UNIVERSITY OF CALIFORNIA,
IRVINE

Modeling Flow for Automated System Design and Exploration

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Andreas Gerstlauer

Dissertation Committee:
Professor Daniel D. Gajski, Chair
Professor Rainer Dömer
Professor Michael Franz

2004

© 2004 Andreas Gerstlauer

The dissertation of Andreas Gerstlauer
is approved and is acceptable in quality
and form for publication on microfilm:

____

____

____

Committee Chair

University of California, Irvine
2004

To my dad.

Contents

List of Figures viii

List of Listings x

List of Tables xii

List of Acronyms xiii

Acknowledgments xvi

Curriculum Vitae xviii

Abstract of the Dissertation xxiii

1 Introduction 1

1.1 Design and Synthesis 2
1.2 System Design 4
1.3 Abstraction Levels and Design Models 5
1.4 Design Language 6
1.5 Problem Definition 7
1.6 Dissertation Overview 8
1.7 Related Work 10
1.7.1 Design Methodologies 10
1.7.2 Computation Synthesis 10
1.7.3 Communication Synthesis 12
1.7.4 Design Environments 12

2 Modeling Flow 14

2.1 Overview 14
2.1.1 Design Flow 16
2.1.2 Modeling 17
2.2 Design Models 20
2.2.1 Specification 22
2.2.2 Architecture 23

2.2.3 Communication . . . 26
2.2.4 Implementation . . . 28
2.3 Design Methodology . . . 32
2.4 Design Process . . . 34
2.5 Design Environment . . . 35
2.5.1 Modeling . . . 36
2.5.2 Refinement . . . 37
2.5.3 Exploration . . . 37
2.5.4 Synthesis . . . 37
2.6 Summary . . . 38

3 System Specification **40**
3.1 Modeling Guidelines . . . 40
3.1.1 Computation . . . 41
3.1.2 Communication . . . 43
3.2 Modeling Style . . . 44
3.2.1 Computation . . . 46
3.2.2 Communication . . . 48
3.3 Design Example . . . 49
3.4 Summary . . . 51

4 Computation Design **52**
4.1 Overview . . . 52
4.2 Partitioning . . . 54
4.2.1 Processing Element Layer . . . 54
4.2.2 Memory Layer . . . 63
4.3 Scheduling . . . 75
4.3.1 Static scheduling . . . 75
4.3.2 RTOS Layer . . . 77
4.4 Summary . . . 87

5 Communication Design **90**
5.1 Overview . . . 90
5.1.1 SoC Communication . . . 92
5.1.2 Communication Layers . . . 94
5.2 Network Design . . . 99
5.2.1 Application Layer . . . 99
5.2.2 Presentation Layer . . . 102
5.2.3 Session Layer . . . 105
5.2.4 Transport Layer . . . 108
5.2.5 Network Layer . . . 109
5.3 Link Design . . . 112
5.3.1 Link Layer . . . 112
5.3.2 Stream Layer . . . 116
5.3.3 Media Access Layer . . . 124

5.3.4 Protocol Layer . . . 134
5.4 Summary . . . 143

6 Backend **146**
6.1 Hardware Design . . . 146
6.1.1 Overview . . . 147
6.1.2 Superstate Model . . . 149
6.1.3 Behavioral RTL . . . 151
6.1.4 Structural RTL . . . 152
6.2 Software Design . . . 156
6.2.1 Overview . . . 158
6.2.2 C Model . . . 162
6.2.3 Instruction Set Simulation Model . . . 164
6.3 Summary . . . 166

7 Design Environment **168**
7.1 Overview . . . 168
7.1.1 Simulation . . . 170
7.1.2 Profiling . . . 170
7.1.3 Refinement . . . 170
7.1.4 Synthesis . . . 171
7.1.5 User Interface . . . 171
7.2 Specification Capture . . . 173
7.2.1 Modeling . . . 173
7.2.2 Simulation . . . 174
7.3 Profiling and Estimation . . . 176
7.3.1 Profiling Flow . . . 176
7.3.2 Metrics . . . 177
7.3.3 Visualization . . . 180
7.4 Databases . . . 183
7.4.1 Database Format . . . 183
7.4.2 Allocation and Selection . . . 185
7.5 Computation Design . . . 186
7.5.1 Partitioning . . . 187
7.5.2 Scheduling . . . 188
7.6 Communication Design . . . 188
7.6.1 Network Design . . . 189
7.6.2 Link Design . . . 190
7.7 Backend . . . 190
7.7.1 Hardware Design . . . 191
7.7.2 Software Design . . . 192
7.8 Summary . . . 193

8 Experimental Results **194**
8.1 Overview 195
8.1.1 Modeling and Simulation 195
8.1.2 System Design 196
8.2 Vocoder System 198
8.2.1 Modeling and Simulation 199
8.2.2 PE Modeling 205
8.2.3 OS Modeling 206
8.2.4 Communication Modeling 208
8.3 JPEG Encoder Subsystem 210
8.3.1 Modeling and Simulation 210
8.3.2 PE Modeling 214
8.4 Baseband System 216
8.5 Summary 220

9 Summary and Conclusions **222**

Bibliography **226**

List of Figures

1.1 Y-Chart. 2
1.2 System design. 5

2.1 System-level design. 15
2.2 System design flow. 16
2.3 Abstraction levels and models. 21
2.4 Specification model example. 23
2.5 Architecture model example. 25
2.6 Communication model example. 28
2.7 Behavioral implementation model example. 30
2.8 Structural implementation model example. 31
2.9 System design methodology. 32
2.10 Design process. 35
2.11 SoC Design Environment (SCE). 36

3.1 Specification model top-level structure. 45
3.2 System design example specification model. 50

4.1 Computation design flow. 53
4.2 Behavior partitioning. 55
4.3 Behavior partitioning refinement. 56
4.4 PE model. 64
4.5 Distributed variable partitioning. 66
4.6 Shared memory variable partitioning. 70
4.7 Partitioned model. 74
4.8 Static scheduling refinement. 76
4.9 Scheduled model. 78
4.10 RTOS modeling layers. 79
4.11 Model refinement example. 81
4.12 Dynamic scheduling implementation. 85
4.13 RTOS model simulation traces. 86
4.14 Architecture model. 88

5.1 Communication design flow. 91

5.2 Communication architecture example. 97
5.3 Communication model refinement. 98
5.4 Application model. 101
5.5 Session model. 104
5.6 Transport model. 107
5.7 Link model. 111
5.8 Stream model. 115
5.9 Media access model. 122
5.10 Protocol model. 133
5.11 Communication model. 141

6.1 Hardware design flow. 147
6.2 Hardware refinement. 148
6.3 Software design flow. 158
6.4 Software implementation layers. 160
6.5 Software refinement. 161

7.1 SCE tool flow and architecture. 169
7.2 SCE graphical user interface (GUI). 172
7.3 Model capture and browsing. 174
7.4 Simulation output and traces. 175
7.5 Profiling and estimation flow. 177
7.6 Weight table editor. 179
7.7 Visualization of design metrics. 181
7.8 Connectivity display. 182
7.9 Design quality dialog. 183
7.10 Database browser and component allocation. 186
7.11 Behavior and variable mapping. 187
7.12 Scheduling dialog. 188
7.13 Channel routing. 189
7.14 Link parameter dialog. 190
7.15 RTL scheduling and binding dialog. 191

8.1 Vocoder modeling results. 200
8.2 Vocoder simulation results. 201
8.3 Vocoder PE exploration. 206
8.4 JPEG encoder modeling results. 211
8.5 JPEG encoder simulation results. 212
8.6 Baseband implementation detail added during model refinement 218
8.7 Baseband simulation times. 218
8.8 Baseband simulated delays. 218

List of Listings

3.1 Specification model top-level code. 45

4.1 Specification model. 57
4.2 Grouped model. 58
4.3 Handshaking synchronization behaviors. 59
4.4 PE model. 60
4.5 Timing refinement. 62
4.6 *DSP* PE distributed variable refinement. 67
4.7 *SI* PE distributed variable refinement. 67
4.8 Top-level distributed variable refinement. 68
4.9 Message-passing refinement. 69
4.10 Top-level shared memory refinement. 71
4.11 Shared memory behavioral model. 71
4.12 *DMA* PE shared memory refinement. 72
4.13 *ColdFire* PE shared memory refinement. 72
4.14 Static scheduling refinement. 76
4.15 Interface of the RTOS model. 80
4.16 Task modeling. 82
4.17 Task creation. 83
4.18 Synchronization refinement. 84

5.1 Communication model PE refinement. 98
5.2 Presentation layer. 102
5.3 Shared memory presentation model. 103
5.4 Memory access presentation layer. 105
5.5 Transducer model. 110
5.6 Link layer. 113
5.7 Shared memory link model. 114
5.8 Data stream layer. 117
5.9 Memory stream layer. 118
5.10 Control stream translator. 119
5.11 Stream layer for non-OS processor. 120
5.12 Stream layer for processor with OS. 121

5.13 Interrupt handler task for control streams. 123
5.14 Media access layer data transactions. 126
5.15 Media access layer for memory access in the master. 127
5.16 Media access layer for memory slave. 128
5.17 Media access layer arbitration. 129
5.18 PE hardware abstraction layer (HAL). 130
5.19 PE hardware abstraction layer (HAL) with interrupt sharing. 131
5.20 Media access layer slave interrupt polling. 131
5.21 Data transfer protocol layer. 136
5.22 Arbitration protocol layer. 137
5.23 Interrupt protocol layer. 138
5.24 Processor core hardware model. 139
5.25 Bus-functional processor model. 140

6.1 Custom hardware behavioral model. 150
6.2 Custom hardware SFSMD model. 151
6.3 Custom hardware FSMD model. 153
6.4 RTL netlist custom hardware model. 154
6.5 Custom hardware controller. 155
6.6 Custom hardware datapath. 156
6.7 Custom hardware controller behavior hierarchy. 157
6.8 Application software C code. 163
6.9 Software C model. 164
6.10 Instruction set simulation (ISS) model. 165

List of Tables

2.1 System-level design steps. 18

4.1 Computation design steps. 89

5.1 Communication layers. 95
5.2 Communication design steps. 144

6.1 Accellera RTL styles. 152
6.2 Backend design steps. 166

8.1 Baseband computation design parameters. 196
8.2 Baseband communication design parameters. 197
8.3 Vocoder modeling results. 200
8.4 Vocoder simulation results. 201
8.5 Vocoder OS simulation results. 207
8.6 Vocoder simulated communication delays. 209
8.7 JPEG encoder modeling results. 211
8.8 JPEG encoder simulation results. 212
8.9 JPEG encoder PE exploration. 215
8.10 Baseband system modeling results. 217

List of Acronyms

ALU Arithmetic Logic Unit. A block of combinatorial logic that performs arithmetic and logic operations.

Behavior An entity that encapsulates and describes computation or functionality in the form of an algorithm.

CAD Computer Aided Design. Design of systems with the help of and assisted by computer programs, i.e. software tools.

CE Communication Element. A system component that is part of the communication architecture for transmission of data between PEs, e.g. a transducer, an arbiter, or an interrupt controller.

Channel An entity that encapsulates and describes communication between two or more partners in an abstract manner.

FSM Finite State Machine. A model that describes a machine as a set of states, a set of transitions between states, and a set of actions associated with each state or transition.

FSMD Finite State Machine with Datapath. An FSM in which each state contains a set of expressions over variables.

GUI Graphical User Interface. A graphical interface of a computer program that allows visual entry of commands and display of results.

HAL Hardware Abstraction Layer. The lowest layer of software in a processor implementing the interface to the processor's hardware.

HDL Hardware Description Language. A language for describing and modeling blocks of hardware.

HW Hardware. The tangible part of a computer system that is physically implemented.

IP Intellectual Property. An IP component is a pre-designed system component that is stored in the component database.

ITRS International Technology Roadmap for Semiconductors. Roadmap published by SEMATECH identifying and predicting the technological challenges and needs facing the semiconductor industry over the next 15 years.

MoC Model of Computation. A set of rules that define the interaction and composition of components.

OS Operating System. A piece of software that manages and controls functionality in a computer system.

PE Processing Element. A system component that performs computation (data processing), e.g. a software processor, a custom hardware component, or an IP.

RTL Register-Transfer Level. A level of abstraction at which computation is described as transfers of data between storage units (registers) where each transfer involves processing and manipulation of data.

RTOS Real-Time Operating System. An operating system that provides predictable timing and timing guarantees.

SEMATECH Global consortium of leading semiconductor manufacturers.

SCE SoC Environment. Tool set for automated, computer-aided design of SoC and computer systems in general.

SLDL System-Level Design Language. A language for describing complete, heterogeneous, mixed hardware/software computer systems at high levels of abstraction.

SoC System-On-Chip. A complete computer system implemented on a single chip or die.

TLM Transaction Level Model. A model of a system in which communication is abstracted into channels and described as transactions at a level above pins and wires.

Transducer A CE that translates between incompatible communication protocols and connects two or more incompatible communication media, e.g. a bus bridge.

VHDL VHSIC Hardware Description Language. An HDL commonly used for hardware design at RTL and logic levels.

VHSIC Very High Speed Integrated Circuit.

Acknowledgments

The work in this thesis was to a large extent shaped and influenced by all the members of the SpecC/SCE group in the Center for Embedded Computer Systems (CECS) at UC Irvine. First and foremost, this work certainly would not have been possible without the leadership, direction and vision of my advisor, Prof. Daniel Gajski, who started the project and contributed many of the initial ideas. I would like to thank him for his support, guidance—both academically and otherwise—and patience in dealing with my stubbornness over the years.

The foundation of this work was laid by the fathers of the SpecC language, Prof. Rainer Dömer and Prof. Jianwen Zhu. In particular, I would like to thank Prof. Dömer for serving on my committee, for all his support throughout this project and for many fruitful discussions that always proofed helpful in bringing me back on track. In addition, I would also like to thank Prof. Michael Franz for serving on my committee and for his valuable comments on improving this dissertation.

Many details of this thesis were determined in discussion with my colleagues and co-authors in the team developing the SCE system: Samar Abdi, Lukai Cai, Junyu Peng, Dongwan Shin and Haobo Yu. Furthermore, I would like to thank Alexander Gluhak, David Berner, Martin von Weymarn and Shuqing Zhao who as visiting researchers or former students helped tremendously in the development of the GUI and of several design examples.

Finally, I want to thank all the former and current members, students and office-mates in CECS and in the computing support group of the department who provided often desperately needed technical, administrative or research support, glimpses into a world outside the box and a productive and fun working environment in general. Furthermore, I am grateful for the help and support of the CECS staff who are instrumental in running the center so friendly, smoothly and efficiently.

On a personal level, life in Irvine would have been so much less bearable without all the friendships made here over the years. In addition to all the people mentioned above, this includes

the many friends in the graduate student body and the international community at UCI. Especially Sumit Gupta has been both a helpful colleague and an invaluable friend, roommate and partner in many activities throughout our journey together to seek those elusive Ph.D. degrees.

Curriculum Vitae

Andreas Gerstlauer

Education

2004	**Ph.D.**, Information & Computer Science, University of California, Irvine
1998	**M.S.**, Information & Computer Science, University of California, Irvine
1997	**Dipl.-Ing. (M.S.)**, Electrical Engineering, University of Stuttgart, Germany

Experience

1998-2004	Graduate Research Assistant	Center for Embedded Computer Systems, University of California, Irvine
1997-1998	Teaching Assistant	Information & Computer Science, University of California, Irvine
1996-1997	Graduate Research Assistant	Integrated Systems Engineering, University of Stuttgart, Germany
1995-1996	Research Assistant	Institute for Microelectronics Stuttgart, University of Stuttgart, Germany
1993-1994	Research Assistant	Communication Networks & Computer Engineering, University of Stuttgart, Germany
1989-1997	Senior Software Engineer	Embedded Software Development, Ehrler Prüftechnik, Germany
1994	Summer Intern	Böblingen Instruments Division, Hewlett-Packard GmbH, Germany

Publications

Books

- A. Gerstlauer, R. Dömer, J. Peng, D.D. Gajski, "System Design: A Practical Guide with SpecC," Kluwer Academic Publishers, 2001.
- D.D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC: Specification Language and Methodology, Japanese Edition," CQ Publishing, Japan, 2000.
- D.D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, "SpecC: Specification Language and Methodology," Kluwer Academic Publishers, 2000.

Book Chapters

- A. Gerstlauer, H. Yu, D.D. Gajski, "RTOS Modeling for System-Level Design," in *Embedded Software for SoC*, edited by A.A. Jerraya, S. Yoo, N. When, D. Verkest, Kluwer Academic Publishers, 2003.
- A. Rettberg, F. Rammig, A. Gerstlauer, D. Gajski, W. Hardt, B. Kleinjohann, "The Specification Language SpecC within the PARADISE Design Environment" in *Architecture and Design of Distributed Embedded Systems*, edited by B. Kleinjohann, Kluwer Academic Publishers, 2001.

Conference Papers

- L. Cai, A. Gerstlauer, D. Gajski, "Retargetable Profiling for Rapid, Early System-Level Design Space Exploration," *Design Automation Conference (DAC)*, June 2004.
- A. Gerstlauer, H. Yu, D. Gajski, "RTOS Modeling for System-Level Design," *Design, Automation & Test in Europe (DATE)*, March 2003.
- A. Gerstlauer, D.D. Gajski, "System-Level Abstraction Semantics," *International Symposium on System Synthesis (ISSS)*, October 2002.
- W. Mueller, R. Dömer, A. Gerstlauer "The Formal Execution Semantics of SpecC," *International Symposium on System Synthesis (ISSS)*, October 2002.

- S.B. Saoud, D.D. Gajski, A. Gerstlauer, "Co-design of Emulators for Power electric Processes Using SpecC Methodology," *Annual Conference of the IEEE Industrial Electronics Society*, November 2002.

- S.B. Saoud, D.D. Gajski, A. Gerstlauer, "Co-design of Embedded Controllers for Power Electronics and Electric Systems," *International Symposium on Intelligent Control*, October 2002.

- S.B. Saoud, D.D. Gajski, A. Gerstlauer, "Seamless approach for the design of control systems for Power Electronics and Electric Drives," *International Conference on Systems, Man and Cybernetics*, October 2002.

- R. Dömer, A. Gerstlauer, D. Gajski, "SpecC Methodology for High-Level Modeling," *IEEE/-DATC Electronic Design Processes Workshop*, April 2002.

- A. Gerstlauer, S. Zhao, D. Gajski, A. Horak, "SpecC System-Level Design Methodology Applied to the Design of a GSM Vocoder," *Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI)*, April 2000.

Professional Documents

- R. Dömer, A. Gerstlauer, D. Gajski, "SpecC Language Reference Manual, Version 2.0," SpecC Technology Open Consortium (STOC), December 2002.

- R. Dömer, A. Gerstlauer, D. Gajski, "SpecC Language Reference Manual, Version 1.0," SpecC Technology Open Consortium (STOC), March 2001.

Technical Reports

- L. Cai, A. Gerstlauer, D. Gajski, "Retargetable Profiling for Rapid, Early System-Level Design Space Exploration," UC Irvine, Technical Report CECS-TR-04-04, October 2003.

- D. Shin, A. Gerstlauer, R. Dömer, D. Gajski, "C-based Interactive RTL Design Methodology," UC Irvine, Technical Report CECS-TR-03-42, December 2003.

- S. Abdi, J. Peng, H. Yu, D. Shin, A. Gerstlauer, R. Dömer, D. Gajski, "System-on-Chip Environment (SCE Version 2.2.0 Beta): Tutorial," UC Irvine, Technical Report CECS-TR-03-41, July 2003.

- A. Gerstlauer, “Communication Abstractions for System-Level Design and Synthesis,” UC Irvine, Technical Report CECS-TR-03-30, October 2003.
- A. Gerstlauer, L. Cai, D. Shin, R. Dömer, D.D. Gajski, “System-On-Chip Component Models,” UC Irvine, Technical Report CECS-TR-03-26, August 2003.
- A. Gerstlauer, K. Ramineni, R. Dömer, D.D. Gajski, “System-On-Chip Specification Style Guide,” UC Irvine, Technical Report CECS-TR-03-21, June 2003.
- H. Yu, A. Gerstlauer, D. Gajski, “RTOS Scheduling in Transaction Level Models,” UC Irvine, Technical Report CECS-TR-03-12, March 2003.
- D. Gajski, J. Peng, A. Gerstlauer, H. Yu, D. Shin, “System Design Methodology and Tools,” UC Irvine, Technical Report CECS-TR-03-02, January 2003.
- S. Abdi, J. Peng, R. Dömer, D. Shin, A. Gerstlauer, A. Gluhak, L. Cai, Q. Xie, H. Yu, P. Zhang, D. Gajski, “System-On-Chip Environment (SCE): Tutorial,” UC Irvine, Technical Report CECS-TR-02-28, September 2002.
- A. Gerstlauer, D.D. Gajski, “System-Level Abstraction Semantics,” UC Irvine, Technical Report CECS-TR-02-17, July 2002.
- A. Gerstlauer, “SpecC Modeling Guidelines,” UC Irvine, Technical Report CECS-TR-02-16, April 2002.
- J. Peng, L. Cai, A. Gerstlauer, D.D. Gajski, “Interactive System Design Flow,” UC Irvine, Technical Report CECS-TR-02-15, April 2002.
- W. Mueller, R. Dömer, D.D. Gajski, “The Formal Execution Semantics of SpecC,” UC Irvine, Technical Report CECS-TR-02-04, January 2002.
- W. Mueller, R. Dömer, A. Gerstlauer, “The Formal Execution Semantics of SpecC,” UC Irvine, Technical Report ICS-TR-01-59, November 2001.
- A. Gerstlauer, “SpecC Modeling Guidelines,” UC Irvine, Technical Report ICS-TR-00-48, August 2000.
- A. Gerstlauer, “Communication Software Code Generation,” UC Irvine, Technical Report ICS-TR-00-46, August 2000.

- D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, “The SpecC Methodology,” UC Irvine, Technical Report ICS-TR-99-56, December 1999.

- L. Cai, J. Peng, C. Chang, A. Gerstlauer, H. Li, A. Selka, C. Siska, L. Sun, S. Zhao and D. Gajski, “Design of a JPEG Encoding System,” UC Irvine, Technical Report ICS-TR-99-54, November 1999.

- A. Gerstlauer, S. Zhao, D.D. Gajski, A. Horak, “Design of a GSM Vocoder using SpecC Methodology,” UC Irvine, Technical Report ICS-TR-99-11, March 1999.

- A. Gerstlauer, S. Zhao, D.D. Gajski, “VHDL+/SpecC Comparisons - A Case Study,” UC Irvine, Technical Report ICS-TR-98-23, May 1998.

Abstract of the Dissertation

Modeling Flow for Automated System Design and Exploration

by

Andreas Gerstlauer

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2004

Professor Daniel D. Gajski, Chair

Raising the level of abstraction is widely seen as the solution for closing the productivity gap in the design of embedded computer systems and systems-on-chip (SoCs). However, system design thus far suffers from a lack of rigorous, structured, and comprehensive design processes from abstract specification down to cycle-accurate implementation. In order to provide an automated design process, a well-defined design flow with clear and unambiguous abstraction levels, models, and transformations is required. The key to the success of this approach are properly defined design models. Arbitrary models without clear semantics do not enable synthesis and verification. In addition, synthesis requires clear definitions of the target architecture and a set of synthesis steps to transform the input model into the target model. In this dissertation, we define such input and target models, a set of design steps, design decisions, model transformation, and intermediate models that are necessary for an automated system design flow.

The design flow has been implemented in the form of an interactive system-on-chip design environment (SCE). Automating tedious and error-prone tasks while reaping human expertise and insight results in a flow with the necessary transparency, controllability, observability and productivity. As part of this work, architecture, organization, tool flow, design model management, interfaces, and databases of the design framework have been established. Models in the flow have

been defined such that they can be automatically generated and model refinement tools have been integrated into the design environment. Finally, a graphical user interface (GUI) has been developed that aids and steers the designer in the decision making process, provides visualization of design models, and supports interactive, graphical decision entry for each design step.

Experiments using the design environment prove the feasibility and effectiveness of a seamless, comprehensive, and automated design flow supporting rapid, early design space exploration. Results obtained by applying the design flow to several industrial-strength examples show the tradeoffs and benefits of intermediate models at each step and demonstrate that required productivity gains can be achieved while supporting a wide range of realistic target implementations.

Chapter 1

Introduction

As we enter the era of ubiquitous computing where we will find computer systems embedded into all parts of our environment, the demand for efficient processes and methodologies for designing and implementing such embedded computer systems is rising steadily. In general, systems grow in complexity and size, and technological improvements as predicted by the International Technology Roadmap for Semiconductors (ITRS) [60] will allow us to create systems-on-chip (SoCs) with billions of transistors on a single die. On the other hand, traditional design processes, methodologies and tools in place today will not enable us to design and implement such systems within reasonable cost and time limits as demanded by ever increasing market pressures.

In order to tackle and close this emerging productivity gap, both raising the level of abstraction for the design of complete systems and massive reuse of pre-designed, pre-existing intellectual property (IP) components have been proposed as solutions. With higher levels of abstraction, the number of design objects decreases exponentially. On the other hand, decisions made at such high levels can have huge impacts on the final quality of the implementation. Therefore, system design at high levels of abstraction allows the designers and tools to focus on the critical aspects without being overwhelmed by unnecessary details. Thus, large parts of the design space can be explored in short amounts of time.

The key to any of these approaches are well-defined design flows with clear and unambiguous abstraction levels, models and transformations. Only a rigorous, well-defined flow enables efficient design space exploration by both humans or tools. Furthermore, such a formalized definition of the design process allows application of design automation for synthesis and verification in order to achieve the required productivity gains. Finally, clearly defined design models are the basis for interoperability across tools within a complete design environment. So far, however, no

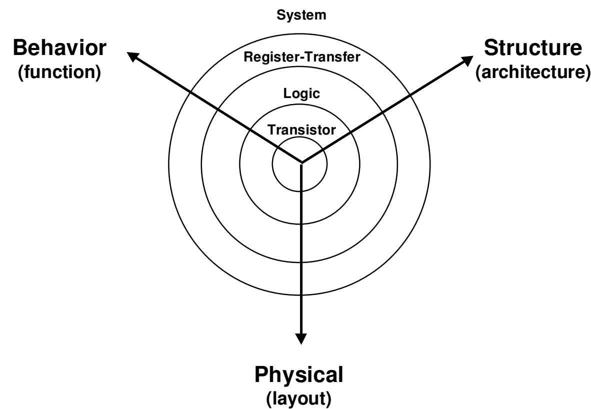


Figure 1.1: Y-Chart.

such well-defined, formalized modeling flows existed and system design has been done in an unstructured, ad-hoc manner based largely on designer experience.

1.1 Design and Synthesis

A classification of the process of designing computer systems in general is available through the Y-Chart [35]. The Y-Chart distinguishes between three different views or ways of describing a design:

(a) A *behavioral view* describes the functionality of the design in terms of abstract concepts, independent of any implementation details.

Building blocks of a behavioral description are abstract entities that do not represent physical components. Each block describes a piece of functionality that takes inputs, processes them and finishes after producing its output. In a behavioral view, such blocks are arranged hierarchically to model the control and data dependencies between them.

Parallelism in a behavioral description does not imply true concurrency in hardware. Again, behavioral blocks are abstract representations of algorithms that are free of implementation assumptions.

(b) A *structural view* describes the design architecture as a netlist of lower-level components and their connectivity.

Building blocks of a structural description represent real, physical objects. As such, each of the blocks is active all the time, constantly processing data. In a structural view, the system is

modeled as a set of non-terminating, concurrent processes representing the way the system is composed out of tangible lower-level components. Dependencies have to be modeled as part of the processes' functionality by inserting synchronization as needed.

Since the processes of a structural description represent real hardware, the parallel composition of the processes reflects the true concurrency available among the set of physical components on the chip or the board.

(c) A *physical view* describes the spatial layout of the lower-level components on the chip. A physical view describes the floorplan of how the components and their interconnect are placed and routed on the chip.

The Y-Chart defines several levels of granularity of design objects. With lower levels, the design process focuses on more and more detailed aspects of the system. At each level, the designer works with a specific set of objects. Objects at higher levels of abstraction are hierarchically composed of lower-level design objects. At the transistor level, a gate is composed as a netlist of transistors. At the logic level, a block of combinatorial logic is described as a set of boolean equations or as a netlist of gates. At the register-transfer level (RTL), a processors is described as a piece of sequential program code or as a netlist of RTL components. Finally, at the system level, a system is described as a hierarchical task graph or as a netlist of processors.

In general, design is the process of moving from a behavioral to a structural (and eventually physical) description at a certain level, implementing the desired functionality through an architecture of subcomponents. At the next lower level of abstraction, the subcomponents, in turn, are designed by moving from a behavioral description to a structural (and physical) description of the subcomponent. For example, at the system level, the designer will create a system architecture consisting of a set of processors connected through system busses that implements the desired system functionality. The processing element's functionality, in turn, is implemented by designing a microarchitecture of functional units, registers files, and so on for the processor at the register-transfer level.

A design flow can be bottom-up or top-down. In a bottom-up approach, design moves from the lowest level of abstraction up to the system level by assembling previously designed components such that the desired behavior is achieved at each level. In a top-down approach, design starts with a specification of the system behavior and moves down in the level of abstraction by mapping the desired behavior at each level onto a set of components and specifying the behavior of each component for the next level.

In order to automate the design process with CAD tools, the models and transformations of the design methodology must be formalized. Languages with special support to describe different views of the design at different levels of abstraction in a formal and efficient manner are needed. In addition to the application of formal methods for verification, an executable language allows validation through simulation of the models.

Once the models for the different design views at different abstraction levels are formally defined, CAD tools can automate parts of the design process. Specifically, the formalized process of deriving a structural description from a behavior description of the desired functionality is called *synthesis*. The synthesis processes at the highest levels of abstraction are:

(a) System synthesis
Given a specification of the system behavior, synthesize a system architecture consisting of processing elements and system busses that implements the desired functionality [36, 28].

(b) High-level/behavioral synthesis
Given a behavioral description of a processor, synthesize a microarchitecture implementation out of RTL components like functional units, register files, and so on [34, 76].

(c) Logic synthesis
Given a description of the functionality of an RTL component, synthesize a gate netlist that implements the combinatorial/sequential logic for the component [76, 33].

1.2 System Design

Based on the general overview provided in the previous section, system design is therefore defined as the process of moving from a behavioral description of the system to a structural description at the system and eventually register-transfer/instruction set level. As shown in Figure 1.2, system design starts with a system specification that describes the desired system functionality as a process graph. During system synthesis, the specification is mapped into a system architecture described as a netlist of system components. Finally, in a backend process, each of the components in the system is then brought down to a cycle-accurate implementation by synthesizing its behavior in hardware or software on top of the component's custom or fixed micro-architecture.

