar rule section. A negation construct can be used to specify complemented token constants, i.e. they specify all bit-combinations except the specific constant token. The negated token is expanded into its complementary set. The *condition* construct can be used to specify synchronisation situations, where a main-stream is parsed and the decision between two identical productions should be made using an extra condition. This is a useful feature that allows the grammar to count and can also be used to terminate loops. *Interrupt* sequences allows to handle events local to parts of the protocol. The *reset* sequence allows to handle start-up and initialisation situations that can happen anywhere within the protocol.

The first version of ProGram [paper I] had four rudimentary exceptions in the grammar rule section: the *redirection* construct, the *other-* and *any-bit* constructs, plus an *error* construct to support specification of error conditions in the input sequences. The *Other-bit* construct is used to specify else-situations in the grammar. The *Any-bit* construct is used to specify ignore-situations in the grammar. The other two were extended in [paper IV]. The redirection construct was extended to increase the readability and expressiveness of the grammar description and the *error* condition was extended to include error sequences. Error sequences allow the possibility to detect and recover from error situations during run-time execution of the protocol.

A. Redirection Exception

A common situation when describing the controlling FSM of a communication protocol is that at different parts of the FSM, the FSM is sensitive to different signals. To be able to describe this situation, we need to be able to parse different input streams at different places in the protocol. This can be modelled in two ways in a grammar-based environment. Either we describe the valid combinations of all inputs at all times in the protocol, letting a don't-care symbol denote unused inputs, or we have an exception handling construct for changing the input stream. The first case will be a very cluttered description, and if the number of inputs are large, not very readable. Also, it would make it troublesome to use this type of specification in our case because of our message scheduling methodology. Since all bit-patterns are word-aligned onto their respective input port sizes, the sequence in which bits are specified is important. In this case, bits end up on a different port than that it was specified for. To have the input patterns end up on the right input would then require a change in the input description of our programming language. We have adopted the second approach. The *redirection* construct changes input streams between tokens. The semantics of this construct is shown in Figure 4.1. Parsing of the clock stream is treated as a special case of input parsing where only *any-bit* tokens are allowed, i.e. time is advancing with steps of one for every clock period, regardless of the value on the input stream.

There are three types of redirections. Redirections of type one change the bit-stream locally for all alternatives of a production rule. Redirections of type two change the bit-stream, when the production rule is called. This works as a parameterisation, and makes it possible to use production rules in a generic manner, enabling the same set of rules to be parsed from any set of inputs. Redirections of type three change the bit-stream locally on the alternative where it is specified.

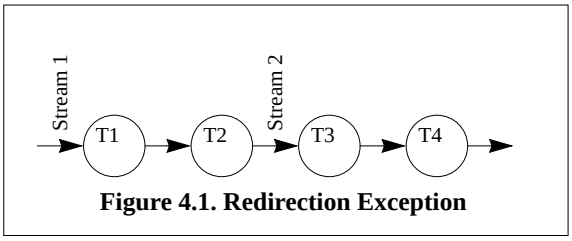


Figure 4.1. Redirection Exception

B. Condition Exception

Another common situation when describing controlling FSMs, is that at some point in time, it could be necessary to synchronize with an external source, for instance to delineate the frames of an ATM protocol. At other times, the controlling FSM may need to count something, like the number of incoming cells or the number of turns in a loop. This could of course be modelled using the redirection exception described in the previous section. However, in these cases the redirection scheme becomes somewhat cluttered and the main intent of the description is lost. Therefore, we have introduced the *condition* construct make these situations explicit. The semantics of the condition construct is shown in Figure 4.2.

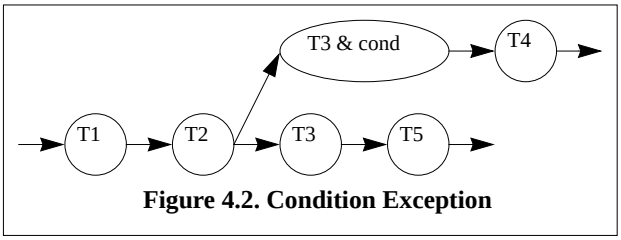


Figure 4.2. Condition Exception

Transitions not having a condition associated with it is treated as a non-overlapping don't-care situation. For example, Figure 4.2 shows the transition T3 and the transition T3 & cond(...). At a first look, the patterns seem to overlap. But, since they should be treated as non-overlapping, we could either extract out the complement of the condition and automatically associate it with the transition T3, or we can use a priority scheme for specifying this. For simplicity, we have chosen the latter scheme for the condition construct. Priority is in the order they were specified. Thus, specifying the T3 transition before the T3 & cond(...) would hide the condition-statement completely while the reverse order, with T3 & cond(...) first and the T3 second, would accomplish the non-overlapping don't care situation we would like to achieve.

A condition can be either left associative or right associative. A left associative condition is associated with the first terminal token to the left of the condition. Thus, it can be used to terminate a generic hierarchical loop, since only the exit token would be visible from the outside. A right associative condition is associated with the first terminal token to the right of the condition. Thus, it can be used when a generic hierarchical sequence should not be executed until some synchronisation condition is true.

C. Interrupt and Reset Exception

For some protocols, it might be desirable to be able to specify distributed conditions. This is the case if, during the execution of a sequence, a reset or an interrupt occurs. Since the reset is of a more global nature, it takes affect anywhere in the whole FSM. The interrupt is more of a

local nature since it in some cases is desirable to mask them out and not allowing them to execute. This situation could be modelled with the condition construct, but again, to have to specify the condition for all transition tokens would clutter the specification and the main intent of the description would be lost. Therefore, we have introduced two new constructs. The *reset* construct affects the whole state machine and terminates execution anywhere in the FSM and continues parsing the reset sequence ones the condition associated with the reset becomes false again. The *interrupt* construct works as a distributed condition statement, affecting all transition tokens associated with the same hierarchy level and below where the interrupt was specified. The semantics of the interrupt statement is shown in Figure 4.3.

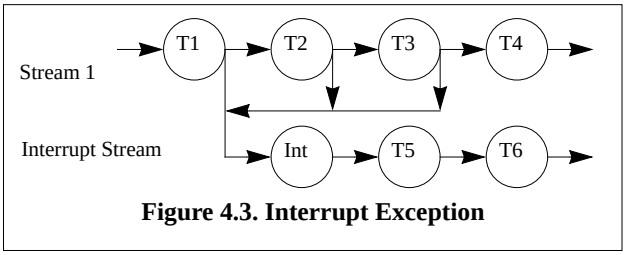


Figure 4.3. Interrupt Exception

Both interrupts and resets have precedence in the order they were specified in order to be able to specify priority between exceptions of the same type. Thus, it is possible to specify more than one reset signal; this makes it possible to specify different behaviour for power-on situations and warm start resets. The same thing holds for interrupts, making it possible to specify several interrupts of different priority, for instance to differ between a system interrupt and a user interrupt or to immediate seize execution to handle a hardware failure or some illegal operation occurring elsewhere in the system.

D. The Other-bit Exception

The ability to specify complementary situations are important to keep the code size small in any programming language. These situations are usually described using an ELSE-statement in most programming languages. It is the basic construct for specifying exceptions to the normal execution flow. For data communication protocols, it is also important to have an efficient construct for specifying sequences of complementary transitions. In the ProGram language, complementary sequences are specified using the *other-bit* construct. The *other-bit* construct represents all other transitions not already specified at the same level the *other-bit* construct was specified. The semantics of the other-bit statement is shown in Figure 4.4.

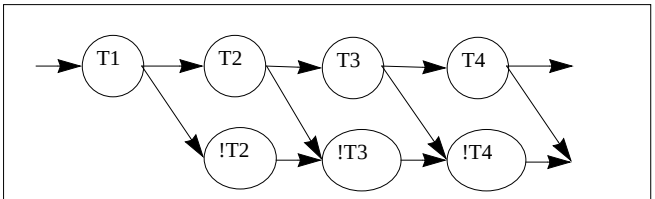


Figure 4.4. Other Exception

A related transition construct is the *any-bit* construct. The *any-bit* statement allows the designer to specify don't care situations in the input sequence, i.e. a situation where a transition should be taken regardless of the value on the current input stream. The any-bit construct represents all possible values that the transition could take. Thus, the any-bit statement is overlapping with ordinary transitions, which means that the construct must be split during NFA to

DFA conversion to weed out overlapping bit-patterns.

E. Error Exception

One of the most critical exceptions that needs to be handled by data communication protocols is error recovery. The error can be caused by loss of sync, a noisy environment or a faulty transmission sent by another controlling FSM. In summary, there must be a way to specify how the controlling FSM of a data communication protocol should react onto illegal sequences fed to it. This situation could of course be modelled with the *other-bit* construct described in the previous section. However, doing so would not highlight the main intent of the situation. Also, the error is more of a global situation since it can happen anywhere in the protocol. Thus, some way of specifying a hierarchy of errors is needed.

This is modelled using the *error* construct in the ProGram language. Any token sequence that is not in the grammar will result in an error exception. The sequence following the error statement describes how the controlling FSM should react on the error. The sequence is associated to all bit sequences that come at the same depth or later, counted from the start of the input frame. Thus, error statements occurring later in the transition sequence of the protocol takes precedence over the error statements that come earlier. The semantics of the error statement is shown in Figure 4.5.

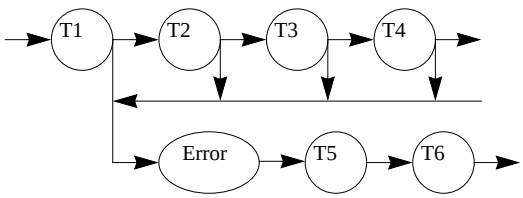


Figure 4.5. Error Exception

In the case when not all possible input sequences have been specified, there is an implicit error that will trap these sequences. The protocol will then end up in the default error state. The default error state then restarts the parsing from the beginning of the input frame.

4.4.2. An Illustrative Example

To illustrate how the exception handling constructs are used in practice, an example is shown in Figure 4.1, where a hypothetical processor is described using ProGram. The example illustrates the use of the new constructs in ProGram for exception handling. The processor is a standard CISC processor, which reads one instruction at the time from the main memory and executes it. It has two interrupt lines, the non-maskable interrupt (NMI) and the interrupt request (IRQ) line. The IRQ can be masked out internally, thus disallowing the interrupt, by setting the IRQ enable flag. Upon reset, the processor should initialize its internal registers first before starting the execution of instructions. Furthermore, the processor has a memory management unit (MMU), that checks that all memory accesses are performed on a valid memory address. If the MMU detects an illegal address, it signals a bus error to the processor. Upon a bus error, the processor should immediately cease the execution of the current instruction and perform a bus error recovery.

The Processor decodes and execute instructions coming at the input instr. Upon a *reset* excep-

```
%start hypothetical_processor(instr) clock(clk)

hypothetical_processor: instruction
                      | interrupt(bus_error) trap_bus_error
                      | reset(res) (clk) start_up_sequence reset_fetch;

instruction:          valid_instructions instruction_fetch(clk)
                      | error trap_unknown_instruction;

valid_instructions:   div_instruction
                      | add_instruction
                      | ... e.t.c.
                      | interrupt(division_by_zero) trap_division_by_zero_error;

instruction_fetch:     nmi_interrupt_fetch
                      | irq_interrupt_fetch
                      | normal_fetch;

reset_fetch:          bit
                            { address=START_ADDRESS; }
nmi_interrupt_fetch:bit & cond(NMI=1) push_registers_onto_stack
                            { address=NMI_ADDRESS; }
irq_interrupt_fetch:  bit & cond(IRQ=1) & cond(IRQ_enable=1) push_registers_onto stack
                            { address=IRQ_ADDRESS; }
normal_fetch:         bit { address=(address+1); }
```

Figure 4.1. Exception handling constructs used for modelling of different parts in a hypothetical example processor

tion, the processor switch from parsing the instr input stream, to start parsing the clk stream. This is described using one of the *redirection*-type exceptions. Then, the processor runs the start up sequence to initialize the internal registers of the processor, followed by an instruction fetch of the first instruction. The instruction decoding is described using the production instruction. If the MMU detects any illegal address on the buses, the parsing of the instruction being executed should be terminated immediately. This is modelled by an *interrupt*-exception, that traps the bus_error signal from the MMU.

The production instruction decodes the valid instructions. Any unknown instructions to the processor is trapped by the *error*-exception. After parsing a valid instruction, an instruction fetch is performed. The input stream is switched to the clk stream, i.e. time is incremented in steps of one regardless of the value on the input, using another of the *redirection*-type exceptions. The next address that should be selected depends on the value of the of the external ports NMI and IRQ. If the NMI-port is high, the internal registers are pushed onto the stack, and the NMI_ADDRESS is selected. If the IRQ-port is high and the internal IRQ_enable register is high then the internal registers are pushed onto the stack and the IRQ_ADDRESS is selected. Otherwise the normal address is selected and the address counter is incremented by one. The extra information needed to decide which address that should be selected is described using the *condition*-exception. If more than one selection is possible, i.e. the IRQ and the NMI goes high at the same time, the branch that is listed first has the highest priority. That means that in case of conflict, the NMI-branch is selected first, the IRQ-branch second and the normal branch third.

4.5.Recursion

ProGram can be classified as an extended regular grammar with attributes and conditions. A regular grammar with attributes and conditions can implement any context free grammar [56, 58, 105]. Even more so, it is as complex as any programming language. The conditions and the attributes makes it context-sensitive since the grammar can know what it has already parsed. It can therefore implement a larger design space than a context-free grammar can do.

Although the ProGram language allows context-free grammar specifications, the ProGram compiler cannot implement a context free grammar description directly. The functionality has to be rewritten first and then implemented.

This is best illustrated with an example. Assume the classical context free grammar example of identifying a string that consists of any number 0's greater than one followed by exactly the same number of 1's [57]:

```
S: 0 S 1
   | 0 1
```

Using a regular grammar with attributes and conditions, the same sequence can be specified as:

```
S: { count=0; } S0 S1;
S0: 0 { count++; } S0
   | 0 { count++; } 1;
S1: 1 && cond(count=1)
   | 1{ count--; } S1;
```

In the context free case, the stack is considered to be part of the controller and infinite. In the regular case, the stack is considered to be a part of the datapath and, in this case, implemented as a counter, but it is still infinite. Context free grammars implemented on a computer will have the computer memory as a limiting factor. In ProGram, the size of the counter is set to a fixed value in the header, but it could as well be implemented to output the width as a generic parameter in the produced VHDL-code. In that way, the designer would not have to redo the ProGram compiler phase before logic synthesis. The width of the counter would then be given by the designer when doing logic synthesis.

As an example of a sequence that cannot be recognized by a context-free grammar there is another classical example [57]:

$$L = 0^n 1^n 2^n$$

This language cannot be written in a context-free grammar, but is easily implemented using a standard programming language or with a regular grammar with attributes and conditions:

```
S: { N=0; } S0 { count=N; } S1{ count=N; } S2;
S0: 0 { N++; } S0
   | 0 { N++; } 1;
S1: 1 && cond(count=1)
   | 1{ count--; } S1;
S2: 2 && cond(count=1)
   | 2 { count--; } S2;
```

4.6.Conclusion

This chapter has presented an overview over the ProGram language. Although the language has proven to be quite useful, many things and constructs could be improved. For instance, the semantics of the interrupt exception resembles more a traditional break exception than the way an interrupt exception is usually conceived in the software world. There are also some exception handling constructs that are currently missing and that could prove to be useful in the future. One is a sync exception, that would work like a condition except that is would wait until the condition is fulfilled. Another is an implicit wait alternative. The wait alternative would wait on the current symbol until one of the succeeding symbols occur instead of generating an error. A third category of exceptions that might be useful to have are the local_error, local_break or local_interrupt. Instead of terminating the execution and move to the end of the protocol frame, they would restart from the current production rule.

One thing that is clumsy in the ProGram language is the usage of right recursion for specifying loops. Many loops have a simple structure and it would be much more convenient to use the Kleene Closure operation of regular expressions for shorthand notation. Also, a better way is needed to specify bounded loops with a simple counter. The current way with conditions becomes unnecessary cluttered with details. The branch containing the condition could easily be inferred from the description.

Another thing that needs to be investigates is the expressiveness of the ProGram language. It is currently used to specify protocols in a type 3 grammar extended with attributes and conditions. Using the condition constructs of the language it is also possible to rewrite type 2 grammar specifications. One thing that deserves a closer examination is if it is possible to also rewrite type 1 and type 0 grammar descriptions into ProGram. If it is, and the steps needed to perform the rewriting is possible to automate, the more powerful grammar descriptions types might provide a platform for a more intuitive way of describing exceptions in fewer lines of code.