In general, the system design process is too complex to be completed in one single step. The gap between pure functionality and implementation is too big for either humans or tools to handle. When combined, the large number of interrelated tasks, concerns and design issues leads to

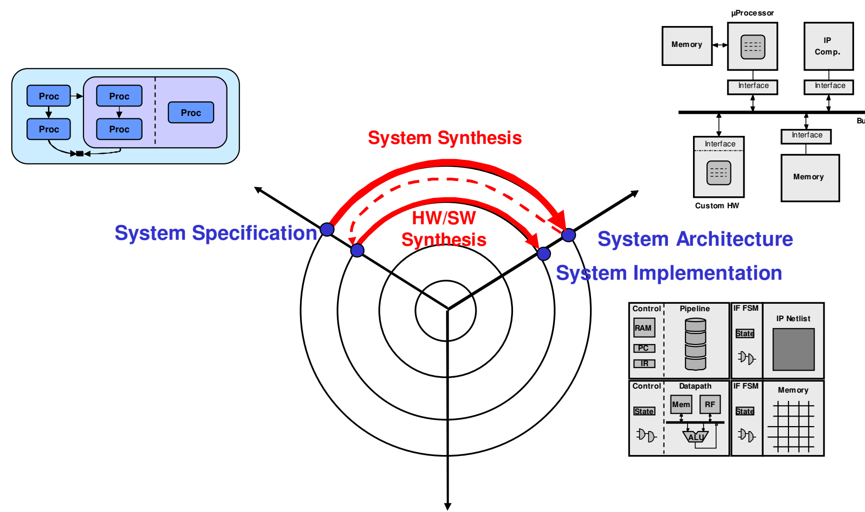


Figure 1.2: System design.

exponential growth of problem complexities that exceed any mental or machine capacity. In order to bridge the gap, the system design process therefore has to be broken down into smaller, manageable steps, breaking the overall problem into individual, contained subproblems that each can be solved by relatively simply algorithms at the expense of loss of global optimality.

1.3 Abstraction Levels and Design Models

With each step in the system design process, the system is gradually brought closer to an implementation by introducing more and more detail into the system description. Therefore, each step in the system design process corresponds to a certain abstraction level. An abstraction level is defined by the amount of implementation detail it contains where implementation detail is both structure (space) and order (time).

A model is then a description of the system at a certain level of abstraction, usually captured in a machine-readable form, i.e. in some language. The purpose of design models in between design steps is twofold:

(a) A model serves as a representation of the chosen implementation as the result of the previous design step. As such, it provides a description and abstraction of the reality of already designed physical aspects of the system. If details of certain implementation aspects are not yet available, the model can be augmented by back-annotating estimates about such characteristics.

(b) A model serves as the specification of the desired behavior for the following design steps. As such, it describes the functionality to be implemented without restricting how these parts will be implemented later. In addition to functionality, the model can include other constraints and requirements for this purpose.

Ideally, a model should clearly and unambiguously describe all of its aspects. It should allow to distinguish between parts that are already implemented and parts that are not. In the former case, the model should identify the structure of the implementation. In the latter case, the model should define what has to be implemented. Only if all of these aspects are properly formalized, human and tools will be able to process models efficiently and design automation for synthesis and verification becomes possible. Note that the use of specific languages to capture models (see Section 1.4) has a large influence on model ambiguity. If the language does not allow to distinguish among different concepts, ambiguities have to be resolved outside the language via meta-information (e.g. annotations), or other conventions and rules.

1.4 Design Language

In order to enable seamless exchange of models and in order to apply automated tools, design models have to be captured in machine-readable, formalized form, i.e. in some well-defined language. Furthermore, all design models within the flow should be executable for validation of design descriptions through simulation at any given point in the flow. Finally, it is desirable to capture all models throughout the flow in a single language. In contrast to heterogeneous approaches in which a system is specified in one language and then transformed into another, one common language removes the need for tedious translations throughout the design process. In addition, such homogeneous approaches allow to reuse a single testbench for validation of all models in the flow.

Due to these reasons, a number of system-level design languages (SLDLs) have been developed in recent years. Out of the large number of proposals, two languages are emerging as major candidates for standardization: SystemC [53, 79] and SpecC [38, 91]. Both candidates are C-based languages that are build on top of well-established C++ or C frameworks, respectively. SystemC is a library-based approach that implements system-level modeling concepts in the form a C++ class library [78, 56]. SpecC, on the other hand, extends standard ANSI-C with around ten new keywords for system modeling, thus creating a new language supported by a dedicated compiler and simulator [27, 77, 111].

Deriving new SLDLs from existing programming languages allows to leverage legacy knowledge and code in order to reduce the learning curve and reuse existing system models. In contrast to normal programming languages, SLDLs have to have support for modeling of special features required to describe complete systems consisting of a mix of hardware and software. On top of the software modeling capabilities offered by the underlying C language base, SLDLs therefore support the concepts of concurrency, synchronization, time, exception handling, hierarchy, and state machines needed for hardware and system modeling [25].

For our research, we chose SpecC as the SLDL for describing design models in the design flow [41]. SpecC was chosen due to its simplicity and completeness. By implementing all required system-level concepts in an orthogonal, clear and unambiguous manner, the language remains small yet allows to capture all possible system models from specification down to implementation. Note, however, that even though examples of models and corresponding code in this work are using SpecC syntax and semantics, the concepts presented here can be applied in an equivalent manner to any other SLDL that supports the required system-level modeling features.

1.5 Problem Definition

System design today is mostly done in an ad-hoc and unstructured manner, largely based on designer experience. Design decisions are made without proper exploration and evaluation of design alternatives. Based on informally specified decisions, systems are then implemented directly at the lowest, cycle-accurate level. At such low levels, significant exploration, evaluation and validation become infeasible. In order to achieve the required productivity gains, design processes and design space exploration therefore have to be moved to higher levels of abstraction.

High-level system design starts from an abstract, functional specification and gradually derives a system implementation from this specification. As outlined previously, the semantic gap between specification and implementation is too big to be completed in one step. In order to bridge the gap, the design process has to be broken down into smaller, manageable steps. The problem is therefore to define such a design flow of successive design steps. The number of steps and their order have to be chosen such that individual steps can be implemented by automated tools. In addition, steps should be largely independent such that decisions can be made reliably and intermediate design models between steps can provide accurate feedback. The goal is to provide a minimal, orthogonal set of steps and intermediate models for design automation and exploration.

In order to provide an automated design process, a well-defined design flow with clean and unambiguous abstraction levels, models, and transformations is required. The key to the success of this approach are properly defined design models. High-level models must be implementable while providing an abstraction of implementation detail that allows rapid, early, and reliable exploration of critical design issues. Arbitrary models without clear semantics do not enable synthesis and verification for an automated path to implementation. In addition, synthesis requires clear definitions of the target architecture and the set of synthesis steps to transform the input model into the target model. In this dissertation, we aim to define such input and target models, a set of design steps, and intermediate models that are necessary for an automated system design flow.

All in all, the resulting design flow should support a wide variety of realistic system applications and target architectures. The formalized nature of the process should enable design automation for decision making and model refinement. Finally, together with design automation, high-level models should enable exploration of large parts of the design space in short amounts of time.

1.6 Dissertation Overview

In this dissertation, we propose a system modeling flow and methodology that is the basis for an automated, interactive system design environment. We provide the necessary rigorous structuring of the step-by-step system design process from abstract specification down to cycle-accurate implementation along a well-defined, linear set of design steps and design models. To achieve the required productivity gains for rapid system design and design space exploration, a flow of successive design models is defined that supports a wide range of target implementations, allows early validation of critical design issues, and enables design automation for model refinement and decision making.

Starting with a separation of computation and communication, we identify and classify implementation details and implementation decisions of different system design tasks. Based on dependencies between decisions and orthogonality of concerns, tasks and decisions within each task are ordered and grouped into a sequence of design steps and corresponding abstraction levels such that critical design issues can be addressed early while being able to obtain reliable feedback. Highly related decisions are grouped together, design tasks are broken into independent, manageable steps in order to bridge the design gap, and steps are ordered according to their dependencies. We develop design models after each step that accurately represent the corresponding implementation decisions

while abstracting unnecessary or unknown implementation detail. For each design step, we define the necessary design decisions and model transformations such that decisions can be automated and models can be transformed automatically through successive refinement. Consequently, the set of design decisions and refined models defines the target architecture for each step and for the overall design flow.

The design flow has been implemented in the form of the system-on-chip design environment (SCE). Automating tedious and error-prone tasks while reaping human expertise and insight results in an interactive, automated flow with the necessary transparency, controllability, observability and productivity. As part of this work, we defined the overall architecture, tool flow, and interfaces of an environment that enables integration and interoperability of various tools for automation of the design process under a common framework. Models in the flow have been defined such that they can be automatically generated through successive refinement. Corresponding model refinement tools have been integrated. In addition, a graphical user interface (GUI) has been developed. The GUI aids and steers the designer in the decision making process, provides visualization of design model characteristics and metrics, supports interactive, graphical decision entry for each design step, and allows the user to employ automated decision-making algorithms selectively on parts of the design.

Experimental results obtained by applying the design flow to several industrial-strength examples demonstrate the feasibility and effectiveness of the design flow and design environment. By showing tradeoffs and benefits of design models in terms of accuracy and speed, results confirm the choice of design steps and intermediate design models for rapid, early design space exploration. Furthermore, experiments show that complete systems can be designed in a short amount of time for realistic applications and target architectures.

The rest of this dissertation is organized as follows. Chapter 2 describes the overall design flow and the corresponding design methodology as implemented by the SoC design environment. In Chapter 3, requirements, rules, and guidelines for specifying a system at the start of the design process are given. Chapter 4, Chapter 5 and Chapter 6 then describe in detail the different steps and design models that comprise computation, communication and backend design tasks, respectively. In Chapter 7, the implementation of the SoC design environment (SCE) including its architecture, user interface, databases, and tool flow is described in more detail. Experimental results of applying the design flow and design environment to the system design examples are shown in Chapter 8. Finally, Chapter 9 summarizes and concludes the dissertation.

1.7 Related Work

1.7.1 Design Methodologies

There are several approaches dealing with classification and structuring of the design process [58, 65]. However, none of these defines an actual flow with models at specific points.

Traditionally, abstracted models of a design are used mainly for simulation purposes. In such simulation-centric approaches, the designer is responsible for manually rewriting the model at a fixed level of abstraction to adjust to changes in the design. There has been a lot of work done on horizontal integration of different models for simulation. At lower levels, different languages or implementations are integrated for co-simulation [20, 40]. At higher levels, different models of computation are combined into common simulation environments for specification [9]. However, none of these approaches attack the vertical integration of models that is needed for a synthesis-centric design flow with refinement of higher-level models into lower-level ones.

1.7.2 Computation Synthesis

1.7.2.1 Partitioning

Automation and optimization of partitioning of functionality onto a set of processing elements (PEs) is a well-studied problem. Early hardware/software co-design approaches were based on architectural templates usually consisting of one processor assisted by a custom hardware co-processor [29, 55]. More recently, general partitioning for heterogeneous, distributed multiprocessor systems has been solved using optimal, but exponential-complexity, integer linear programming (ILP) [84] or non-optimal heuristic [101, 22] approaches.

Similar to partitioning of behavior onto PEs, there are a number of approaches dealing with exploration of storage to memory mapping, considering different memory hierarchy configurations [54, 81, 16]. In all cases, however, automated partitioning approaches by itself do not provide a path to final implementation. Therefore, they are complementary to our design flow in the sense that the corresponding algorithms can be plugged into our design environment and applied to design models at corresponding design steps as determined by the user.

1.7.2.2 Scheduling

In the embedded software world, there is a well-established dynamic scheduling theory as part of real-time operating system (RTOS) design [10, 67, 72]. Variants of earliest deadline first

(EDF) scheduling for aperiodic task sets [59] or of rate monotonic (RM) scheduling for periodic tasks [71] are universally employed today. Even though these approaches can provide necessary timing guarantees, they are generally limited to pure software solutions on a single processor. As such, they are integrated into modern RTOS implementations. Therefore, in our design flow they provide the targets for dynamic scheduling of tasks on each PE in the system.

General scheduling approaches that deal with distributed multiprocessor systems are usually limited by specific assumptions. Static scheduling, for example, is feasible for specific models of computation (MoC) like synchronous dataflow (SDF) [69]. For multiprocessor dynamic scheduling, there exist a number of schedulability analysis approaches that aim to validate pre-determined scheduling strategies [105, 21, 103]. More recent work has been dealing with analysis of systems with multiple, mixed scheduling strategies [86, 114]. Similar to the situation for partitioning algorithms, all of these analysis and optimization approaches are complementary to the design flow proposed in this work and can be integrated into our design environment at corresponding stages.

1.7.2.3 RTOS Targeting

In terms of targeting real-time operating systems (RTOS) for dynamic scheduling of software tasks mapped into a processor during computation synthesis, traditionally work has been focusing on automatic RTOS and code generation for embedded software. Gauthier *et al*. [39] give a method for automatic generation of application-specific operating systems and corresponding application software for a target processor. Cortadella *et al*. [19] propose a way of combining static task scheduling and dynamic scheduling in software synthesis. While both approaches mainly focus on software synthesis issues, the papers do not provide any information regarding high level modeling of the operating systems integrated into the whole system.

There are a limited number of approaches dealing with high-level modeling of operating systems for early validation of dynamic scheduling effects during system design. Tomiyama *et al*. [93] show a technique for modeling fixed-priority preemptive multi-tasking systems based on concurrency and exception handling mechanisms provided by SpecC. However, the model is limited in its support for different scheduling algorithms and inter-task communication, and its complex structure makes it very hard to use. Finally, Desmet *et al*. [23] propose a high-level model of a RTOS called SoCOS. Instead of being written on top of existing SLDLs, however, SoCOS requires its own proprietary simulation engine. Therefore, it is difficult to integrate the model into overall system design models and design flows.

1.7.3 Communication Synthesis

Recently, SLDLs have been proposed as vehicles for so-called transaction-level modeling (TLM) of systems to provide communication abstraction [18, 15]. However, no general definition of the level of abstraction and the semantics of transactions in such models have been given. Furthermore, TLM proposals so far focus on simulation-purposes only and they lack a path to vertical integration of models for implementation and synthesis.

There are several approaches dealing with automatic generation, synthesis and refinement of communication [104, 17, 74, 96, 92]. None of these approaches, however, provide intermediate models breaking the design gap into smaller steps required for rapid, early exploration of critical design issues. Furthermore, to our knowledge, there is no approach that deals with methodical and automated implementation of communication over network-oriented, non-traditional communication structures.

Finally, in [89], the authors show an approach for modeling of communication at different levels of abstraction with automatic translation between levels based on message composition rules. However, they do no propose an actual design methodology and their approach is, for example, limited in its support for arbitration and interrupt handling in traditional bus-based architectures.

1.7.4 Design Environments

There are several approaches which aim to provide complete design environments that allow designers to bring an initial specification down to an implementation ready for final manufacturing.

The SpecSyn system [36] developed by Prof. Gajski's group at UC Irvine in the early nineties is based on the SpecCharts [94] language which is the predecessor of SpecC. SpecCharts are an extension of VHDL for system design and as such oriented more towards hardware design and limited in terms of support for complex embedded software. In contrast, SpecC is based on ANSI C with extensions for hardware modeling and as such supports complex SoC designs consisting of arbitrary combinations of hardware and software in a unified manner.

In the mid nineties, several co-design environments with focus on the hardware/software partitioning problem emerged. As mentioned previously, such hardware/software co-design approaches are based on architectural templates consisting of a microcontroller assisted by a custom hardware co-processor. As such, the systems are limited in their support for heterogeneous, distributed multiprocessor target architectures.

The COSYMA environment [80] developed by Prof. Ernst's group at TU Braunschweig is among the earliest examples of such systems. COSYMA is a hardware/software co-design system which uses an extension of C called C^X as its input language. It employs a number of algorithms for automatic partitioning and scheduling of specifications onto the architectural template.

In the COSMOS system [95] developed by Prof. Jerraya's group at TIMA laboratory in Grenoble, specifications written in the telecommunication standard language SDL [64] are converted into an internal representation called SOLAR [66] on which a set of transformations are defined that the user can apply to manually refine the specification into an implementation.

Finally, the POLIS system [4] developed at UC Berkeley under Prof. Sangiovanni-Vincentelli is another example of a hardware/software co-design environment. POLIS uses Esterel [6] as its input language and is based on an internal co-design finite state machine (CFSM) model which is, however, limited in its support for complex system designs, e.g. in terms of arbitrary bus protocols for communication. VCC from Cadence Design Systems, Inc. was a commercial product that was developed in the late nineties (discontinued in 2003) based on ideas from the POLIS system.

More recently, design environments have been developed that aim to provide support for more complex system designs beyond limited architectural templates. An example of such an environment is the OCAPI system which started development at IMEC in Belgium in the late nineties. The OCAPI system [98, 87] is based on an object-oriented modeling of designs using a C++ class library. OCAPI allows designers to model system with emphasis on dataflow-dominated designs at different levels of abstraction in a unified manner with easy refinement between models. Furthermore, it supports automatic code generation for both software and hardware with focus on targeting reconfigurable hardware devices (FPGAs).

Chapter 2

Modeling Flow

In order to implement an automated system design process in the form of a system design environment, a well-defined design flow and corresponding design methodology have to be developed. Starting from an identification and classification of necessary implementation concepts, the design process has to be broken into design tasks and input, output and target design models for each task have to be defined. The well-defined, formalized set of design models and design tasks between models defines a corresponding design methodology that is the basis for implementation of the automated system design flow through a design environment.

In this chapter, we define the overall design flow and design methodology that is the basis for our system design environment. Following an overview of system design issues and design tasks in Section 2.1, the four major design models at the core of the design flow are defined in Section 2.2. The resulting design methodology is summarized in Section 2.3. Finally, the design process for implementation of the design flow and the resulting system design environment are introduced in Section 2.4 and Section 2.5, respectively. Details of the implementation of each design task and of the environment itself will then be presented in the remaining chapters of the dissertation.

2.1 Overview

An overview of the system design process and corresponding design issues and design tasks is shown in Figure 2.1 Generally, system design starts with a set of requirements where different parts are possibly captured in domain-specific models of computation (MoCs). However, in order to feed into a global design and synthesis flow, requirements have to be combined into a single, unambiguous system specification. As outlined in the introduction (Chapter 1), design at the

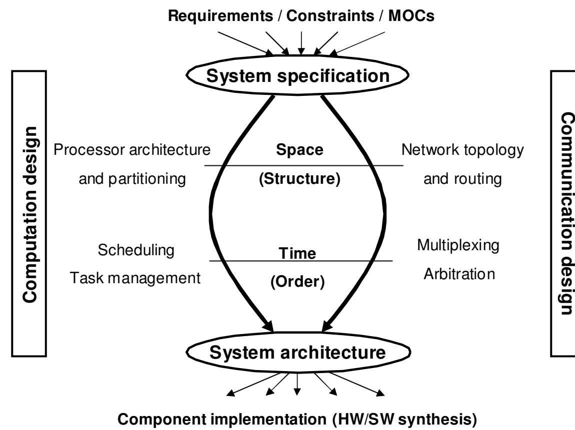


Figure 2.1: System-level design.

system level is then the process of implementing this specification down to a system architecture of components connected via wires. From there on, each of the component is then further implemented through software and hardware synthesis in a backend process at the RT level.

Based on the observation that computation and communication in a system are to a large part orthogonal and hence can be separated, the system design process can be divided into the two major tasks of computation design and communication design. Furthermore, separation of computation and communication is necessary for support of intellectual property (IP) components in design space exploration [25].

Starting with the system specification, the actual design process therefore consists of separate computation and communication design tasks. In each case, synthesis and design means that space (where?) and time (when?) of computation and communication have to be resolved. In terms of computation, the behavior of the specification has to be partitioned onto a set of hardware or software processors. Since each processor has a single thread of control only, execution on the inherently sequential then needs to be scheduled statically or dynamically. In terms of communication, the network topology of stations connected via communication media has to be defined and system channels have to be routed over this network. Then, interfaces to the shared media have to be implemented by resolving multiple accesses to the same media via multiplexing and arbitration.

Even though interactions between computation and communication tasks are minimized, there are still strong dependencies. Specifically, since partitioning of computation determines the amount of communication, computation design needs to be performed before communication design. Similarly, within each task, space issues generally have to be resolved first in order to determine how much functionality has to be shared and ordered in time on each processor or medium.

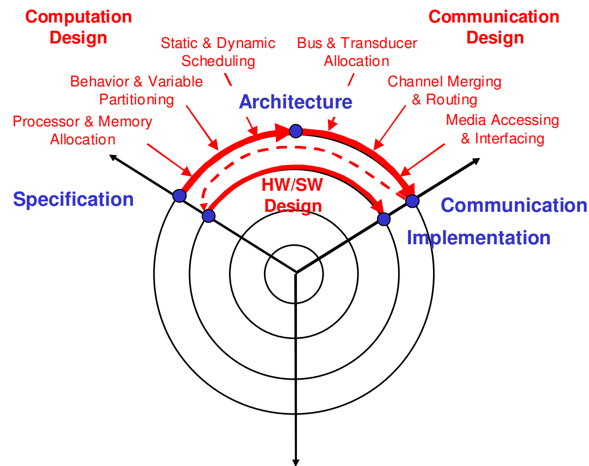


Figure 2.2: System design flow.

2.1.1 Design Flow

The resulting system design flow in the Y-Chart is shown in Figure 2.2. System-level design starts with the behavioral system specification. The first design task is computation design. In order to map computation as represented by behavioral blocks and variables in the specification onto a processing architecture, computation design requires allocation of processors and memories, mapping of behavior and variables onto those components, and finally scheduling of execution on the processors.

After computation design, an intermediate virtual architecture description of the system is generated in which processors communicate through transactions over abstract channels. The virtual architecture is a mixed behavioral/structural description. It defines the computation structure but leaves communication at a behavioral level.

Then, communication design implements communication between processors and memories over a network of busses or other communication media. Communication design requires allocation of busses and bus transducers (bus bridges), merging and routing of channels over links between processors and transducers, and interfacing of processors and transducers to bus media. The result is a fully structural, bus-functional description of the system showing all the details of communication between components at the system level.

Finally, in a backend process, each of the PEs in the system is implemented separately by implementing its behavior in hardware or software on top of the PE's synthesized or fixed micro-architecture. On the hardware side, the backend process follows a standard high-level synthesis approach [34, 76] requiring allocation of RT units (functional units, registers and register files,

local memories and busses), scheduling of operations into clock cycles, and binding of operations, variables and transfers to functional units, storage units and busses. On the software side, code for the target processor has to be generated, compiled and linked against a selected target operating system.

2.1.2 Modeling

Given the basic division of the system design process into computation, communication and backend design tasks, details of each design task have to be defined as a basis for any implementation of the design flow in an automated way. In a first step, design models at the input and output of each design task have to be defined. More specifically, corresponding abstraction levels as determined by the amount and granularity of implementation detail represented in each model have to be specified. Clear definitions of inputs and outputs of each task then serve as a specification for the implementation of each design task. In general, characteristics and features of each input, output and intermediate model have to defined such that interactions between tasks are minimized, automation of design tasks becomes feasible, and reliable feedback for design space exploration is provided.

Details of system-level design steps are listed in Table 2.1 in terms of necessary design decisions, resulting implementation detail in the form of structure and order, and applicable quality metrics for evaluation and exploration. Generally, for each design step, a new design model can be generated that reflects the design decisions made by introducing corresponding implementation detail. However, if steps are dependent on each other they have to be combined into a single model in order to obtain useful feedback. On the other hand, independent steps should be separated into different models in order to break the problem into smaller parts, manage complexity and separate concerns. With each new model, a minimal number of new features should be introduced such that they will not be influenced by later steps and can be evaluated and decided at a high level of abstraction. The goal is therefore to organize steps into a minimal, orthogonal set of design models such that each model provides the designer with reliable feedback for design and exploration.

The first step in system design is the definition of the processing architecture. Allocation of processors defines the hardware execution platform, trading off system cost versus true concurrency available in the system. In order to evaluate the quality of the allocation, however, raw processing performance of the architecture has to be matched against the characteristics (e.g. available parallelism) of the application. Therefore, processor allocation has to be combined with behavior

Design Decision	Implementation Detail / Granularity		Quality Metrics
	Structure / Space	Order / Time	
Processor Allocation	Processor Architecture	-	Cost, Concurrency
Behavior Partitioning	Execution host, Synchronization	Execution time	Code size, Traffic, Behavior delay
Memory Allocation	Memory Architecture	-	Cost, Memory bandwidth
Variable Partitioning	Storage, Communication	Memory access time	Memory size, Traffic, Behavior delay
Static Scheduling	Task grouping, Task graph	Serialization, Behavior order	Task delay, Processor utilization
Dynamic Scheduling	OS scheduler, Task creation	Context switches	Processor delay, Utilization
Network Allocation	Network topology	-	Cost, Bus bandwidth
Channel Routing	Logical links	Packet transfers	Transmission delay
Media Sharing	Media access, Arbitration	Bus cycles, Transactions	Latency, Bus utilization
Media Interfacing	Protocols, Wires, Interrupt handling	Signal events	Protocol timing

Table 2.1: System-level design steps.

partitioning. Given the processor architecture, functional blocks of the specification can be mapped onto those processors. As a result, behavioral blocks are mapped onto their execution hosts, additional synchronization is inserted as necessary to preserve dependencies, and code at the leaves of the hierarchy is annotated with estimated execution delays. The combined model after processor allocation and behavior partitioning then allows to validate correctness of the partitioning and synchronization implementation. Furthermore, it can provide feedback about relative performance of different mappings of individual behaviors to processors.

Definition of the memory architecture is performed in a similar manner. Allocation of memories has to be combined with partitioning of variables into those memories in order to obtain feedback about the application-specific design quality. Together, memory allocation and variable partitioning determine the actual memory sizes necessary to store all system variables and the message traffic necessary for communication of variable values between processors and memories.

While memory allocation can be done together with processor allocation, variable partitioning depends on the results of processor allocation and behavior partitioning which declare local

processor memories and expose variables shared between behaviors mapped to different processors. Therefore, combined processor allocation and behavior partitioning has to preceeded by combined memory allocation and variable partitioning. Even though processor issues can be resolved separately from memory issues, they should be combined into a single allocation and partitioning step since overall system communication and traffic is directly influenced by both. The resulting model after allocation and partitioning validates implementation of mapping, synchronization and communication while providing feedback about basic cost, relative performance and overall traffic.

After allocation and partitioning, accurate delay estimates are available at the level of individual functional blocks. In order to obtain feedback about overall input-to-output delays, scheduling of execution on the processors is necessary. Static scheduling serializes and orders behaviors within each task in a processor. Dynamic scheduling then orders tasks at runtime under control of a scheduling algorithm. As a result of behavior and task ordering, estimated task and overall processor delays become available. Since overall delays depend on both, static and dynamic scheduling are combined into a single scheduling step.

Scheduling in general depends on the mapping of behaviors to processors and as such has to be preceeded by allocation and partitioning. Compared to the model after allocation and partitioning, only the model after scheduling can provide accurate feedback about delays in each processor and overall processing delays. Therefore, allocation, partitioning and scheduling all have to be performed[1] to obtain a model that supports reliable evaluation and exploration of design performance and quality, especially for computation-dominated designs. Hence, the resulting models is the architecture model at the output of the computation design task.

Communication design has to start from a model that accurately specifies the system communication traffic to be implemented. As such, communication design depends on allocation and partitioning. The input of communication design is therefore the architecture model at the output of computation design. Note that for communication-oriented designs, the scheduling step can be deferred after communication design and communication design can be started with the model after allocation and partitioning.

Similar to computation design, communication design starts with allocation of a network architecture consisting of busses or other communication structures connected by transducers (e.g. bridges) based on cost and capacity factors. Given the communication architecture, channels in the input model can be mapped onto this network and routed over logical links between stations.

[1]Not necessarily together in one step, e.g. for simplicity of algorithms, allocation, partitioning and scheduling can be done sequentially in two or three steps, trading off complexity and optimality.

Again, network allocation and channel routing depend on each other and have to be performed in one step. The resulting model is accurate down to individual data packet transfers and associated packet transmission delays over each logical link. It allows to validate the raw network performance and correctness of the network and routing implementation.

Similar to scheduling during computation design, accurate estimates about overall delays and performance are only available after ordering of communication on each shared medium. Media sharing requires data slicing, addressing and arbitration to implement media accesses. As a consequence, the resulting model is accurate down to individual media word/frame transaction (e.g. bus cycles) and provides reliable feedback about overall communication delays. The model after computation design, network design and media sharing is a transaction-level model (TLM) that supports reliable validation, evaluation and exploration of total system cost, quality and performance including computation and communication.

A transaction-level model, however, can not directly feed into the backend design process. For synthesis of system components in custom hardware, timing- and pin-accurate specifications of component interfaces have to be available. Therefore, media sharing is combined with media interfacing to produce the final output model of the system design process. Media interfacing inserts bus protocol descriptions into each component, implements interrupt handling via interrupt controllers and connects components via pins and wires. The result after computation and communication design is the final communication model. The communication model serves as input to the backend design process, allows validation of system connectivity and provides accurate feedback down to the level of single value change events on the pins and wires of the system netlist.

2.2 Design Models

The abstraction levels and models corresponding to the system design flow outlined in the previous section are shown in Figure 2.3. The specification model is the start of the design flow, unifying different domain-specific models of requirements and constraints. It free of any implementation detail and hence a purely functional, untimed system description. It defines a partial order in the system, based on causality only. With the specification model, the designer defines, validates and evaluates the functionality of the system design to be implemented without having to consider any implementation issues.

The architecture model defines the interface between computation and communication design. It describes the system as a structure of processors communicating through abstract trans-

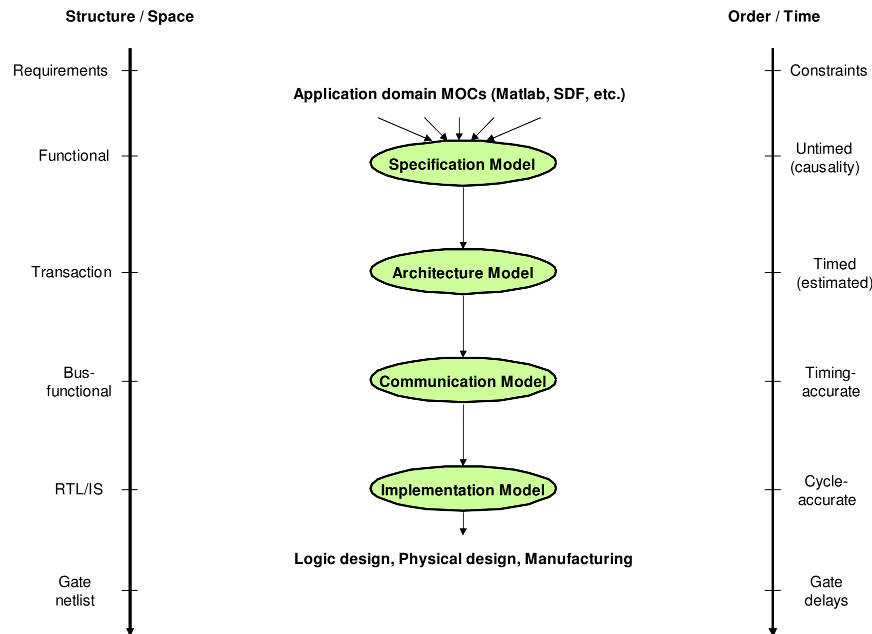


Figure 2.3: Abstraction levels and models.

actions. Additional order is introduced into the system through estimated execution delays on the processors. Using the architecture model, the designer can evaluate and explore the raw processing performance of the targeted system architecture. As results will show (see Chapter 8), with the architecture model, rapid, early exploration of the processor design space with no or little additional overhead is possible.

The result of design at the system level at the interface to the backend design process is the communication model. Starting with the communication model, each system component can then be further designed separately at lower levels. The communication model is a completely structural description of the system architecture of processors connected by wires. It is timing-accurate in the interactions between processors as observed through the events on the connecting wires. Using the communication model, the designer can validate and explore the interaction between system components in a timing-accurate manner.

Finally, the implementation model is the output of the backend design process and the sign-off to traditional logic, physical and manufacturing design processes. The implementation model provides a structural description of the micro-architecture of each processor at the RTL or instruction-set level. A complete implementation model is a cycle-accurate description of the whole system. As such, the implementation model allows the validate and explore the final, cycle-accurate performance and timing of the system.

In the following, we will define each of the four models at the core of the design flow in detail [47]. For each model, concepts, features and the amount of implementation detail represented within will be shown. The four models define the input, output and target architecture for each design task between models. As such, the formalized definition of design models forms the basis for implementation and automation of the design flow in the design environment.

2.2.1 Specification

The specification is a behavioral description of the system. It describes the desired functionality free of any implementation details. The specification is composed without any implications about the structure of the implementation. Objects in the specification model are abstract entities that perform computation on data and then terminate. Apart from timing constraints, there is no notion of time, i.e. behavioral objects execute in zero time. Objects are ordered only based on their dependencies.

At the specification level, a design consists of computation and communication. Computation is described by a hierarchical composition of behaviors. Behaviors communicate by transferring data messages over channels. More formally, a specification model is a triple

$$\langle B, V, C, R \rangle$$

consisting of a set of behaviors B, a set of variables V, a set of channels C, and a connectivity relation $R \subseteq B \times (C \cup V)$ that defines connections of behaviors to channels and variables.

Behaviors form a semigroup $(B, \circ)$ under the composition operation $\circ \in \{\triangleright, \|, |, \vee\}$. Behaviors $b1, b2 \in B$ can be composed sequentially ($b1 \triangleright b2$), concurrently ($b1 \| b2$), in a pipelined loop ($c : b1 | b2$), or in a mutually exclusive way ($c : b1 \vee b2$) where the pipelined and alternative compositions are guarded by additional conditions c. Blocks at the leaves of the hierarchy contain basic algorithms that perform computations. Such leaf behaviors contain a description of the algorithm using, for example, a standard programming language such as C. Hence, the code in the leaves describes how the behavior processes its input data to produce its output data using expressions over variables with different data types as supported by the programming language. Throughout the system design process, leaf behaviors will remain untouched, forming indivisible units for the purpose of exploration and refinement. In general, models describe how the system is composed out of the basic building blocks—the leaf behaviors—on top of any underlying language.

An example of a simple yet typical specification model is shown in Figure 2.4. In the example of Figure 2.4, the specification is a serial-parallel composition of *b1* followed by the con-

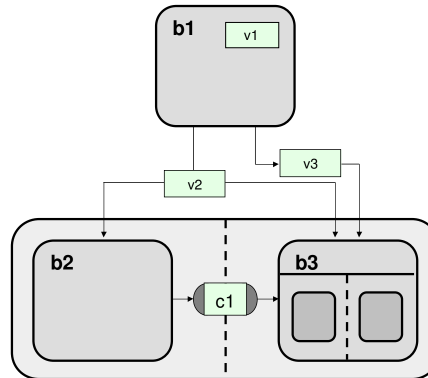


Figure 2.4: Specification model example.

current execution of *b2* and *b3*. *b3*, in turn, is a parallel composition of two additional subbehaviors. Behavior *b1* encapsulates a local variable *v1* to store locally needed state information. At its outputs, *b1* produces data and writes it to global variables *v2* and *v3*. Data in *v2* is then consumed by *b2* and *b3* whereas *v3* is read by behavior *b3* only. Finally, the concurrent behaviors *b2* and *b3* communicate via a channel *c1* to send data from *b2* to *b3* during their execution.

In summary, the purpose of the specification model is to clearly and unambiguously describe the system functionality. The system is composed of self-contained blocks with well-defined interfaces enabling easy composition, rearrangement, and reuse. All dependencies are explicitly captured through the connectivity between behaviors and no hidden side effects exist. The parallelism available between independent blocks is exposed through their concurrent or pipelined composition. Computation and communication are abstracted as a composition of functions over data. They are separated into behaviors and channels, respectively, allowing for a separate implementation of both concepts.

2.2.2 Architecture

The architecture model is the result of mapping computation onto actual processing elements (PEs). It represents the allocation and selection of PEs and the mapping of behaviors onto PEs. It is a mix of a structural description of system computation and a behavioral description of system communication.

The architecture model redefines the computational part of the design. Formally, an architecture model is a triple

$$\langle PE, C, R \rangle$$

where computation is described as a set of concurrent PEs. PEs are structural objects representing physical components and as such are non-terminating. In general, the set of PEs in the system, $PE = P \cup IP \cup M$, consists of a set of hardware or software general-purpose processors, a set of IPs, and a set of memories, respectively. Communication as the set of channels C and the connectivity relation R between leaf behaviors and channels remains essentially untouched.

A processor $p \in P$ is defined as a triple

$$\langle B_p, V_p, C_p, R_p \rangle$$

that executes the set of behaviors B_p mapped onto it. Behaviors inside processors communicate via sets of local channels C_p and local variables V_p as defined by the connectivity relation $R_p \subseteq B_p \times (C_p \cup V_p)$. Due to the inherently sequential nature of structural objects such as processing elements, behaviors inside a processor have to be serialized. In a static scheduling approach, the order of behaviors is fixed and represented as artificial control dependencies of a purely sequential composition of behaviors inside the PE, i.e. processor behaviors form a semigroup $(B_p, \triangleright)$ under sequential composition only. In a dynamic scheduling approach, the order of behaviors is determined at run-time. Behaviors are composed into tasks and operating system models inside the PEs provide an abstraction of the dynamic scheduling behavior for dispatching tasks during runtime.

In contrast to general purpose PEs, a pre-designed intellectual property (IP) component $ip \in IP$ is defined as a triple

$$\langle B_{ip}, V_{ip}, A_{ip} \rangle$$

where the pre-defined, fixed functionality B_{ip} and storage V_{ip} is encapsulated in an IP adapter A_{ip}. The adapter abstracts the IP's internal communication interface and provides a set of canonical channel interfaces for communication with the IP at the behavioral (data message) level. At the system level, behaviors communicate directly with those adapters, i.e. the system connectivity relation $R \subseteq B \times (C \cup A)$ connects processor behaviors $B = \bigcup_{p \in P} B_p$ to channels C or IP adapters $A = \bigcup_{ip \in IP} A_{ip}$. Note that a special case of IPs are dedicated memory components which do not provide any functionality apart from (read and write) access to stored data V_{ip}.

An exemplary architecture model corresponding to the specification example from Figure 2.4 is shown in Figure 2.5. *b1* and *b3* are mapped onto processing elements *pe1* and *pe2*, respectively. *b2* is implemented by an existing IP component that provides the same functionality. A vendor-supplied black-box description *ip1* encapsulates a simulation, analysis and synthesis model of the IP while allowing integration into the system through an IP adapter interface.

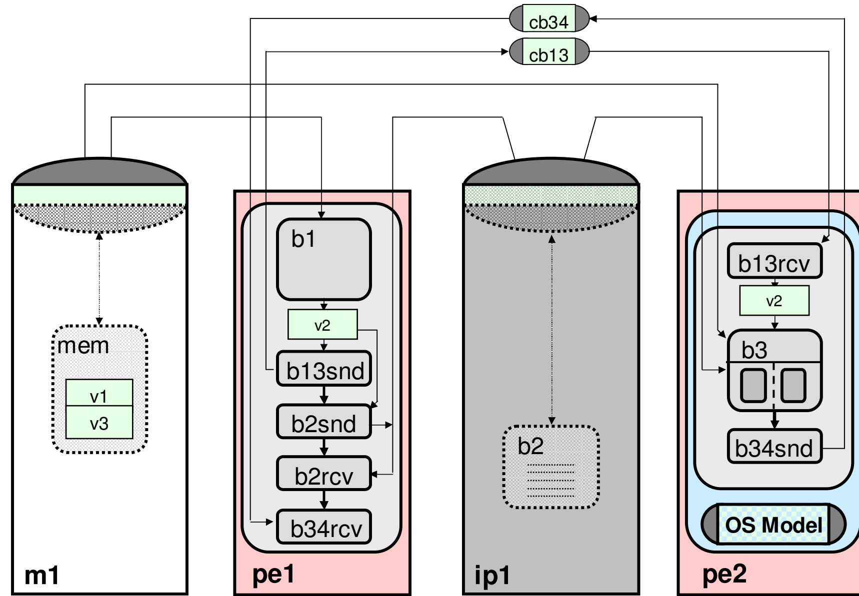


Figure 2.5: Architecture model example.

A system memory *m1* holds variables *v1* and *v3* and provides read and write access through its channel interface. On the other hand, local copies of the variable *v2* have been created in *pe1* and *pe2*. In addition, communication and synchronization blocks *bXXsnd* and *bXXrcv* have been inserted to preserve the original execution semantics. Execution of formerly sequential blocks mapped to concurrent PEs is synchronized, and updated variable values are communicated to keep local copies in sync.

Inside *pe2* an operating system channel is inserted that manages dynamic scheduling of the two concurrent subbehaviors inside *b3*. Tasks inside *pe2* use operating system services for task management and synchronization by communicating with the OS model channel.

Finally, behavioral blocks inside *pe1* and *pe2* communicate via global channels *cbXX* or by accessing the channel interfaces of *m1* and *ip1* directly.

In summary, the architecture model refines computation by grouping behaviors and mapping them onto a PE structure while largely preserving the original behavioral communication. PEs contain a behavioral description of their functionality. Behaviors inside PEs execute in order through static or dynamic scheduling. In addition, the architecture model introduces the notion of time for the computation mapped onto the PEs, further increasing the partial order among PEs. Based on estimated execution times on the target PEs, behaviors are annotated with delay information. Therefore, true parallelism at the architecture level is only available through the set of concurrent PEs.

The architecture model describes the implementation of the computation on the PEs of the system architecture. It is a structural view of the system's computational aspects. On the other hand, the architecture model contains behavioral descriptions of the PEs that will feed into the lower parts of the design flow. Finally, at the architecture level, communication between the PEs is exposed for implementation in the following steps.

2.2.3 Communication

The communication model is a structural description of the complete system for both computation and communication. In addition to allocation and selection of PEs as part of the multiprocessing model, the communication model represents the allocation and selection of busses and the mapping of global channels onto busses. As a result, the system is modeled as a netlist of components connected via bus wires. It is obtained by adding bus protocols to all channels, splitting channels, and inlining them into each PE as bus drivers.

Based on the multiprocessing model definition, the communication model redefines the global communication part of the system. An communication model is defined as a triple

$$\langle S, W, c \rangle$$

where S is the set of network stations or system components, W is the set of bus wires, and $c : \bigcup_{p \in S} O_p \mapsto W$ is the port mapping function connecting component ports to bus wires. In general, the set of communication model components, $S = PE \cup CE$, is a combination of the sets of processing elements $PE = P \cup IP \cup M$ (consisting of general-purpose processors, IPs and memories) and communication elements $CE = T \cup A \cup IC$ (consisting of transducers, arbiters and interrupt controllers).

Behavioral processor descriptions are transformed to bus-functional models by adding bus drivers. A processor $p \in P$ in the communication model is a quintuple

$$\langle B_p, V_p, C_p, D_p, O_p, R_p \rangle$$

where B_p is the scheduled set of behaviors executing on the processor, V_p is the set of local variables, C_p is the set of local channels, D_p is the set of bus driver channel interfaces, O_p is the processor's set of ports, and $R_p \subseteq B_p \times (C_p \cup V_p \cup D_p)$ is the connectivity relation that has been extended to define the connection of behaviors to channels, variables and bus drivers. Bus drivers describe a processor's implementation of the data messages over the bus protocols on the processor's ports.

Inside the processor, bus drivers provide a behavioral message interface to its behaviors and the behaviors connect to those channel interfaces for all bus communication.

For IP components, bus-functional or structural IP models can be directly integrated into the communication model. Bus-functional IP models are equivalent to the definition of bus-functional processor models shown above. Structural IP models, on the other hand, are defined as netlists of RTL components. A structural $ip \in IP$ is a quadruple

$$\langle U_{ip}, B_{ip}, O_{ip}, c_{ip} \rangle$$

where U_{ip} is the set of RTL units, B_{ip} is the set of local busses, O_{ip} is the set of ports, and c_{ip} is the connectivity function mapping ports of RTL units to busses and external IP ports. In the communication model, bus-functional and structural IP models can be used interchangeably allowing, for example, mixed-level co-simulation. Again, note that memory components can be treated as a special case of IPs.

If necessary, special transducer PEs that translate between incompatible protocols need to be inserted into the communication model. A transducer interfaces to two busses via two sets of ports and contains bus drivers for each protocol. Hence, a transducer is defined as a processor with two sets of ports and two sets of bus driver channel interfaces.

Finally, the communication model can contain arbiters which mediate conflicting bus accesses in case of multiple masters on a bus. Arbiters implement a certain arbitration protocol on their bus ports through internal bus drivers. Therefore, equivalent to scheduling of computation on PEs in the multiprocessing model, arbiters serialize accesses to the inherently sequential busses. Arbiters usually come in the form of IPs and as such can be defined as bus-functional or structural processor models.

The communication model example corresponding to the previously shown architecture model is shown in Figure 2.6. In the example, the memory *m1* is connected to processor *pe1* via the processor's bus *bus1* while *ip1* and co-processor *pe2* are connected via *bus2*. Inside *pe1* and *pe2*, behavioral blocks connect to bus drivers that implement message-passing over the bus wires. Transducers *t1* and *t2* translate between incompatible bus protocols. *t1* acts as a bridge between busses *bus1* and *bus2*. *t2* interfaces the IP with its proprietary protocol to *bus2*. The channel interface of *ip1* in the communication model is moved into *t2* where it implements communication with *ip1* over the exposed wires of the IP bus. Finally, an additional PE *arbiter1* that regulates conflicting accesses of *pe1* and *t1* on *bus1* is inserted.

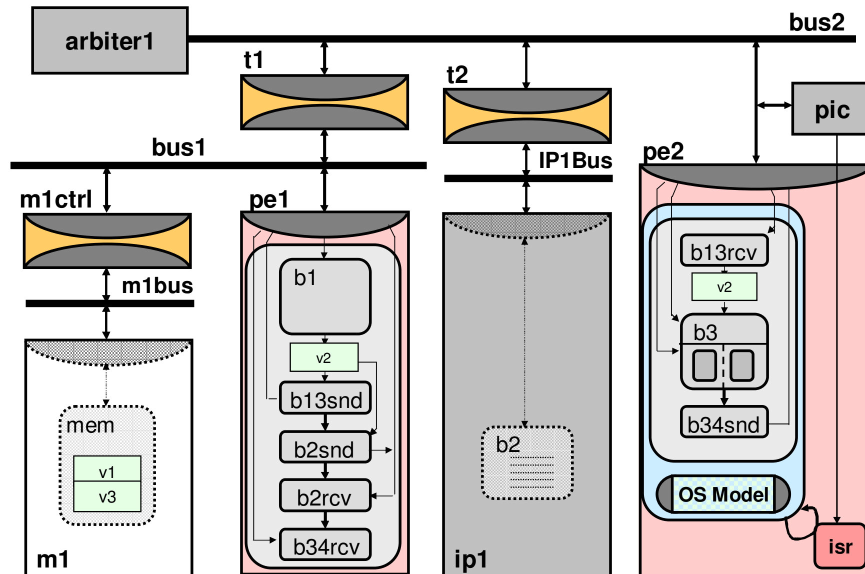


Figure 2.6: Communication model example.

In summary, the communication model refines communication into an implementation over busses and transducers. Computation inside the PEs, on the other hand, remains largely untouched. The structural nature imposes a total order on the communication over each bus. Furthermore, the partial order between busses is refined by introducing bus protocol timing. Therefore, the communication model is timing-accurate in terms of both computation and communication.

2.2.4 Implementation

The implementation model is the result of scheduling the functionality mapped onto the PEs (both, computation and communication functionality) into register transfers per clock cycle. Therefore, the implementation model is a cycle-accurate model at the register-transfer level.

For each PE, the implementation model defines the datapath, the control logic and the clock frequency at which the component runs. In general, the implementation model requires allocation of a datapath, binding of operations, variables, and transfers onto functional units, registers/memories and busses, and the scheduling of register-transfers into clock cycles.

For custom hardware PEs, high-level synthesis creates the implementation model of the hardware PE from the code of the behaviors and adapters inside the PE behavior of the communication model. For programmable processors, the code of the behaviors in the communication model is converted into C code and compiled into assembly code to create the implementation model.

The implementation model supports two views of the PEs in the design: a behavioral RTL view and a structural RTL view [70]. In both cases, the steps of allocation, binding and scheduling are required to derive the implementation model. The difference is that the behavioral RTL view does not explicitly represent the datapath architecture and the binding information. However, it corresponds closely to the original C code in the communication model. The structural RTL view, on the other hand, explicitly describes the structure of data path plus control unit. Therefore, structural RTL is closer to the implementation and forms the immediate input to logic synthesis.

2.2.4.1 Behavioral RTL

Behavioral RTL specifies the operations performed in each clock cycle without explicitly modeling the units in the PE's datapath. Instead, operations in each cycle are described in the form of finite state machine with datapath (FSMD) models. Therefore, behavioral RTL is close to the original, sequential C code. Essentially, behavioral RTL is obtained by scheduling the operations in the C code into clock cycles.

Depending on the type of PE, different styles are needed for the implementation models of the PEs at the behavioral RTL level. For programmable processors, the operations performed in each clock cycle are defined by the assembly code compiled for that PE. On the other hand, for custom hardware PEs the operations in each clock cycle can be explicitly modeled.

The behavioral RTL model for the given design example is shown in Figure 2.7. For custom hardware and IP components such as *pe1* and *ip1*, the behavioral RTL model contains FSMD models describing the cycle-accurate behavior of both computation and communication inside the PEs. Similarly, transducers *t1* and *t2* and the arbiter *arbiter1* are replaced with descriptions that contain FSMD models of their bus interfaces' protocol implementation in hardware. Finally, the memory *m1* is modeled as a state machine that answers incoming requests in a similar manner.

In contrast to hardware PEs, the behavioral RTL model of programmable processors is based on the execution of assembly output generated by compiling the code of the PE behavior in the communication model. Therefore, the behavioral RTL model for the programmable component *pe2* implements an instruction set simulation (ISS) of the assembly code. The simulation model of the processor executes the instruction stream synthesized through code generation, compilation and linking in the backend. Existing ISS models of processors can be plugged into the system through wrappers that drive and sample the ports of the PE behavior based on any I/O instructions encountered during runtime.

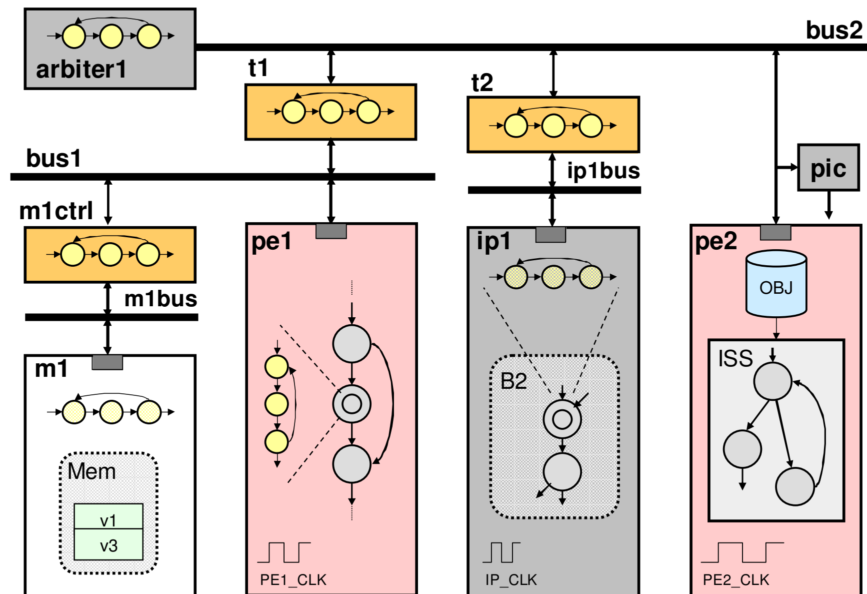


Figure 2.7: Behavioral implementation model example.

2.2.4.2 Structural RTL

A structural RTL view of the PEs in the implementation model accurately reflects the microarchitecture internal to the system PEs. As a result of the high-level synthesis process, structural RTL explicitly models the allocation of RTL components, the scheduling of register transfers into clock cycles, and the binding of operations, variables and assignments to functional units, register/memories and PE busses. The result is an RTL netlist of sequential and combinatorial logic inside each PE. Structural RTL is the input to traditional logic synthesis which in turn will derive a gate-level netlist from the netlist of units inside each PE.

A structural RTL representation is usually used for custom hardware PEs which have to be synthesized further. Since structural RTL represents the hardware microarchitecture of PEs, at this level there is no difference between models for custom hardware or programmable processors. In both cases, structural RTL is a netlist of functional units, busses, memories and registers. However, in case of predesigned components (IPs, programmable off-the-shelf processors, memories) the level of detail for further synthesis of the hardware is not needed. A more abstract behavioral RTL model is sufficient for effective simulation.

A structural RTL model example for the previously introduced design is shown in Figure 2.8. For each processing element *pe1*, *ip1*, and *pe2*, the structural RTL model consists of controller and datapath netlists where the controller drives the datapath through a set of control

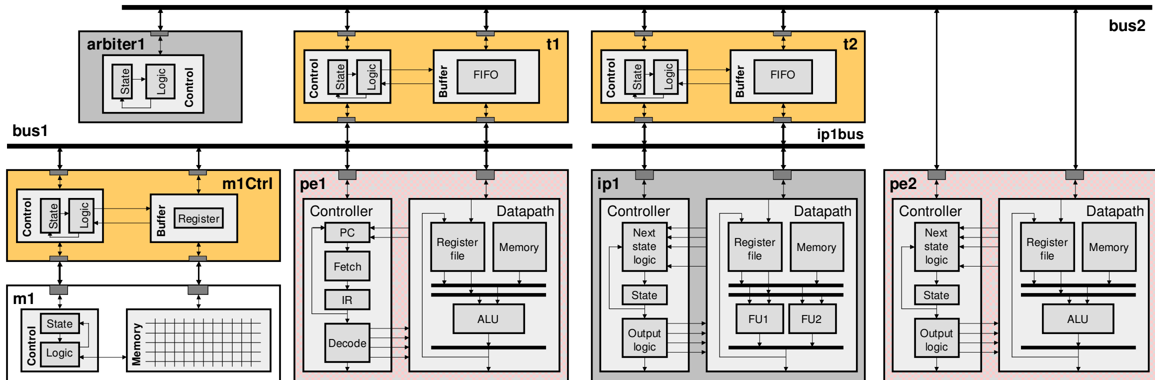


Figure 2.8: Structural implementation model example.

and status wires. In each case, datapaths are given as netlists of RTL components such as register files, memories and functional units connected by busses. For hardware PEs such as *ip1* and *pe2*, controller FSMs are hardwired in the form of state registers and next state and output logic blocks. For programmable PEs like *pe1*, the controller contains instruction fetch and decode logic blocks in addition to program counters (*PC*) and instruction registers (*IR*).

Structural RTL models of transducers, arbiters and memories all contain FSM implementations for control in the form of state registers and control logic. In the case of transducers (*t1*, *t2*, and *m1Ctrl*), the datapath consists of simple queues for buffering between interfaces. For the memory *m1*, the memory cell array is the datapath driven by the controller FSM.

2.2.4.3 Implementation Model

At the top level, the implementation model is equivalent to the communication model. The system is a set of concurrent, non-terminating PEs communicating via busses and wires. Internally, on the other hand, PEs represented by the PE behaviors, are further refined and turned into a model of the PE's microarchitecture. The minimal requirement for the PEs in the communication model is that they provide a cycle-accurate description of events on their ports through a behavioral microarchitecture model. Alternatively, more detailed PE models can be used in the communication model, e.g completely structural RTL descriptions.

PE behaviors are interchangeable between communication and implementation model. This allows mixed-level simulations in which a cycle-accurate PE behavior is part of an otherwise bus-functional simulation of the design and vice versa. Therefore, different parts of the system can be simulated at different levels of detail, e.g. allowing to quickly validate individual PE's.

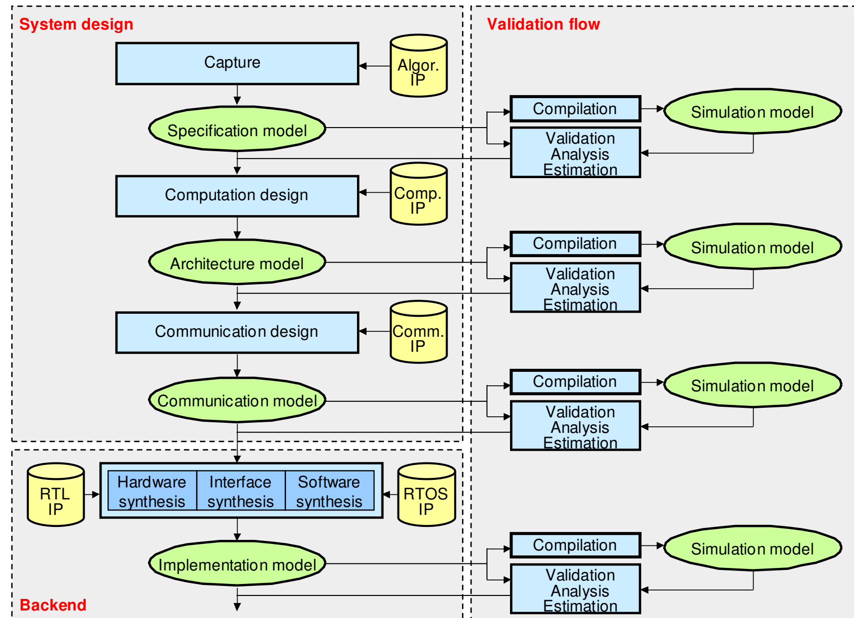


Figure 2.9: System design methodology.

In summary, the implementation model is a cycle-accurate model of the system implementation of both, the communication between the PEs and the microarchitecture inside the PEs. In contrast to the bus-functional communication model, the computation inside the PEs is refined down to the register-transfer level. As a result of high-level synthesis of custom hardware and compilation of software for programmable components, the implementation model is the basis for further refinement down to the gate level through logic synthesis or instantiation of hard IP cores.

2.3 Design Methodology

In summary, the resulting overall system design methodology is shown in Figure 2.9. The methodology is a set of four models and three transformation steps that take a system specification down to an RTL/IS implementation [32, 46, 38, 37].

System design starts with the capture of the intended functionality in the form of an executable specification. The specification model describes the desired system behavior as well as performance, power, cost and other constraints. Based on the general design flow outlined in the previous sections, the system design flow is divided into two major tasks: computation design and communication design. Starting from the specification model, the design is gradually mapped onto a selected target computation and communication architecture through a series of well-defined steps.

Computation design selects a set of processing elements (PEs) and maps the computation behavior and storage inside the specification onto those PEs for implementation. Furthermore, computation design decides the order of computation over time through static or dynamic scheduling of concurrent functionality on the inherently sequential PEs. Computation design refines the specification into the intermediate architecture model. The architecture model describes the virtual computation architecture in the form of structural PE components that communicate via abstract, behavioral message-passing channels.

Communication design then refines the abstract communication between PEs into an actual implementation over real busses or other communication structures. During communication design, the topology of the communication network is defined, additional communication elements (CEs) like bus bridges or arbiters are inserted, and abstract channels are mapped and routed over this network. Inside the stations connected to the network, implementations of channel transactions over the protocols of the selected media are created. As the result of communication design, the communication model of the system is generated. The communication model is a fully structural, bus-functional description of the system computation and communication architecture as a set of components connected via pins and wires.

The communication model as the result of the system design process is then handed off to the backend tools. The communication model specifies the desired functionality of computation and communication inside each of the components of the system architecture. In the backend tools, each component is synthesized separately by implementing its behavior in custom hardware or software on top of the component's synthesized or fixed RTL or instruction-set microarchitecture. At the end of the backend process, the final implementation model of the system is generated. The implementation model is a cycle-accurate description of the whole system at the register-transfer or instruction-set level. This description, in turn, serves as the basis for manufacturing of the system through traditional logic synthesis and physical design processes.

All models in the methodology are captured in the SpecC SLDL. In a validation flow that is orthogonal to the design flow, models can therefore be simulated, verified, analyzed, and estimated at any point in the flow in order to validate functionality and design quality. For example, at the specification level system functionality is validated. At the architecture level, the computation structure is checked and computing performance is estimated. The communication level allows validation of component communication and associated timing and overhead. Finally, at the implementation level, clock-accurate performance, power, and other metrics are available for final sign-off before manufacturing.

2.4 Design Process

In general, a design methodology is defined as a set of design models and a set of transformations between models. With each step or task in the design methodology, a design model at a certain level of abstraction is transformed and a new design model at the next lower level of abstraction is produced. Again, if design tasks are implemented and automated by corresponding design tools, output models are said to be synthesized.

In general, each design task consists of multiple steps to be performed. In any case, however, all design tasks can be separated into two distinct steps:

(a) A process of making implementation decisions, e.g. choosing components out of databases and determining the mapping of functionality onto these components for implementation. The output of this process are design decisions (e.g. the list of components and the specification-component mapping). Usually, there is a wide variety of possible implementations and decisions have to be made in consideration of any given constraints by aiming to optimize design quality under a cost function.

(b) A process of refining the design model to reflect the implementation decisions. Given the design decisions, design models are refined such that the output model describes the chosen implementation. At lower levels of abstraction, model refinement is usually a straightforward process. For example, in logic synthesis, the design is represented as a gate netlist in a canonical form. However, at higher levels of abstraction, model refinement requires significant effort to generate an efficient representation due to the large variance in possible design representations.

Within each design task, each of the two steps can be manual or automated. In either case, a well-defined formal framework is necessary in order to apply automation. The primary goal of any automation should be to remove the need for tedious, error-prone model rewriting. Then, optimization algorithms can be applied on top of model refinement for automated decision making. Especially at higher levels, however, where optimality of solutions is hard or impossible to prove, the use of such algorithms should be guided by the user in order to leverage human experience and insight.