5. The ProGram Hardware Compiler

This chapter presents an overview of the ProGram Compiler. The chapter is based on papers I, III, V, VI, VII and VIII.

5.1. Introduction

Grammar-based approaches have several advantages over procedural specification styles when it comes to modelling and verification of communication centric designs. For instance, with procedural specification styles it is difficult to change data sequencing. For data communication protocols, the ability to describe complex state machines in an easy way is crucial. Currently used languages do not capture communication behaviour in an easily understandable manner. Protocols have long been modelled using grammars in the software world. Recently, grammar-based HDLs for describing data communication protocols for hardware synthesis has appeared and some promising results have been shown. This chapter presents our approach for performing grammar-based hardware synthesis on data communication protocols, the ProGram Compiler.

The ProGram language supports a description style that is implementation independent of the port sizes. The ProGram Compiler is therefore well adapted to the synthesis philosophy presented in Figure 1.8 on page 9. The figure is reproduced below in Figure 5.1 for simplicity. The designer specifies the functionality of the communication protocol in the grammar rules section. Design Space Constraints are specified in the header section. The actual implementation of the protocol is derived by the synthesis tool from the interface constraints given by the designer. The design goals are viewed as constraints that should be passed on or given directly to the logic synthesis tool. The ProGram Compiler produces RT-Level VHDL Code, that is generated in a style that ensure good synthesis results.

The outline of this chapter is as follows. In section 5.2., the internal representation used in the ProGram Compiler is presented together with some related work on internal representations. In section 5.3., the synthesis steps that the compiler follows to produce synthesizable VHDL code is presented. Finally, in section 5.4. the chapter is summarized and some conclusions are drawn.

5.2. Internal Representation

Most tools have their own internal representation. First, in section5.2.1., the internal representation that is used in the ProGram Compiler is presented. Then, in section 5.2.2., a short overview over other existing internal representations is given.

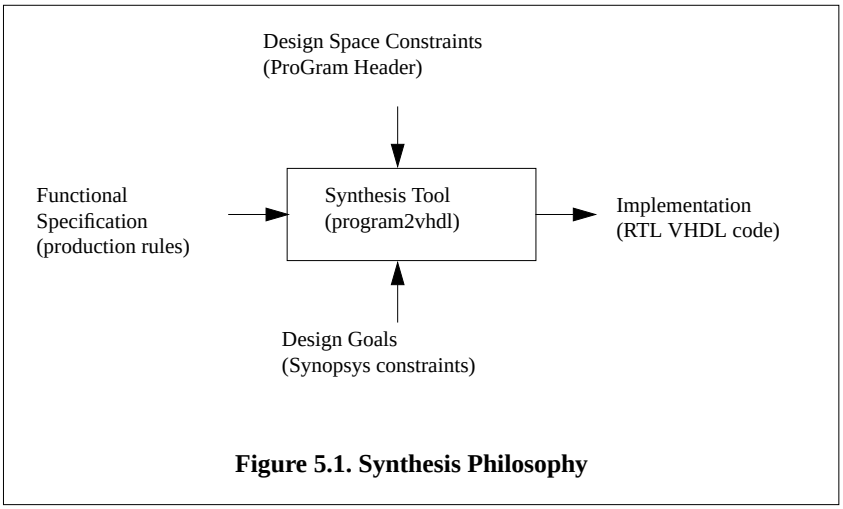


Figure 5.1. Synthesis Philosophy

5.2.1. The Grammar DAG - The Transition Hyper Graphs

Finite State Machines (FSMs) are usually modelled using State-Transition Graphs (STGs). FSMs can also be described using a regular grammar. The grammar G describes all legal sequences of strings. This is called the language L(G) of the grammar G. If several sentences within the language begins with the same words, the grammar is Non-deterministic. To be able to build an FSM, the grammar must be made deterministic. This can be done in two ways. The first way is to build a Non-deterministic Finite Automaton (NFA), i.e. an STG, from the grammar. The NFA is then converted into its Deterministic equivalent (DFA) before minimization. However, having an STG as an internal representation is not natural when dealing with grammars. Since the grammar describes the sequence of transitions in the FSM, it is much easier to build a graph where the transitions are the nodes and directed edges represent the sequence in which the transitions occur. The second way to represent the grammar is to build a Transition Graph (TG). This TG is also non-deterministic and needs to be reduced into its deterministic equivalent before minimization.

The advantage of using a TG is that it alleviates a designer from having to think of the implementation in terms of the number of states he/she would need. Instead the designer can focus on what is more important: the sequence in which things should occur and what should happen in the sequence. This is easily captured by the Transition Graph. The TG describes only the partial order of the sequence, i.e. the order in which transitions occur, not the space required to store the sequence. Thus, it represents the timing aspects of the design. The STG describes the storage space of the sequence, i.e. how many states that are needed to represent the sequence. Thus, it represents the implementation aspects of the design.

Edges in the TG represent states. In addition, all out-going edges must represent the same state since a transition cannot be made into more than one state in a deterministic representation of an FSM. In-going edges does not necessarily represent the same state. However, they may do so if the transitions they came from have identical futures. If they don’t share the same state but may do so, merging them is the same thing as performing state minimization. Since multiple edges share a state, the Transition Graph can be viewed as a Hyper Graph of the FSM, where the nodes represent the transitions and the hyper-edges represent the states. Thus, the *Transition Hyper Graph* (THG) is the dual of the more commonly used State Transition

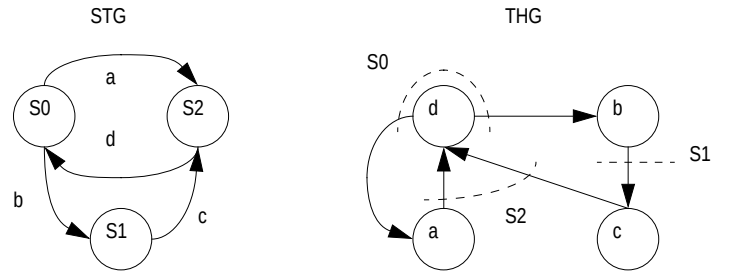


Figure 5.2. STG vs THG

Graph, see Figure 5.2.

The internal representation of ProGram is a Directed Acyclic Graph were the nodes represent the transitions in an FSM and the edges represents the sequence in which the transitions should take place. Loop edges are marked but are not followed. A directed graph is a graph in which all edges in the graph has an explicit direction. An acyclic graph does not have any loops inside, which makes it easier to handle. Thus, the grammar DAG is not a true acyclic graph,. It is rather pseudo-acyclic since there could be loop-edges in the graph, but the loop edges are marked with an attribute and are not followed during synthesis. Since the nodes in the graph represents the transitions in an FSM, the grammar DAG is also a Transition Graph.

The Grammar DAG represents a series-parallel graph in which parallel subgraphs correspond to the alternatives for a grammar symbol and subgraphs in series correspond to the sequence of tokens within an alternative. The basic structure of the Grammar DAG, taken from the manchester encoder example in [paper III], is shown in Figure 5.3. The start symbol marks the start of an input *Frame*. The exit symbol marks the end of the input frame. The start and the exit symbol represent an implicit infinite loop, which restarts parsing of the next the input frame once the previous one has completed execution. Each input symbol can have output assignments (actions) associated with it. An input symbol that relates to a specific bit-pattern is called a token to separate them from *Error*-, *Interrupt*- and *Reset*-symbols. States are marked on out-going edges. Since the start and the exit symbol represent the restart symbol of the input frame, all edges going into the exit symbol and all edges leaving the start symbol are marked with the same state.

5.2.2. Other Internal representations

Petri-Nets and their extensions have been effectively used to represent behaviour of digital systems. Petri-Nets can be used for representing sequential or parallel behaviour. It can also be used for representation of behavioural hierarchy. The dynamic behaviour of the system is described by flow of tokens in the system. Computations are described by actions associated with firing of a transition. Petri-nets have been extended for more conveniently representing digital systems [75]. Modified Petri-nets are called R-Nets. R-Nets have been used as internal representation in some high level systems based on the hardware description languages RACHNA [75] and IDEAL [76]. How structure and structural hierarchy can be described using petri-nets or R-nets, is not clear. Communication between parallel threads is performed through shared variables.

Many high level systems use a graph representation that unifies the control flow and the data flow in a system. This graph is called a Control and Data Flow Graph (CDFG). The basic con-

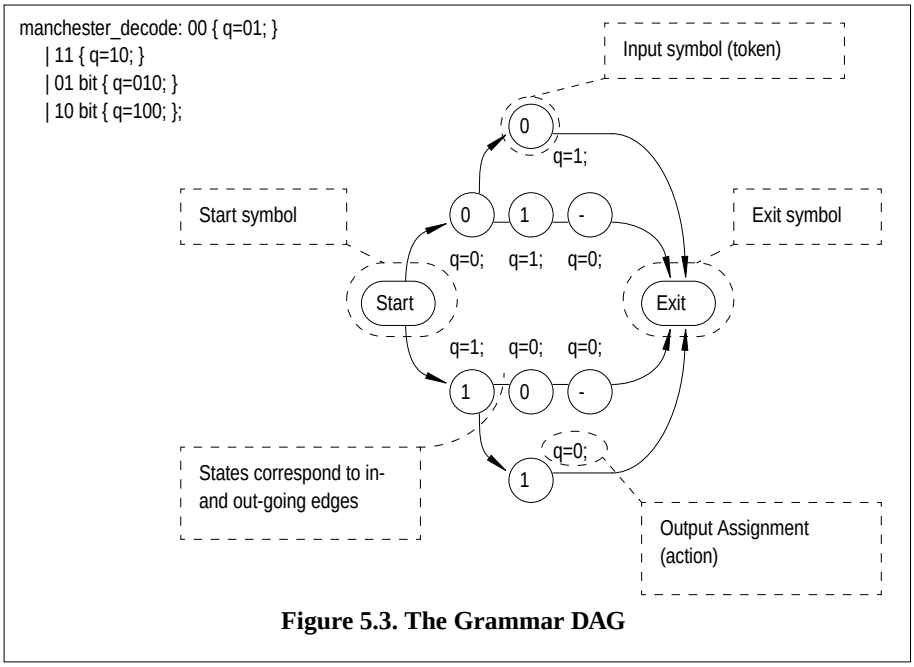


Figure 5.3. The Grammar DAG

cepts required in such a representation are described in [10, 77]. The CDFG consists of two types of nodes, Control nodes and Data nodes. Control nodes are connected to each other to form a control flow graph. The control flow graph part (control nodes) of CDFG captures sequencing, conditional branching, parallelism and looping in the system behaviour. Each node in the control flow graph is linked to a data flow block consisting of one or more sequences of data computations. Each sequence of data computations is represented by a directed acyclic graph (DAG) of data nodes. Each data node represents either an arithmetic/ relational/logical operation or a read/write to a memory or a port. Variations and extensions of CDFG's have been used by many researchers.

SOLAR is intended as an intermediate design representation for control dominated mixed HW/SW systems [78, 79]. It is based on hierarchical, concurrently executing finite state machines, allowing for the specification of hierarchical communicating FSMs. Tree structures allow for modular specifications.

The program state machine (PSM) model integrates a hierarchical, concurrent FSM with programming language concepts [13]. A PSM description consists of a hierarchy of program states. A program state is a generalization of a state in a traditional FSM. It represents a distinct mode of computation. At any given time only a subset of program states will be active, i.e. actively carrying out their computations. Transitions between sequential program states are modelled either as transition immediate (TI) arcs or as transition on completion (TOC) arcs. The TI arcs are traversed as soon as the associated condition becomes true, no matter which sub-program states are active or where the computation is at that time. This is used to model resets and exceptions. TOC arcs are traversed when the program state has finished its activity.

Codesign Finite State Machines (CFSM) [82] is a network of communicating FSMs with a specific communication mechanism. The FSMs in the network are not synchronized with each

other, i.e. the state changes in the FSMs are unrelated with each other and the time to compute the next state can vary and be different for different FSMs. In fact, it is unbound as long as the implementation is not known. The model is very control oriented and does not support modelling of complex data types and data transformations efficiently. The communication mechanism between the FSMs is based on timed events. An event is defined by a name, a value and a non-negative integer denoting the time of occurrence of the event. Events are transmitted in a send-and-forget manner and the sender does not expect nor receive any acknowledgment. There is, however, an implicit storage element for each event type, which ensures that events remain available until a new event overrides the previous one. There are two types of events: Trigger events and Pure value events. Trigger events implement the basic synchronization mechanism between CFSMs. They can be used only once to cause a transition of a given CFSM. Pure value events cannot directly trigger a transition but can be used to choose among several possibilities involving the same set of trigger events.

COSYMA uses an internal representation called Extended Syntax Graph (ESG) [83], which combines concepts of data flow graphs and syntax graphs. A system is described as a directed acyclic graph describing a sequence of declarations, definitions and statements. ESG is intended for systems descriptions in C/C++.

Flow charts [80, 81] have been used for many years in system design as an informal description of algorithms. A flow chart is an informal graphical description of an algorithm built as a directed graph. Usually four types of nodes - start, stop, operation and decision - are used. But, depending on the problem area, other, different, types can also be used. The flow charts have been widely used to describe algorithms in early phases of design for both software and hardware, mainly because of their capability to give a good overview of the algorithm but also to simplify checking the logic of the algorithm. There exist also some special expansions of flow chart for hardware design - sequential machine flow charts, behaviour finite state machines and algorithmic state machine charts, to mention some of them.

XFC (Extended Flow Chart) [86] and IRSYD (Internal Representation for System Description) [84, 85], based on Flow Charts, are representations developed here at ESDLab. Historically XFC came first as an internal representation used in the in-house high level synthesis tool dedicated for control and memory centric systems. The basic idea in XFC is a unified graph representation for control flow and data flow. Differently from a typical CDFG, the XFC has the control flow described explicitly and data flow described implicitly. The IRSYD is essentially an extension of the XFC to capture hierarchical and concurrent sub-systems. The other extensions deal with higher abstraction levels of data representation, inter-process communications, etc. The IRSYD also has mechanisms to preserve semantic information of the source language.

5.3. Synthesis Steps

After the ProGram description has been parsed and the corresponding internal representation built, a series of synthesis steps, shown in Figure 5.4, transforms the ProGram description to Register Transfer Level VHDL. Every start rule in the ProGram description correspond to a separate state machine. Therefore, the listed synthesis steps is performed once for every start rule. The synthesis steps can be roughly divided into four main phases. During phase 1, the Grammar DAG corresponding to the production rules in the ProGram description is built. Dur-

for each start rule do

1) Build the grammar DAG
   a) Create DAG
   b) Connect Interrupt Ends
   c) Reduce dummy Exit symbols

2) Input Stream Alignment
   a) Word Align Terminals to their resp. input stream
   b) Link Interrupts
   c) Reduce the Grammar (Convert it from NFA to DFA)
   d) LinkErrors

3) Output Assignment Scheduling
   a) Word Align Action assignments to their resp. output stream
   b) Split the DAG into a Tree
   c) Schedule Upwards
   d) Schedule Downwards

4) State Minimization & Code generation
   a) Perform State Minimization
   b) Mark States
   c) Output the FSM in Synthesizable VHDL code

Figure 5.4. The synthesis procedure.

ing phase 2, all input streams are split into pieces the size of their respective input stream, i.e. the tokens are word aligned onto their respective input stream. The resulting grammar DAG is unfortunately non-deterministic. The grammar is therefore reduced in order to produce a deterministic equivalent. During phase 3, the output sequences of the grammar DAG are partitioned into assignments the size of their respective output stream, i.e. the actions are word aligned onto their respective output streams. The resulting list of assignments compose an output sequence. Since there cannot be more than one assignment to a single output at given point in time, the assignments need to scheduled over the available control steps. During phase 4, the state machine represented by the grammar DAG is minimized, states are marked and the state machine is output in synthesizable FSM-style behavioural VHDL code. The produced FSM is a Mealy machine with latched outputs.

5.3.1. Phase 1 - Building the Grammar DAG

The Grammar DAG is constructed in three steps. In step 1 a), the parsed ProGram description is converted into a DAG. Hierarchical symbols are expanded. Since the last symbol at a lower level of hierarchy should be followed by the first symbol in the rule coming after one just expanded, some way of linking is needed to connect the symbols. To do that, a dummy exit node is inserted at the end of every hierarchical rule, see Figure 5.5. Recursive rules, i.e. rules that contain alternatives calling itself, are not connected to the exit node but instead to the first symbol on every alternative for the rule. To keep the graph acyclic, the end symbol of the loop alternative is marked as a loop node. Loop connections are not followed during the synthesis steps.