A general view of the resulting design process is shown in Figure 2.10. Starting with the specification, the design is brought down to an implementation through gradual, successive refinement of design models. In each step, design decisions are made either by the user through a graphical user interface (GUI) or with the help of automated synthesis tools implementing optimiz-

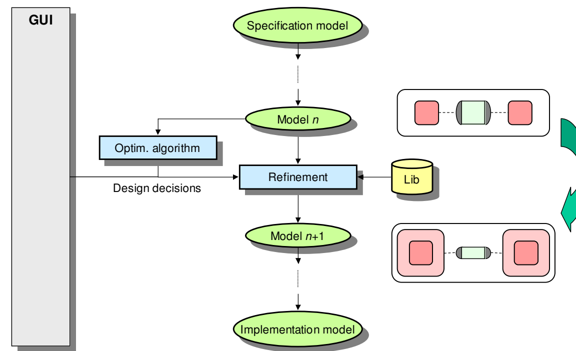


Figure 2.10: Design process.

ing algorithms. Based on the design decisions, a refinement tool will generate a new design model from the input model automatically.

In general, model refinement is based on a layering of implementation functionality. With each refinement step, a new implementation layer is inserted into the model. Depending on the modeling language, implementation layers can be directly represented as additional layers of hierarchy. Each new level of hierarchy covers one or more objects at lower levels and/or adds functionality in the form of additional design objects. Keeping implementation functionality organized as a stack of layers increases observability and transparency of the results of the design process. Note that in the backend process that follows system design and implements each system component down to the cycle-accurate level, layers still can be merged or combined for optimizations across layers.

2.5 Design Environment

The design methodology and design process defined in the previous sections are the basis for implementation of an automated system design flow in the form of a system design environment. The SoC Design Environment (SCE) is based on a philosophy of automating tedious, error-prone manual tasks of the design process while benefitting from and reaping human experiences, knowledge and insight whenever applicable. To maximize productivity, relative strengths and weaknesses of humans and computers are balanced and exploited. The goal of SCE is not to implement a fully automated push-button solution but to keep the designer in the loop by providing the necessary transparency, controllability, and observability of the design process at any stage of the flow.

Towards this goal, SCE is based on a separation of synthesis and design into a decision-making process on the one hand and a process of executing or implementing a decision on the

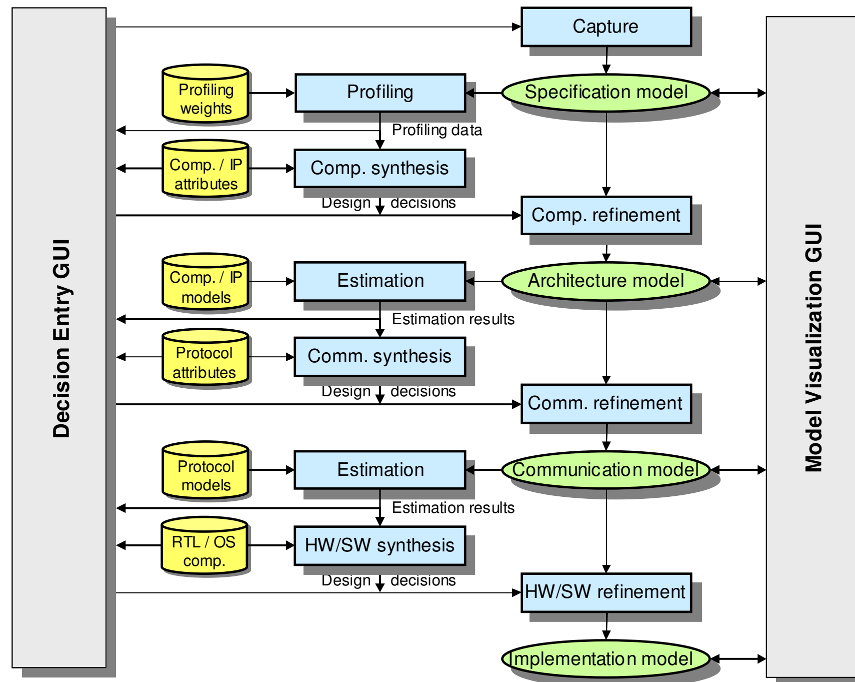


Figure 2.11: SoC Design Environment (SCE).

other hand. Decision making is an interactive and iterative process. SCE provides feedback about critical design issues that aid and steer the user in the decision making process at all stages of the design flow. Execution of design decisions, however, is automated. SCE implements decisions through fully automatic model refinement, avoiding the need for time-consuming rewriting of models completely. As a result, turn-around times are almost instantaneous and a large number of design decisions can be explored in a short amount of time.

An overview of the SoC Design Environment (SCE) is show in Figure 2.11 [1, 46]. In the following sections, we will outline how SCE supports the system design process with varying degrees of design automation for modeling, refinement, exploration and finally fully automated synthesis. After step-by-step descriptions of each design task in the flow, details about the actual implementation of the design environment will be shown in Chapter 7.

2.5.1 Modeling

At the core of the design environment are the four system design models at four levels of abstraction from specification down to implementation. The environment allows the designer to capture, visualize and manipulate the four SoC models through the modeling part of its framework

and user interface. At any step in the design flow, the designer can browse the design hierarchy, edit the source code of the design, or manually refine the design by adding, deleting or changing design objects. Furthermore, design models can be validated through simulation at any time through the compiler and simulator built into the environment.

2.5.2 Refinement

The environment supports automatic refinement of design models following the steps of the design methodology by integrating corresponding model refinement tools. Given design decisions entered by the designer through the user interface or through manual annotation of design models, the refinement tools will automatically generate design models at the next lower level of abstraction that implement and represent the chosen design. Therefore, no manual rewriting of models is ever necessary and only the specification model at the input of the flow has to be supplied. All other models will be derived from this golden model automatically.

2.5.3 Exploration

In order to support the designer in the decision making process and to enable fast design space exploration, early and rapid feedback about a variety of design quality metrics is needed. To this effect, the design environment incorporates additional profiling and estimation tools that perform corresponding analysis of system design models.

Profiling and estimation data computed by the corresponding tools is presented and visualized to the user through the graphical user interface. The user interface compiles, organizes and summarizes data provided by the tools into tables, bar and pie charts. Thereby, it facilitates directed queries of critical design data to identify hotspots, isolate problems, and evaluate trade-offs.

2.5.4 Synthesis

In the final stage, the design environment supports fully automated synthesis by allowing the designer to employ algorithms for automated decision making through corresponding synthesis tools. Synthesis tools automatically generate decisions for input into refinement using internal optimization algorithms. A synthesis tool operates on a given part of a design model including annotated meta-data like estimation metrics or constraints, runs an algorithm to automatically explore the design space in order to find an optimal solution, and back-annotates the resulting implementation decision into the design model.

Through the user interface, the designer is given the option to selectively apply synthesis tools to all or part of the design at any time. With the help of synthesis tools, the designer can save tedious work, rapidly explore parts of the design space, receive hints about possible design alternatives, and ultimately increase productivity. The user can apply different algorithms to different parts of the design, apply non-optimal algorithms to non-critical parts of the design, or make partial decisions and let algorithms fill in the blanks. At any time, however, the use of automated decision making is under the control of the designer, and via the user interface, designers are always able to observe, modify, and override decisions made by the algorithms before executing them through refinement.

2.6 Summary

In this chapter, we defined the system design flow and corresponding design methodology that are the basis for implementation of an automated system design environment. The design methodology divides the system design process into three major design tasks of computation design, communication design and backend design. Specification, architecture, communication and implementation models at the inputs and outputs of the design tasks have been defined in detail. Finally, the SoC Design Environment (SCE) that implements and automates the proposed the design flow has been introduced.

In the following chapters, we will provide detailed descriptions and specifications for each of the design tasks within the design environment. Specifically, the flow of input, output and intermediate models that breaks each tasks into individual design steps will be given in a well-defined manner. As such, the remainder of the dissertation serves as a specification for implementation of each design task within the environment and of the environment itself.

The proposed design flow and design environment form a comprehensive approach at raising the level of abstraction in embedded systems design, supporting both computation and communication abstraction. The definition of the design flow is based on a separation of concerns that minimizes interactions between levels, reduces refinement between models, and supports rapid, early exploration of critical design issues with a variety of components and IPs. As results will prove (see Chapter 8), the two-step approach to the system design flow supports rapid design space exploration by focusing on critical decisions at early stages while providing quick and reliable feedback.

The contributions of this chapter are definitions of necessary and sufficient design tasks and design models in the design flow in a formalized way. The four models define the general framework on which the detailed design flow is based. Models are sufficient in the sense that they cover the complete flow from specification to implementation. On the other hand, intermediate models are necessary to break the flow into manageable tasks and to bridge the semantic gap based on orthogonality of computation, communication and cycle-accurate design concepts. Furthermore, intermediate models are needed for rapid, early exploration by trading off unnecessary implementation detail to provide quick feedback about important overall system design issues. Finally, formalization of the models is required to enable interoperability and design automation. In summary, a formalized framework of models and transformations based on the definitions presented in this chapter is the foundation for the vertical integration of models through synthesis and verification within the design environment.

Chapter 3

System Specification

The system specification model is the starting point of the design flow and is captured by the designer to describe the desired system functionality and associated requirements. All other design models will be generated automatically from the specification model through a sequences of interactive refinement steps. As such, the specification model needs to precisely and unambiguously describe the desired system behavior. Furthermore, the specification model defines the possible design space for exploration. Therefore, quality of implementation results depends to a large extent on the characteristics of the specification model.

In this chapter, we define how to describe a valid system specification that can serve as the input to the design flow [48]. First, a set of general guidelines for writing proper specification models will be given in Section 3.1. Then, in Section 3.2 specific and detailed rules and restrictions imposed on the specification model style are defined. Finally, Section 3.3 shows an example of a specification model of a real system that will serve as the design example throughout the rest of the dissertation.

3.1 Modeling Guidelines

A key aspect of the specification model is to separate computation from communication. On the one hand, this is a requirement for composability of a system out of components including the reuse of pre-existing IP components. On the other hand, this separation of concerns allows to implement computation and communication in two separate steps of the design flow.

3.1.1 Computation

In terms of computation, the specification is hierarchically composed of so-called behaviors. Behaviors are arranged sequentially, concurrently, or in a mix of both, i.e. in a pipelined fashion. Behaviors at the leaves of the hierarchy contain basic algorithms in the form of straight-line C code that perform arithmetic and logical operations on data. In addition to temporary data, leaf behaviors will encapsulate any permanent storage required by the algorithm.

3.1.1.1 Granularity

The basic, indivisible units of granularity for design space exploration are SpecC behaviors. That is, during the design process the specification will be partitioned along behavior boundaries but behaviors at the leaves of the hierarchy form the smallest, indivisible units for exploration. Therefore, leaf behaviors contain basic algorithms in the form of C code, reading from their inputs, processing a data set, and producing outputs.

Algorithms of the specification model are split into leaf behaviors along the boundaries defined between reading and writing of data structures. On the other hand, all the code needed to process a complete, consistent data set should be kept together in one leaf behavior.

Also, the ratio of communication to computation should be minimized yet the size of the leaf behaviors be kept small and manageable with well-defined, sensible interfaces and possible reuse in mind. As a rule of thumb, what would be a traditional C function will become a leaf behavior with typically half a page to maximally two pages of code.

3.1.1.2 Hierarchy

At each level of hierarchy, the system should be composed of self-contained blocks with well-defined interfaces enabling easy composition, rearrangement, and reuse. Closely related functionality is grouped through hierarchy. Higher-level behaviors encapsulate tightly coupled groups of subbehaviors such that the ratio of external to internal communication is minimized. On the other hand, the number of subbehaviors per parent should be kept small and manageable. As a guideline, behaviors typically have 2-5 children on average.

At each level, the behavior hierarchy should be clean. Different behavioral concepts should not be mixed within the same level. A behavior is either a hierarchical composition of subbehaviors or a leaf behavior with sequential code. Similarly, a hierarchical behavior is either a

sequential, parallel, pipelined or FSM composition of subbehaviors but does not contain arbitrary C code.

3.1.1.3 Encapsulation

In general, information should be localized as much as possible. This includes code (functions, methods), storage (variables), and communication (port variables, channels). Each hierarchical unit (behavior) encapsulates and abstracts as many local details as possible, hiding them from higher levels. Hierarchical behaviors encapsulate dependencies and communication of a group of subbehaviors, providing only an interface to their combined functionality.

At the leaves, behaviors encapsulates all the code and storage needed by the algorithm. As mentioned above, global, static variables become member variables of the leaf behavior. Furthermore, global functions that are called out of leaf behaviors should be avoided. Instead, depending on size and number of callers, consider converting functions into separate leaf behaviors that get instantiated as subbehaviors of the caller. Otherwise, global functions can be moved into the calling behavior where they become local methods. An exception are small helper functions with a few lines of code that are used ubiquitously and can be considered basic operations (on the same level as additions or multiplications).

3.1.1.4 Concurrency

Any concurrency available between independent behaviors should be exposed through their parallel or pipelined composition. That is, all behaviors that do not have any control or data dependencies (or data dependencies only across iterations) should be arranged to execute in a concurrent fashion. Furthermore, the behavior hierarchy should be constructed in such a way as to maximize the number of independent behaviors and hence the available parallelism.

Dependent behaviors, on the other hand, should generally not be arranged in a concurrent fashion. Instead, their dependencies should be captured explicitly through transitions. An exception are, for example, rare (control) dependencies between otherwise highly independent top-level tasks. In those cases, communication and synchronization are modeled using channels between the tasks.

In general, concurrent behaviors in the specification model should reflect the available parallelism in the specification. Therefore, they should be as independent as possible. Data or control dependencies between behaviors at the specification level should be explicitly captured through the behavior hierarchy. Instead of concurrent behaviors that communicate or synchronize through

variables or events, the behaviors should be split into independent parts that can run in parallel and dependent parts that have to be executed sequentially.

3.1.1.5 Time

The specification model is untimed and all behaviors execute in zero logical time. Therefore, the only events in the system are events for synchronization in order to specify causality. The ordering of events in the system is based on causal relationships only and there is no notion of time. The system is partially ordered based on causality as determined by the explicit or implicit dependencies between behaviors. As the design flow progresses, timing information that will be added to the system will successively introduce additional order based on delays.

Apart from the untimed behavior, however, the specification model can contain constraints for execution times of parts of the specification. During the design process, it has to be assured that any delays introduced into the model do not violate any of the constraints.

3.1.2 Communication

In terms of communication, exchange of data between behaviors in the specification model is encapsulated into SpecC channels that connect behaviors through ports. Channels describe how data and synchronization messages are transfered between two communication partners in an abstract way.

3.1.2.1 Semantics

In general, behaviors at the specification level communicate via message-passing channels. Behaviors exchange data by sending and receiving messages over communication channels with appropriate semantics. In the case of a sequential composition, message-passing degenerates to simple variables. Data is exchanged by reading from and writing to the variable. In the case of a parallel composition with simple synchronization only, the synchronization is implemented via a single event. In the general case of data communication between concurrent behaviors, however, a message-passing channel is instantiated.

The specification model instantiates channels out of a SpecC channel library with predefined, known semantics. The library contains channels with abstract communication semantics like buffered and unbuffered message-passing, FIFOs, shared-memory semaphores/mutexes, and so

on. By using the predefined channels out of the library, commonly needed communication functionality is available for integration into the specification model.

Note that the specification models of channels do not imply any specific implementation of their abstract semantics. The code inside the channel is for simulation of the correct semantics during execution only. It is the task of communication design to refine those abstract channels into an actual implementation of the desired semantics using the available system bus protocols and PE interfaces.

3.1.2.2 Dependencies

Data dependencies should be reflected explicitly in the behavioral hierarchy as transitions between behaviors, either through a sequential composition or conditionally using the `fsm` statement. In this case, channels degenerate to simple variables connecting behaviors, and the need for implicit synchronization through message-passing is eliminated.

All dependencies are explicitly captured through the connectivity between behaviors and no hidden side effects exist. Global variables should be avoided completely. Static variables accessed from a single leaf behavior become member variables of that behavior. Global variables used for communication have to be turned into explicit dependencies in the form of connectivity as behaviors are only allowed to exchange data through their ports.

If the relationship of concurrent behaviors in the specification model extends beyond synchronization through pure events and necessitates some actual form of data communication, the specification needs to clearly separate such communication from the normal computation by encapsulating communication functionality in the form of channels.

3.2 Modeling Style

In general, the specification input model is written in SpecC and as such has to adhere to the syntax and semantics of the SpecC language [27]. However, to form a valid specification model that can be input into the design flow, additional rules and restrictions on top of the SpecC base have to be adhered to as defined in this section. Note that unless otherwise noted here, any valid SpecC code is an acceptable specification model.

Figure 3.1 and Listing 3.1 show an example template for a valid specification model. A specification model has to be an executable SpecC model, i.e. it has to define a `Main` behavior

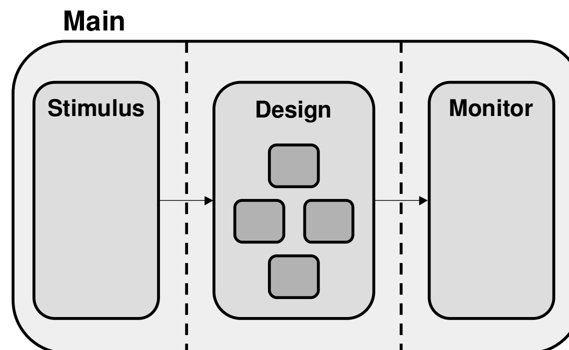


Figure 3.1: Specification model top-level structure.

```
   import "c_double_handshake";

   behavior Stimulus(i_sender input) {               // Stimuli creator
     void main(void) {
 5     // while (...) { ... ; input.send(...) ; ... }
     }
   };

   behavior Monitor(i_receiver output) {             // Output monitor
10   void main(void) {
       // while (...) { ... ; output.receive(...) ; ... }
     }
   };

15 behavior Design(i_receiver input, i_sender output) {  // System design
     // ...

     void main(void) {
       // fsm { ... }
20   }
   };

   behavior Main() {                                 // Top level
     c_double_handshake input, output;
25
     Stimulus  stimulus(input);
     Design    design(input, output);
     Monitor   monitor(output);

30   int main(void) {
       par {
         stimulus.main();
         design.main();
         monitor.main();
35     }
     }
   };
```

Listing 3.1: Specification model top-level code.

(line 23). Usually, a specification model consists of a testbench that surrounds the actual design to be implemented. Typically, a testbench consists of stimulating (`Stimulus`, line 3) and monitoring (`Monitor`, line 9) behaviors that are executing concurrently to the actual design (`Design`, line 15) in the top-most `Main` behavior, and that drive the design under test and check the generated output against known good values.

The design to be implemented is defined by a single SpecC behavior (`Design`) which in turn can be hierarchically composed out of a tree of subbehaviors. For a valid specification model, all the behaviors that are part of this tree have to comply with the rules and restrictions for describing computation and communication that will be defined in the following sections. Note, however, that these restrictions do not apply to the testbench part. Therefore, the testbench can be freely described using any valid SpecC code. For example, while the code of the design to be implemented has to be available completely in SpecC source form, the testbench can link against external translation units (libraries) for additional functionality.

3.2.1 Computation

The computational part of the specification is described through the execution semantics of the hierarchy of SpecC behaviors that form the design to be implemented. For a valid specification model, this behavior hierarchy has to be clean. A clean hierarchy is defined as a tree of behaviors in which every behavior is either a leaf behavior or a hierarchical composition of subbehaviors as defined in the following sections.

3.2.1.1 Leaf Behaviors

In each leaf behavior, the behavior `main()` method contains a piece of straight-line, plain ANSI-C code. Specifically, the following rules define the restrictions that apply to leaf behaviors.

Rule 3.1 *A leaf behavior must not contain any channel or behavior instances. It can, however, contain instances of variables.*

Rule 3.2 *A leaf behavior has exactly one method, the* `main()` *method.*

Generally, the `main()` *method contains any plain, valid ANSI-C code. Of the SpecC-specific types, expressions and statements, only the following are permitted:*

(a) calls to channel methods through behavior ports of interface type (see also Section 3.2.2.1),

(b) `notify` *and* `wait` *statements on behavior ports of* `event` *or* `signal` *type,*

(c) declarations of and operations on variables or ports of `bit`*,* `long long`*,* `long double`*, or* `bool` *basic type, and*

(d) `do-timing` *constructs to specify constraints.*

Rule 3.3 *For leaf behaviors that should be implementable in hardware, depending on the capabilities of the backend tool used for hardware design, additional restrictions might apply (e.g. most tools can not synthesize pointers).*

If any of these restrictions are violated, the corresponding leaf behavior will be limited to a software implementation. In order to allow the greatest possible flexibility for exploration, these restrictions should be followed as much as possible for all leaf behaviors.

Rule 3.4 *Generally, leaf behaviors can make calls to global functions. However, leaf behaviors that call global functions can only be mapped to PEs that provide a native implementation of each global function in the processor library (i.e. as a link-level library in software or a dedicated functional unit in hardware). Therefore, global functions should be avoided completely as much as possible.*

3.2.1.2 Hierarchical Behaviors

A hierarchical behavior is a composition of several subbehavior instances in a sequential, parallel, pipelined or FSM fashion. More specifically, the following rules must be followed when composing hierarchical behaviors.

Rule 3.5 *A hierarchical behavior has exactly one method, the* `main()` *method, and the* `main()` *method contains exactly one statement that is either*

- *a* `seq`*,*
- *a* `par`*,*
- *a* `pipe`*, or*
- *a* `fsm` *statement.*

Rule 3.6 *A hierarchical behavior generally contains instances of subbehaviors that execute inside the hierarchical behavior's composition statement (by calling subbehavior* `main()` *methods).*

However, each subbehavior instance can be called at most once inside the composition. Subbehavior instances communicate through ports, variables and channel instances of the hierarchical behavior mapped to subbehavior ports (see Section 3.2.2.2).

Rule 3.7 *For the expressions in the arguments of a `pipe()` statement and in the `if()` statements of `fsm` transitions the same rules and restrictions as for the C code in leaf behaviors (Section 3.2.1.1) apply.*

3.2.2 Communication

All communication in the specification model, both inside the design to be implemented and between the testbench and the actual design, is described through variables and channels that connect ports of behaviors.

3.2.2.1 Behavior Interfaces (Ports)

The list of ports of a behavior defines the interface between the behavior and its environment, i.e. behaviors are only allowed to communicate with other behaviors through their ports.

Rule 3.8 *Behaviors can have ports of standard (variable with direction) type or of interface type. In case of standard ports, ports that are of pointer type are not allowed. For ports of interface type, only interfaces that are part of the standard SpecC channel library are allowed.*

Rule 3.9 *Behaviors are not allowed to export any methods, i.e. they cannot implement any interfaces.*

Rule 3.10 *Behaviors are not allowed to (directly or indirectly, e.g. through a call to a global function) access variables and channels that are outside of their local scope. Therefore, code inside behaviors can only reference variables or call methods of interfaces that are defined inside the behavior as ports or local instances. Hence, accesses of global variables or channels are forbidden.*

3.2.2.2 Connectivity (Variables and Channels)

Inside hierarchical behaviors, the connectivity of subbehaviors instances is defined by mapping ports of the hierarchical behavior or instances of variables and channels onto the ports of the subbehaviors.

Rule 3.11 *Ports of subbehavior instances inside a behavior can only be connected to the ports of the parent behavior or to variables or channels instantiated inside the parent. Hence, it is not allowed to map other subbehavior instances onto a subbehavior port.*

Rule 3.12 *Given the restrictions on standard port types (see Section 3.2.2.1), variables used for connections (i.e. mapped to ports) must not be of pointer type.*

Rule 3.13 *Variables with storage class* `piped` *are only allowed inside hierarchical behaviors with a* `pipe` *composition (see Section 3.2.1.2) to connect subbehaviors that act as pipeline stages.*

Rule 3.14 *Only channels out of the standard SpecC channel library may be instantiated and mapped to ports.*

3.3 Design Example

An example of a system specification model is shown in Figure 3.2. The design being used as an example is a simplified mobile phone baseband application which combines a JPEG encoder for processing of digital still pictures taken by a camera and a voice encoder/decoder (Vocoder) for speech processing on the transmission path. The JPEG encoder is an implementation of the corresponding lossy image compression standard [7]. The Vocoder, on the other hand, implements the speech transcoding standard that is part of the GSM set of standards for mobile telephony [30]. At the top level, the baseband system specification model therefore consists of concurrent functional blocks for the JPEG encoder and the Vocoder. A channel *Ctrl* between the two blocks is used to send control messages from the JPEG encoder to the vocoder.

The *JPEG* encoder [14, 106] is triggered by an external control signal. After initialization of encoding tables and image headers in *JPEGInit*, *JPEGEncode* encodes the incoming picture in a double-nested pipeline. In the outer loop, stripes of raw pixels are received from the external camera and encoded. The inner loop consists of a four-stage pipeline that encodes each stripe in blocks of 8x8 pixels through discrete cosine transformation (*DCT*), *Quantization*, and *Huffman* encoding.

The voice encoder/decoder (*Vocoder*) [99, 52, 51], on the other hand, internally runs encoding and decoding behaviors in parallel, assisted by four pre- and post-processing behaviors for framing, conversion, etc. On the encoding side, voice samples from the microphone are received (*SpchIn*) as frames of 160 samples corresponding to 20 ms of speech. In the *Coder*, linear prediction analysis (*LP_Analysis*) and open loop analysis (*Open_Loop*) are performed on a frame basis.

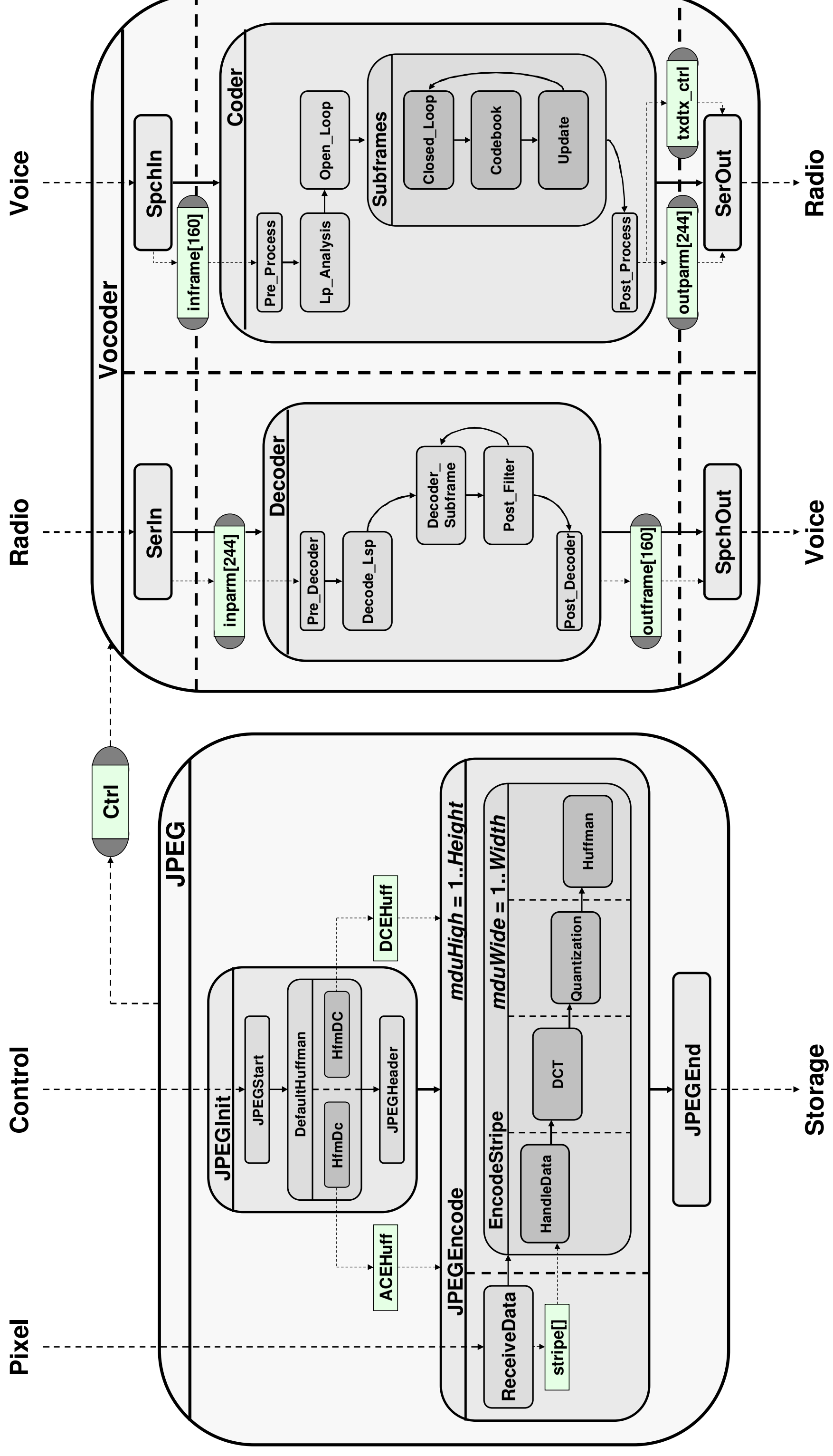


Figure 3.2: System design example specification model.

Each frame is further subdivided into 4 subframes of 40 samples (5 ms of speech) each for closed loop analysis (*Closed_Loop*), codebook search (*Codebook*) and filter update (*Update*) on a subframe basis. At the output of the encoder (*SerOut*), 244 bits of speech parameters are produced for each incoming frame of speech. On the decoding side, the vocoder receives packets (244 bits) of speech parameters (*SerIn*) and the *Decoder* synthesizes speech on a frame and subframe basis in a reverse process to produce speech frames (160 samples) at the output (*SpchOut*).

In the rest of the dissertation, we will use this example design to illustrate the design flow from specification down to implementation. As the design process moves along, the specification model will gradually be refined by adding more and more implementation detail until the bus-functional implementation is reached.

3.4 Summary

In this chapter, requirements, guidelines and rules for specification of systems have been given. Formal, executable specifications captured in the form of SLDL code are the starting point for deriving implementations from the specification in a systematic way [75]. After outlining general guidelines for specifying desired system functionality in a clear, unambiguous and organized manner, specific rules for writing SpecC specification models that can feed into the design flow have been defined. Furthermore, an industrial-size design example has been introduced in the form of its specification model that will serve as the guiding example throughout the rest of the dissertation.

The contributions of this chapter are detailed definitions of requirements and rules for specification of systems such that specifications can be analyzed and synthesized with the help of automated design tools.

Chapter 4

Computation Design

Computation design is the first part of the system design process. It derives a system architecture model from the specification model. The purpose of computation design is to implement the computation in the specification on a set of processing elements (PEs) and memories. In the specification, computation is described in an abstract manner as an arbitrary hierarchical composition of behaviors that process data. At the end of computation design, the system architecture is described as a set of non-terminating, concurrent PEs that each execute pieces of sequential code.

In this chapter, we describe and define the different steps of the computation design process. In Section 4.1, an overview of the computation design flow including its subdivision into partitioning and scheduling design tasks is given. Then, individual steps of these two design tasks are described in detail in Section 4.2 and Section 4.3, respectively. As outlined previously (Chapter 2, Section 2.4), the design process is based on a layering of implementation functionality in the design models and with each step a new layer is introduced into the refined design models. Correspondingly, for each step, the details of its layers as applied to the system design example introduced in Chapter 3, Section 3.3 will be shown.

4.1 Overview

As described in the introduction (Chapter 1, Section 1.2), design generally requires that the where (space) and when (time) of functionality has to be decided. As a result (Figure 4.1), computation design is separated into two tasks for partitioning and scheduling. Partitioning and scheduling tasks respectively resolve space and time issues for implementation of the computation in the system.

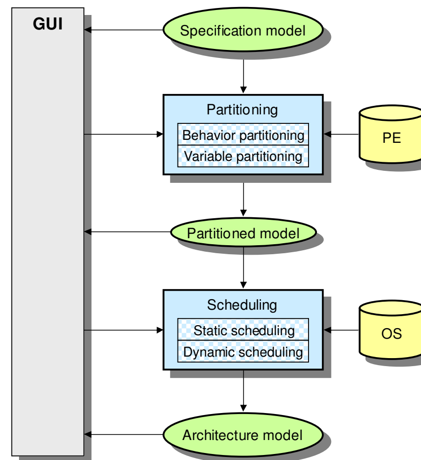


Figure 4.1: Computation design flow.

The computation design process starts with partitioning. During partitioning, system computation as represented by the behaviors and variables in the abstract system specification is implemented on a computation architecture consisting of a set of processing elements (PEs) and memories. Behavior and variable partitioning require allocation and selection of PEs and system memories out of the PE database. Then, behaviors are mapped onto PEs and variables are mapped into local or global memories. Behavior and variable partitioning insert an additional layer of hierarchy in the form of PE and memory behaviors into the design. Original behaviors and variables are then grouped under those PEs and memories according to the selected mapping, additional synchronization and communication is inserted to preserve execution semantics and to exchange updated variable values via message passing, and timing is refined by annotating behaviors with estimated execution delays on their target PEs. In the resulting partitioned design model, the system is represented as a set of non-terminating, concurrent components that communicate at an abstract message-passing level.

After partitioning, scheduling is performed. Due to the inherently sequential nature of PEs, processes mapped to the same PE need to be serialized. Processes are scheduled statically or dynamically depending on the nature of the PE and on the level of data inter-dependencies. During static scheduling, a fixed child order is defined for a subset of concurrent behaviors inside PEs. In the design, the selected behaviors are serialized and statically ordered in time. After static scheduling, each remaining group of concurrent behaviors is dynamically scheduled. Behaviors are converted into tasks on top of an operating system and dynamic scheduling parameters like priorities are assigned to each task.

In case of dynamic scheduling, a representation of the dynamic scheduling implementation is required. In the real system, dynamic scheduling will be handled by a real-time operating system (RTOS). Therefore, a high level model of the underlying RTOS is needed for inclusion into the system model during computation design. Corresponding RTOS models are stored inside an OS database. RTOS models provide an abstraction of the key features that define a dynamic scheduling behavior independent of any specific RTOS implementation.

The scheduling step refines the unscheduled, partitioned system model into the final architecture model. Statically scheduled behaviors inside PEs are sequentialized and their children are rearranged in the selected order. In general, for each PE in the system, an RTOS model corresponding to the selected dynamic scheduling strategy is taken out of the database and instantiated in the PE. Concurrent processes remaining inside the PEs after static scheduling are converted into tasks with assigned priorities. Synchronization as part of communication between processes is refined into OS-based task synchronization.

The resulting architecture model consists of multiple PEs communicating via abstract message-passing channels. Each PE runs one or more tasks on top of its local RTOS model instance. Therefore, the architecture model can be validated through simulation or verification to evaluate different system partitions and different scheduling approaches (e.g. in terms of timing) as part of system design space exploration.

4.2 Partitioning

System partitioning consists of two steps: behavior partitioning and variable partitioning. Behavior partitioning introduces the processing element (PE) layer into the design model which describes the mapping of specification behaviors onto PEs. Variable partitioning, on the other hand, introduces the memory layer which describes the mapping of storage represented by variables into local and global system memories.

4.2.1 Processing Element Layer

The PE layer is inserted into the design model as the result of behavior partitioning. Behavior partitioning is the process of determining on which physical component each piece of specification functionality will be implemented. Behavior partitioning requires allocation of a set of processing elements (PEs) and mapping of specification behaviors onto the allocated PEs. This

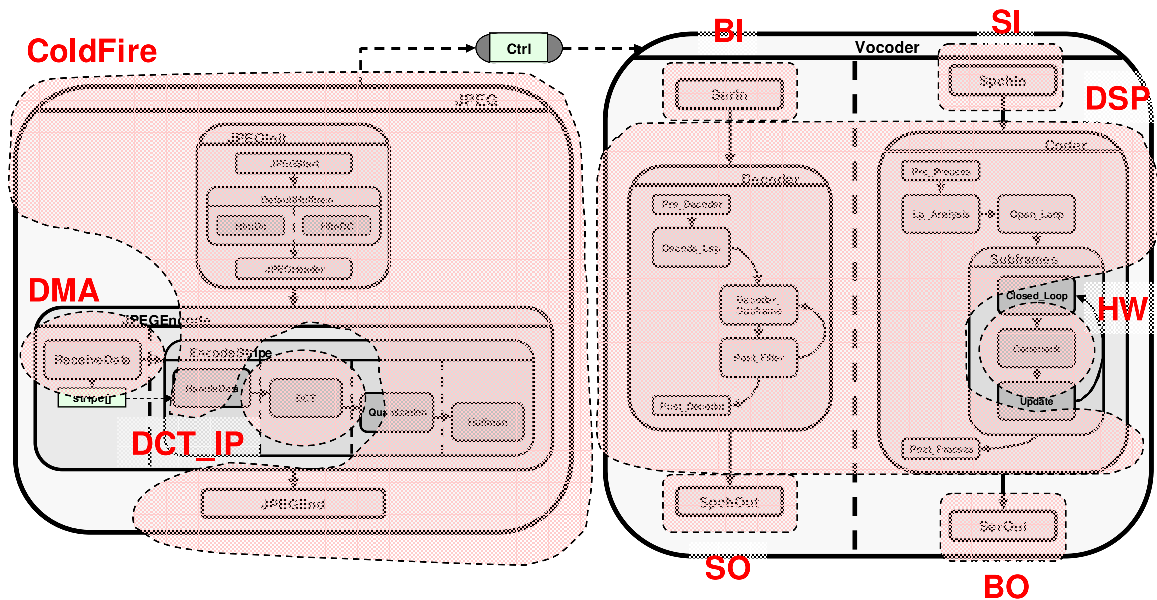


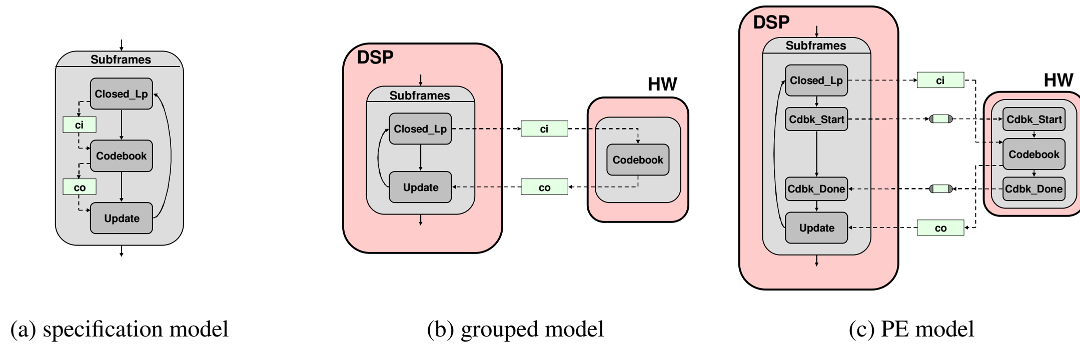
Figure 4.2: Behavior partitioning.

process determines the groups of behaviors that will define the functionality to be implemented by each PE.

A processing element in general is a system component that can perform computation or data processing operations. As such, PEs can be general-purpose programmable software processors, synthesizable custom-hardware processors, or intellectual property (IP) components with fixed functionality. PEs are allocated and selected out of a PE database. At this stage, the design process is only concerned with aspects of the computational functionality of PEs. Therefore, the PE database contains behavioral PE models for allocation and import into the design during behavior partitioning [44][1]. Behavioral PE models define basic characteristics like attributes and parameters of PEs. Furthermore, in case all or part of a PE's computation functionality is pre-defined or fixed (i.e. IP components), its behavioral model will contain code that describes the high-level functional aspects of the PE (see Section 4.2.1.2).

In the design model, PE allocation and behavior mapping is modeled by inserting an additional level of hierarchy—the PE layer—at the top of the behavior hierarchy. At the top level, a set of concurrent behaviors representing the PEs of the system architecture are introduced by importing PE behavioral models out of the database. The leaf behaviors are grouped under those newly added PE behaviors according to the selected mapping, replicating the original behavior

[1]The PE database contains additional models for each PE at lower levels for later stages in the design process, e.g. bus-functional models describing PE interfaces for communication design (see Chapter 5)



(a) specification model (b) grouped model (c) PE model

Figure 4.3: Behavior partitioning refinement.

hierarchy in each PE as necessary. In order to preserve the execution semantics of the original specification, synchronization is added between PEs for each pair of sequential behaviors mapped to concurrent PEs. Finally, in the process of regrouping behaviors, communication between behaviors mapped to different PEs becomes system-global communication and is moved to the top-level that contains the PE behaviors.

Figure 4.2 shows the chosen behavior partitioning for our system design example from Chapter 3, Section 3.3. On the JPEG side, a (Motorola) *ColdFire* PE is allocated and the majority of the JPEG encoding behaviors are mapped onto this PE for execution in software. The *ColdFire* processor is assisted by a custom hardware *DMA* PE which handles receiving of incoming pixel data in parallel to software execution in the *ColdFire* processor. Furthermore, to accelerate encoding, the DCT pipeline stage is mapped onto a predesigned *DCT_IP* intellectual property (IP) hardware PE. On the vocoder side, a digital signal processor *DSP* is allocated to run encoding and decoding of speech in software. The *DSP* is assisted by a custom hardware coprocessor *HW* for acceleration of the codebook search on the encoding side. Furthermore, four custom I/O processors, *BI*, *SI*, *SO*, and *BO* handle framing, buffering, and processing of incoming and outgoing bit and speech streams.

In the following sections, we will describe PE refinement of design models, modeling of IP components and the PE design model as the result of the behavior partitioning process.

4.2.1.1 Model Refinement

Model refinement for behavior partitioning is illustrated in Figure 4.3 using the example of the *Subframes* behavior hierarchy inside the vocoder subsystem's encoder. At the specification level (Figure 4.3(a) and Listing 4.1), the vocoder's top-level design (Listing 4.1(b)) directly executes the vocoder application (Listing 4.1(a)) which executes the *Subframes* behavior inside the *Coder*

```
behavior Subframes()
{
  CI ci;
  CO co;
5
  Closed_Lp closed_lp(cdbk_in);
  Codebook  codebook(cdbk_in, cdbk_out);
  Update    update(cdbk_out);

10  void main(void) {
    closed_lp.main();
    codebook.main();
    update.main();
  }
15 };

behavior Coder()
{
  ...
20  Subframes subframes;

  void main(void) {
    ...
    subframes.main();
25  }
};
```

(a) application

```
behavior Vocoder()
{
  Coder   coder;
  Decoder decoder;
5
  void main(void) {
    par {
      coder.main();
      decoder.main();
10    }
  }
};
```

(b) design

Listing 4.1: Specification model.

hierarchy (line 24). The *Subframes* behavior itself is composed out of *Closed_Lp*, *Codebook* and *Update* subbehavior instances (line 6 through line 8). For simplification, it is assumed that the behaviors communicate via two variables: *ci* (line 3) from *Closed_Lp* to *Codebook* and *co* (line 4) from *Codebook* to *Update*.

As part of the behavior partitioning process explained previously, it has been decided to map the *Codebook* behavior into the custom hardware coprocessor whereas the rest of the vocoder will run on the DSP in software (see Figure 4.2). As part of refinement, behaviors are grouped accordingly under a new layer of corresponding *DSP* and *HW* PE behaviors (Figure 4.3(b)), synchronization behaviors and channels are inserted to preserve execution semantics (Figure 4.3(c)), and leaf behaviors are back-annotated with (estimated) timing information. The result of refinement is the PE model that represents and implements the selected behavior partitioning.

Grouping Grouping inserts the new layer of PE behaviors into the design and replicates the original application behavior hierarchy inside each PE according to the selected mapping (Listing 4.2). At the top level (Listing 4.2(c)), the design is refined into a parallel composition of two newly instantiated behaviors, *DSP* and *HW*, representing the PEs of the system architecture.

```
behavior Sbfrm_DSP(out CI ci, in CO co) {
  Closed_Lp closed_lp(ci);
  Update    update(ci);

  void main(void) {
    closed_lp.main();
    update.main();
  }
};

behavior Coder_DSP(out CI ci, in CO co) {
  ...
  Sbfrm_DSP subframes(ci, co);

  void main(void) {
    ...
    subframes.main();
  }
};

behavior DSP(out CI ci, in CO co) {
  Coder_DSP   coder(ci, co);
  Decoder_DSP decoder;

  void main(void) {
    par {
      coder.main();
      decoder.main();
    }
  }
};
```

(a) *DSP* PE

```
behavior Sbfrm_HW(in CI ci, out CO co)
{
  Codebook codebook(ci, co);

  void main(void)
  {
    codebook.main();
  }
};

behavior Coder_HW(in CI ci, out CO co)
{
  Sbfrm_HW subframes(ci, co);

  void main(void)
  {
    subframes.main();
  }
};

behavior HW(in CI ci, out CO co)
{
  Coder_DSP coder(ci, co);

  void main(void)
  {
    coder.main();
  }

};
```

(b) *HW* PE

```
behavior Vocoder() {
  CI ci;
  CO co;

  DSP dsp(ci, co);
  HW  hw(ci, co);

  void main(void) {
    par {
      dsp.main();
      hw.main();
    }
  }
}
```

(c) design

Listing 4.2: Grouped model.

```
behavior Sender(i_send c) {
  void main(void) {
    c.send();
  }
};
```

(a) sender

```
behavior Receiver(i_receive c) {
  void main(void) {
    c.receive();
  }
};
```

(b) receiver

Listing 4.3: Handshaking synchronization behaviors.

Inside the *DSP* behavior (Listing 4.2(a)), the original application behavior hierarchy of the specification model is replicated. *DSP*-specific copies of the *Subframes* (line 1) and *Coder* (line 11) behaviors are created. Inside the *Sbfrm_DSP* behavior, however, the instance of the *Codebook* behavior has been removed.

Conversely, inside the *HW* behavior (Listing 4.2(b)), the application behavior hierarchy is replicated with only the *Codebook* instance (line 3) inside the *Sbfrm_HW* behavior (line 1). Note that large parts of the behavior hierarchy inside the *HW* PE are empty (e.g. the complete decoder side and other behaviors inside the encoder) and subsequently have been removed.

As a consequence of grouping, variables *ci* and *co* that are shared between behaviors mapped to different PEs become global variables and are now instantiated at the top level (Listing 4.2(c), line 2 and line 3). Leaf behaviors inside each PE are connected to the variables via ports and properly routed port mappings throughout the PE's behavior hierarchy.

Synchronization As a general rule, synchronization needs to be added during the behavior partitioning process in order to preserve the execution semantics of the original specification. Sequential behaviors mapped to different PEs will run in parallel on the concurrent components of the architecture. In order to maintain the proper execution order of behaviors, pairs of synchronization behaviors that communicate via a channel are inserted. For each sequential behavior transition that crosses component boundaries after partitioning, synchronization behaviors are inserted into the behavior hierarchy in the source and destination PEs.

Each synchronization behavior pair implements the semantics of the original transition using a channel with handshaking semantics connecting the behaviors (Listing 4.3). On the source side (Listing 4.3(a)), the behavior sends a handshake over the channel to initiate the transition and to pass control to the other side (line 3). On the destination side (Listing 4.3(a)), the behavior receives the handshake over the channel to wait for the sender before continuing execution (line 3).

In the refined design model (Listing 4.4), two synchronization channels and behavior pairs are inserted for the PE-crossing transition on start and transition on completion of the *Codebook*

```
behavior Vocoder() {
  CI ci;
  CO co;
  c_handshake c_start, c_done;
5
  DSP dsp(c_start, ci, co, c_done);
  HW  hw(c_start, ci, co, c_done);

  void main(void) {
10  par {
      dsp.main();
      hw.main();
    }
  }
15 }
```

(a) design

Listing 4.4: PE model.

behavior. Behavior instances *cdbk_start* on the *Sbfrm_DSP* and *Sbfrm_HW* side (Listing 4.4(b), line 5 and Listing 4.4(c), line 5, respectively) implement the transition from *Closed_Lp* to *Codebook*. Similarly, behavior instances *cdbk_done* (line 6 and line 7) implement the transition from *Codebook* back to *Update* on the *DSP* side.

Handshaking is routed through ports of the behavior hierarchy on each PE up to the top level (Listing 4.4(a)) where the corresponding handshaking channels *c_start* and *c_done* connecting the two PEs are instantiated (line 4). Note that channels implementing one-way handshaking semantics are taken out of the standard SpecC channel library.

Timing After behaviors have been partitioned onto PEs, the concept of time is introduced into the model. The computation represented by the behaviors is refined to include execution times on the target components. As a result, behavior executions are further ordered beyond the pure causality of the specification.

Behavior execution delays can be based on estimated execution times derived from a model of the target component. Alternatively, execution delays can describe a timing budget allocated to each behavior. These budgets will later serve as timing constraints for the behavior implementation on the target PEs.

Execution times can be specified on different levels of granularity, ranging from the statement level to the behavior level. Execution delays at the behavior level are used to model average or worst-case execution times of the corresponding behavior. On the other hand, execution times at the basic-block level can accurately model even data-dependent delays.

```
   behavior Sbfrm_DSP(i_send      c_start,
                      out CI ci, in CO co,
                      i_receive   c_done) {
     Closed_Lp  closed_lp(ci);
 5   Sender     cdbk_start(c_start);
     Receiver   cdbk_done(c_done);
     Update     update(co);

     void main(void) {
10     closed_lp.main();
       cdbk_start.main();
       cdbk_done.main();
       update.main();
     }
15 };

   behavior Coder_DSP(i_send      c_start,
                      out CI ci, in CO co,
                      i_receive   c_done) {
20   ...
     Sbfrm_DSP  subframes(c_start, ci,
                          co, c_done);

     void main(void) {
25     ...
       subframes.main();
     }
   };

30 behavior DSP(i_send      c_start,
                out CI ci, in CO co,
                i_receive   c_done) {
     Coder_DSP    coder(c_start, ci,
                        co, c_done);
35   Decoder_DSP  decoder;

     void main(void) {
       par {
         coder.main();
40       decoder.main();
       }
     }
   };
```

(b) *DSP* PE

```
   behavior Sbfrm_HW(i_receive  c_start,
                     in CI ci, out CO co,
                     i_send     c_done)
   {
 5   Receiver  cdbk_start(c_start);
     Codebook  codebook(ci, co);
     Sender    cdbk_done(c_done);

     void main(void)
10   {
       cdbk_start.main();
       codebook.main();
       cdbk_done.main();
     }
15 };

   behavior Coder_HW(i_receive  c_start,
                     in CI ci, out CO co,
                     i_send     c_done)
20 {
     Sbfrm_HW  subframes(c_start, ci,
                         co, c_done);

     void main(void)
25   {
       subframes.main();
     }
   };

30 behavior HW(i_receive  c_start,
              in CI ci, out CO co,
              i_send     c_done)
   {
     Coder_HW  coder(c_start, ci,
35                   co, c_done);

     void main(void)
     {
       coder.main();
40   }

   };
```

(c) *HW* PE

Listing 4.4: PE model (continued).

```
behavior Update(in CO co) {
  void main(void) {
    ...

5   if(co.x) {
      ...

    }
    ...
10
  }
};
```

(a) specification model

```
behavior Update(in CO co) {
  void main(void) {
    ...
    waitfor(UPDATE_BB1_DELAY);
5   if(co.x) {
      ...
      waitfor(UPDATE_BB2_DELAY);
    }
    ...
10  waitfor(UPDATE_BB3_DELAY);
  }
};
```

(b) PE model

Listing 4.5: Timing refinement.

Execution time is introduced into the refined model by annotating the behaviors with `waitfor` statements (Listing 4.5, line 4, line 7, and line 10). In addition to providing feedback about timing during simulation, the `waitfor` statements serve as input to other validation, verification or analysis tools, e.g. as specification of execution delays for static timing analysis.

4.2.1.2 IP Components

As part of the behavior partitioning process, intellectual property (IP) PEs with predesigned, fixed functionality can be included in the design. IP components are allocated and selected together with other PEs out of the IP database. Out of the list of specification behaviors with matching functionality, a behavior can then be mapped onto the IP for implementation. At minimum, the interface of a matching behavior has to be compatible with the IP interface in terms of data and transitions going in and out of the behavior via its ports.

For each IP component, the PE database contains a behavioral IP model that acts as a black-box simulation model for inclusion into the design during computation synthesis. The behavioral IP model simulates the high-level, data-accurate computational functionality of the IP including estimated or abstracted timing as observed at its external interface. On the other hand, the behavioral IP model should exclude unnecessary implementation details in order to achieve the fastest possible simulation speeds. The IP model is a black-box model used for simulation. Therefore, it only needs to be functionally correct in terms of the input and output values that can be observed at its ports.

Ports of behavioral IP models are of abstract variable or channel type. Behavioral IP models in the database have to have ports on the same level as ports of PE behaviors in the design created during refinement. To validate a behavior-to-IP mapping, the list of ports of a behavioral

IP model is matched against the list of ports that would have been created for a corresponding PE behavior during regular refinement. As part of refinement, the behavioral IP model can then simply be plugged into the design replacing its regularly refined PE behavior.

In general, at each step of the refinement process either a behavioral model out of the database or a PE behavior created during refinement can be inserted into the design to represent an IP. For practical reasons, the IP database usually contains the behavioral model needed for the final partitioned model only (see Section 4.2.2.3). Any intermediate models are generated by refinement.

4.2.1.3 PE Model

The PE model for the given design example at the output of the behavior partitioning process is shown in Figure 4.4. An additional processing element (PE) layer of hierarchy has been inserted. At the top level the design is composed out of PE behaviors running concurrently: *ColdFire*, *DMA* and *DCT_IP* for the ColdFire subsystem running JPEG encoding, and *DSP*, *HW*, *BI*, *SI*, *BO* and *SO* for the DSP subsystem running voice encoding/decoding. Original specification behaviors are grouped under their respective PEs according to the selected mapping. Variables shared between behaviors mapped to different PEs have become global variables connecting PEs. Finally, additional synchronization behaviors (not shown in all cases due to space reasons) and global handshaking channels have been inserted for each PE-crossing behavior transition.

4.2.2 Memory Layer

The memory layer is inserted into the system design model as a result of variable partitioning. Variable partitioning is the process of determining the physical memory in which system variables are stored. Specifically, global variables shared between PEs that have been created during behavior partitioning represent global storage that has to be mapped to actual memories in the system architecture. In addition, local variables inside the PE behavior hierarchies have to be assigned to physical storage.

In general, variables can be mapped to local memories that are part of the PEs or to dedicated global, shared memory components that are allocated as part of variable partitioning[2]. Unless specifically mapped to global memories, local variables inside the PEs are assigned to the local memory of the PE they are instantiated in. For global variables, on the other hand, a mapping has to

[2]Note that in special cases, regular PEs can be implemented to act as shared memories. However, due to the resulting inefficiencies, such implementations are usually not preferred.

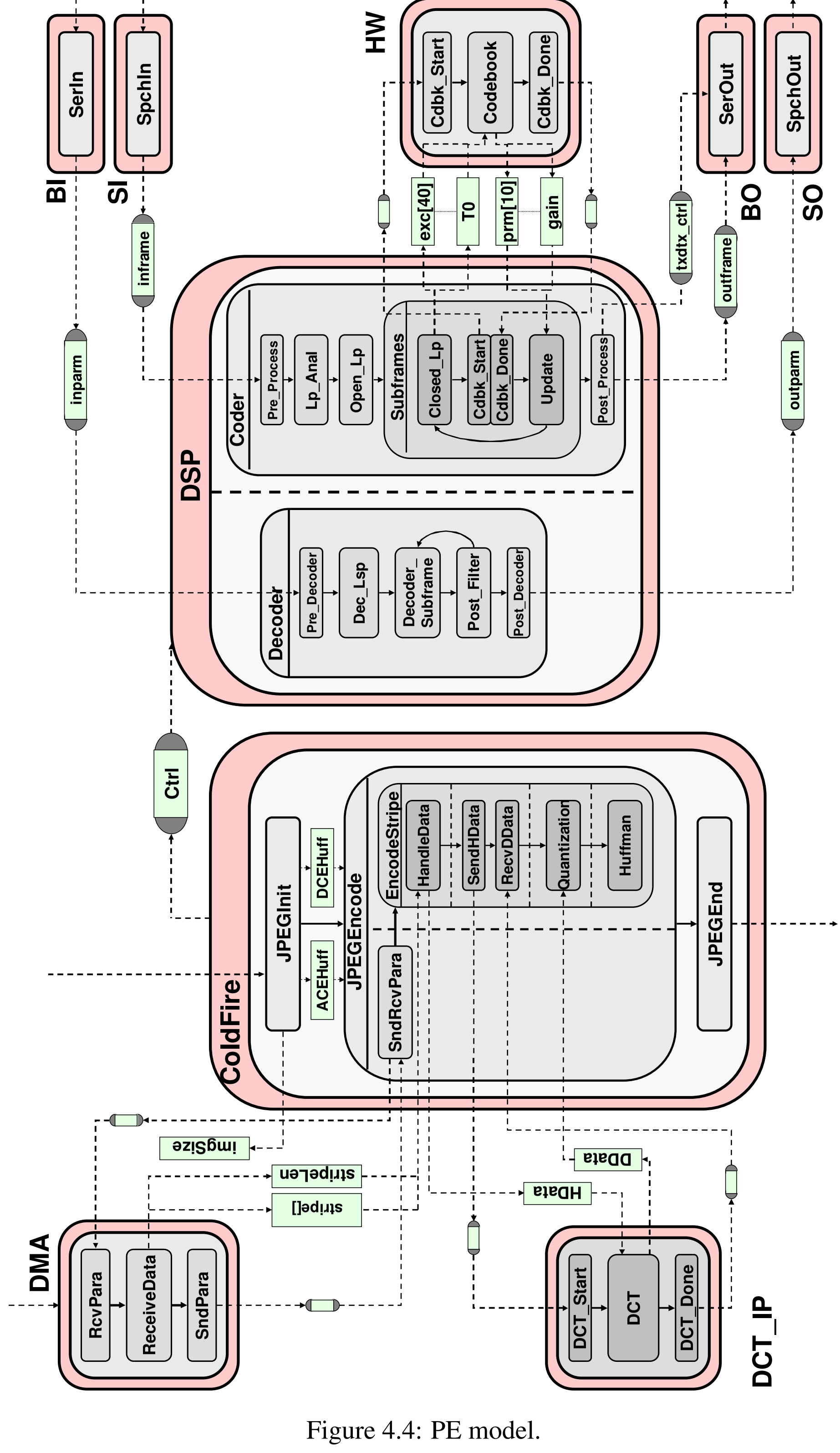


Figure 4.4: PE model.

be chosen in any case. If mapped to a global memory, all PEs accessing the shared variable will be connected to the memory. If mapped to local memory in a so-called distributed variable partitioning, a local copy of the variable is created in each PE accessing the variable. Then, communication is inserted to keep local copies inside the PEs in sync[3].

Memories are allocated and selected out of the PE database where they are stored as special PEs. In general, each PE can contain a local memory as part of its microarchitecture. Behavioral models of PEs in the database define the amount of local memory available in a PE, if any. Therefore, PE behaviors for regular PEs inserted during behavior partitioning double as memory behaviors representing the PE local storage. All variables inside the PE behavior hierarchy will be implemented inside the PE's local memory. For memory components, behavioral models out of the database are imported and instantiated as system memory behaviors during variable partitioning. A behavioral memory model turns the component into a server providing global storage. By exporting its local storage through an interface, other PEs can access the variables inside [44].

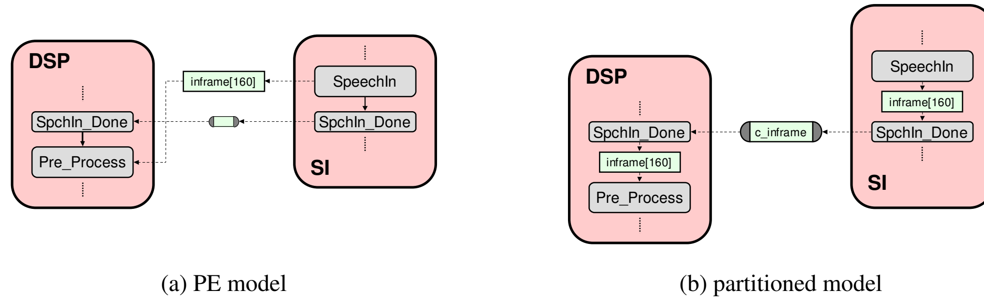
In the design model, variable partitioning is modeled by inserting a memory layer into the system architecture. Memory behaviors representing memory components are added to the system architecture at the top level of the design. Memory behaviors act as containers for variables, and variables are grouped under the memory behaviors according to their selected mapping to physical storage. A memory behavior acts a storage server and other PEs are connected to the memory behavior to access variables stored within. In case of a distributed scheme, variables are moved into local PE memories and existing synchronization channels and behaviors are refined to exchanged updated variable values.

In the following sections, we will illustrate distributed and shared memory variable refinement processes followed by a description of the final partitioned design model at the output of the variable partitioning process.

4.2.2.1 Distributed Variable Refinement

Model refinement for distributed variable partitioning is illustrated in Figure 4.5 using the example of the *inframe* array variable shared between the *DSP* and *SI* PEs. In the PE model (Figure 4.5(a)), as result of behavior partitioning, the *SpeechIn* and *Pre_Process* behaviors have been mapped onto the *SI* and *DSP* PEs, respectively. In the process, the *inframe* variable shared between the two behaviors has become a global variable connecting the two PEs. In addition,

[3]Note that in the case of local variables, the default behavior of mapping them to local memory is equivalent to a distributed variable partitioning scheme. As there is only one PE accessing the variable, no synchronization is necessary.



(a) PE model (b) partitioned model

Figure 4.5: Distributed variable partitioning.

synchronization behaviors *SpchIn_Done* that are handshaking via a corresponding channel between the PEs have been inserted.

In the refined model after distributed variable partitioning (Figure 4.5(b)), local copies of the *inframe* array variable are instantiated inside both *DSP* and *SI* PEs. In order to preserve the shared semantics of the variable and to keep local instances in sync, updated data values have to be exchanged between PEs at synchronization points. Therefore, the existing synchronization behaviors and the existing synchronization channel are refined to communicate updated values of *inframe* together with transferring control over a single message-passing channel *c_inframe* from *SI* to *DSP*.