Interrupt sequences are also terminated with dummy exit nodes. But, in contrast with hierarchical dummy exits, interrupt dummy exits are not connected to the previous level. According to the interrupt semantics, control should not be returned to the previous level once the inter-

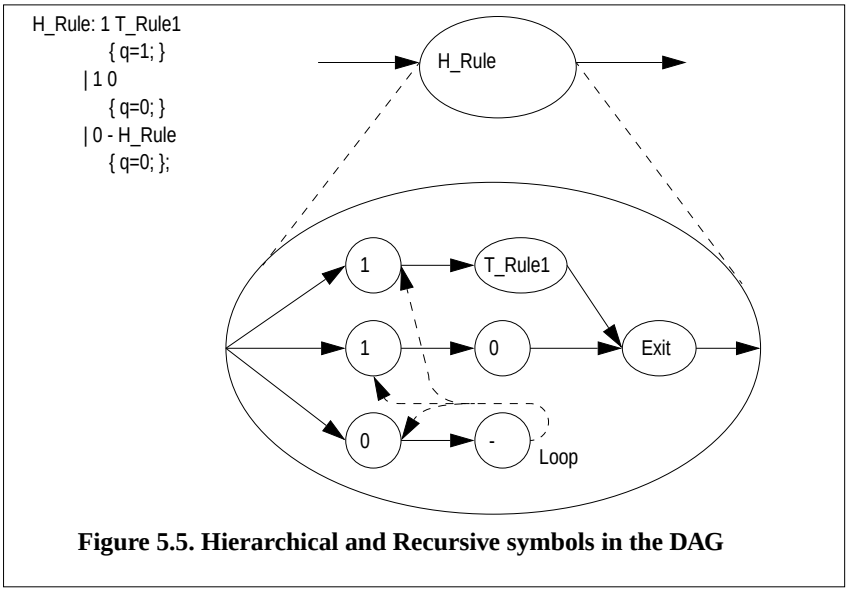


Figure 5.5. Hierarchical and Recursive symbols in the DAG

rupt exception has been handled. Instead, it should start parsing a new input frame. In step 1b), the interrupt dummy exits are therefore connected to the frame exit symbol. Reset and error sequences are handled in the same way as interrupt sequences.

The dummy exit symbols must then be reduced from the graph to be able to split and merge token symbols across rule hierarchy. This is performed in step 1 c). After the dummy exit symbols have been reduced from the graph, two dummy tokens remain: the entry token marking the start of the input frame and the exit token marking the end of the input frame.

5.3.2. Phase 2 - Input Stream Alignment

During the input stream alignment, the input tokens are partitioned into chunks with the same size as the input port the stream is parsing. The input stream alignment is performed in four steps. During step 2 a), all input tokens are word aligned to their respective input stream. Input tokens that are wider than the width of the input stream are partitioned into chunks the size of the input stream. Input tokens that are smaller than the width of the input stream are merged with their successors to form a bigger token. If the total width of all tokens is not an integer multiple of the port width, the protocol is not completely specified and an error is reported. Output assignments are kept as close to their insertion point as possible, i.e. if the token is split, the output assignment is kept with the part of the original token closest to the end of frame token; if the token is merged with another token, the output assignment is kept with the token closest to the frame entry token, see Figure 5.6., Splitting token sequences.

Tokens marking the start of an interrupt sequence are copied to every token within the rule where they were specified. During word alignment, interrupt tokens are duplicated and copied to ensure that all created partitions also get an interrupt token marker. However, to be able to merge tokens, these interrupt successor sets must be merged with the normal successor set of the token. This is done during step 2 b). Since interrupts have priority over normal tokens, they are inserted before the normal successor tokens. Thus, we have two classes of tokens, interrupts and normal tokens. Priority among tokens of the same class is in the order in which they appear. Only tokens of the same class can be merged.

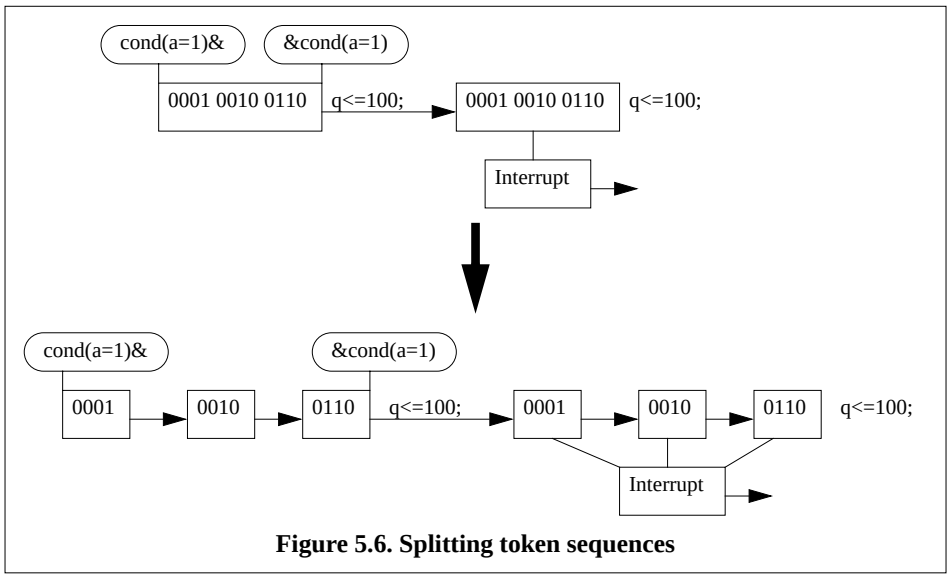


Figure 5.6. Splitting token sequences

At this point, the Grammar DAG represents a Non-deterministic Finite Automaton (NFA). Original tokens that begin with the same sequence and differs only at a later stage may have been partitioned into sequences starting with a series of identical tokens. Since only one look-ahead-symbol is allowed for LL(1) parsing, the non-deterministic sequences must be reduced. This is performed during step 2 c). Successors with overlapping transitions are split and successors with identical transitions merged, see Figure 5.7. If one of the successor tokens is an *Any-bit* token and it overlaps with one of the other successor tokens, the any-bit token must be split since any-bit tokens match all bit-patterns. Algorithms for subtracting a bit-pattern from another can be found in many books on logic synthesis, see for instance [73, 74]. After the grammar has been reduced, only unique transitions remain. The Grammar DAG has been converted into a Deterministic Finite Automaton (DFA).

Some situations during the token merging requires special attention. If a token has multiple predecessors, one copy of the token for every predecessor is created before merging. The reason for this comes from the flattening of the rule hierarchy. Nodes with multiple predecessors are created if there are multiple alternatives within the expanded rule. If one of the alternatives in addition contain a loop node, a sequence could be created starting from the one alternative and ending up in the loop in another one. By always splitting tokens that have multiple predecessors, this situation is avoided.

Splitting multiple predecessors causes problem if two alternatives should be merged and they both contain a loop. In this case, both loops would unroll indefinitely if the tokens in the two loops are identical. This could for instance be the case if one loop recognize any number of token T1 followed by a token T4 and the other loop recognize any even number of token T1 followed by the token T2. This situation is shown in Figure 5.8. To avoid this situation, the loops are unrolled until the token with multiple predecessor is a loop exit marker. If they are, both loops have been unrolled to their least common denominator of turns. During the state minimization procedure presented in section 5.3.4., the redundant transitions are removed leaving the deterministic transition graph shown in Figure 5.8.

Another situation that cause trouble is when one of tokens to be merged is the end of frame,

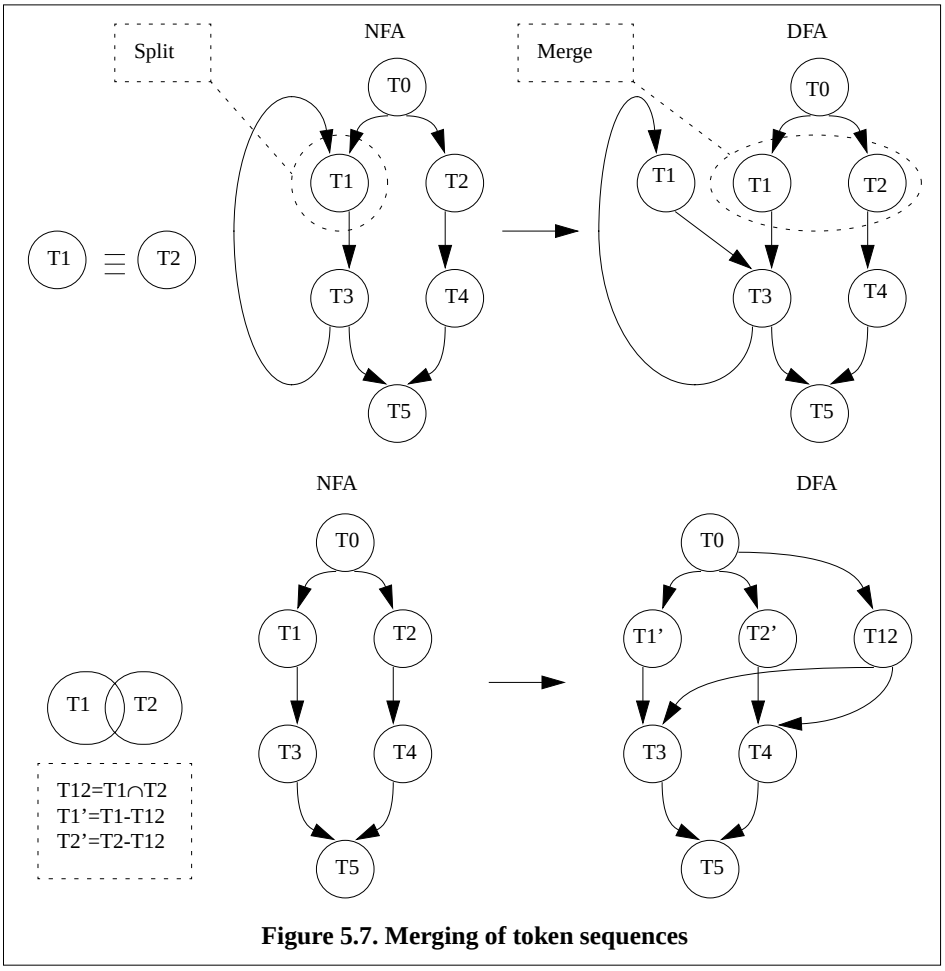


Figure 5.7. Merging of token sequences

see Figure 5.9. This can be the case if one alternative is part of another message. This situation in principle cannot be solved by copying tokens from the start of the input frame. This introduces latency since which branch to select cannot be determined until the first token of the next frame has been read. If the first token of the next frame is identical with the token in the first frame, more tokens are copied until the situation is resolved. However, this is a potential source for state explosion and is therefore not allowed in the current implementation.

If one of the successor tokens is an *other-bit* marker, the successor set of the other-bit token is copied to the successor sets of the other tokens. Then, if the cover over the other bit-patterns at the current level is full, the *other-bit* node pattern is already covered by the other tokens at this level, and is therefore deleted. The other-bit token merging is shown in Figure 5.10.

Extra care must be taken if two *Other-bit* tokens from two different alternatives should be merged. Since the *Other-bit* token represents all bits that has not been previously matched, two *Other-bit* tokens may represent different sets of bit-patterns because they come from two different successor sets that were filled differently. When a token from a second alternative is selected for merging, it is compared with all tokens in the first alternative until either a match is found, the end-of-set is reached or an *Other-bit* token is found. This means that the selected token is either in the first set, will be placed in the first set or is covered by the *Other-bit* token.

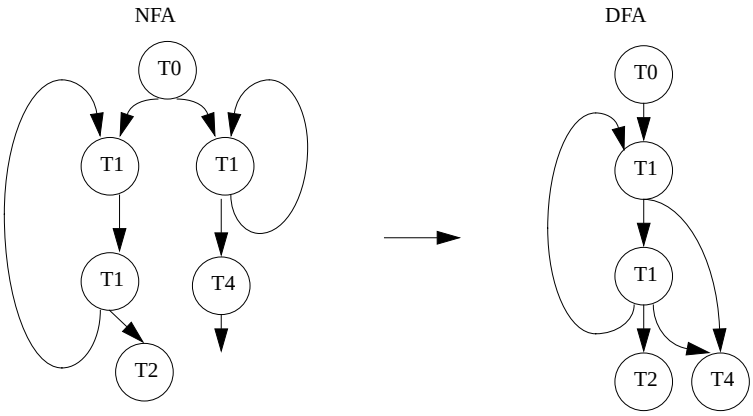


Figure 5.8. Loop merging

If the selected token is in the Other-bit set, the token is added to the first set. The successors of the first set's Other-bit token is then added to the successor set of the added token. Example: The alternative (11), Others (3 patterns) is merged with the alternative (11,10), Others (2 patterns). The (11) from the second set is merged with the (11) from the first set without problem. When the (10) from the second set is merged, it does not match the (11) and must therefore be in the Other-bit token's set. The (10) can therefore be added to the first set if the Other tokens successors are added to the successor set of the (10)-token. The two Other-bit tokens can then be merged since they both have been checked against the same set of tokens and therefore must cover the patterns. The result of this operation is then (11)(10), Others(2 patterns).

Although this seems to be a good way of merging two Other-bit tokens, the effects of the Other-bit merging on the state minimization algorithm, presented later in section 5.3.4., is currently not known. A problem is that the number of patterns that the Other-bit covers need to be stored with the token, otherwise it would be difficult to decide if two preceding Other-bit tokens are identical and can be merged during state minimization. This requires the tag to be an arbitrarily large integer, since it must be large enough to store the number of combinations and there is really no limit to the number of bits in a single tokens. The number of combinations covered inside an Other-bit token is of the order $2^{nr\_of\_bits}$.

The last step of the Input Alignment phase is step 2 d), where the Error-tokens are linked. Error tokens are stored at the place in the rule hierarchy where they were specified. Since an Error token is defined to trap all errors that happen at the point or later where it was specified, the error token must be distributed to all tokens that comes after the insertion point. Error tokens are therefore propagated toward the Frame Exit token. Error tokens that reside closer to the Frame Exit token has priority over ones that are closer to the Frame Entry token.

5.3.3. Phase 3 - Output Assignment Scheduling

Output Assignment Scheduling is performed in four steps. In step 3 a), all actions are word aligned. Action assignments are split and partitioned into assignments the size of their respective output port. Datapath actions, i.e. actions that make up the calculation primitives to perform a computation of an expression, are not split in the current implementation, which is a limitation of the current approach. This will be changed in the future as soon as a good method for splitting the datapath has been found. Instead, datapath actions are assumed to have the

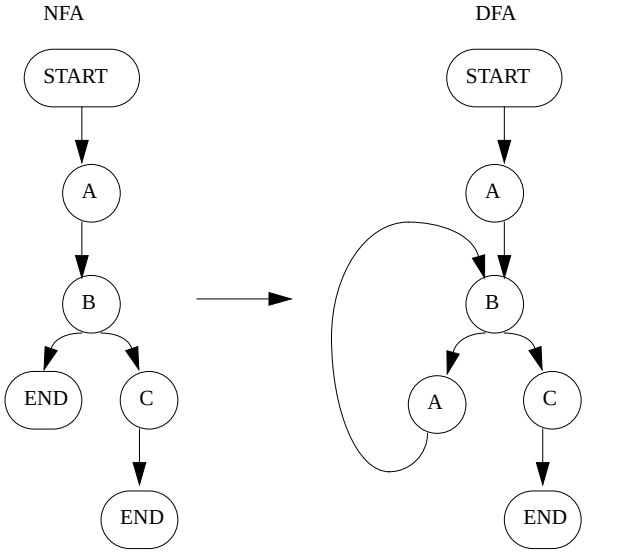


Figure 5.9. Sequences needing Overlapping Input frames

same size as the port. As a consequence, design space exploration of port sizes cannot be performed for designs having a datapath unless they are partitioned and scheduled manually.

In step 3 b), a tree splitting operation is performed. The reason for this is to compensate for a flaw in the current schedule algorithms. They cannot handle scheduling conflicts caused by reconverging paths in some hierarchical graphs. The graph is simplified by splitting all reconverging paths into a tree. Tree-splitting is a costly operation since it may result in a combinatorial explosion of paths, which threatens to blow up the graph. Since ProGram descriptions that result in situations where the current schedule algorithms does not work are rare, and the operation is so costly, it was decided that this synthesis step should be included as an option in the synthesis procedure.