Mapping During distributed variable partitioning, variables are mapped into local memories of all PEs accessing the variable. A local copy of the variable is created inside each PE connecting to the shared variable. As mentioned previously, behavioral PE models for regular PEs double as containers for variables representing the data stored in the PE's local memory. Therefore, during distributed variable refinement local instances of mapped variables are created inside the connected PE behaviors. Behaviors inside the PEs then operate on the data in the local memory instead of accessing a global variable.

Listing 4.6 and Listing 4.7 show the refinement of the models for the *DSP* and *SI* PEs, respectively. In both cases, a local instance of the *inframe* array is created in the PE behavior (Listing 4.6(b), line 4 and Listing 4.7(b), line 4), replacing the PE behavior port of the same name (Listing 4.6(a), line 2 and Listing 4.7(a), line 2). As a result, subbehaviors inside the PE automatically connect to the newly created local copy of the variable instead of accessing it via the corresponding port. In addition, as will be explained later, synchronization behaviors are replaced with refined versions that connect to the new local variable and communicate updated data values over the external PE interfaces (Listing 4.6, line 5 and Listing 4.7, line 6).

```
   behavior DSP(i_receive si_done,
                in short inframe[160],
                ...)
   {
 5   Receiver    spchin_done(si_done);
     Pre_Process pre_process(inframe);
     ...

     void main(void)
10   {
       spchin_done.main();
       pre_process.main();
       ...
     }
15 };
```

(a) PE model

```
   behavior DSP(i_short160_receive c_inframe,
                ...)
   {
     short       inframe[160];
 5   Receiver    spchin_done(c_inframe,
                             inframe);
     Pre_Process pre_process(inframe);
     ...

10   void main(void) {
       spchin_done.main();
       pre_process.main();
       ...
     }
15 };
```

(b) partitioned model

Listing 4.6: *DSP* PE distributed variable refinement.

```
   behavior SI(i_send si_done,
               out short inframe[160],
               ...)
   {
 5   SpeechIn speechin(inframe);
     Sender   spchin_done(si_done);

     void main(void)
     {
10     speechin.main();
       spchin_done.main();
     }
   };
```

(a) PE model

```
   behavior SI(i_short160_send c_inframe,
               ...)
   {
     short inframe[160];
 5   SpeechIn speechin(inframe);
     Sender   spchin_done(inframe,
                          c_inframe);

     void main(void) {
10     speechin.main();
       spchin_done.main();
     }
   };
```

(b) partitioned model

Listing 4.7: *SI* PE distributed variable refinement.

```
behavior Vocoder() {
  short inframe[160];
  c_handshake si_done;
  ...

  SI  si(si_done, inframe, ...);
  DSP dsp(si_done, inframe, ...);
  ...

  void main(void) {
    par {
      si.main();
      dsp.main();
      ...
    }
  }
}
```

(a) PE model

```
behavior Vocoder()
{
  c_short160_handshake c_inframe;
  ...

  SI  si(c_inframe, ...);
  DSP dsp(c_inframe, ...);
  ...

  void main(void) {
    par {
      si.main();
      dsp.main();
      ...
    }
  }
}
```

(b) partitioned model

Listing 4.8: Top-level distributed variable refinement.

At the top level of the design hierarchy (Listing 4.8), the global *inframe* variable and the synchronization channel *si_done* (Listing 4.8(a), line 2 and line 3) are replaced with a message-passing channel *c_inframe* of appropriate type (short integer array of size 160) that handles synchronization and communication of update variable values in the refined model after partitioning (Listing 4.8(b), line 3). In addition, connectivity of PEs is refined accordingly. As a result, in the partitioned model PEs communicate via channels only and there are no more global variables.

Message Passing In general, the implementation of global variables in a distributed variable partitioning scheme is characterized by moving global variables into local PE memories where the behaviors then operate on those local copies. In order to preserve the original, shared semantics of the global variable, communication to exchange updated data values at synchronization points becomes necessary. Adding communication ensures that guarantees about the state of variables at ports of behaviors are preserved during refinement.

In order to keep local copies in sync, the existing synchronization behavior pairs are modified to include shared variable updates. For each inter-component transition, the corresponding behavior pair will transfer both control and data from the source to the destination end. As part of the message passed over the channel, all the data shared between the behaviors along that transition is included in the transfer.

As shown in the updated code for the synchronization behaviors (Listing 4.9(b)), the refined synchronization behaviors are connected to the local copies of the shared variable through

```
behavior Sender(i_send c)
{
  void main(void) {
    c.send();
5 }
};

behavior Receiver(i_receive c)
{
10 void main(void) {
    c.receive();
  }
};
```

(a) PE model

```
behavior Sender(short inframe[160],
                i_short160_send c) {
  void main(void) {
    c.send(inframe);
5 }
};

behavior Receiver(i_short160_receive c,
                  short inframe[160]) {
10 void main(void) {
    c.receive(inframe);
  }
};
```

(b) partitioned model

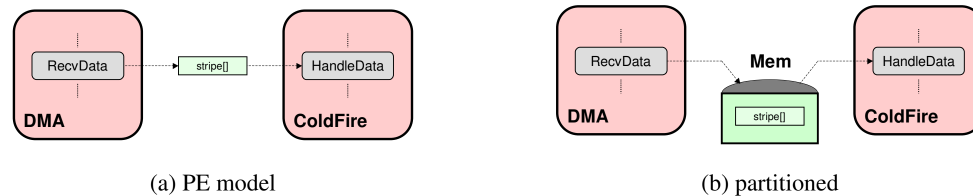
Listing 4.9: Message-passing refinement.

their ports (line 1 and line 9). They assemble and disassemble variable updates into/from the message passed over the channel between them. The source side reads the value of the variable from its input port and sends it over the corresponding channel interface (line 4). On the other end, the destination waits for reception of the message and writes the updated value into the variable through its output port (line 11).

4.2.2.2 Shared Memory Refinement

Figure 4.6 shows model refinement for a shared memory variable partitioning scheme using the example of the *stripe* array variable shared between *DMA* and *ColdFire* PEs in the JPEG subsystem. In the PE model after behavior partitioning (Figure 4.6(a)), the *RecvData* behavior on the *DMA* PE and the *HandleData* behavior on the *ColdFire* PE communicate by accessing the same global *stripe* variable. Note that necessary synchronization for access to the shared resource (either as part of the specification or automatically inserted in the form of synchronization behaviors and channels during behavior partitioning) is not shown here.

During variable partitioning it has been decided to map the *stripe* variable into a dedicated shared memory component that becomes part of the system architecture. As a result, the *stripe* variable—together with other variables mapped to the system memory—is grouped under a newly inserted layer of memory behavior representing the memory component *Mem* (Figure 4.6(b)). The memory behavior acts as a storage server. It exports methods for accessing variables stored inside the memory through an interface. Consequently, variable reads and writes inside the PE's leaf behaviors are refined to access the variable through the memory interface instead.



(a) PE model (b) partitioned

Figure 4.6: Shared memory variable partitioning.

Grouping During shared memory refinement, a new layer of memory behaviors is inserted into the system and global variables between PEs are grouped and encapsulated under these behaviors according to the selected mapping. Listing 4.10 shows the shared memory refinement of the top level of the design hierarchy from the PE model to the partitioned model. Global variables (Listing 4.10(a), line 2) are replaced by instances of system memory behaviors which hold the mapped variables (Listing 4.10(b), line 3) The memory components are inserted into the system architecture next to the other PEs. Instead of accessing global variables, PEs are connected to the shared memory as needed in order to access the variables within (line 4 and line 5).

Memory Behaviors During variable partitioning, behavioral models of dedicated system memory components are allocated and imported out of the PE database. Memory behaviors describe a memory's computational functionality needed at this stage in the design process. A shared memory behavioral model (Listing 4.11) acts as a container and server for variables stored in the memory. Global variables are moved into memories. Iside the memory behavior (Listing 4.11(b)), instances of all variables mapped to the memory are created (line 4).

In order for other PEs to access the variables stored in the memory, the memory behavior exports access methods through a corresponding interface. The interface (Listing 4.11(a)) provides methods for read and write accesses to all data stored in the memory (line 2 and line 3). In general, methods have to be created to allow access to variable data on all possible levels. For example, in case of complex, composite variables (structs, arrays, or unions), access to individual data elements and to whole data blocks (e.g. for struct or array assignments) has to be provided.

The memory behavior implements the interface and its methods by simply mapping each read or write call to a read or write access of the corresponding variable or variable element available in the memory (Listing 4.11(b), line 7 and line 8). Note that a dedicated memory component does not provide any computational functionality. Hence, its PE behavior does not have any ports and its *main* method is empty (line 11).

```
behavior JPEG() {
  char stripe[STRIPESIZE];

  DMA      dma(stripe);
5 ColdFire cf(stripe);
  ...

  void main(void) {
    par
10  {
      dma.main();
      cf.main();
      ...
    }
15 }
}
```

(a) PE model

```
behavior JPEG()
{
  Mem      mem;
  DMA      dma(mem);
5 ColdFire cf(mem);
  ...

  void main(void) {
    par {
10    mem.main();
      dma.main();
      cf.main();
      ...
    }
15 }
}
```

(b) partitioned model

Listing 4.10: Top-level shared memory refinement.

```
interface i_mem {
  char read_stripe(int index);
  void write_stripe(int index, char val);
  ...
5 };
```

(a) memory interface

```
behavior Mem()
  implements i_mem
{
  char stripe[960];
5 ...

  char read_stripe(int index)           { return stripe[index]; }
  void write_stripe(int index, char val) { stripe[index] = val;   }
  ...
10
  void main(void) { }
};
```

(b) memory behavior

Listing 4.11: Shared memory behavioral model.

```
behavior RecvData(char stripe[960]) {
  void main(void) {
    ...
    stripe[i] = d;
5   ...
  }
};

behavior DMA(char stripe[960],
10             ...) {
  ...
  RecvData rcvdata(stripe);
  ...

15 void main(void) {
    ...
    rcvdata.main();
    ...
  }
20 };
```

(a) PE model

```
behavior RecvData(i_mem mem) {
  void main(void) {
    ...
    mem.write_stripe(i, d);
5   ...
  }
};

behavior DMA(i_mem mem,
10             ...) {
  ...
  RecvData rcvdata(mem);
  ...

15 void main(void) {
    ...
    rcvdata.main();
    ...
  }
20 };
```

(b) partitioned model

Listing 4.12: *DMA* PE shared memory refinement.

```
behavior HandleData(char stripe[960]) {
  void main(void) {
    ...
    x = stripe[n];
5   ...
  }
};

behavior ColdFire(char stripe[960],
10                   ...) {
{
  ...
  HandleData hdldata(stripe);
  ...
15
  void main(void) {
    ...
    hdldata.main();
    ...
20 }
};
```

(a) PE model

```
behavior HandleData(i_mem mem) {
  void main(void) {
    ...
    x = mem.read_stripe(n);
5   ...
  }
};

behavior ColdFire(i_mem mem,
10                   ...) {
{
  ...
  HandleData hdldata(mem);
  ...
15
  void main(void) {
    ...
    hdldata.main();
    ...
20 }
};
```

(b) partitioned model

Listing 4.13: *ColdFire* PE shared memory refinement.

Memory Access After grouping global variables into shared memory components, variable accesses inside the PEs have to be refined to access the corresponding memory interface instead. Inside the leaf behaviors of the PEs (Listing 4.12 and Listing 4.13), each access to a global variable in the C code is refined into calls to corresponding read and write methods of the memory interface (line 4). In the process, connectivity of leaf behaviors to global variables through ports of the PE behavior hierarchy is replaced with ports of memory interface type that ultimately connect leaf behaviors to the global system memory where the variables are stored (line 1, line 12 and line 9).

4.2.2.3 Partitioned Model

The resulting partitioned model for the given design example at the output of the variable partitioning process is shown in Figure 4.7. With the exception of the *stripe* variable shared between *DMA* and *ColdFire* PEs on the JPEG side, a distributed variable partitioning scheme has been chosen for all variables in the system. Local variable copies are created inside the PEs and synchronization channels between PEs are replaced by message-passing channels. For the *stripe* variable, a dedicated memory component *Mem* that holds the variable is part of the JPEG subsystem. *DMA* and *ColdFire* PEs connect to the memory interface such that behaviors inside can access the shared data.

For the *DCT_IP* component, a behavioral IP model is imported and inserted out of the IP database. Note that the IP model is based on a distributed partitioning of variables needed for communication with the IP. As described in Section 4.2.1.2, the behavioral model in the database is a black-box simulation model of the IP that is plugged into the system architecture from the database. Based on the distributed variable IP requirements, local copies of variables previously shared with the IP are created inside the connecting *ColdFire* PE. Furthermore, synchronization behaviors are upgraded to full message-passing IP communication.

As a result of variable partitioning, the partitioned model represents the output of the partitioning process in general. Behaviors are partitioned into PEs and variables are partitioned into local and global memories. At the top level of the design, the system is a netlist of PEs connected by stateless channels. Specifically, there are no system-level variables or buffers (i.e. state carried across transactions inside channels). All storage has been mapped into physical memory for implementation. Note that the *Ctrl* channel between subsystems and the channels between the *DSP* PE and its I/O processors were specified to be of stateless message-passing type from the beginning and subsequently did not have to be refined.

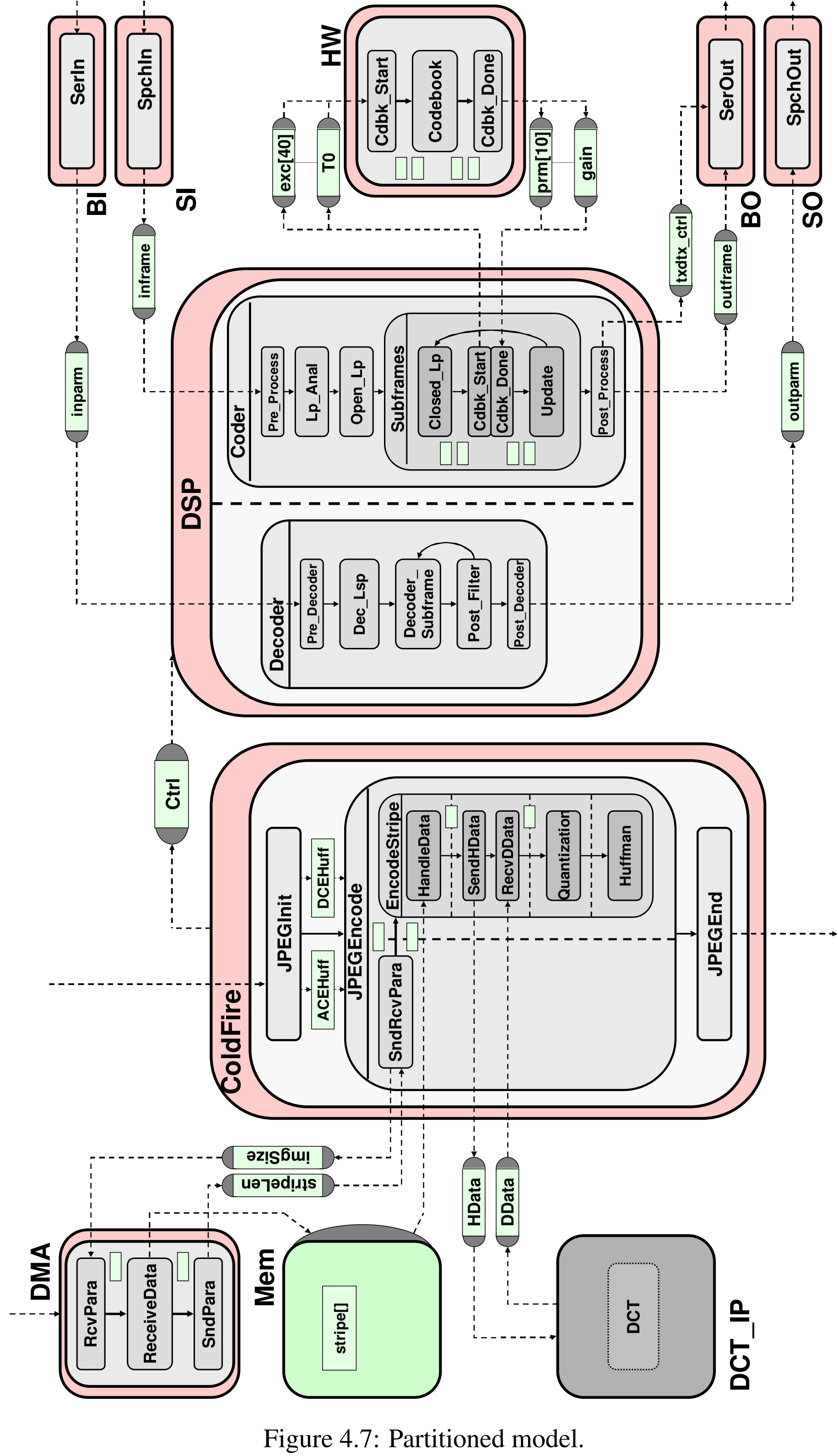


Figure 4.7: Partitioned model.

4.3 Scheduling

After partitioning of behaviors and variables onto PEs and memories, system scheduling has to be performed in order to determine the order of behavior execution on the inherently sequential PEs. PEs are defined to have a single thread of control and true parallelism is available through the concurrent execution of PE components only. Therefore, behaviors inside each PE have to be scheduled to serialize their execution.

Behaviors can be scheduled statically or dynamically. For static scheduling, the order of behaviors is fixed and the design is refined to execute behaviors according to the given order. For dynamic scheduling, the order of behaviors will be determined at runtime under the control of a scheduling algorithm running on the PE. The scheduler will later become part of the embedded real-time operating system (RTOS). As part of dynamic scheduling refinement an RTOS layer is introduced into the design which controls behavior execution and contains an abstracted model of the underlying RTOS scheduling policy.

In the design flow, scheduling is performed in two steps: first, static scheduling of selected parts of the behavior hierarchy in each PE is performed. Then, an RTOS layer is inserted for each supported PE in order to dynamically schedule all remaining concurrent behaviors inside.

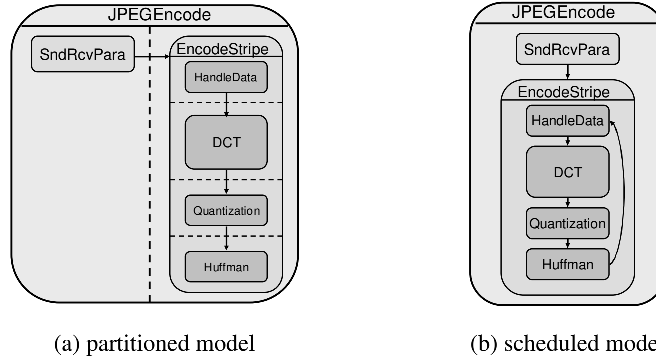
4.3.1 Static scheduling

In a static scheduling approach, behaviors are executed in a fixed and predetermined order. Static scheduling requires the definition of the execution order for a given concurrent composition of behaviors. Any two behaviors running in parallel (i.e. for which their order of execution is undefined) can be statically scheduled. In the process, multiple nested levels of concurrency in the behavior hierarchy can be flattened and behaviors can be scheduled across hierarchical boundaries. The design model is refined to represent static scheduling decisions by arranging behaviors in the selected execution order, possibly removing parts of the behavior hierarchy.

In the following sections, we will illustrate the model refinement process for static scheduling followed by a description of the scheduled design model at the output of the process.

4.3.1.1 Model Refinement

Model refinement for static scheduling is illustrated in Figure 4.8 (Listing 4.14) using the example of the main JPEG encoding block *JPEGEncode*. In the partitioned model before schedul-



(a) partitioned model (b) scheduled model

Figure 4.8: Static scheduling refinement.

```
behavior EncodeStripe(...) {
  piped int b[64];
  ...
  HandleData   hndl(..., b);
  DCT          dct(b, ...);
  Quantization quant(...);
  Huffmann     huff(...);

  void main(void)
  {
    int i;
    pipe(i = 0; i < W; i++)
    {
      hdldata.main();
      dct.main();
      quant.main();
      huff.main();
    }
  }
};

behavior JPEGEncode(...) {
  ...
  SndRcvData   sndrcvData(...);
  EncodeStripe encStripe(...);

  void main(void) {
    par {
      sndrcvdata.main();
      encStrip.main();
    }
  }
};
```

(a) partitioned model

```
behavior EncodeStripe(...) {
  int b[64];
  ...
  HandleData   hndl(..., b);
  DCT          dct(b, ...);
  Quantization quant(...);
  Huffmann     huff(...);

  void main(void) {
    int i = 0;
    fsm {
      hndl:
      dct:
      quant:
      huff: {
        if(++i < W) goto hndl;
      }
    }
  }
};

behavior JPEGEncode(...) {
  ...
  SndRcvData   sndrcvData(...);
  EncodeStripe encStripe(...);

  void main(void)
  {
    sndrcvdata.main();
    encStrip.main();
  }

};
```

(b) scheduled model

Listing 4.14: Static scheduling refinement.

ing (Figure 4.8(a), Listing 4.14(a)), encoding is performed in two nested levels of concurrent subbehaviors. The inner *EncodeStripe* behavior (line 1) encodes a stripe of W 8×8 pixel blocks in a 4-stage pipeline (line 12). The outer JPEG encoder behavior (line 22) then runs encoding of stripes in parallel with receiving pixel data from the outside (line 28).

During the design process, it has been decided to statically schedule the complete JPEG encoding block. In the scheduled model after refinement (Figure 4.8(b), Listing 4.14(b)), the pipelined and parallel behavior compositions are replaced with corresponding sequential execution of subbehaviors in the selected (default) order. The pipeline is converted into a simple loop, using an FSM composition to model looping (line 11). Note that in the process, pipelined variables used for communication between stages are converted into regular variables (line 2). The outer parallel composition is simply replaced by an analogous sequential composition (line 28).

4.3.1.2 Scheduled Model

The refined design model for the given system design example after static scheduling is shown in Figure 4.9. As explained in the previous sections, the main JPEG encoding block inside the *ColdFire* processor has been statically scheduled with the result that all behaviors inside the *ColdFire* PE execute sequentially.

On the *DSP* side, however, no static scheduling has been performed and encoding and decoding blocks still execute concurrently. Therefore, they will be dynamically scheduled as part of the RTOS refinement step that follows static scheduling.

Apart from scheduling behavior executions inside PEs, the design at the top level remains unchanged from the previous partitioned model as a set of concurrent PEs communicating via abstract channels on the message-passing level.

4.3.2 RTOS Layer

As mentioned in the chapter overview, any concurrent processes remaining after static scheduling require dynamic scheduling via an underlying RTOS. At this early design phase, however, using a detailed, real RTOS implementation would negate the purpose of an abstract system model. Furthermore, at higher levels, not enough information might be available to target a specific RTOS. Therefore, an RTOS layer which captures and models the abstracted RTOS behavior is inserted into the system model at this stage [49, 50].

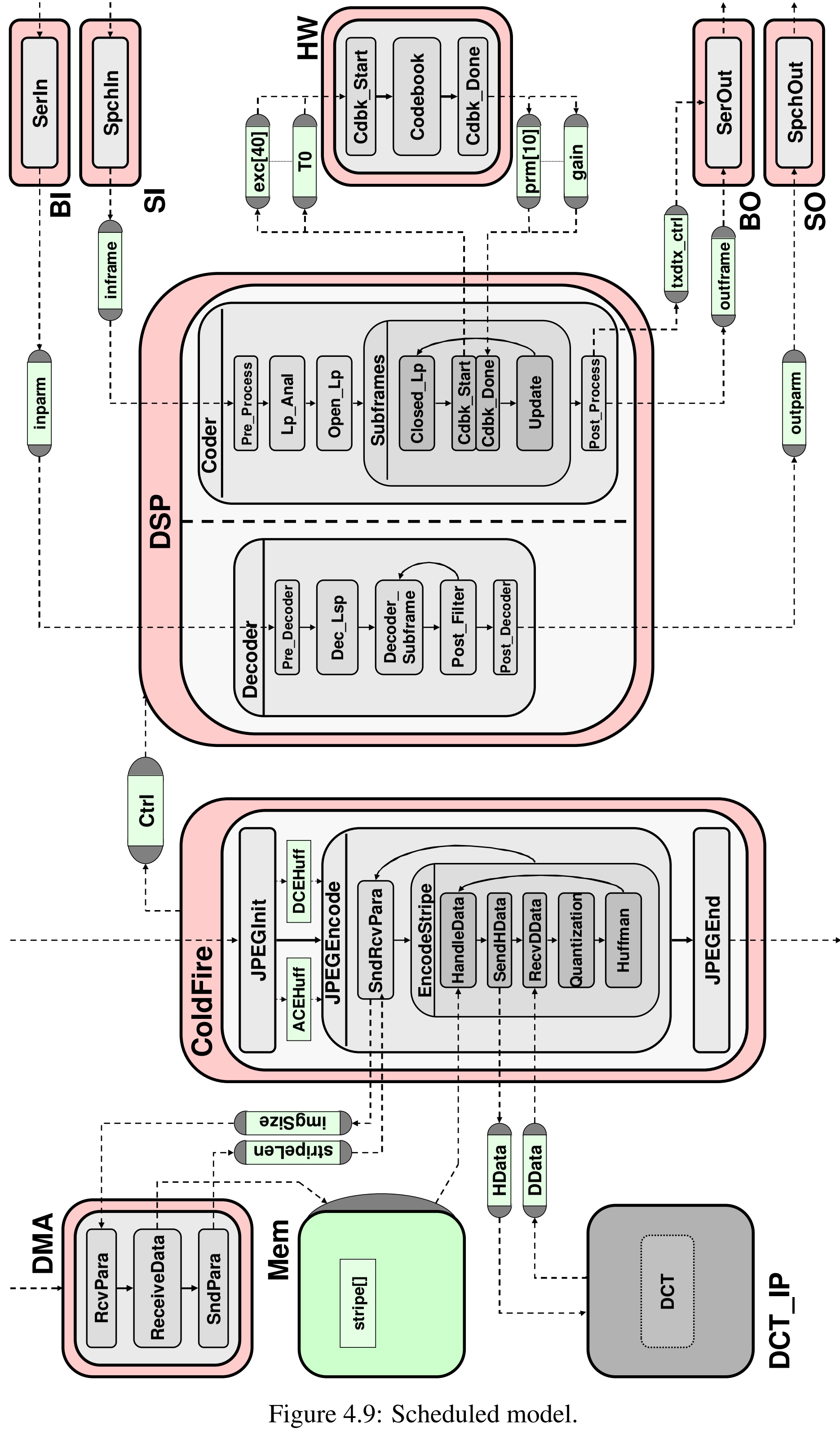
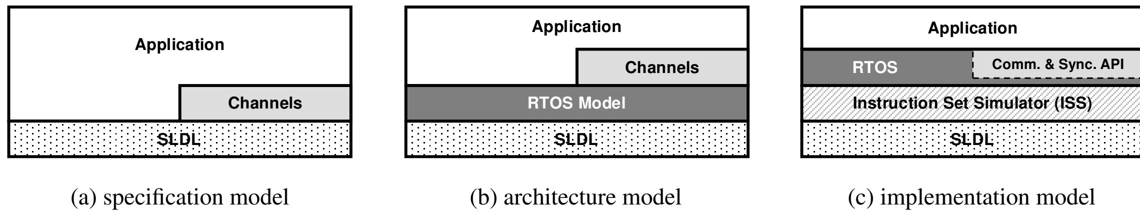


Figure 4.9: Scheduled model.



(a) specification model (b) architecture model (c) implementation model

Figure 4.10: RTOS modeling layers.

The RTOS models is written on top of the existing SpecC SLDL and does not require any specific language extensions. It supports all the key concepts found in modern RTOS like task management, real time scheduling, preemption, task synchronization, and interrupt handling [10]. On the other hand, it requires only a minimal modeling effort in terms of refinement and simulation overhead, thereby allowing to rapidly explore different design alternatives at this early stage in the design process.

Figure 4.10 shows the modeling layers at different steps of the design flow. At the specification level (Figure 4.10(a)), the application is a serial-parallel composition of SLDL processes. Processes communicate and synchronize through variables and channels. Channels are implemented using primitives provided by the SLDL core and are usually part of the communication library provided with the SLDL.

In the architecture model (Figure 4.10(b)), the RTOS model is inserted as a layer between the application and the SLDL core. The SLDL primitives for timing and synchronization used by the application are replaced with corresponding calls to the RTOS layer. In addition, calls of RTOS task management services are inserted. The RTOS model implements the original semantics of SLDL primitives plus additional details of the RTOS behavior on top of the SLDL core, using the existing services of the underlying SLDL simulation engine to implement concurrency, synchronization, and time modeling. Existing SLDL channels (e.g. semaphores) from the specification are reused by refining their internal synchronization primitives to map to corresponding RTOS calls. Using existing SLDL capabilities for modeling of extended RTOS services, the RTOS library can be kept small and efficient. Later, as part of software synthesis in the backend, RTOS calls and channels are implemented by mapping them to an equivalent service of the actual RTOS or by generating channel code on top of RTOS primitives if the service is not provided natively.

Finally, in the implementation model (Figure 4.10(c)), the compiled application linked against the real RTOS libraries is running in an instruction set simulator (ISS) as part of the system co-simulation in the SLDL.

```
interface RTOS
{
    /* OS management */
    void init();
5   void start(int sched_alg);
    void interrupt_return();

    /* Task management */
    Task task_create(const char *name, int type, sim_time period);
10  void task_terminate();
    void task_sleep();
    void task_activate(Task t);
    void task_endcycle();
    void task_kill(Task t);
15  Task par_start();
    void par_end(Task t);

    /* Event handling */
    Task enter_wait();
20  void wakeup_wait(Task t);

    /* Delay modeling */
    void time_wait(sim_time nsec);
};
```

Listing 4.15: Interface of the RTOS model.

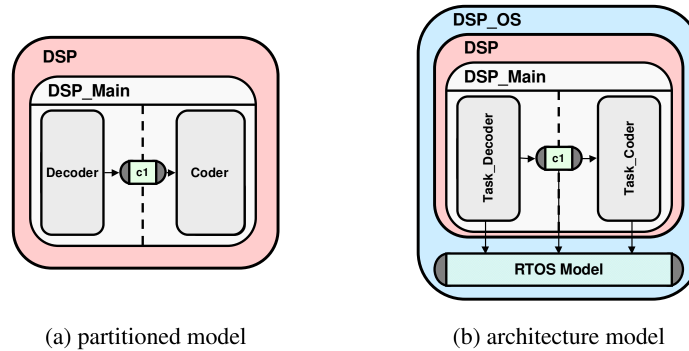
In the following sections we will discuss the interface between application and the RTOS model, the refinement of specification into architecture using the RTOS interface, and the implementation of the RTOS model.

4.3.2.1 RTOS Interface

Listing 4.15 shows the interface of the RTOS model. The RTOS model provides four categories of services: operating system management, task management, event handling, and time modeling.