During step 3 c), Upward Scheduling is performed. During Upward Scheduling, the list of assignments resulting from the word alignment step are distributed over the available tokens. To reduce latency, assignments are scheduled upwards first. To give the programmer some control over where output assignments are scheduled, they are scheduled as close as possible to their insertion point. An alternative would be to always schedule all output assignments As Fast As Possible (AFAP), which would then minimize the reaction latency of all output assignments in the protocol. The problem with this solution is that it would then be difficult to guarantee synchronization between output assignments targeting different ports, for example when an asynchronous interface needs to send a handshake signal together with the data signal or when a frame-based protocol need to send an end-of-frame signal with the last bit in a message, without having to introduce some kind of timing constraints.

After the assignments have been moved up as far as is allowed in the graph, the remaining excess assignments are scheduled downwards. This is performed in step 3 d), Downward Scheduling. As a result, some output assignments may be scheduled later than their insertion point. Although this may potentially cause a loss of synchronisation between two output port assignments, the resulting schedule is close to the insertion point. Therefore, differences in timing between two assignments can be easily detected in the produced VHDL code and the

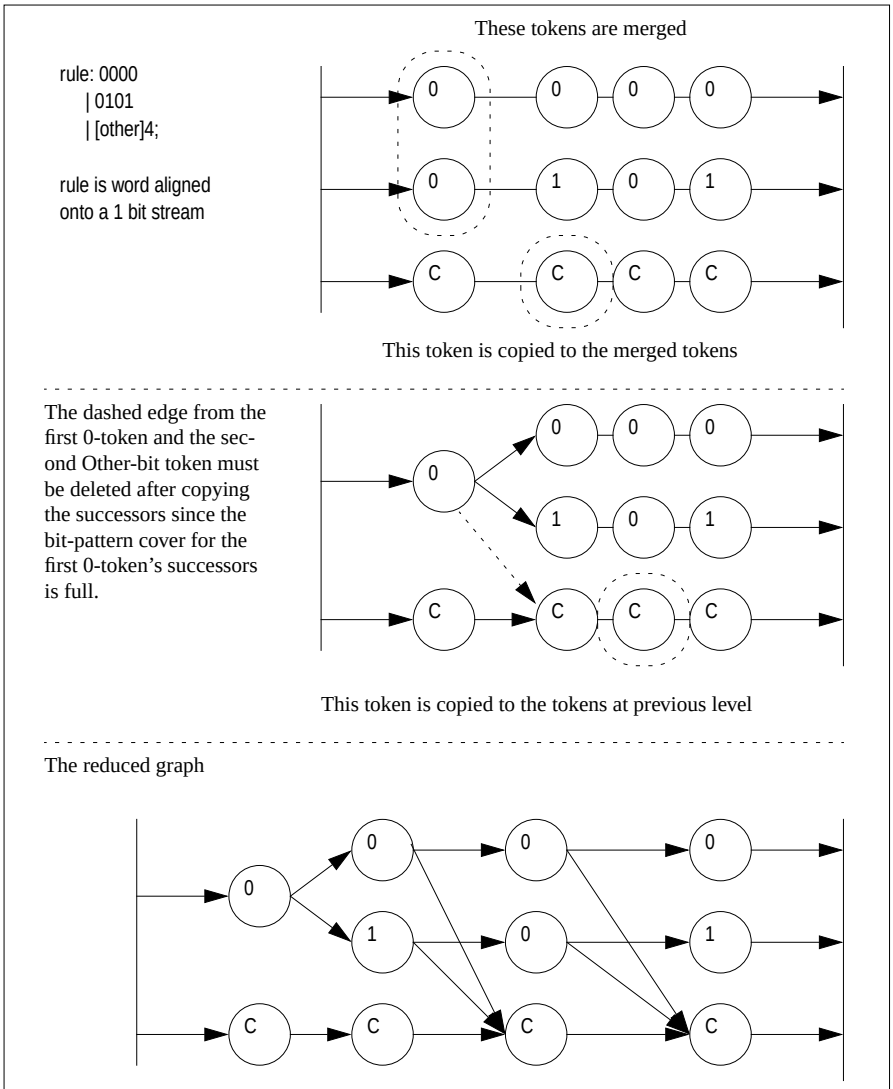


Figure 5.10. Other-bit token merging during grammar DAG reduction

ProGram description can be modified to compensate accordingly.

Differences with earlier papers - The hierarchical graph Scheduling Conflict

The ScheduleUp algorithm presented in [paper III] couldn't handle Other-bit Exceptions properly. The algorithm was extended in [paper V] to handle the ladder-structure that is the consequence of word aligning this exception. It was later found that this modification was not sufficient to handle all situations. Merging of reconverging paths, coming from multiple alternatives within more than three rules, across rule hierarchy boundaries created a situation where the outcome of the scheduling dependent on the order in which nodes were selected and, in some cases, incorrect schedules of assignments. For example, in the algorithm presented in [paper III], excess assignments were moved up. In [paper V], this part was modified

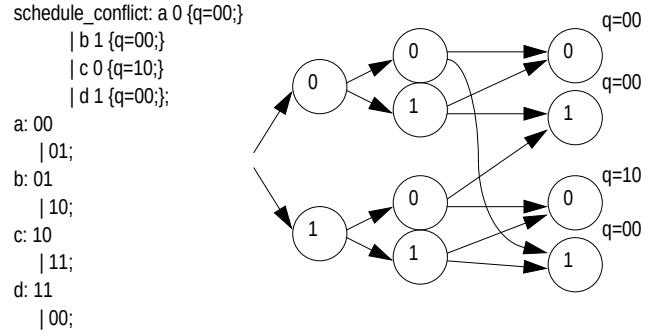


Figure 5.11. Hierarchical DAG Schedule Conflict

to check that all predecessors of a token have successors with the same excess assignments. The excess assignments are then copied to the predecessors end then deleted. As illustrated in Figure 5.11, deleting the excess assignments would lead to an incorrectly scheduled graph should no tree expansion be performed on the graph. The “00” alternative of the production *a* will be visited first, moving up the assignment *q=0*. Its reference is then deleted. When the”01” alternative in the productions *a* and *b* is visited next, the successor it shares with the “00” alternative does not have an excess assignment any more and the assignment *q=0* to the other successor can not be moved up. Deleting the successors after the graph has been scheduled does not help either. Then the assignment *q=1* for the sequences “100” and “110” would conflict with the assignment *q=0* in the other sequences. In fact, no assignment should be moved up since this example has a ring of dependencies. Alternative *a* depends on alternative *b* that depends on alternative *c* e.t.c.

This was remedied in the algorithm presented in [paper VI]. This algorithm zig-zags through dependency-rings to determine the excess assignments. However, the proposed solution was deemed to be too slow. The proposed algorithm has worse time complexity than the other two proposed algorithms. And, if there are rules that have unbalanced branches, i.e. one alternative is shorter than the others, the ScheduleDown algorithm, proposed in the same paper, may anyway have to perform a local tree-split to be able to schedule the graph,. This example is shown in Figure 5.12. It was therefore decided to always add a tree-split operation before scheduling. This solution was suggested in [paper VIII]. With the tree-splitting operation, the original algorithms proposed in [paper III] work for the Other-bit exception as well. Expanding the graph into a tree makes it possible to achieve a correct schedule at the cost of an increase in the graph size.

5.3.4. Phase 4 - State Minimization & Code Generation

The fourth and last phase performs the grammar DAG to FSM mapping and produces synthesizable VHDL code for later logic synthesis. It is divided into three steps. In step 4 a), state minimization is performed on the grammar DAG. Since the grammar DAG is a transition graph, it is easily converted into a THG. Although the state minimization algorithm [paper VII] has worse worst time complexity than ordinary state minimization algorithms performed on the STG [73, 74], the THG algorithm perform better for protocols that have long series of transitions and are very wide, i.e. they do not have so many alternative branches. Many frame-based protocols exhibit these properties. In THG minimization, all out-going edges from a transition node are viewed as a hyper-edge, to which a state should be assigned. The state assigned to an hyper-edge is called a *Hyperstate*. If a node has more than one incoming hyper-

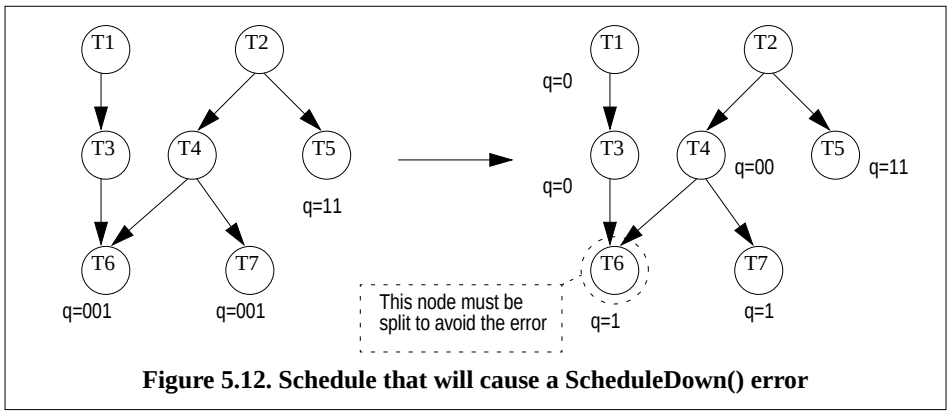


Figure 5.12. Schedule that will cause a ScheduleDown() error

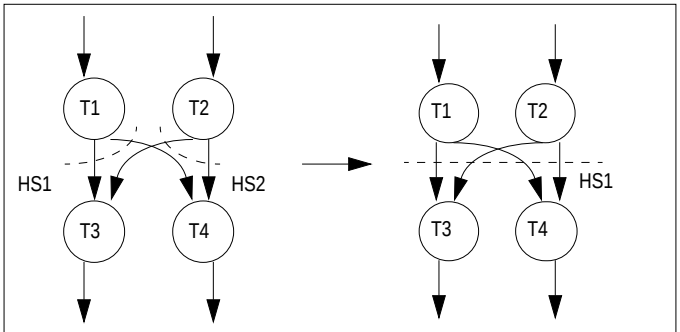


Figure 5.13. Hyperedges with identical futures are merged

edge, as is the case for reconverging nodes, the in-coming hyper-edges have equivalent states if they have the same set of successor nodes. If this be the case, they share the same future and the hyper-edges are merged to form a single, larger, hyper-edge and to permit further minimization, see Figure 5.13. The state minimization is governed by three premises:

1) All out-going edges from a node correspond to the same hyperstate.

2) In-going edges have equivalent hyperstates if their respective source nodes have the same set of successors, i.e. they share the same future.

3) Identical preceding transition nodes with identical futures, i.e. that share hyperstate, must be merged to permit further minimization.

After the THG has been minimized, states are marked on the graph. This is performed in step 4 b). A state number is assigned to the outgoing edges of each node. A state number can be assigned in two ways. The state machine can either be viewed as a single state machine, with a unique number assigned to each state, or it can be viewed as a state machine composed of a branch selector and a depth counter, where the uniqueness of the state is given by the 2-tuple (branch number, depth number). If there are long sequences in the grammar DAG, chances are that the final implementation of the FSM will be very counter-like. Thus, marking the states as the state-tuple (branch, depth) simplifies the synthesis task, although at the risk of having a sub-optimal final result. The designer selects which state marking to use by giving a constraint to the ProGram compiler.

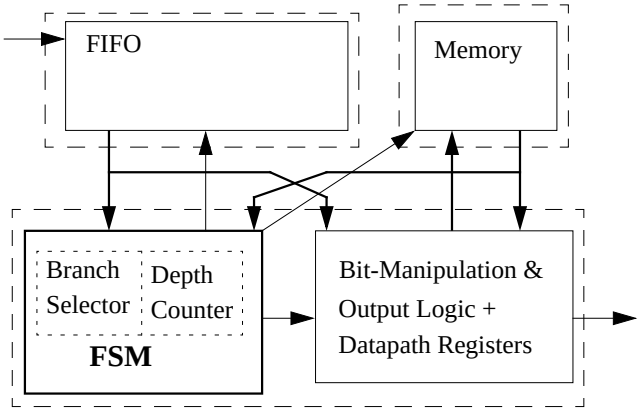


Figure 5.14. The target architecture for one start symbol

In step 4 c), finally, the FSM of the start rule is output in synthesizable VHDL code. The VHDL code is in behavioural RTL VHDL style for ease of debugging. The state machine model used is a Mealy machine with latched outputs. Any reference to previously parsed tokens in the input stream will result in a FIFO. Memories are an optional data construct and is generated only if specified. The target architecture is shown in Figure 5.14.

The VHDL code for the FSM is partitioned into two processes, one for calculating the next state and the values of the output signals and one that is sensitive on the clock, to latch the values of the present state and the output latches. Each start symbol corresponds to an entity and an architecture. A ProGram description, i.e. all start symbols, correspond to a set of concurrent communicating FSMs, one for each start symbol. The concurrent FSMs are connected at the top-level with an entity and an architecture.

5.4. Summary and Conclusion

Grammar-based approaches have several advantages over procedural specification styles when it comes to modelling and verification of communication centric designs like data communication protocols. Protocols have long been modelled using grammars in the software world.

The ProGram language is a grammar-based hardware description language, targeted for specification of data communication protocols. The language supports a description style that is implementation independent of the port sizes. The ProGram Compiler parses the language and produce a RT-level VHDL implementation in of the interface protocol. The actual implementation of the protocol is derived by the synthesis tool from the interface constraints given by the designer. The compiler is therefore well adapted to the implementation independent synthesis philosophy mentioned in chapter 1 on page 9.

Although the generated VHDL code is especially adapted for synthesis by the backend logic synthesis tool and the FSM is close to optimal, the input FIFO and the memory unit used in the target architecture is not optimal and could be improved. The implied FIFO is used for fast storage of input symbols. In some cases, if the memory is fast enough, an ordinary memory could with advantage be used instead of the FIFO to implement the same functionality at a

lower hardware cost. This is the case when the FIFO is used to store an input sequence, and the data is read from the sequence in another order than it was read, and only one data at a time is used for parsing alternatively in the actions. The FIFO solution need some kind of mux structure to supply the data. A standard RAM memory does not need an additional mux structure since the data forwarding from single RAM cells is already part of the RAM memory, and in addition highly optimised. In some cases, the reverse situation could happen. A data structure that has been specified as a memory might be better to implement as a FIFO structure.

6. Evaluation of the ProGram Methodology

This papers presents a summary of results obtained from using the ProGram Compiler on a sample set of designs. The results were presented in papers I, II, III, V, VI, and VIII.

6.1. Introduction

To evaluate the efficacy of the ProGram Compiler a series of experiments has been conducted. Section 6.2., covers experiments performed to test our design space exploration strategy. A single ProGram description of a reduced F4 OAM protocol has been implemented and used to generate different designs by varying the port-size constraints of the inputs and outputs to test our design space exploration methodology.

In section 6.3., results from evaluating the quality of the produced designs is presented. The compactness of the ProGram code was evaluated by coding a set of designs in ProGram, HLS style VHDL code and RTL style VHDL code, and then comparing the code sizes of the different descriptions.

The same set of designs were then synthesized using 1) a commercial available (standard) HLS tool, 2) an in-house HLS tool, specialized for the target domain and 3) the ProGram compiler followed by standard logic synthesis, and the area and timing results were compared. This is presented in section 6.4.

Section 6.5. presents results obtained from testing the Exception Handling constructs. The descriptions of the previous set of designs were analysed in terms of the usage of the introduced exception constructs. Finally, in section 6.6., the chapter is summarized and some conclusions of the experiments are drawn.

6.2. Design Space Exploration of Port Sizes - OAM example

Since the ProGram description is port size independent, it is possible to use the ProGram description to do Design Space Exploration using the port sizes as an additional exploration parameter. Most small protocols have the ports, and thereby their sizes, defined when they are specified. However, in the area of telecommunication, the ATM-protocols are specified independent of the port sizes and implementations differ from vendor to vendor. For instance, UTOPIA [71], the standard ATM interface, specifies 16 bit interface, whereas some ATM vendors [72] implement 128 bit port for a higher bandwidth.