Operating system management mainly deals with initialization of the RTOS during system start where *init* initializes the relevant kernel data structures while *start* starts the multi-task scheduling. In addition, *interrupt_return* is provided to notify the RTOS kernel at the end of an interrupt service routine.

Task management is the most important function in the RTOS model. It includes various standard routines such as task creation (*task_create*), task termination (*task_terminate*, *task_kill*), and task suspension and activation (*task_sleep*, *task_activate*). Two special routines are introduced to model dynamic task forking and joining: *par_start* suspends the calling task and waits for the child tasks to finish after which *par_end* resumes the calling task's execution. The RTOS model



(a) partitioned model (b) architecture model

Figure 4.11: Model refinement example.

supports both periodic hard real time tasks with a critical deadline and non-periodic real time tasks with a fixed priority. In modeling of periodic tasks, *task_endcycle* notifies the kernel that a periodic task has finished its execution in the current cycle.

Event handling in the RTOS model sits on top of the basic SLDL synchronization events. Two system calls, *enter_wait* and *wakeup_wait*, are wrapped around each SpecC wait primitive. This allows the RTOS model to update its internal task states (and to reschedule) whenever a task is about to get blocked on and later released from a SpecC event.

During simulation of high-level system models, the logical time advances in discrete steps. SLDL primitives (such as waitfor in SpecC) are used to model delays. For the RTOS model, those delay primitives are replaced by *time_wait* calls which model task delays in the RTOS while enabling support for modeling of task preemption.

4.3.2.2 Model Refinement

In this section, we will illustrate model refinement based on the RTOS interface presented in the previous section using the example of the *DSP* PE (Figure 4.11). In general, the same refinement steps are applied to all the PEs in a multi-processor system. The unscheduled, partitioned model (Figure 4.11(a)) executes the parallel composition of *Coder* and *Decoder* behaviors in its main application inside the *DSP* processor. Behaviors *Coder* and *Decoder* communicate via a channel *c1*.

The output of the dynamic scheduling refinement process is shown in Figure 4.11(b). As part of refinement, a new OS layer is inserted around each PE in the system. The RTOS model implementing the RTOS interface is instantiated inside this *DSP_OS* layer in the form of a SpecC channel. Behaviors and communication channels use RTOS services by calling the RTOS channel's

```
behavior Decoder()
{
    void main(void)
    {
5       ...
        /* model execution delay */
        waitfor(BLOCK1_DELAY);
        ...
        /* model execution delay */
10      waitfor(BLOCK2_DELAY);
        ...
    }
};
```

(a) partitioned model

```
behavior task_Decoder(RTOS os) implements Init
{
    Task h;

5   void init(void) {
        h = os.task_create("Decoder", APERIODIC, 0);
    }

    void main(void) {
10      os.task_activate(h);
        ...
        /* model execution delay */
        os.time_wait(BLOCK1_DELAY);
        ...
15      /* model execution delay */
        os.time_wait(BLOCK2_DELAY);
        ...
        os.task_terminate(h);
    }
20 };
```

(b) architecture model

Listing 4.16: Task modeling.

methods. Behaviors are refined into three tasks. *DSP_Main* is the main task that executes as soon as the system starts. Immediately after startup, *DSP_Main* spawns concurrent child tasks, *Task_Coder* and *Task_Decoder*, and waits for their completion.

Task Refinement Task refinement converts parallel processes/behaviors in the specification into RTOS-based tasks in a two-step process. In the first step (Listing 4.16), behaviors are converted into tasks, e.g. behavior *Decoder* (Listing 4.16(a), line 1) is converted into *Task_Decoder* (Listing 4.16(b), line 1). A method *init* is added for construction of the task (line 5). All `waitfor` statements within the task's body are replaced with RTOS *time_wait* calls to model task execution delays (line 13 and line 16). Finally, the main body of the task is enclosed in a pair of

```
behavior DSP_Main()
{
    Decoder decoder();
    Coder   coder();
5
    void main(void)
    {


10
        par
        {
          decoder.main();
          coder.main();
15      }

    }

};
```

(a) partitioned model

```
behavior DSP_Main(RTOS os)
{
    Task_Decoder task_decoder(os);
    Task_Coder   task_coder(os);
5
    void main(void) {
        Task t;

        task_decoder.init();
10      task_coder.init();

        t = os.par_start();
        par {
          task_decoder.main();
15        task_coder.main();
        }
        os.par_end(t);
    }
};
```

(b) architecture model

Listing 4.17: Task creation.

task_activate / *task_terminate* calls so that the RTOS kernel can control the task activation and termination (line 10 and line 18).

The second step (Listing 4.17) involves dynamic creation of child tasks in a parent task. Every `par` statement in the code (Listing 4.17(a)) is refined to dynamically fork and join child tasks as part of the parent's execution (Listing 4.17(b)). The *init* methods of the children are called to create the child tasks. Then, *par_start* suspends the calling parent task in the RTOS layer before the children are actually executed in the `par` statement. After the two child tasks finish execution and the `par` exits, *par_end* resumes the execution of the parent task in the RTOS layer.

Synchronization Refinement In the partitioned model, all synchronization in the application or inside communication channels is implemented using SLDL events. Synchronization refinement wraps corresponding event handling routines of the RTOS model around the event-related primitives (Listing 4.18). Each `wait` statement in the code is enclosed in a pair of *enter_wait* / *wakeup_wait* calls to notify the RTOS model about corresponding task state changes (line 12). Note that there is no need to refine `notify` primitives as the state of the calling task is not influenced by those calls (line 9).

After model refinement, both task management and synchronization are implemented using the system calls of the RTOS model. Thus, the dynamic system behavior is completely controlled by the RTOS model layer.

```
channel C1()
{
    event eReady;
    event eAck;

    void send(...)
    {
        ...
        notify eRdy;
        ...

        wait(eAck);
        ...

    }
};
```

(a) partitioned model

```
channel C1(RTOS os)
{
    event eReady;
    event eAck;

    void send(...) {
        Task t;
        ...
        notify eRdy;
        ...
        t = os.enter_wait();
        wait(eAck);
        os.wakeup_wait(t);
        ...
    }
};
```

(b) architecture model

Listing 4.18: Synchronization refinement.

4.3.2.3 Model Implementation

The RTOS model library is implemented in approximately 2000 lines of SpecC channel code [109]. The library contains models for different scheduling strategies typically found in RTOS implementations, e.g. round-robin or priority-based scheduling [90]. In addition, the models are parameterizable in terms of task parameters, preemption, and so on.

Task management in the RTOS models is implemented in a customary manner where tasks transition between different states and a task queue is associated with each state [10]. Task creation (*task_create*) allocates the RTOS task data structure and *task_activate* inserts the task into the ready queue. The *par_start* method suspends the task and calls the scheduler to dispatch another task while *par_end* resumes the calling task's execution by moving the task back into the ready queue.

Event management is implemented by associating additional queues with each event. Event creation (*event_new*) and deletion (*event_del*) allocate and deallocate the corresponding data structures in the RTOS layer. Blocking on an event (*event_wait*) suspends the task and inserts it into the event queue whereas *event_notify* moves all tasks in the event queue back into the ready queue.

In order to model the time-sharing nature of dynamic task scheduling in the RTOS, the execution of tasks needs to be serialized according to the chosen scheduling algorithm. The RTOS model ensures that at any given time only one task is running on the underlying SLDL simulation kernel. This is achieved by blocking all but the current task on SLDL events (Figure 4.12). Whenever task states change inside a RTOS call, the scheduler is invoked and, based on the scheduling

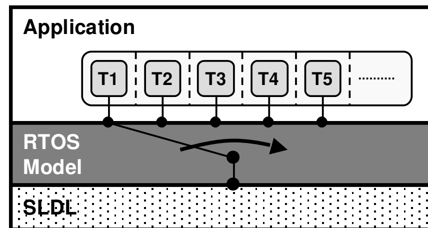


Figure 4.12: Dynamic scheduling implementation.

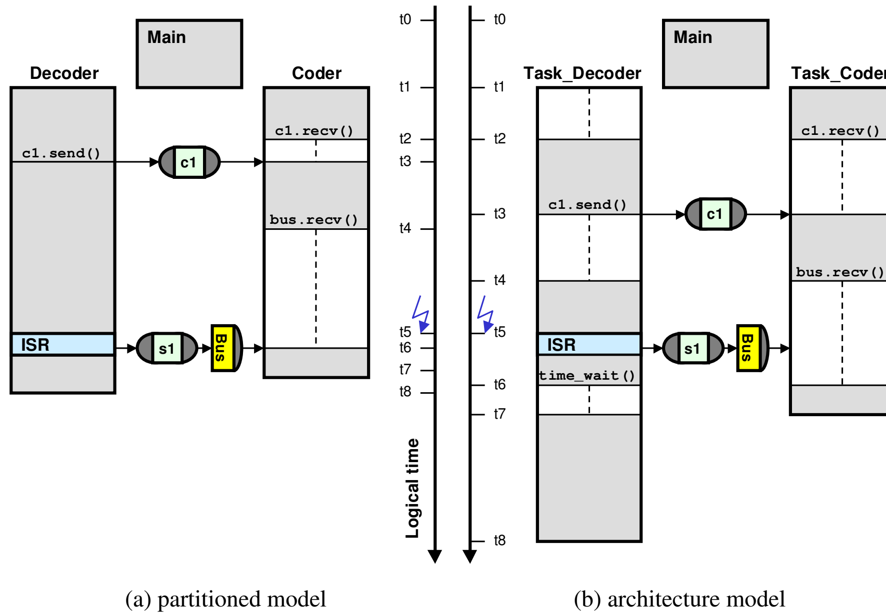
algorithm and task priorities, a task from the ready queue is selected and dispatched by releasing its SLDL event. Note that replacing SLDL synchronization primitives with RTOS calls is necessary to keep the internal task state of the RTOS model updated.

In high level system models, simulation time advances in discrete steps based on the granularity of `waitfor` statements used to model delays (e.g. at behavior or basic block level). The time-sharing implementation in the RTOS model makes sure that delays of concurrent task are accumulative as required by any model of serialized task execution. However, additionally replacing `waitfor` statements with corresponding RTOS time modeling calls is necessary to accurately model preemption. The *time_wait* method is a wrapper around the `waitfor` statement that allows the RTOS kernel to reschedule and switch tasks whenever time increases, i.e. in between regular RTOS system calls.

Normally, this would not be an issue since task state changes can not happen outside of RTOS system calls. However, external interrupts can asynchronously trigger task changes in between system calls of the current task in which case proper modeling of preemption is important for the accuracy of the model (e.g. response time results). For example, an interrupt handler can release a semaphore on which a high priority task for processing of the external event is blocked. Note that given the nature of high level models, the accuracy of preemption results is limited by the granularity of task delay models.

Figure 4.13 illustrates the behavior of the RTOS model based on simulation traces obtained for the *DSP* PE in the design example. For the purpose of illustration, the traces include aspects of PE-internal interrupt handling and bus driver communication that will be inserted as part of communication design as described in Chapter 5. Specifically, the simulated *DSP* PE contains bus driver channels and interrupt service routines (ISR) for external communication where the ISR can communicate with the bus driver through a semaphore *s1*.

Figure 4.13(a) shows the simulation trace of the unscheduled, partitioned model. Behaviors *Decoder* and *Coder* are executing truly in parallel, i.e. their simulated delays overlap. After



(a) partitioned model (b) architecture model

Figure 4.13: RTOS model simulation traces.

executing until time t_2, *Coder* waits until it receives a message from *Decoder* through the channel *c1*. At time t_3, *Decoder* sends a message to *Coder* through channel *c1* which wakes up *Coder* and both behavior continue executing in parallel. At time t_4, *Coder* enters the bus driver and waits for data from another PE. At time t_5, an interrupt happens, execution of behaviors is interrupted and the interrupt service routine is invoked. The ISR wakes up *Coder* sitting in the bus driver through semaphore *s1* at time t_6. *Coder*, in turn, receives its data through the bus driver and both behaviors continue until they finish execution.

Figure 4.13(b) shows the simulation result of the architecture model for a priority based scheduling. It demonstrates that in the refined model *Task_Decoder* and *Task_Coder* execute in an interleaved way. Since *Task_Coder* has the higher priority, it executes unless it is blocked on receiving a message from *Task_Decoder* (t_2 through t_3), it is waiting for an interrupt (t_4 through t_6), or it finishes (t_7). At those points execution switches to *Task_Decoder*. Note that at time t_5, the interrupt wakes up *Task_Coder*, and *Task_Decoder* is preempted by *Task_Coder*. However, the actual task switch is delayed until the end of the discrete time step at t_6 in *Task_Decoder* based on the granularity of the task's delay model. These inaccuracies are unavoidable due to the discrete nature of system simulations at high levels of abstraction. In summary, however, as required by priority based dynamic scheduling, at any time only one task, the ready task with the highest priority, is executing.

4.3.2.4 Architecture Model

Figure 4.14 shows the architecture model as the result of dynamic scheduling refinement. The architecture model is also the final output of the computation design process as a whole. For both software PEs, *ColdFire* and *DSP*, an additional OS layer (*ColdFire_OS* and *DSP_OS*, respectively) has been inserted around the basic PE behaviors. In case of the *ColdFire* processor, no dynamic scheduling is required and the OS layer is empty. However, note that the layer contains an unused dummy OS model stub (not shown in the figure) that provides services for suspending and resuming the processor needed for implementation of interrupt handling during communication design later.

In case of the *DSP* processor, the OS layer contains an instance of the OS model implementing the selected scheduling strategy. Furthermore, parallel behaviors inside the processor have been converted into tasks and the application in general uses the services provided by the OS model for task management, synchronization, and internal and external communication.

4.4 Summary

In this chapter, we presented the computation design flow with well-defined design steps and design models. Starting from an abstract system specification, a virtual architecture model of the design consisting of a set of non-terminating, concurrent processing elements (PEs) communicating through abstract message-passing channels is generated through partitioning and scheduling design tasks.

The computation design flow supports heterogoeneous multi-processor system architectures with an unlimited number of arbitrary PEs like custom hardware processors, software processors, memories, and IPs in a unified manner. Execution of behaviors on PEs can be scheduled statically and dynamically. In the latter case, an abstract model of the underlying RTOS describes the dynamic scheduling behavior of the operating system scheduler for feedback during simulation and validation in general.

The contributions of this chapter include clear definitions of design steps, abstraction levels, and semantics. For each step, necessary design decisions and model transformations have been identified (as summarized in Table 4.1). Furthermore, models of designs, target architectures and implementation details have been developed. As results will show (Chapter 8), intermediate models break the computation design flow into manageable steps while providing feedback for

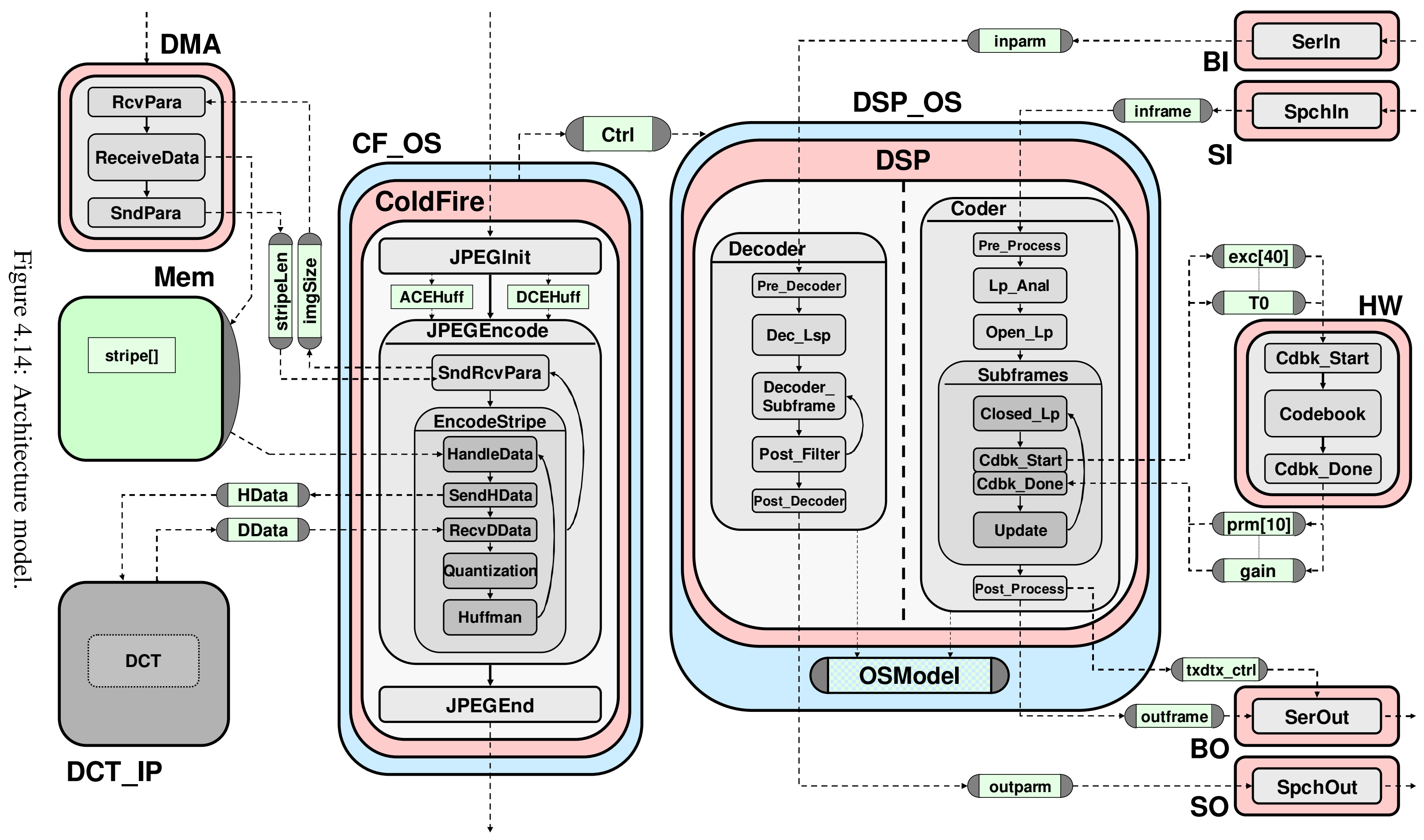


Figure 4.14: Architecture model.

<table>
<tr><th colspan="2">Design step</th><th>Design decisions</th><th>Model transformations</th></tr>
<tr><td rowspan="2">Partitioning</td><td>Behavior partitioning</td><td>(a) PE allocation & selection:
PE = set of ($name$,$type$) tuples
(b) Behavior mapping function m_b:
set of behaviors $B \mapsto PE$</td><td>(a) PE behavior insertion
(b) Behavior grouping
(c) Synchronization insertion
(d) Timing refinement</td></tr>
<tr><td>Variable partitioning</td><td>(a) Memory allocation & selection:
M = set of ($name$,$type$) tuples
(b) Variable mapping function:
subset of variables $V_g \subseteq V \mapsto M$</td><td>(a) Memory behavior insertion
(b) Variable grouping
(c) Message passing insertion
(d) Memory access refinement</td></tr>
<tr><td rowspan="2">Scheduling</td><td>Static scheduling</td><td>(a) Behavior schedules S:
$\forall b \in$ subset $B_s \subseteq B$:
S_b = totally ordered set of children</td><td>(a) Behavior serialization
(b) Behavior flattening
(c) Behavior ordering</td></tr>
<tr><td>Dynamic scheduling</td><td>(a) Scheduling algorithm selection:
$os : PE \mapsto$ set of algorithms OS
(b) Task priority assignment:
$p : B_t \subseteq B \mapsto \mathbb{Z}^+$</td><td>(a) RTOS model insertion
(b) Task creation
(c) Task refinement
(d) Synchronization refinement</td></tr>
</table>

Table 4.1: Computation design steps.

validation and verification of critical design decisions at early stages. Finally, models have been defined such that they can be automatically generated through successive refinement (see Chapter 7) while being able to describe a wide range of possible target architectures.

Chapter 5

Communication Design

Communication design is the second step of the system design process. It derives the system communication model from the intermediate architecture model. Communication design is a process with multiple stages where the system communication is gradually refined from an abstract message-passing down to an actual implementation over pins and wires. In the architecture model, communication is described in purely behavioral or functional manner as synchronous or asynchronous message-passing channels between PEs. At the end of the communication design flow is a structural description of the communication architecture in the form of wires connecting the pins of bus-functional network stations.

In this chapter, we describe and define the different steps of the communication design process. Starting with an overview of the communication design flow, specific requirements of SoC communication, layering of communication functionality and application of layers to SoC design are outlined in Section 5.1. Specific network and link design tasks and their individual steps and layers are then described in detail in Section 5.2 and Section 5.3, respectively.

5.1 Overview

Analogous to the computation design flow, communication design generally requires the resolution of space (where) and time (when) issues for communication between components in the system. Consequently, communication design is separated into two tasks—network design and link design—that roughly divide the design process along those issues.

Figure 5.1 shows the proposed communication design flow. Communication design starts with a virtual architecture model of the system in which processing elements (PEs) that per-

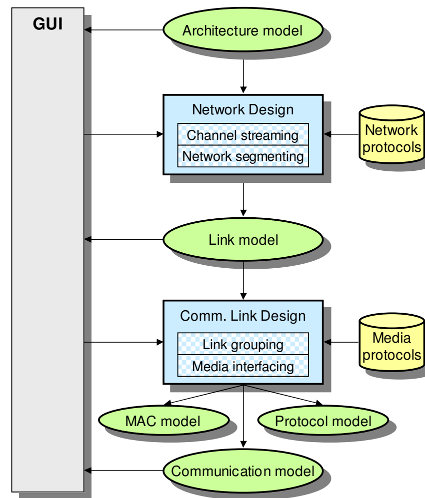


Figure 5.1: Communication design flow.

form computation communicate via abstract channels with synchronous or asynchronous message-passing semantics. The first task of communication design is network design in which the overall network topology is defined and end-to-end channels are mapped into point-to-point communication over multiple subnets. Network design requires conversion and merging of channels into untyped bytes streams and segmenting of the network into several connected subnets. During network segmenting, additional transducer components are allocated to bridge and connect subnets. End-to-end streams are then routed over the network of point-to-point links connecting PEs and transducers within each segment. The result of the network design step is a refined link model of the system. In the link model, PEs and other network stations communicate via logical link channels that carry streams of packets between directly connected components.

After network design, link design is performed for all links within each network segment where each segment can be designed separately. Within each segment, link design implements packet transfers over actual pins and wires of a selected bus or other communication structure. First, logical links within each segment are grouped into physical links and packet transfers for each link are implemented over the common, shared communication medium. Then, media interfaces of each PE and transducer are implemented down to the media access, protocol and finally wire level. In the process, additonal arbiter and interrupt controller components are inserted and connected as necessary.

As a result of the communication design process, a physical model of the system is generated automatically. The physical model is a fully structural model at the system level in which

PEs and other network components communicate and are connected via pins and wires. Apart from the physical model, the communication design process can produce intermediate transaction-level models (TLMs) at the output of the link grouping process. Transaction-level models abstract the pin-level communication in the physical model to the level of media access or individual protocol word/frame transactions. Depending on the parameters of the implementation, automatically generated TLMs of the physical model can be used to trade off accuracy and model complexity in order to accelerate validation through simulation, for example.

5.1.1 SoC Communication

In the architecture model at the input of communication design, reliable, synchronous and asynchronous message-passing communication between entities (PEs) is required. In synchronous communication, both sender and receiver are provided with information about the completion of the transaction, i.e. of delivery of data from sending to receiving entity. Usually, this means that calls on both sides block until it is guaranteed that data has arrived at/from the other end[1]. Around the actual data transfer, this requires synchronization to ensure that both sides have completed the transaction. Note that synchronous communication precludes loss of data but does not guarantee protection against data errors, for example. Synchronous communication is often chosen because it minimizes storage and by default, any lossless communication without buffering has to be synchronous and blocking.

Asynchronous communication, on the other hand, is less restricted in that no feedback about completion of transfers is provided or required. In the general case, communication calls on both sides do not block on successful data delivery, i.e. communication partners are decoupled and do not need to be synchronized. Asynchronous communication enables and is a consequence of buffering of data without overall synchronization. Asynchronous communication therefore acts like a FIFO queue where the queue depth depends on the amount of buffering in the actual implementation[2]. Note that independent of the blocking and non-blocking nature of communication in the synchronous and asynchronous case, calls itself may be blocking or non-blocking depending on how overflow of any available local buffers is handled. For example, if data is simply discarded when buffers are full, asynchronous sends are non-blocking but lossy.

[1] Calls might not block on delivery if data is buffered and some other mechanism (e.g. callbacks) is available to provide feedback about completion.

[2] If the implementation does not make any guarantees at all, queues of depth zero are possible which, if lossless, are equivalent to synchronous communication.

In both cases, reliable communication has to be lossless and error-free, i.e. it is guaranteed that the same data that is put in on the sender side will be received at the other end. Reliability is achieved through flow control and/or error correction. Flow control is error prevention in that it ensures that communication partners can not overrun each other, thus avoiding data loss during the actual data transfer. By matching data rates on both ends (including local delays for processing of data) it guarantees that both sides are free to send and receive. During a transfer, therefore, flow control needs to delay the faster end until it can be made sure that the other side is ready. At the lowest level, flow control requires some appropriate timing guarantees in the implementation, for example by inserting delays or wait states to communicate at a lowest common, fixed data rate. On top of that, information about the state of data processing needs to be exchanged. In order to match data rates and delay the faster end by appropriately blocking callers, it has to be ensured that data is available for sending (ready messages) or that the receiver can accept data and has space to store it (acknowledgments). Note that synchronization for synchronous communication and flow control are related. The implementation of one can ease or even replace the implementation of the other. For example, state information provided by flow control can be used for synchronization. In fact, flow control is a less restricted version of synchronization of individual data elements. If there is no buffering they are equivalent.

Error correction is necessary to deal with unreliable underlying communication structures. Possible errors can include data (bit) errors or complete loss of data. Typically, error correction requires detection of errors together with retransmission of data. Error detection is usually based on (negative or lack of) acknowledgments from receiver to sender together with error checking at the receiving side. Note that error correction can compensate for data loss due to lack of or incomplete flow control. Therefore, if error corrections is necessary for other reasons and if the performance hit can be tolerated (e.g. if the likelihood of overflows is small), it can possibly replace flow control completely.

Both flow control and error correction can profit from intermediate buffering of outstanding data in order to increase performance and throughput by hiding and compensating for communication delays and latencies (e.g. to inject more data while waiting for replies or acknowledgments). Also, intermediate buffers are unavoidable in multi-hop communication architectures that require store-and-forward configurations. However, buffering affects synchronization and flow control, and it requires special handling for their implementation. For example, buffering without additional synchronization results in asynchronous communication, even though communication from buffer to buffer (or between application and buffer) is synchronous. Pairwise synchronization on each leg

does not provide end-to-end synchronous communication unless it can be guaranteed that legs are synchronized overall by intermediate way-stations.

Buffers can be used to even out data rate variations in general and for flow control in particular. Given large enough buffers, explicit flow control can possibly be avoided all together as long as burst of data are guaranteed to fit into the buffers. Otherwise, information about buffer fill states needs to be exchanged. In any case, however, flow control at lower levels needs to ensure that communication between buffers matches the rate at which data is read from one buffer with the rate at which it can be stored in the next. Note that in contrast to synchronization, reliability of communication between buffers implies overall end-to-end reliability. However, cross-influences between different communication streams due to sharing of resources (e.g. buffers) usually requires end-to-end flow control. Flow control prevents that one stream can saturate shared resources. Thus, it avoids unnecessary blocking of others and, in the worst case, the possibility of deadlocks,

All in all, a communication design flow needs to take all these issues into account in order to allow designing an optimal communication architecture for a given SoC application.

5.1.2 Communication Layers

The communication design flow is structured along a layering of communication functionality within each task of the design flow. The implementation of SoC communication is divided into several layers based on separation of concerns, grouping of common functionality, dependencies across layers, and early validation of critical issues for rapid and efficient design space exploration through humans or automated tools.

Layers are stacked on top of each other. A layer provides services to the next higher layer by building upon and using the services provided by the next lower layer. In general, at its interface to higher layers, each layer provides services for establishing communication channels and for performing transactions over those channels. Semantics of transactions and channels vary from layer to layer. Therefore, each layer corresponds to a certain level of abstraction. As design progresses, layers are successively inserted into the design and with each step a refined design model at the next lower level of abstraction is generated.

Table 5.1 summarizes the layers for SoC communication. For each layer, its interface to higher layers, its functionality, and its level of implementation in the backend tools (software, operating system kernel, device driver, hardware abstraction layer (HAL), hardware) is listed. Layering is based on the ISO OSI reference model [62]. However, due to the unique features and

Layer	Interface semantics	Functionality	Impl.	OSI
Application	N/A	• Computation	Application	7
Presentation	PE-to-PE, typed, named messages • `v1.send(struct myData)`	• Data formatting	Application	6
Session	PE-to-PE, untyped, named messages • `v1.send(void*, unsigned len)`	• Synchronization • Multiplexing	OS kernel	5
Transport	PE-to-PE streams of untyped messages • `strm1.send(void*, unsigned len)`	• Packeting • Flow control • Error correction	OS kernel	4
Network	PE-to-PE streams of packets • `strm1.send(struct Packet)`	• Routing	OS kernel	3
Link	Station-to-station logical links • `link1.send(void*, unsigned len)`	• Station typing • Synchronization	Driver	2b
Stream	Station-to-station control and data streams • `ctrl1.receive()` • `data1.write(void*, unsigned len)`	• Multiplexing • Addressing	Driver	2b
Media Access	Shared medium byte streams • `bus.write(int addr, void*, unsigned len)`	• Data slicing • Arbitration	HAL	2a
Protocol	Unregulated word/frame media transmission • `bus.writeWord(bit[] addr, bit[] data)`	• Protocol timing	Hardware	2a
Physical	Pins, wires • `A.drive(0)` • `D.sample()`	• Driving, sampling	Interconnect	1

Table 5.1: Communication layers.

characteristics of SoC communication, layers have been tailored specifically to these requirements. Furthermore, note that layers only serve as a specification of the desired implementation. As part of synthesis of communication within each tool, layers are possibly merged for optimizations across layer boundaries.

5.1.2.1 SoC Layer Stacks

Layers provide a general framework for communication design. However, when synthesizing a specific SoC design, customized implementations of layer stacks are generated. Based on specifics of SoC communication, only part of a layer's functionality is implemented or layers are not needed at all. For example, in traditional architectures, low-latency, reliable busses are used and error correction, flow control, buffering or dynamic routing are not required. Therefore, network and transport layers are empty or largely simplified.

In addition, application and target architecture specific optimizations are applied. Layer functionalities and hardware resources like bus widths, number of interrupt lines, etc. are adjusted to requirements. For example, end-to-end synchronization in the session layer is implemented only if synchronous communication is required by the application and not supplied by underlying links. Packet lengths in the transport layer are determined based on message sizes such that overhead is minimized. At the media access level, arbiters are inserted and customized depending on the number of masters on a medium.

Finally, multiple layers generated within each design task are inlined into PEs together. In the process, layers are merged and optimizations across layer boundaries are applied.