To illustrate how ProGram could be used to specify these kind of protocols in an implementation independent manner, a subset of the F4 OAM protocol [87] was implemented in ProGram and design space exploration was performed. The OAM protocol can be partitioned into designs with port sizes that with an integer multiple gets the protocol's frame length, 424 bits.

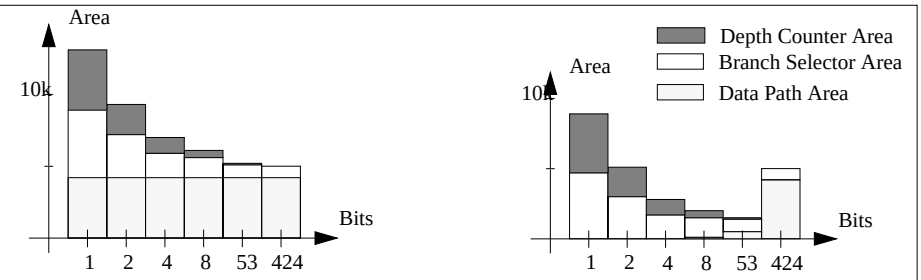


Figure 6.1. Design Space Exploration. a) Area vs. input port width. b) Area vs. input & output port width.

The input and output frames are of the same length. The first results were presented in [paper I] and [paper III]. These results are shown in Figure 6.1. Figure 6.1 a) shows the area results of having the output width constant at 424 bits and b) shows the results of having the input width equal to the output width. The area results for synthesizing the reduced F4 OAM protocol for a) various input widths with the output width held constant and b) for various input widths with the output widths the same as the input widths in are displayed in Figure 6.1 a) and b), respectively.

The first version of the ProGram Compiler was implemented only to use the two-way partition of the state machine into a branch selector and a depth counter. To further enhance the differences, the depth counter was implemented as a one-hot encoded Johnson-counter. The branch selector was implemented using binary encoding. The first version did not merge tokens if they were smaller than the input width, instead it produced an FSM similar to the one an HLS tool would produce, with several states making up one superstate formed by the sample clock. As a consequence, it produced more states than necessary for inputs wider than the specified tokens since it took a couple of clock cycles to decode the transition. For the one bit partition, this also meant that the depth counter was unnecessary since the branch selector contained all states necessary to capture the functionality. Thus, the implementation was not so efficient as could be desired.

Another problem with the implementation of the ProGram compiler was that the state machines also became so big that the backend tool had problems to synthesize them for small port widths. For instance, to be able to synthesize the OAM protocol with 1 and 2 bits as constraints for the input-width, the output flip-flops had to be taken out of the description to reduce the complexity of the synthesis task for the backend tool. The final result was then estimated by adding an area estimate of the missing flip-flops and the datapath that was connected to them, based on the area results obtained using larger port widths.

To tackle these problems, the ProGram compiler was rewritten. The new version merged tokens that were smaller than the port width. In this way, the generated state machines became smaller and unnecessary states were no longer generated. Another state marking algorithm was also written. Instead of partitioning the state machine in two, one single state machine was generated, resulting in a high quality state machine. The Myhill-Nerode theorem from 1958 [49, 52, 53] states that every regular expression has a unique minimum-state DFA. Regular expressions are equivalent to regular grammars. Since ProGram is based on a regular grammar, all ProGram models of a Protocol that can also be described using a regular expression, will result in a minimal FSM after state minimization. Exception Handling constructs on the other hand, make the situation more complicated, especially the condition construct. The condition construct makes it possible to describe counters, which can not readily be mapped to a regular expression without causing a state explosion.

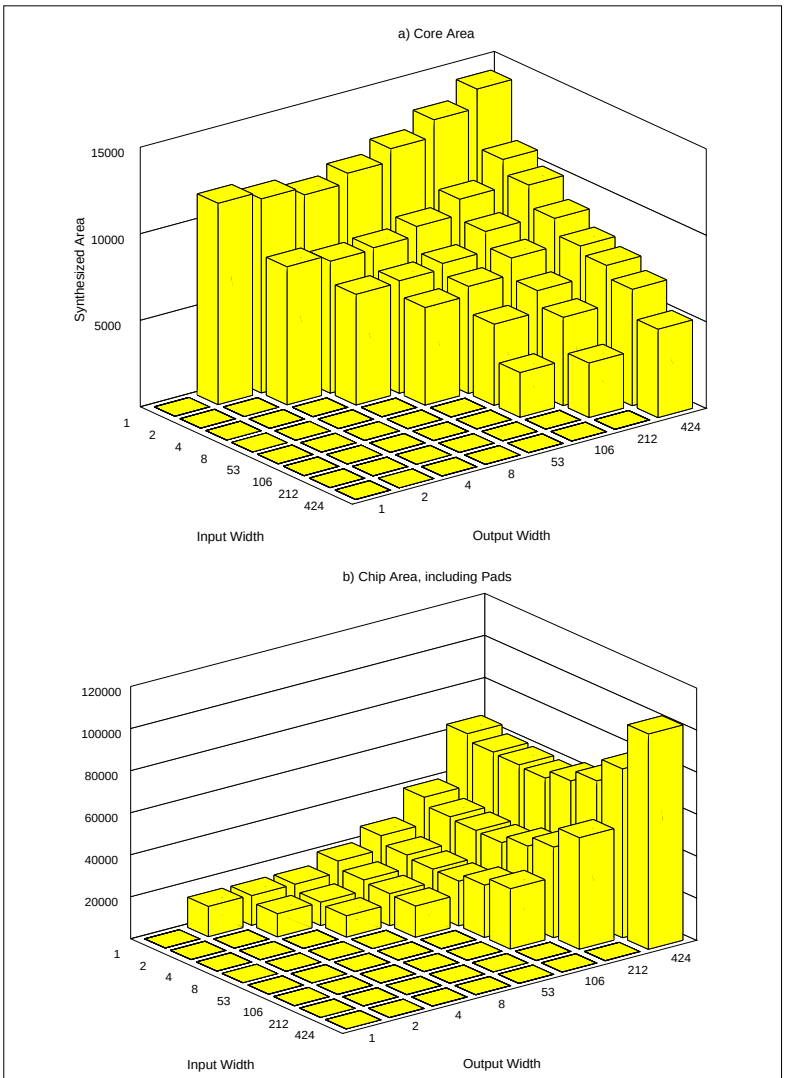


Figure 6.2. Design Space Exploration of Input vs. Output Port Sizes. a) Core Area b) Chip Area, including pads

The experiments with the OAM protocol were redone, this time with the new compiler. The results were presented in [paper VIII] and are shown in Figure 6.2 a) and b). Figure 6.2 a) shows the area for the core functionality, while Figure 6.2 b) displays the area for the whole chip, including the pad area.

Some of the designs that had output width equal to input width turned out to be unschedulable. The output assignments could not be moved up because of scheduling conflicts in the graph and the subsequent down scheduling failed because it reached the end of frame marker. Designs such as these needs control pipelining to be implemented, i.e. to allow the output assignments “spill over” into the next input frame. Designs with output widths smaller than the input width cannot be implemented unless oversampling of the input tokens are performed. Oversampling increases the numbers of control steps available for scheduling by checking

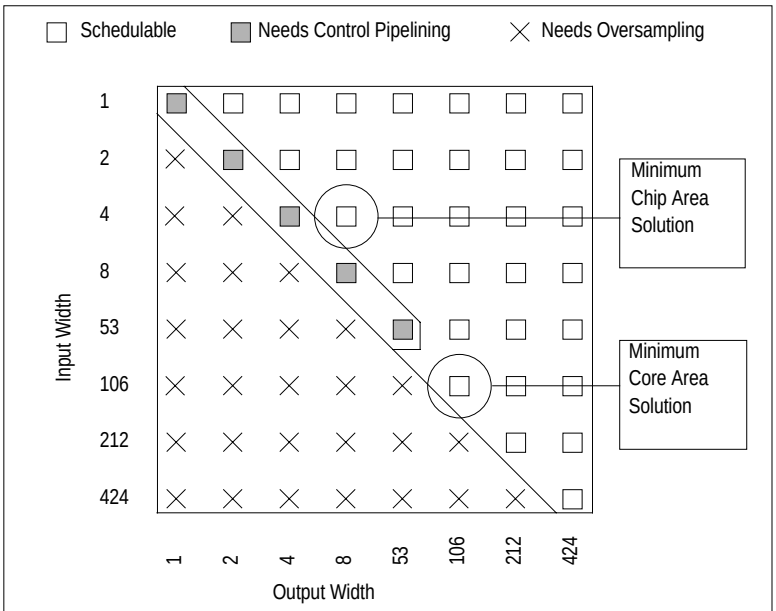


Figure 6.3. Schedulability of the Design Space

each token more times. This can be achieved by duplicating the input tokens, in this way making more tokens available for the scheduling algorithms. The schedulability of the design space is shown in Figure 6.3.

The design with minimum core area is the one with 106 inputs and 106 outputs, which has a core area of 2581 Gates. The design with minimum chip area is the one with 4 inputs and 8 outputs, which has a chip area of 10082 Gates. The designs with 1 and 2 bit input port sizes did not meet the timing requirements. The designs with 4 bit input port size barely met the timing constraint while all others had a large margin. This means that designs with 8 bits and more allows for a higher throughput than was specified.

Another encouraging result from these experiments is that the ProGram compiler is also very fast. For large examples with many states, the ProGram compiler generated the VHDL code five orders of magnitude faster than the backend tool could synthesize it. Thus, the time that the compiler takes for generating the VHDL code is negligible compared to the times it takes to synthesize the generated code.

6.3. ProGram vs. CMIST and HLS area results

To evaluate the quality of the synthesis procedure, the area results of synthesizing the Intel 8251A Personal Communication Interface [88] and the manchester encoder protocol using ProGram was compared with the results obtained from 1) a commercially available standard HLS-tool, and 2) an HLS-tool specialized for the target domain (CMIST). These results were presented in [paper II]

The functionality of the I8251A PCI was partitioned into five sub-protocols, of which two

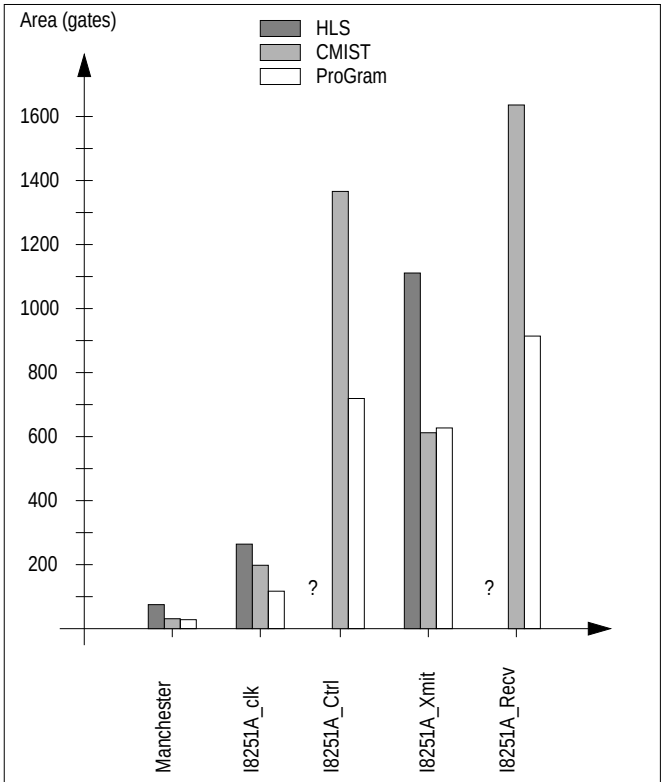


Figure 6.4. Synthesized Area. HLS, CMIST vs. ProGram

were identical. The two were implemented as a single protocol : the I8251A_clk protocol, a programmable clock divider protocol, used for generating the send and receive clock. The other three were the I8251A_xmit protocol, for sending data over the external interface; the I8251A_recv protocol, for receiving data over the external interface and the I8251A_ctrl protocol, for handling the CPU interface protocol and internal book-keeping for the I8251A_xmit and I8251A_recv protocol.

The synthesized area results are shown in Figure 6.4. The designs are ordered by the number of ProGram lines of code needed to describe the behaviour. Because of the identical functionality, and consequently also the same area, of the two clock protocols, only one of them are shown in the figure. The two sub-protocols I8251A_recv and I8251A_ctrl could not be synthesized in the standard HLS case, due to a bug in the commercial HLS-tool.

It should be noted that comparing the results with a common HLS is slightly unfair since HLS was developed for the computation centric domain of algorithms, and not for the communication centric domain of data communication protocols. The comparison with the CMIST tool set is fair, since it was specially developed to take this domain into account. The CMIST tool set was 2% better in one case, the same in one, and ProGram was significantly better for the other three cases.

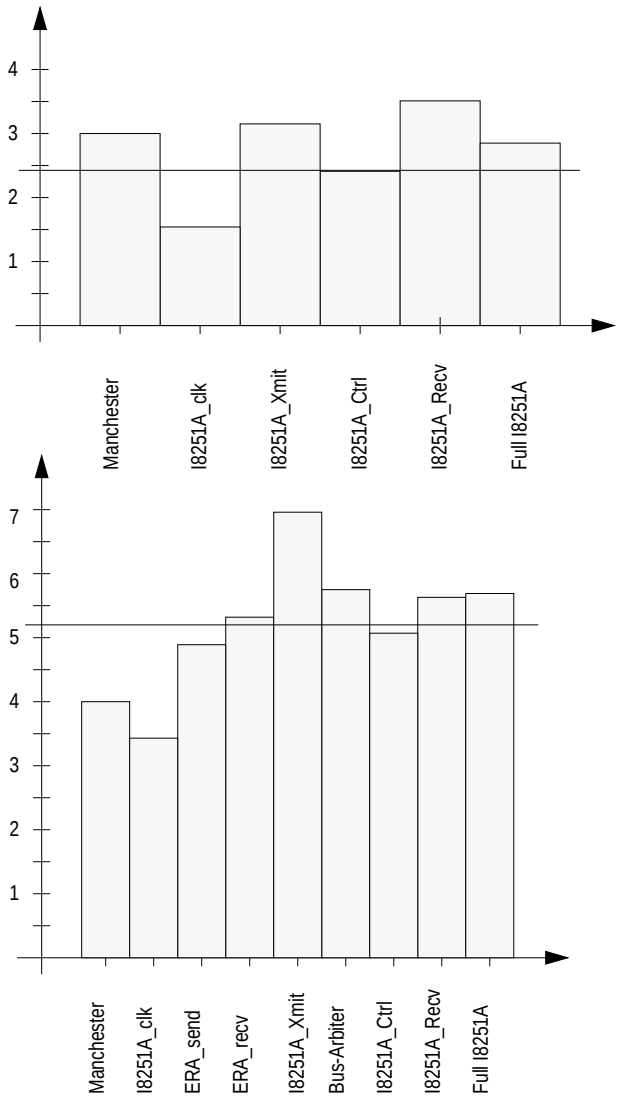


Figure 6.5. Code size ratios. a) HLS VHDL vs ProGram code size ratio. b) RTL VHDL vs ProGram code size ratio

6.4. ProGram vs VHDL code sizes

To evaluate the compactness of the ProGram code, four protocols of various sizes were implemented in different coding styles. The selected protocols were a manchester encoder, an industrial relevant protocol supplied from Ericsson Radio Systems (ERA), a bus-arbiter protocol and the Intel 8251A Personal Communication Interface. The selected coding styles for comparison were 1) VHDL targeted for HLS and 2) VHDL targeted for Behavioural RTL synthesis.

The ERA protocol was divided into two separate sub-protocols: one send and one receive pro-

tocol. The largest protocol (the I8251A) was partitioned in the same way as was described in section 6.3.

The results of these comparisons were presented partly in [paper II] and in [paper VIII]. They are shown in Figure 6.5 a) and b), respectively. Figure 6.5 a) shows the ratio between the HLS VHDL vs. the ProGram code size and b) shows the ratio between the Behavioural RTL VHDL vs. the ProGram code size. The (sub-) protocols are ordered with increasing ProGram code description size. The average ProGram vs HLS-code ratio is 2.7 and the average ProGram vs RTL VHDL-code ratio is 5.2. The average values are indicated with a solid line in the figures.

The I8251A was written as a single protocol in ProGram and some signals and buses are shared among the sub- protocols. Therefore, some values for the sub-protocols of the I8251A are approximate. In these cases, the lines of code needed to describe the interface, i.e. the entity description and local signal declarations, has been excluded in the line count to give a fair comparison with the VHDL coding styles.

6.5. Exception Handling results

To test the efficacy of our exception handling, a few protocols that had been written in ProGram and used several of the exception constructs were analysed. These were the ERA protocols, the Intel 8251A PCI protocols, and the bus-arbiter mentioned in the previous section. The results from the analysis of the exception handling constructs were published in [paper V] and [paper VI], and are summarized in Table 1. The two clock-protocols in the I8251A did not contain any exception constructs and are therefore not listed in the table.

Table 1. Modelling Results

Protocol	ProGram code (# of Lines)	RTL VHDL code (# of Lines)	Code Ratio	Types of Exceptions used
ERA_receiver	41	218	5.32	reset, error, condition
ERA_sender	37	181	4.89	reset, condition
Bus-Arbiter	59	339	5.75	reset, error, redirection, condition
I8251A_send	~52	~362	~6.96	reset, other-bit, redirection, condition
I8251A_recv	~92	~518	~5.63	reset, other-bit, redirection, condition
I8251A_ctrl	~68	~345	~5.07	reset+sequence, other-bit, redirection, condition
Full I8251A	370	2106	5.69	reset+sequence, other-bit, redirection, condition

The number of lines for the I8251A_recv, I8251A_send and the I8251A_ctrl listed in Table 1 are the number of lines of code discounting the declarations, i.e.it contains only the number of lines needed to describe the functionality. The reason for giving approximate numbers is that the three remaining protocols of the I8251A were written together as a single protocol, as was mentioned in the previous section. Therefore, it would be hard to know which lines should be

accounted for and which one should not. Instead, the number of lines needed to describe the functionality in ProGram and the number of lines to describe the functionality in the generated FSM Style RT-Level VHDL was compared. As can be seen, ProGram allows to specify the same functionality with a code ratio ranging from 4.9 and 7.0.

The mostly used exception handling constructs were the reset and the condition, closely followed by other-bit and redirection. The error construct was used only once. The interrupt construct was never used. The reason that the interrupt was not used was because this kind of exception is most suited to handle abort situations in hardware, a functionality which none of the implemented designs exhibited. A typical example of an abort situations is the situation when data is written to a buffer queue and the buffer queue overflows, and no additional data can be written to that queue in case of overflow. Then the writing procedure must be aborted immediately, no matter where the sending FSM is in its execution.

6.6. Conclusion

We have presented a methodology for specifying data communication protocols in an implementation independent manner and synthesizing hardware from such specifications. The approach is promising. The synthesis tool developed can be used to explore alternative realisations with different widths of the I/O ports.

From the experiments it can be seen that VHDL generation time is very small but the time it takes to perform State Machine and Logic Synthesis is very long, particularly for small bit widths, because the state space of the design becomes large.

Grammar-based Hardware synthesis has very compact descriptions compared to existing methods. The grammar description is on average 2.7 times smaller than the same functionality described using Behavioural VHDL-code and 5.2 times smaller than the one needed to describe the functionality using RTL FSM style VHDL-code.

The advantage of using a grammar-based description is that it captures RTL functionality on a higher-level which results in fewer lines of code.

The experiments on the quality of the produced code are promising, and though the number of synthesized designs are small, the conclusion must be that the quality of the produced code is very high. For small designs, the produced VHDL code is as good as the code produced by an experienced designer. Larger examples also have as good code quality as the one produced by an experienced designer, but for these examples it is not sure that the ProGram description of the protocol is the best possible, i.e would result in the smallest/fastest/least power-consuming possible implementation. Most protocol specifications can be specified in several ways, each and every one of them fulfilling the functionality of the specification.

7. Thesis Summary & Directions for Future Work

This chapter summarises the contributions in this thesis and describes some directions for future work.

7.1. Summary of the work presented in the thesis

A data communication protocol is an agreement between two or more communication parties about the exchange of messages in order to provide some service. The protocol specify the language used to communicate over the interface between the involved systems. A natural way to specify an interface protocol is in terms of a grammar expressed in the BNF notation and annotated with actions. The vocabulary of messages used to implement the protocol correspond to the grammar rules. The encoding format of each message in the vocabulary corresponds to the token terminals of the grammar. The procedure rules guarding the consistency of message exchanges are also embedded in grammar rules. The service provided by the protocol corresponds to the actions in a grammar. Assumptions about the environment can be viewed as port width and throughput constraints posed to the synthesis process.

This thesis presents a grammar-based language called ProGram, that allows a designer to describe frame-based protocols in a port-size independent manner together with methods for performing hardware synthesis from such descriptions. The ProGram language and the ProGram compiler has established a working methodology for design space exploration of port sizes. The code of the language is compact and the results after hardware synthesis are comparable to hand-written designs.

In [paper I], the ProGram language was introduced and the concept of design space exploration of port sizes. Synthesis algorithms for building the internal representation and performing design space exploration of input port sizes were presented. The methodology was extended with synthesis algorithms for performing design space exploration of output port sizes in [paper III].

To reduce the effort spent for specifying a protocol, exception handling constructs were introduced in [paper IV]. In [paper V], new synthesis algorithms were presented that could cope with the exception handling constructs.

The hierarchical scheduling problem was first described in [paper VI] together with an evaluation of using the exception handling in the ProGram language for modelling different designs. Another method for solving the hierarchical schedule problem was presented in [paper VIII]. This paper also presents results from exploring the design space of a typical frame-based protocol example from the telecom world. Results obtained from a case study comparing the ProGram methodology with conventional HLS was performed in [paper II].

Algorithms for performing state minimization on the internal representation used in ProGram was presented in [paper VII].

7.1.1. Directions for Future Work

Currently, the ProGram methodology targets data communication protocols. Data communication protocols are control centric and contain little or no datapath. However, many protocols also contain some datapath is them. To be able to handle datapaths some open issues remain. The ProGram Compiler cannot split and merge datapath operations when it performs the design space exploration. As a result, sub-optimal results are achieved. After datapath operations has been split, they need to be scheduled over the available control-steps. This is a common task performed by standard HLS. Therefore, a connection to HLS needs to be established.

The ProGram language still has some deficiencies in the description style. The exceptions constructs described in the conclusion of chapter 4, would boost the descriptive power of the language. Another deficiency comes from that it is currently not possible to synchronize the scheduling of output assignment to different output ports without explicitly writing them into the desired control step.

Another open issue that has not been dealt with is the similarity of the ProGram language with Process Algebras. If a link can be established, formal properties of the ProGram designs can be proved. Also, the automatic selection of clock for protocols running on different clocks need to be established. Cosimulation with third party languages could be useful to be able to do rapid virtual prototyping of control dominated designs.

The rest of this chapter will deal with open issues in the ProGram methodology that remains to be solved. In section 7.2, unsolved issues concerning the connection to HLS is covered. Section 7.3 covers the issues concerning output assignment scheduling. Section 7.4 covers other unsolved issues like formal verification, low power, clock protocols, matlab cosimulation etc. Finally, in section 7.5, some conclusions are drawn.

7.2. Connection to HLS

A situation that the ProGram compiler does not handle, is delays in the datapath. It is assumed that all computations scheduled into a control step is faster than the system clock. This is a disadvantage if there are several computation steps involved in computing the result of an expression. The only thing that matters from the protocol point of view is that the output assignments are done at a specific time and place within the protocol. Thus, the placement of the datapath operations do not matter at all. They can be placed anywhere in the protocol as long as the input signals that the expression depends on are available and that the result is computed before the result should be assigned to the output. The datapath operations can therefore be scheduled over the available control steps. This would also save area since operations in the datapath can be reused. Scheduling of datapath operations is a standard step performed in High-Level Synthesis. In the case of protocols, a speculative scheduler for the datapath is the appropriate choice. A speculative scheduler has the potential to allow more time for calculating the result of an expressions. It allows datapath operations to be scheduled before the decision point of selecting an alternative. Thus, sub-results of the expression may be computed even if they are not needed.

7.2.1. Intelligent Library Functions

Computations in the datapath might also be wider than the output stream. In these cases the

computation itself can be split and scheduled over the available control steps. This would again save area since sub-parts of the datapath operations can be reused. This is also a kind of automatic bit-parallel to bit-serial conversion. For instance, a description of an adder written in the right way would become fully parallel if the input port widths are set as wide as the word size of the addition. On the other hand, if the port width is set to one, the add operation would be split into one-bit adder-primitives. The adder-primitives, most likely full-adders, would then be scheduled over the available control steps, generating a fully bit-serial adder.

For these reasons, some kind of connection to ordinary HLS is needed. The ProGram compiler would then extract the functionality from the datapath and the cycle budget, i.e. the number of cycles that the functionality is allowed to be scheduled over. This is then fed to the HLS tool and the results from performing HLS is then back-annotated into the protocol. The Compiler would then output the Protocol FSM and the resulting datapath.

A problem that exist with this methodology is that the way in which operations should be split is not easy. In the adder case mentioned above, full adders were suggested as an adder primitive which is a correct decision for one-bit port widths. However, if the port widths grow wider, at some point a ripple-carry structure on the adder primitives is not longer the best solution, especially if the timing constraints are such that the delay of the ripple-carry structure is to long. Then, a carry-look-ahead adder or even a carry-save adder primitive would be a better choice. The problem grows more complicated the larger and more complex the datapath. The best way to handle this situation is probably to let every datapath operation to be the equivalent of the Object construct used in many programming languages. The object would have synthesis methods associated with it that tells it how to react onto given constraints and technologies. The object would then always “know” how to calculate the best way to split itself should such a situation arise [93].

7.2.2. Pipelining of the datapath

Another issue to solve is the pipelining of the datapath. If there is not enough time for the datapath to complete its operation because of the frame rate constraints, the datapath must be pipelined. A problem with this is that the length of the pipeline might become longer than the input frame itself. In these cases, letting the input frame FSM control the output pipeline might become unwieldy, although possible. A better way would be to separate the output controller from the input frame FSM, and let the input frame FSM give a token to the output.

In some cases, the access to pipelined signals is desired. Examples of this are found in the world of digital filtering. For instance, to describe an FIR filter, the ideal way to describe the filter is to describe it as a sum of products. Each product is a constant multiplied with the delayed input. To be able to easily describe sum of products, generating constructs for the datapath, with the same functionality as the traditional for-loop is also needed.

Another thing that is missing is a construct for having transparent ports, i.e. ports and local signals that do not have registers at the output. These are useful to describe pure datapath assignments, assignments that will never need a controller associated with it. There might also be a need for pure datapath assignments with registered signals and ports. This construct should yield the same structure as the transparent construct, with the difference of the registered output.

All the pure datapath constructs mentioned is needed if ProGram should be used it in a real life design environment. ProGram was developed with data communication in mind only. It needs

strengthening on the datapath side. Since ProGram is a protocol grammar, written in a functional style using production rules, these datapath additions should be in the same style as the control path productions. Otherwise, the style of the description changes when the designer should write the control description compared to then he/she should write the datapath description. There is already a language for these kind of datapath descriptions, it is called Silage, and has been used as description language for HLS in the Hyper [89] and Cathedral [90] systems. Since ProGram also has the feature of WordAligning both input and outputs, the language cannot be adopted as is, but must be adapted to suit the style of ProGram. However, since it is a pure functional description language, it has great similarities with the production rules used in ProGram, so many constructs and syntactical details will remain the same.

7.2.3. Datatypes

ProGram does not currently support datatypes. Datatypes are usually used as an abstraction by the designer to make modelling of systems that contain many computations easier. An addition is still an addition, no matter the type. The type infers the actual implementation.

ProGram was developed from the communication point of view instead of the computation point of view. From the communication point of view, the actual type of an incoming stream of bits is not important. The order in which the bits come in and go out is much more important. Instead, the datatypes can be modelled in the same way as assembly code is written. Any special treatment of the incoming bits is then reflected in the name of the instruction treating the bits, in ProGram's case the name of the function implementing the functionality. Thus, an addition of two real numbers cannot be written as q=(a+b) but must instead be written as q=add_real_real(a,b). This is not a serious problem as long as all code is written in the ProGram language. However, it may become a serious problem if the ProGram compiler should include datapath code written in a third-party language. Then, some way of mapping the ProGram data streams, which do not have type information, onto the external function's arguments, which have type declarations, must be devised.

7.2.4. Memory handling

The current way of specifying memories is not satisfactory. In the current implementation, the size of the memory is inferred by the number of lines used in the memory's address bus. Since there are many applications that do not use a full-sized memory, memory area can be saved if the number of memory cells are given and the number of address lines to the memory is inferred instead. Also, only a single indexing of memory records is currently used. In many cases, the modelling of a design is easier if multiple indexing were allowed.

The size of the data-bus is not given explicitly. This will be added as a constraint given by the designer in the future to allow for design space exploration of memory sizes. There are two reasons that memory exploration was left out from the original ProGram language. The first was that there are many research groups around the world addressing this problem. The second was the observation that one of the key problems with memory optimization is that the sequence in which the data is read from the memory affects the order in which data should be packed together in the memory. If a protocol were to parse something already existing in the memory, the order in which the designer specifies that things should be read from the memory is most likely sub-optimal. From the protocol point of view, the sequence in which data is read from memory is not so crucial as long as there is enough space to schedule the actions. Thus, there is huge potential for possible optimization by reordering memory references.

Another issue is that only one type of memory is allowed currently. This is rather inefficient since there are many types of memories that may be accessed in the actions. In addition to standard RAM memories, ROM memories should be possible to specify as well as FIFO memories. In some cases it might be more efficient to implement a data structure in a FIFO than in a RAM memory. The reverse situation might also happen. An efficient memory handler must be able to identify these situations and take appropriate actions.

7.3. Output Assignment Scheduling Issues

7.3.1. Input-Output frame pipelining

A situation that is not captured by the ProGram compiler is the case when an output frame is identical in size as an input frame and the output cannot be scheduled because of a conflict in the grammar DAG. If the frames are of equal size, there is a possibility to schedule the graph by allowing the output frame to “spill over” into the next input frame. To be able to do this, new output scheduling algorithms are needed. A simple way to solve this problem would be to let the down scheduling algorithm continue from the start of frame token if it reaches the end of frame token, adding a mux in the front to tell which of the end branches that was reached. The scheduler needs to align the output frames to assert that no output frame can overlap over any output assignment in any of the input frame alternatives.

The OAM protocol can be partitioned into designs with port sizes of integer multiples of the protocols frame length, 424 bits. The input and output frames are of the same length. Designs with an output port size less than the input port size can therefore not be scheduled by the current implementation of the ProGram compiler since there will be more outputs to schedule than there are tokens available. This could be remedied if the input bits would be oversampled, thus duplicating every input token a number of times to make up for the differences in communication port rates. In this way more tokens would be available for the scheduling algorithms to schedule the output assignments.

Another limitation is that the output scheduler cannot schedule output assignments across loop boundaries. The reason for this limitation is that it is not possible to determine the number of iterations in a loop beforehand in the case of dynamic loops, where the number of iterations depends on an external signal. However, it is possible to schedule output assignments across loop boundaries in the case of static loops. To do that, it is needed to keep track of the current iteration to distinguish between output assignments in the different iterations. One simple solution to this problem would be to unroll the loop before scheduling is performed. This would however increase the number of states significantly if the loop is large or it has a large number of iterations.

7.3.2. Other output scheduling methods

There is at least two other ways to schedule the output assignments than has been described in this thesis. One way is to go through all possible paths, collect the length of all paths, the length of all output assignments and their insertion place, respectively and to pose the scheduling problem as an Integer Linear Programming (ILP) problem. This way has certain advantages since it would use a well-known method for solving the problem. It would also be easier to handle the input-output frame pipelining since the output assignments can be placed after the final insertion place has been calculated. A disadvantage is that collecting the timing infor-

mation and going through all possible paths is a time consuming task. So are the methods for solving ILP problems.

Another way would be to disconnect the output frames from the input frames, and move around a pointer to the output frame during scheduling. In conjuction with the pointer, an integer is needed to keep track of which sub-part of the output assignment is to be output with the token pointing to it. This solution has two major benefits: 1) during scheduling, only pointers are moved which speeds up the algorithm considerably; and 2) the input-output frame pipelining becomes easier to handle since the output assignments can be placed after the final insertion place has been calculated, just as with the ILP method. A disadvantage is that it will still be needed to perform both up and down scheduling since the position numbers need to be updated if there are conflicts in the graph.

To be able to determine which scheduling algorithm is best, a case study needs to be performed. The existing scheduling algorithms should be compared with the two suggested above. The best scheduling algorithm is the fastest one. A complexity analysis is not enough because the speed of the algorithm might depend on the problem. For instance, the scheduling algorithms suggested in [paper VIII] need a tree-split operation, an operation that is not needed with the ILP method since it goes through all possible paths. However, it is needed to incorporate some kind of tree-splitting operation in the pointer-method since it might move the pointer across a boundary where there are scheduling conflicts. On the other hand, the pointer method has great similarities with the already existing ones so it is most probable easy to implement.

7.3.3. Output Assignment Constraints

A problem with the current compiler is that it is not possible to synchronize different output assignments with each other in an easy way. Sometimes a control signal is supposed to be sent together with another signal, a signal that will be scheduled over the available control steps. The most logical place in the code to put the control signal assignment is the same place where the assignment to the other signal is placed and let the control signal follow the one that is being scheduled. Today, this is not possible. Even if a the control signal is filled up with memory symbols or filled up with the control signals off-value to match the frame length for that alternative, that is no guarantee for the control signal to be scheduled correctly. Since the control signal most probably will have the same value for all alternatives, making it easy to schedule, the associated signal might have conflicts and, because of the conflicts, that signal will be scheduled at a later time than the control signal. Thus, there is a need for some kind of scheduling constraint that tells the scheduler to schedule a group of signals together.

Another constraint that is needed is a constraint to tell the scheduler to evenly distribute a schedule over the available states. The present scheduler can only schedule the output assignment in bursts. That is, a signal is scheduled as close to the insertion point as possible and all the assignments are placed in consecutive available states. Sometimes the output frame should be sent at an integer fraction of the input frame. Thus, the relation between input and output frame must be known. In addition, the receiving protocol at the other end might run on a slower clock, requiring that the output frame must be equally distributed over the input frame. Also, sometimes a fixed timing relation is desired between the input and the output frame. Thus, some timing constraints would be needed to tell the compiler what latencies are acceptable and what are not.

7.4. Other Issues

7.4.1. Introducing Hierarchy and library functions for a Reuse methodology

To be able to define a complete methodology based on the ProGram language, constructs for reusing parts of the ProGram code, independent of the current code level, would be very useful. A problem with the current ProGram language is that the name of the productions, input and output ports of a protocol are hard coded. This makes it troublesome to reuse a protocol, or parts of it, in another design. The sub-protocol might have “hidden” production rule names that conflict with names in a bigger protocol being written. The sub-protocol might have local redirections that override the input stream of the top production. The sub-protocol’s output assignments should be send on parts of a bus, etc. Today, this can only be done by importing the ProGram code and replace conflicting names with new ones, which is inefficient.

A good way to handle these problems would be to have hierarchy and library functions included in the ProGram language. A simple automatic naming hierarchy would solve the problem of reusing code from a one protocol inside another, but it does not solve the problem with the hardcoded local redirections of inputs not the problem with hardcoded output assignments. To solve this some kind of remapping mechanism similar to the function call’s parameter mapping commonly used in ordinary programming languages like C++ and Pascal is needed.

7.4.2. Formal semantics - exploiting the similarity with Process Algebras

An untouched issue is the Formal semantics of ProGram. The language has been developed in an ad-hoc manner, adding new constructs whenever they were needed. As a consequence, the semantics of the language is not clear at all points. An example of this is for instance how the *Other-bit* token should be interpreted during token merging. Defining the formal semantics of the language would help designers understand the interpretation of the language constructs and also help tool developers when writing synthesis algorithms for the language. It is also possible that defining the formal semantics will discover some conflicts that are hidden within the language itself.

Another issue not dealt with today is the similarity between ProGram and process algebras, like LOTOS [24]. LOTOS has many constructs that are similar to ProGram’s in many respects. LOTOS is also a formal language, so finding common cross-points between ProGram and LOTOS is a good starting point in defining a formal semantics for ProGram itself.

Control-flow expressions [67], is another good starting point. It is a subset of a process algebra and it is targeted for describing control-flow intensive designs like data communication protocols. Many of the algebraic constructs have a direct relation to the internal representation used in ProGram. Also, some of the synthesis steps performed in the ProGram compiler can be related to the algebraic operations allowed in the control-flow expression algebra.

7.4.3. Formulating a design methodology using ProGram

To really use ProGram in real life design projects, a robust design methodology must be defined. A step on the way is to determine how the protocols should be validated. This has already been researched and has been published in [94]. Another step on the way would be to find good estimation algorithms. A good estimation algorithm that could estimate the results

of the subsequent logic synthesis would speed up the design process. Thus, a designer could specify a design, check which port size gives the “best” solution and synthesize only this one. The problem of estimation is complicated because of the extra freedom in selecting input and output port widths. It is important to note here that a good estimation algorithm does not necessarily have to be exact, i.e. give estimates that are close to the real values. It is much more important to have high fidelity, i.e. that the relative differences between the estimates and the relative difference in reality is the same. After all, design space exploration is performed to determine which design gives the minimum area. The exact area can always be obtained later after logic synthesis. An initial study has been performed [91], but the quality of the estimation algorithms could need an improvement.

7.4.4. Importing of Matlab and c-functions for describing the datapath

Importing datapath functions written in a third-party language is a good way of including Intellectual Property. Also, in many cases, the modelling of dataflow functionality is easier in a procedural language such as Matlab or C. Especially Matlab is very common for modelling of signal processing applications. After modelling the design in Matlab, the matlab code can be converted into a bit-true model, in which the real-type operations have been replaced with fixed-point integer type operations. Importing a bit-true model into ProGram would allow to add the reset and configuration-behaviour as well as producing a rate-true model of the design. Then, a way to pipeline, split and schedule the included datapath operation over the available control steps is needed. Once this possible, a synthesizable model can be produced. An initial study has been performed to formalise the concept [92], but the implementation work has yet to be performed.

7.4.5. Automatic clock selection for protocols with multiple frame rates

Protocols that changes parsing stream, and the streams have different input rates, can be implemented in several ways. The design can be driven with a high frequency clock, and using idle states to solve the problem that the protocol should run at different speeds in different parts of the protocol. This implementation style has the disadvantage that the clock lines of all flip-flops in the design are toggling at a high frequency, thus having a high power consumption.

Another way to solve this problem is to have multiple clocks driving the system and selecting the most appropriate clock that should drive the system whenever needed. The advantage of this solution is that the design could always be run at the most optimal speed. Since the clock lines of the flip-flops will not always be toggling at the higher frequency, this solution will consume less power. Unfortunately, using different clocks for driving the protocol will make the design harder to test.

An initial study of this idea and its consequences has been performed and was published in [96]. In the study, the clock protocols were designed by hand and the insertion point of the clock selection signal inside the protocol was done manually by the designer. On the other hand, this process should be possible to automate under some conditions, especially if the relation between the different frame rates of the input streams that are parsed is known.

7.4.6. Mapping of ProGram code to a General Purpose Protocol Processor

Industry is very fond of using standard components. With the emergence of Systems-on-a-chip

and IP-based methodologies, forced into existence by ever reducing time-to-market constraints, this statement becomes even more true. ProGram might be used by industry in real life for generating glue logic between these IP-block, although using ProGram for this purpose might be overkill. Interfacing between incompatible protocols does not usually need ProGram's ability to describe port-size independent protocols. This port-size independence is much better utilized when designing data communication IP-blocks themselves. On the other hand, generating the IP-block from a ProGram description might compromise the testing of the full system since the generated components must be validated themselves before the system is validated. For industry, it would be much better if a pre-tested component with a well-tested structure could be used. The well-tested structure could then have a reprogrammable memory that would allow the company to download the functionality and change it if bugs are discovered or functions should be added. Reprogrammable structures saves money for the manufacturing company and also makes the design more versatile.

These kind of structures has been commonly used in the world of processors. General purpose processors are used to solve common computation tasks and DSPs are commonly used to boost performance for image processing applications. What is missing is a special kind of processor family, targeted for data communication. A case study to see how the internal structure of such a process could look like has been presented in [95]. A question that still remains is what language to use for this type of processor and how the compilation process should be handled. In a sense, this approach is very similar to the one that Bloks suggested in [58]. He describes a grammar processor and the programming language ProGrIL. A description of the protocol in the ProGrIL language is used to derive the microcode of the grammar processor. Direct compilation of the microcode is of course a solution but there is no reason that the grammar description could not be compiled into a protocol assembler instead.

To utilize ProGram in this manner is possible as well, but to map a protocol description onto a fixed processor structure would require a rethinking in the synthesis strategies. For instance, can ProGram's feature of different giving port-widths as constraints for the compiler still be utilized? How should the ProGram constructs be mapped into the assembly language of the protocol processor? Many questions needs to be raised and many problems needs to be solved before data communication protocols can be implemented on a general purpose protocol processor.

7.5. Conclusions

The ProGram language and the ProGram compiler has established a working methodology for design space exploration of port sizes. The code of the language is compact and the results after hardware synthesis are comparable to hand-written designs.

Nevertheless, when all thesis work comes to an end, there are usually still some open issues that remain unsolved. So also with this thesis. This chapter has reflected some of the thoughts and observations that has arisen during this thesis work.

Some data communication protocols contain a datapath. To be able to handle datapaths, datapaths need to be split and merged, before scheduling them over the available control steps. Therefore, a connection to HLS needs to be established. The ProGram language still has some deficiencies in the description style and because it is not possible to synchronize the scheduling of output assignment to different output ports. Other open issues include how to prove formal properties of the language and how to automatically select the clock if the protocol is

running on different clock speeds at different times. Cosimulation with third party languages could be useful to be able to do rapid virtual prototyping of control dominated designs. The shear amount of unsolved and open issues that remain is so big that the work that has been left out is probably enough for two or three additional PhD-theses. And there is no doubt that, as the research of these topics evolve, more open issues and unanswered questions will arise.

8. References

[1] G. Moore, “Nanometers and Gigabucks - Moore on Moore’s Law”, University Video Corporation Distinguished Lecture, 1996. (http://www.uvc.com/).

[2] P. Michel, U. Lauther, P. Duzy, editors, “The Synthesis Approach to Digital System Design”, Kluwer Academic Publishers, 1992, ISBN 0-7923-9199-3.

[3] D. D. Gajski, R. Kuhn, “Guest Editors’ Introduction: New VLSI Tools”, IEEE Computer, vol. 16, no. 12, pp. 11-14, Dec. 1983.

[4] D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, R. L. Blackburn, “Algorithmic and Register-Transfer Level Synthesis: The System Architect’s Workbench”, Kluwer Academic Publishers, Boston, 1990.

[5] A. Hemani, Docent lecture.

[6] F. J. Ramming, *Systematischer Entwurf Digitaler Systeme*, B.G. Teubner, Stuttgart 1989 (in german).

[7] F. J. Ramming, “A Multi-Level Cybernetic Model of the Design Process”, In Proc. of IFIP Working Conference on Methodologies for Computer System Design, 1985.

[8] W. Ecker, M. Hofmeister, S. März-Rössel, “The Design Cube: A Model for VHDL Design Flow Representation and its Application”, In *High-Level System Modelling: Specification and Design Methodologies*, chapter 3, R. Waxman, J.-M. Berge, editors, Current Issues in Electronic Modelling, vol. 4, Kluwer Academic Publishers, 1996.

[9] A. Jantsch, S. Kumar, A. Hemani, “The Rugby Model: A Framework for the Study of Modelling, Analysis and Synthesis Concepts of Electronic Systems”, In Proc. of DATE’99, Munich, Germany, 1999.

[10] D. Gajski, N. Dutt, A. Wu, S. Lin, “High-Level Synthesis - Introduction to Chip and System Design”, Kluwer Academic Publishers, 1992.

[11] M.C. McFarland, A.C. Parker, R. Camposano, “The High Level Synthesis of Digital Systems”, In. Proc. of the IEEE, vol. 78, no. 2, pp. 301-318, Feb., 1990.

[12] R.A. Walker, R. Camposano, A Survey of High-Level Synthesis Systems, Kluwer Academic Publishers, Boston, 1991.

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

[14] F. Vahid, S. Narayan, D. D. Gajski, “SpecCharts: A VHDL frontend for embedded systems”, IEEE Trans. on CAD, vol. 14, pp 694-706, 1995

[15] D. D. Gajski, S. Narayan, L. Ramachandran, F. Vahid, “System Design Methodologies: Aiming at the 100 h Design Cycle”, IEEE Trans. on VLSI Systems, vol. 4, pp 70-82, March 1996.

[16] B. Svantesson, P. Ellervee, A.Postula, J. Öberg, A. Hemani, “A Novel Allocation Strategy for Control and Memory Intensive Telecommunication Circuits”. The 9th International Conference on VLSI Design, pp. 23-28, Bangalore, India, January 3-6, 1996.

[17] W. Wolf, “The FSM Network Model for Behavioral Synthesis of Control-Dominated Machines”, In Proc. of the 27nd Design Automation Conference, pp. 692-697, 1990.

[18] R. Camposano, “Path-based Scheduling for Synthesis”, IEEE Transactions on Computer-Aided Design, Vol. 10, No. 1, pp. 85-93, January, 1991.

[19] S. H. Yuang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu, J. F. Wang, “A Tree-Based Scheduling Algorithm for Control-Dominated Circuits”, In Proc. of the 30th ACM/IEEE Design Automation Conference, pp 578-582, June, 1993.

[20] N. Wehn, J. Biesenack, T. Langmaier, M. Münch, M. Pilsl, S. Rumler, P. Duzy, “Scheduling of Behavioural VHDL by Retiming Techniques”, In Proc. of EuroDAC’94, pp. 546-551, Sept. 1994.

[21] T.-Y. Yen, W. Wolf, “An Efficient Graph Algorithm for FSM Scheduling”, IEEE Transactions on VLSI Systems, Vol. 4, No. 1, pp. 98-112, March 1996.

[22] P. Ellervee, A. Kumar, B. Svantesson, A. Hemani, “Segment-Based Scheduling of Control Dominated Applications in High Level Synthesis”. International Workshop on Logic and Architecture Synthesis, pp. 337-344, Grenoble, France, December 16-18, 1996.

[23] P.J. Denning, B. Metcalfe, editors, “Beyond Calculation”, ACM press, 1997, ISBN 0-387-94932-1.

[24] Edited by K. J. Turner, “Using Formal Description Techniques”, John Wiley & Sons Ltd., 1993.

[25] G. J. Holzmann, “Specification and validation of Protocols”, Prentice-Hall International Inc., 1991.

[26] A. S. Tanenbaum , “Computer Networks”, 3rd edition, Prentice Hall Press, March, 1996, ISBN: 0133499456.

[27] J.-M, Daveau, G.F. Marchioro, T.B. Ismail, A.A. Jerraya, “Protocol Selection and Interface Generation for HW-SW Codesign”, In Transactions on VLSI Systems, Vol. 5, No. 1, pp. 136-144, March, 1997.

[28] T.B. Ismail, M. Abid, A. Jerraya, “COSMOS: A CoDesign Approach for Communicating Systems”, In Proc. of CODES’94, pp. 17-24, 1994.

[29] J. Madsen, B. Hald, “An Approach to Interface Synthesis”, In Proc. of the 8th International Symposium on System Synthesis (ISSS), pp. 16-21, 1995.

[30] J.-M, Daveau, T.B. Ismail, A.A. Jerraya, “Synthesis of System-Level Communication

by an Allocation-Based Approach", In Proc. of the 8th International Symposium on System Synthesis (ISSS), pp. 150-155, 1995.

[31] P. Chou, R. Ortega, G. Borrielo, "Interface co-synthesis techniques for embedded systems", In Proc. of ICCAD'95, pp. 280-287, 1995.

[32] B. Lin, S. Vercauteren, H. De Man, "Embedded Architecture Co-Synthesis and System Integration", In Proc. of CODES'96, pp. 2-9, 1996.

[33] F. Vahid, L. Tauro, "An Object-Oriented Communication Library for Hardware-Software CoDesign", In Proc. of CODES'97, pp. 81-86, 1997.

[34] M. Eisenring, J. Teich, "Domain-Specific interface generation from dataflow specifications", In Proc. of CODES 98, pp 43-47, Seattle Washington, March 15-18, 1998.

[35] M. Eisenring, J. Teich, "Interfacing Hardware and Software", In R.W. Hartenstein, A. Keevallik, editors, Field-Programmable Logic and Applications, volume 1482 in Lecture Notes in Computer Science, pp. 520-524, Springer-Verlag, 1998.

[36] J. S. Sun, R. W. Brodersen, "Design of System Interface Modules", In Proc. of the International Conference on Computer-Aided Design (ICCAD), pp 478-481, 1992.

[37] B. Lin, S. Vercauteren, "Synthesis of concurrent system interface modules with automatic protocol conversion generation", In International Conference on Computer-Aided Design (ICCAD), pp. 101-108, November, 1994.

[38] S. Narayan, D.D. Gajski, "Interfacing Incompatible protocols using interface process generation", In Proc. of DAC'95, pp. 468-473, 1995.

[39] J. A. Rowson, A. Sangiovanni-Vincentelli, "Interface-based design", In Proc. of DAC'97, pp. 178-183, 1997.

[40] J. Smith, G. De Micheli, "Automated Composition of Hardware Components", In Proc. of DAC'98, pp. 14-19, San Francisco, California, 1998.

[41] R. Passerone, J. A. Rowson, A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols", In Proc. of DAC'98, pp. 8-13, San Francisco, California, 1998.

[42] A.P. Ravn, J. Staunstrup, "Interface Models", In Proc. of CODES'94, pp. 157-164, 1994.

[43] Z. Chaochen, C.A.R. Hoare, A.P. Ravn, "A calculus of durations", Information Proc. Letters, 40(5), Dec., 1991.

[44] W.D. Tiedemann, "An approach to multi-paradigm controller synthesis from timing diagram specifications", In Proc. of European Design Automation Conference, pp 522-527, 1992.

[45] W. Grass, S. Lenk, "Formal Specification of communication units using timing diagrams", In Designers Track of DATE'98, pp. 273-277, Paris, France, 1998.

[46] W. Grass, S. Lenk, C. Sontheim, “Design of Control Dominated Hardware Based on Formal Methods”, In Proc. of EuroMicro’98, pp. 357-363, Västerås, Sweden, Aug. 25-27, 1998.

[47] A. Bharati, V. Chaitanya, R. Sangal, “Natural Language Processing - A Paninian Perspective”, Prentice-Hall of India Private Limited, 1995, ISBN 81-203-0921-9.

[48] J.S. Gero, E. Tyugu, Editors, “Formal Design Methods for CAD”, Elsevier Science B.V., 1994, ISBN 0-444-81970-3.

[49] James L. Hein, “Theory of Computation - An Introduction”, Jones and Bartlett Publishers, Inc., 1996, ISBN 0-86720-497-4.

[50] Kleene, S.C., “Representation of events by nerve nets”, In “Automata Studies”, ed. C.E. Shannon and J McCarthy, Princeton University Press, Princeton, N.J., 1956, pp. 3-42.

[51] Rabin, M.O., and D. Scott, “Finite Automata and their decision problem”, In IBM Journal of Research and Development 3 (1959), pp. 114-125.

[52] Myhill, J., “Finite automata and the representation of events”, WADD TR-57-624, Wright Patterson AFB, Ohio, 1957, pp. 112-137.

[53] Nerode, A., “Linear automaton transformations”, In Proc. of the American Mathematical Society 9 (1958), pp. 541-544.

[54] S. C. Johnson, “YACC, Yet another compiler compiler”, Computing Science Tech. Rep. 32, AT&T Bell Lab., Murray Hill, 1975.

[55] M. E. Lesk, “Lex-A lexical analyzer generator”, Computing Science Tech. Rep. 39, AT&T Bell Lab., Murray Hill, 1975.

[56] A.V. Aho, R. Sethi, J.D. Ullman, “Compilers, Principles, Techniques and Tools”, Addison-Wesley Publishing Company, 1986.

[57] Alfred V. Aho, Jeffrey D. Ullman (Contributor), “Foundations of Computer Science (Principles of Computer Science Series)”, C Edition edition, W H Freeman & Co, Jan., 1995, ISBN 0716782847.

[58] R.H.J. Bloks, “A Grammar Based Approach towards the Automatic Implementation of Data Communication Protocols in Hardware”, Ph. D. thesis, Eindhoven University of Technology, Sept. 1993.

[59] A. Seawright, F. Brewer, “Synthesis from Production-Based Specifications”, Proc. of the 29th DAC, pp 194-199, Anaheim, June 1992.

[60] A. Seawright, F. Brewer, “High Level Symbolic Construction Techniques for High Performance Sequential Synthesis”, Proc. of the 30th DAC, pp 424-428, Dallas, June 1993.

[61] A. Seawright, F. Brewer, “Clairvoyant: A Synthesis System for Production-Based Specification,” IEEE Trans. on VLSI Systems, vol. 2, pp 172-185, June 1994.

[62] A. Seawright, Grammar-Based Specifications and Synthesis for Synchronous Digital

Hardware Design, Ph. D. Thesis, Univ. of California, Santa Barbara, June 1994.

[63] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, and J. Buck, “A System for Compiling and Debugging Structured Data Processing Controllers”, Proc. of Euro-DAC`96, Geneva, Switzerland, September 1996.

[64] The Synopsys Protocol Compiler’s Reference Manual

[65] G. Economakos, G. Papakonstantinou, P. Tsanakas, “AGENDA: An Attribute Grammar Driven Environment for the Design Automation of Digital Systems”, In Proc. of DATE’98, pp. 933-934, Paris, France, Feb. 23-26, 1998.

[66] G. Economakos, I. Poulakis, G. Papakonstantinou, P. Tsanakas, “An Attribute Grammar Based Interactive High-Level Synthesis Tool”, In Proc. of International Workshop on IP Based Synthesis and System Design, Grenoble, France, December, 1998.

[67] C.N. Coelho, G. deMicheli, “Analysis and synthesis of concurrent digital circuits using control-flow expressions”, In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 15, No. 8, pp. 855-876, Aug. 1996.

[68] M. O’Nils, J. Öberg, A. Jantsch, “Grammar Based Modelling and Synthesis of Device Drivers and Bus Interfaces”, (poster paper), In Proc. of EuroMicro’98, Vol. I, pp. 55-58, Västerås, Sweden, Aug. 25-27, 1998.

[69] M. O’Nils, A. Jantsch, “Operating System Sensitive Device Driver Synthesis from Implementation Independent Protocol Specification”, In Proc. of DATE’99, Munich, Germany, 1999.

[70] M. O’Nils and A. Jantsch, “Synthesis of DMA Controllers from Architecture Independent Descriptions of HW/SW Communication Protocols”, In Proc. of the Twelfth International Conference on VLSI Design, Jan. 1999.

[71] “UTOPIA, An ATM-PHY Interface Specification, Level 1, Version 2.01”, The ATM Forum, March 21, 1994.

[72] Andrew Seawright, informal email conversation, Oct. 9th, 1998.

[73] G. De Micheli, “Synthesis and Optimization of Digital Circuits”, McGraw-Hill, Inc., 1994, ISBN 0-07-016333-2.

[74] Z. Kohavi, “Switching and Finite Automata Theory”, McGraw-Hill, Inc., 1978.

[75] Anshul Kumar and P.C.P. Bhatt, “A structured Language for Digital System Design”, *Proc. International Symposium on Computer Architecture*, La Boule, France, 1979.

[76] Manu Lauria, Shashi Kumar, Anshul Kumar, “A partitioning scheme for multiple PLA based Control part Synthesis”, *Proc. of Intl. Conference on VLSI Design*, Bangalore, 1992.

[77] N. Dutt, T. Hadley, and D. Gajski, “An Intermediate Representation for Behavioural Synthesis”, Proceedings of Design Automation Conference, pp. 14-19, 1990.

[78] Ahmed A. Jerraya and K. O’Brien, “SOLAR: An Intermediate Format for System-

Level Modelling and Synthesis", Codesign: Computer-aided Software/Hardware Engineering, IEEE Press, Jerzy Rozenblit and Klaus Buchenrieder, 7, pp. 145-175, 1995.

[79] Tarek Ben Ismail and Ahmed Amine Jerraya, "Synthesis Steps and Design Models for Codesign", IEEE Computer, pp. 44 - 52, February 1995.

[80] S.C.Lee. Digital Circuits an Logic Design. Prentice-Hall, Inc, 1976, ISBN 0-13-212225-1.

[81] N.Fenton, G.Hill. Systems Construction and Analysis: A Mathematical and Logical Framework. McGraw-Hill International Limited, 1993, ISBN 0-07-707431-9.

[82] M. Chiodo, P. Giusto, A. Jurecska, A. Sangiovanni-Vincentelli, L. Lavagno. Hardware-software codesign of embedded systems. IEEE Micro vol. 14, no. 4, pp. 26-36, Aug. 1994.

[83] The COSYMA system. Technische Universität Braunschweig. "http://www.ida.ing.tu-bs.de/projects/cosyma/"

[84] P. Ellervee, S. Kumar, A. Jantsch, A. Hemani, B. Svantesson, J. Öberg, I. Sander. IRSYD - An Internal representation for System Description (Version 0.1). Internal Report TRITA-ESD-1997-10, Royal Institute of Technology (KTH), Dept. of Electronics, ESDLab, Stockholm, Sweden, 1997.

[85] P. Ellervee, S. Kumar, A. Jantsch, B. Svantesson, T. Meincke. "IRSYD: An Internal Representation for Heterogeneous Embedded Systems". In the Proc. of the 16th NorChip Conference, pp. 214-221, November 9-10, 1998, Lund, Sweden .

[86] P. Ellervee, A. Kumar, B. Svantesson, A. Hemani, "Internal Representation and Behavioural Synthesis of Control Dominated Applications", In Proc. of the 14th NorChip conference, pp. 142-149, Nov. 1996.

[87] "B-ISDN Operation and Maintenance interface principles and functions", ITU-T Recommendation I.610.

[88] "Intel 8251A Programmable Communication Interface", Datasheet, Intel Corporation, 1981.

[89] "Hyper 2.0 - Documentation and Reference Manual", University of California at Berkeley, 1993.

[90] J. Vanhoof, K.V. Rompaey, I. Bolsens, G. Goossens, H. De Man, "High-Level Synthesis for Real-Time Digital Signal Processing", Kluwer Academic Publishers, Netherlands, 1993.

[91] J. Öberg, "Espresso-based Estimation Methods in the ProGram Compiler", TRITA-ESD-1999-02, Technical Report, ESDLab, Dept. of Electronics, Kungliga Tekniska Högskolan, Stockholm, Sweden, 1999.

[92] A. Hemani, J. Öberg, A. Kumar Deb, D. Lindqvist, B. Fjellborg, "System Level Prototyping of DSP ASICs using grammar based approach", Accepted for inclusion in the Proc. of Rapid System Prototyping (RSP'99).

[93] J. Öberg, A. Kumar, A. Jantsch, “An Object-Oriented Concept for Intelligent Library Functions”, In Proc. of the VLSI Design‘98 Conference, pp. 355-358, Chennai, India, Jan. 4-7, 1998.

[94] J. Öberg, A. Jantsch, A. Hemani, “Validation of Interface Protocols Using Grammar-based Models”, In the Proc. of the IEEE International High Level Design Validation and Test Workshop (HLDVT’98), pp. 40-46, La Jolla, California, Nov. 12-14, 1998.

[95] A. Jantsch, J. Öberg, A. Hemani, “Is there a Niche for a General Purpose Protocol Processor?”, In Proc. of NorChip’98, pp. 93-100, Lund, Sweden, Nov. 9-10, 1998.

[96] J. Öberg, P. Ellervee, A. Hemani, “Grammar-based Modelling of Clock Protocols for Low-Power Implementation: A Case Study”, In Proc. of NorChip-98, pp. 144-153, Lund, Sweden, Nov. 9-10, 1998.

[97] G. Borrielo, R. H. Katz, “Synthesis and Optimization of Interface Transducer Logic”, In Proc. of the ICCAD’87, 1987.

[98] G. Borrielo, “A New Interface Specification Methodology and its Applications to Transducer Synthesis”, PhD thesis, University of California, Berkeley, May 1988.

[99] G. Borrielo, “Specification and Synthesis of Interface Logic”, In R. Camposano, W. Wolf, Editors, “High-Level VLSI Synthesis”, Kluwer Academic Publishers, Boston, 1991.

[100] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[101] M. Fujita, H. Fujisawa, “Specification, Verification and Synthesis of Control Circuits with Propositional Temporal Logic”, In Proc. of CHDL, pp. 265-279, 1989.

[102] H. Nakamura, M. Fujita, S. Kono, H. Tanaka, “Temporal Logic Based Fast Verification System Using Cover Expressions”, In Proc. of VLSI’87, pp. 99-111, North-Holland, 1987.

[103] J. Nestor, D. Thomas, “Behavioural Synthesis with Interfaces”, In Proc. of the International Conference on Computer-Aided Design (ICCAD), pp. 112-115, 1986.

[104] J. S. Sun, M. B. Srivastava, R. W. Brodersen, “SIERA: A CAD Environment for Real-time Systems”, In 3rd Physical Design Workshop, May, 1991.

[105] J. Harangozó, “An approach to describing a data link level protocol with a formal language”, In Proc. of the 5th Symposium on Data Communications, Snowbird Utah, Sept. 27-29, 1977.

[106] G. de Jong, B. Lin, “A communicating Petri net model for the design of concurrent asynchronous modules”, In Proc. of DAC’94, June 1994.

A. Appendix

This chapter contains a summary of the author's contribution to the appended papers.

I. Grammar-based Hardware Synthesis of Data Communication Protocols

J. Öberg, A. Kumar, A. Hemani, In Proc. of ISSS-96, pp. 14-19, (1996)

Invented the concept of design space exploration of port sizes, most of the language notation that differs ProGram from YACC, performed all experiments and wrote most of the paper.

II. Comparing Conventional HLS with Grammar-Based Hardware Synthesis: A Case Study

J. Öberg, P. Ellervee, A. Kumar, A. Hemani, In Proc. of NorChip-97, pp. 52-59, (1997)

Performed the HLS and ProGram experiments and wrote most of the paper.

III. Scheduling of Outputs in Grammar-based Hardware Synthesis of Data Communication Protocols

J. Öberg, A. Kumar, A. Hemani, In Proc. of DATE-98, pp. 596-603, (1998).

Invented the synthesis algorithms, performed all experiments and wrote most of the paper.

IV. Specification of Exception Handling in Grammar-based Hardware Synthesis

J. Öberg, A. Kumar, A. Hemani, (poster paper), In Proc. of EuroMicro'98, Vol. I, pp. 38-41, (1998).

Invented the exception handling notation and wrote most of the paper.

V. Synthesis of Exception Handling in Grammar-based Hardware Synthesis

J. Öberg, A. Kumar, A. Hemani, In Proc. of APCHDL-98, pp. 135-140, (1998).

Invented the synthesis algorithms, performed all experiments and wrote most of the paper.

VI. Specification and Synthesis of Exception Handling in Grammar-based Hardware Synthesis

J. Öberg, A. Kumar, A. Hemani, S. Kumar, Accepted for publication in the Journal of Electrical Engineering and Information Science, Korea.

Invented the synthesis algorithms, performed all experiments and wrote most of the paper.

VII. Transition Graph representation of FSMs and its application to State Minimization in the ProGram Compiler

J. Öberg, P. Ellervee, S. Kumar, TRITA-ESD-1998-01, parts of the paper will be rewritten and submitted either as a Transactions brief or to Electronic Letters.

The idea evolved during discussions among all three authors. Wrote most of the paper.

VIII. Grammar-Based Hardware Synthesis from Port Size Independent Specifications

J. Öberg, A. Kumar, A. Hemani, Submitted to IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

Invented the methodology, performed all the experiments and wrote most of the paper.

Other publications from ESD Lab

Licentiate theses:

- ***Design and performance evaluation of asynchronous micropipline circuits for digital radio***, Bengt Oelmann, ISRN KTH/ESD/R--96/17--SE, 1996.
- ***Hardware/Software Partitioning of Embedded Computer Systems***, Mattias O'Nils, ISRN KTH/ESD/R--96/17--SE, 1996.
- ***An Adaptable Environment for Improved High-Level Synthesis***, Johnny Öberg, ISRN KTH/ESD/R--96/14--SE, 1996
- ***ASIC Implementable Synchronization and Detection Methods for Direct Sequence Spread Spectrum Wideband Radio Receivers***, Henrik Olson, ISRN KTH/ESD/R--97/11--SE, 1997.

Doctoral theses:

- ***Design Techniques and Structures for ATM Switches***, Tawfik Lazraq, ISBN 91-7170-703-4, 1996.
- ***Switched-Current Circuits: from Building Blocks to Mixed Analog-Digital Systems***, Bengt Jonsson, ISRN KTH/ESD/AVH--99/1--SE, 1999.

Johnny Öberg

ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols

KTH 1999