An example of communication layers for a typical SoC communication architecture is shown in Figure 5.2. The example shows a part of a system with two processing elements *PE0* and *PE1*. At the application level, *PE0* communicates with *PE1* through two message channels, *c1* and *c2*. Furthermore, *PE1* exchanges data with another remote PE in the system through a channel *c3*. In the presentation layer, abstract data types in the messages are converted into canonical byte format in both PEs. In the session layer, messages from *c1* and *c2* are merged into a single data stream between *PE0* and *PE1*. The transport layer splits large messages into streams of packets in order to have uniform, smaller message sizes for buffering at lower layers. In the network layer, *PE0* and *PE1* are directly connected via a single logical link whereas *PE1* exchanges data with other PEs via a logical link connected to a network interface which in turn will route the data packets via its outgoing logical link(s).

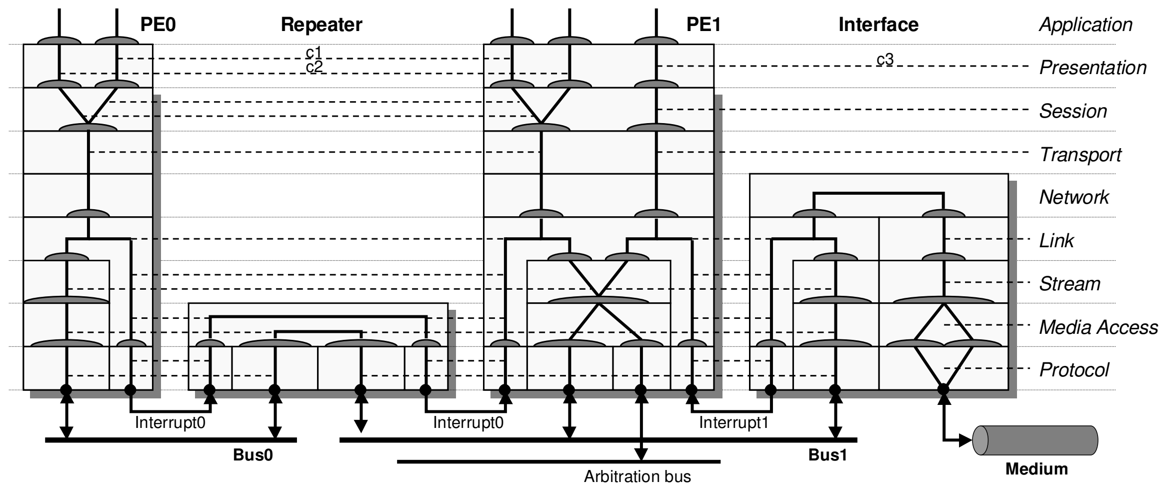
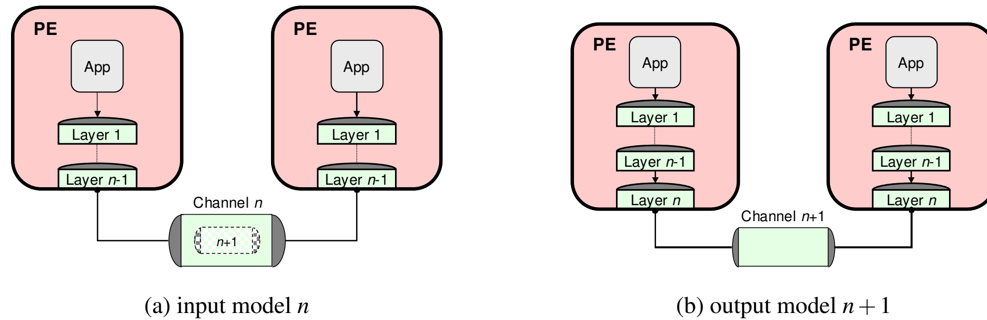


Figure 5.2: Communication architecture example.

For both links between *PE0* and *PE1* and between *PE1* and the interface, bus-based communication is used. *PE1* is declared as the master whereas *PE0* and the interface act as slaves. Correspondingly, control streams in the link layer perform handshaking from slave to master and data streams perform transfers under the control of the master. In the stream layer, the two data streams in *PE1* are then multiplexed over the single medium the PE is connected to. In the media access layer, data stream packets are sliced into bus words. Furthermore, inside the media access layer of bus master *PE1*, arbitration calls to request and release the bus are inserted around each bus transfer. For the link between *PE0* and *PE1*, the bus is split into two segments. A repeater is inserted as part of the media access layer. The repeater connects incompatible bus protocols and passes bus words and handshake events transparently between them. Finally, the protocol layer implements the protocols for driving and sampling the wires of *Bus0* connecting *PE0* and repeater, and of *Bus1* connecting repeater, *PE1*, and the interface. The protocol layer also implements protocols for handshaking via interrupts and for bus requests over the arbitration bus.

The links to other PEs in the system going in and out of the interface are implemented over a shared, long-latency, error-prone network medium. The link layer implements buffering, error correction and flow control to provide high-performance, reliable link communication by interleaving data and control over a single mixed byte stream supported by the medium. The stream layer assigns medium addresses to each link. The media access layer then splits packets into media frames while participating in the arbitration protocol on the medium (e.g. through collision detection). Finally, the protocol layer converts the media frames into bit streams on the physical wires, including any special services required for arbitration (e.g. listen-while-send).



(a) input model n (b) output model $n+1$

Figure 5.3: Communication model refinement.

```
behavior PE(i_layer_n  layer_n)
{
  Layer_n-1 layer_n-1(layer_n);
  Layer_n-2 layer_n-2(layer_n-1);
  ...
  Layer_1    layer_1(layer_2);

  App        app(layer_1);

  void main(void)
  {
    app.main();
  }
};
```

(a) input model n

```
behavior PE(i_layer_n+1 layer_n+1)
{
  Layer_n    layer_n(layer_n+1);
  Layer_n-1 layer_n-1(layer_n);
  Layer_n-2 layer_n-2(layer_n-1);
  ...
  Layer_1    layer_1(layer_2);

  App        app(layer_1);

  void main(void) {
    app.main();
  }
};
```

(b) output model $n+1$

Listing 5.1: Communication model PE refinement.

5.1.2.2 Model Refinement

Refinement of design models from architecture to communication models is organized based on layering of communication functionality. With each step in the design flow, an additional layer of communication functionality is inlined into the PEs of the design model. By replacing the communication between PEs (as represented by the channels connecting PEs) with channels that model the behavior and semantics of transactions at the interface of the next lower layer in an abstract manner, a new system model at the next lower level of abstraction is generated (Figure 5.3). For each layer inserted during the design process, a refined design model is therefore generated.

Communication layers are inserted into the PEs of the design model during refinement in the form of special adapter channels (Listing 5.1). Adapter channels implement the interface of the layer. They connect to the interface of the next lower layer through a corresponding port. An adapter then implements the layer's functionality in the code of its methods on top of calls to the next lower layer's methods. As part of model refinement, instances of layer adapters are created

inside the PEs, linking the PE's higher layers to the external communication channel interface at the next lower layer.

5.2 Network Design

The network design task implements end-to-end communication semantics between PEs. In the architecture model at the input of network design, PEs communicate by passing messages over synchronous or asynchronous channels. At the output of network design, network stations in the link model communicate by exchanging packets over point-to-point logical links between them. Starting with the application layer generated from the initial specification by preceeding computation design tasks, implementations of presentation and session layers are inserted as part of channel streaming whereas network segmenting introduces transport and network layers into the design.

5.2.1 Application Layer

The application layer corresponds to the computation functionality of the system. It defines the behavior of the application implemented by the computation design process. During the computation design process, the initial specification of the desired system behavior has been mapped onto a set of PEs. Inside each PE, the parts of the initial specification mapped onto that PE form the PE's application layer. Thus, the application layers describe the processing of data in the PEs.

5.2.1.1 Model Refinement

Since the application layer is equivalent to the computation functionality, no specific refinement of the architecture model at the output of computation design is required. In the model, PEs exchange data by passing messages over named channels with synchronous or asynchronous semantics where channels with different names are used to distinguish among data of different origin.

The SoC communication design flow supports implementation of reliable synchronous and asynchronous communication channels found in the application. Implementation of synchronous communication provides the corresponding guarantees about completion of transactions. For asynchronous communication, no such feedback is made available. In fact, neither are any guarantees about buffer sizes made in that case. Hence, asynchronous communication might end up

being implemented in a synchronous form. If a defined amount of buffering is required, queues of appropriate depth need to be implemented as part of the application.

In case of IP components in the design, the purely behavioral IP component models of the architecture model are replaced with models required for the communication design flow. IP models for communication design encapsulate a structural (bus-functional) model of the component in a wrapper that implements all layers of communication with the IP. Since the IP's communication protocols are pre-defined and fixed, its communication can not be designed arbitrarily and the wrapper provides the necessary functionality to be gradually inserted into the IP's communication partners as design progresses. Corresponding IP models with communication wrappers are imported out of the IP library and inserted into the design model as part of application layer model refinement.

5.2.1.2 Application Model

The application or presentation model for the design example is shown in Figure 5.4. The application model is the starting point for communication design and is mostly equivalent to the architecture model that was the result of the computation design process. However, for the *DCT_IP* component, a bus-functional model plus *DCT_Adapter* wrapper has been plugged into the model.

The application model specifies the communication functionality to implement. For each data item (variable) communicated between PEs, the model contains a corresponding typed message-passing channel. Communication and channels at the application level are always reliable. In this example, all channels between PEs are specified to be asynchronous, i.e. the application does not require synchronous messages. In general, channels at the application level specify the synchronization requirements but not the amount of buffering for implementation of the channels. Instead, the amount of buffering in application-level channels will depend on their implementation in lower layers. In the application model, channels can therefore have any amount of buffering, e.g. some average, fixed number. On the other hand, if information about the implementation is available, channels can be annotated with estimated buffer sizes for feedback during simulation.

Similar to the architecture model, the shared memory PE in the application model the is modeled as a special channel. The memory channel encapsulates all data items (variables) mapped into the shared memory component. At its interface, the memory channel provides two methods per data item for reading and writing the item's value from/to memory.

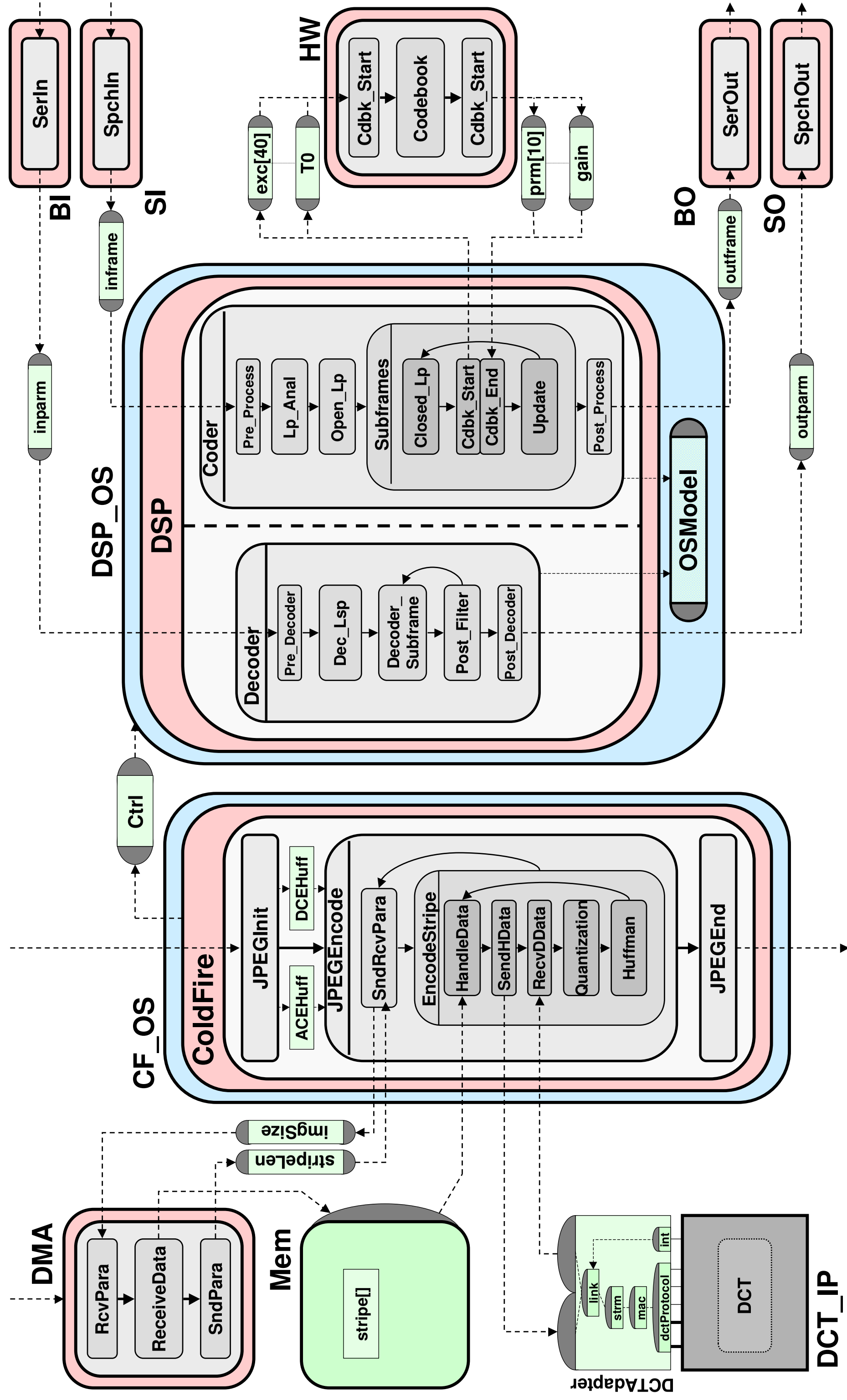


Figure 5.4: Application model.

```
channel PresentInt(i_tranceiver session)
  implements i_int_send, i_int_receive
{
  void send(int val) {
    uint32_t data;
    data = htonl(val);
    session.send(&data, sizeof(data));
  }

  void receive(int *val) {
    uint32_t data;
    session.receive(&data, sizeof(data));
    *val = ntohl(data);
  }
};
```

Listing 5.2: Presentation layer.

5.2.2 Presentation Layer

The presentation layer provides services to establish named channels between PEs and to reliably send and receive messages of arbitrary, abstract data types over those channels. Each presentation layer channel carries messages of a fixed type where a sequence of messages of the same type can be transfered repeatedly over a named channel. In general, the presentation layer provides services for both synchronous and asynchronous channels, depending on the application requirements.

The presentation layer becomes part of the application software and is responsible for data formatting. It converts abstract data types in the application to blocks of ordered bytes as defined by the canonical byte layout requirements of the lower (network) layers. For example, the presentation layer takes care of PE-specific data type conversions and endianess (byte order) issues.

5.2.2.1 Model Refinement

Presentation layer model refinement requires insertion of presentation layer implementations into the PEs. Presentation layers are inserted in the form of adapter channels that provide a presentation layer API to the application while connecting to and calling session layer methods below (Listing 5.2). As outlined previously, the presentation layer performs data formatting for every message data type found in the application. For each application layer channel, corresponding presentation layer adapters are instantiated inside the PEs, converting the abstract data types into byte blocks based on network byte layout conventions. Since the presentation layer becomes part of the application, its adapter channels are instantiated inside each PE's application layer.

```
interface IShm {
  void read(unsigned long ofs, void* data, unsigned long len);
  void write(unsigned long ofs, const void* data, unsigned long len);
};
```

(a) memory interface

```
behavior Mem()
  implements IShm
{
  char mem[MEM_SIZE];    // storage
5
  void read(unsigned long ofs, void* data, unsigned long len) {
    memcpy(data, mem + ofs, len);
  }
  void write(unsigned long ofs, const void* data, unsigned long len) {
10    memcpy(mem + ofs, data, len);
  }

  void main(void) { }
};
```

(b) memory behavior

Listing 5.3: Shared memory presentation model.

As part of the presentation layer implementation, shared memory models are refined down to an accurate representation of the byte layout in the memory (Listing 5.3). All variables stored inside the memory are replaced with and grouped into a single array of bytes (Listing 5.3(b), line 4). As part of the presentation layer, layout and addressing of variables inside the memory is defined based on the parameters (e.g. alignment) of the chosen target memory component. Instead of accesses to individual variables, the refined memory behavior supports read and write accesses to blocks of bytes at given offsets (see memory interface shown in Listing 5.3(a) and implementation of access methods in the memory behavior in Listing 5.3(b), line 6 and line 9).

The presentation layers inside the PEs accessing the memory are then responsible for converting application variables into size and offset for shared memory accesses (Listing 5.4). Given offsets of each variable in memory, the shared memory presentation layer translates variable accesses into corresponding memory accesses (line 6 and line 11).

5.2.2.2 Session Model

The resulting session model of the design example is shown in Figure 5.5. In the session model, PEs are connected through session channels that carry streams of untyped (byte-level) messages. Channels in the session model are always reliable. Depending on the requirements at

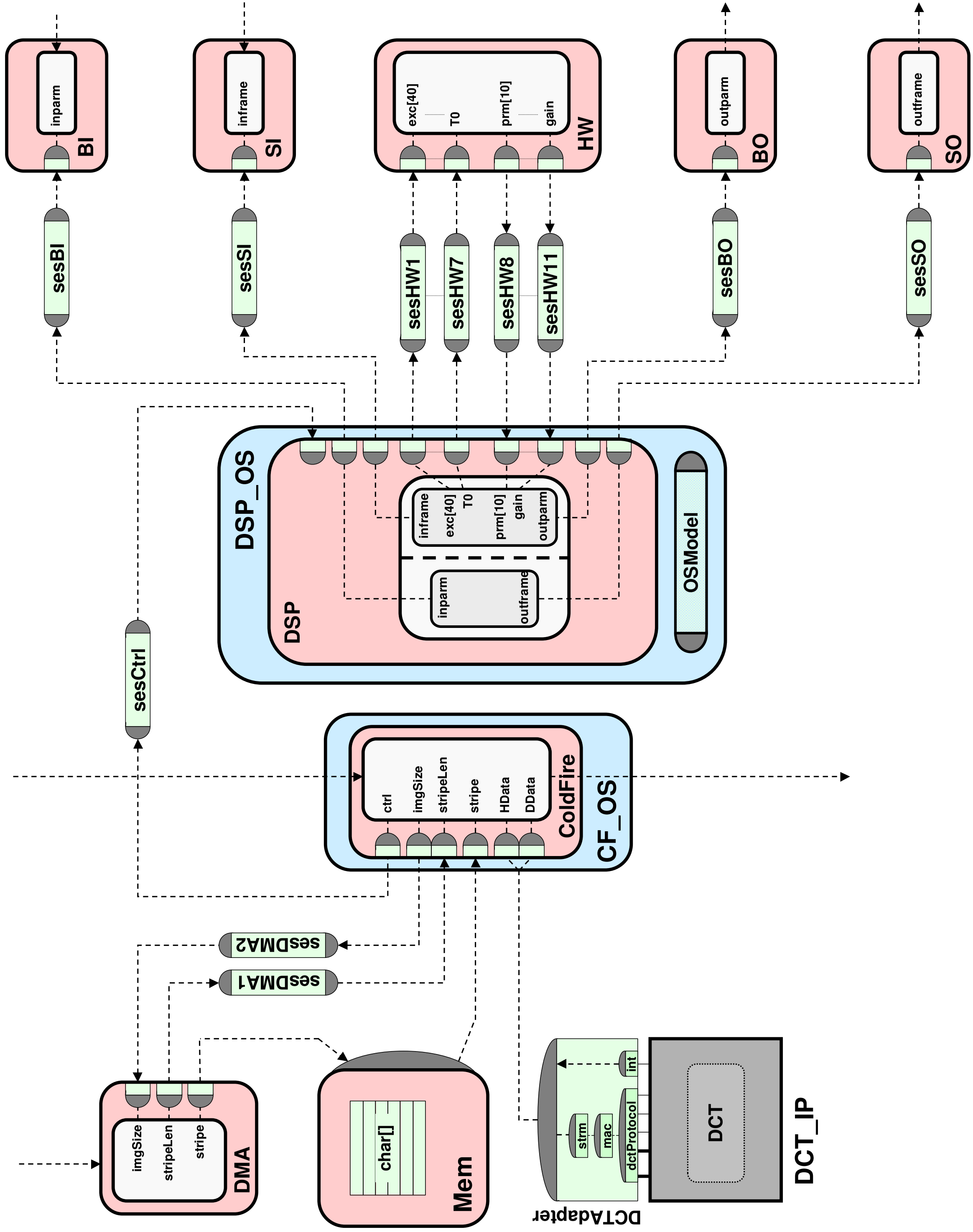


Figure 5.5: Session model.

```
channel ShmStripe(IShm shm)
  implements i_mem
{
  char read_stripe(int index) {
5   unsigned char val;
    shm.read(OFS_STRIPE + index, &val, 1);
    return val;
  }

10 void write_stripe(int index, char val) {
    shm.write(OFS_STRIPE + index, &val, 1);
  }
};
```

Listing 5.4: Memory access presentation layer.

the application (and hence presentation) level, session model channels are synchronous or asynchronous. In this example, the application requires asynchronous communication only. Therefore, all channels in the example session model asynchronous.

Inside the PE's application layers, instances of presentation layer adapters have been created for each external communication channel. Note that for the IP component, presentation layer adapters were part of the IP wrapper and have been inlined into the *ColdFire* PE communicating with the IP. Finally, the memory component behavior has been refined down to a byte-accurate description of its contents.

5.2.3 Session Layer

The session layer provides named channels over which untyped messages can be transfered reliably. Session layer messages are uninterpreted, ordered blocks of bytes. Session layer channels are used to distinguish among communication end-points in the system application where each channel can carry an ordered sequence of messages. Channels are synchronous or asynchronous and the session layer generally supports both types of channels.

The session layer is at the interface between application software and operating system. If the layers below are asynchronous, the session layer will implement end-to-end synchronization to provide any synchronous communication required above. Furthermore, it is responsible for multiplexing messages of different channels into a number of end-to-end sequential message streams. Messages of different channels at the session layer interface inside a PE are usually statically or dynamically ordered. If all communicating PEs transmit messages over different session layer channels in a pre-defined order, messages can be directly merged into a single stream. In general, however, the session layer merges channels into multiple concurrent streams or it implements

name resolution for multiplexing arbitrary messages over a single stream, separating them in space. The stream layer does not necessarily implement separation in time and messages can enter streams concurrently.

5.2.3.1 Model Refinement

Session layer refinement generally requires insertion of adapter channels with session layer implementations inside the PEs. At their interface to higher layers, session layer adapters provide services to simultaneously handle multiple sessions over a single shared underlying transport channel that the session layer connects to.

In many cases of traditional bus-based communication architectures, however, the session layer is empty. If the application only requires asynchronous communication, the session layer does not need to implement any extra end-to-end synchronization. Furthermore, if all communication between the same end-points is sequential, complex multiplexing of concurrent communication over the same transport stream is not necessary. In these cases, each transport will only carry messages that are already sequentialized (ordered in time) by the application. Since ordered messages going over the same transport can be multiplexed directly, merging of sessions over transports is resolved through simple corresponding connectivity in the operating system layer of PEs[3].

5.2.3.2 Transport Model

The transport model of the design example after session layer refinement is shown in Figure 5.6. In the transport model, PEs are connected through transport channels that carry streams of untyped (byte-level) messages. In general, channels in the transport model are always reliable. However, the transport model does not specify the amount of synchronization and buffering in its channels, i.e. channels are generally asynchronous and can have any amount of buffering. On the other hand, if their implementation is known or can be estimated, semantics of channels in the transport model can be selected to reflect and abstract the behavior of their implementation.

In this example, session layers are empty since no extra synchronization or merging of communication is necessary. Therefore, session layer functionality is implemented through simple direct connectivity of presentation layer adapters to external transport channels inside the PEs.

[3]For hardware PEs that don't have an OS layer, an extra hardware layer that will absorb all further communication implementation is inserted.

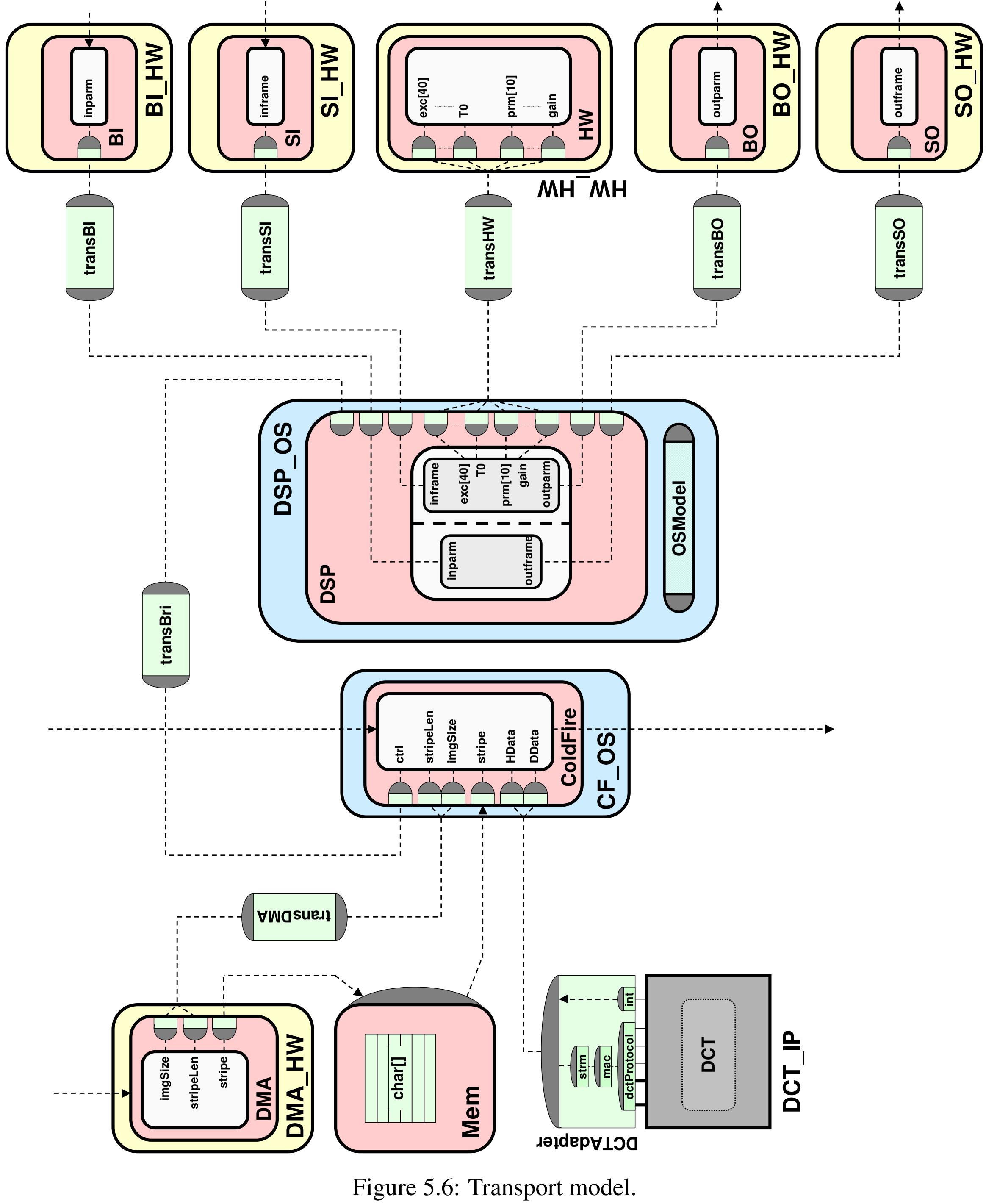


Figure 5.6: Transport model.

5.2.4 Transport Layer

The transport layer provides services to reliably transmit end-to-end streams of arbitrary, untyped messages (blocks of bytes). Channels at the transport layer define the communication pipes between PEs in the system over which individual communication sessions are handled by the layers above. Transport layer channels are generally asynchronous where the amount of buffering generally also depends on the layers below. Note that in case of no buffering, transport channels are effectively synchronous.

The transport layer implements end-to-end data flow as part of the operating system kernel. It splits messages into smaller packets to reduce intermediate minimum buffer sizes, for example. Depending on the links and stations in lower layers, the transport layer implements end-to-end flow control and error correction to guarantee reliable transmission.

5.2.4.1 Model Refinement

Refinement of design models for insertion of transport layers is a straightforward process. Corresponding layers of adapter channels that implement transport layer functionality are inserted into the operating system layer of the PEs. For each transport channel required by higher layers, a transport layer adapter that connects to its corresponding external network interface is instantiated.

5.2.4.2 Network Model

A network model of the design will be generated as a result of implementing the transport layer and inlining its functionality into the PEs. In the network model, PEs are connected via channels that carry streams of packets. Depending on the underlying layers, network channels are generally asynchronous and unreliable.

In the case of the design example, the transport layer is empty because message sizes are small and deterministic such that extra packeting of messages is not necessary. In addition, the implementation of lower layers is known to provide reliable communication without sharing of resources (buffers), i.e. no end-to-end flow control or error correction is needed. Therefore, the network model of the design example is semantically and syntactically equivalent to its transport model (Figure 5.6) and not explicitly shown here.

5.2.5 Network Layer

The network layer provides services for establishing end-to-end paths that can carry streams of packets. Depending on the layers below, the network layer may or may not guarantee reliable delivery. In the general case, transactions are best-effort only. Furthermore, channels are asynchronous depending on both the layers below and the amount of buffering introduced in the network layer itself.

The network layer completes the high-level, end-to-end communication implementation in the operating system kernel. It is responsible for routing of end-to-end paths over individual point-to-point links. Assuming reliable stations and links, routing in SoCs is usually done statically, i.e. all packets of a channel take the same fixed, pre-determined path through the system. In general, however, the network layer can implement dynamic routing on a connection or packet-by-packet basis to deal with changing underlying conditions.

In all cases, the network layer is responsible for separating different end-to-end paths going through the same stations. In a simple implementation, a dedicated logical link is established between two stations for each channel routed through them, assuming the underlying layers support a large enough number of simultaneous logical links between all pairs of stations. In the general case, multiple connections are routed and multiplexed over a single logical link and the network layer implements additional addressing to distinguish different end-to-end connections.

5.2.5.1 Model Refinement

As part of network layer refinement, network layer adapter channels are inserted into the operating system layers of PEs. As outlined above, network layer adapters generally multiplex and route multiple simultaneous end-to-end packet streams over individual incoming and outgoing point-to-point logical links.

As part of the network layer implementation, additional communication stations (transducers) with intermediate buffering are introduced as necessary. Transducers create and bridge subnets, splitting the system of connected PEs into smaller, directly connected groups. Transducers are inserted into the system architecture as additional system components running in parallel with other components at the top level of the design. Transducer components in the network layer are represented by behavioral transducer models (Listing 5.5). A transducer behavior connects to two or more logical links (line 1). It implements the network layer functionality for bridging subnets and routing packets between them by receiving packets over an incoming link (line 7) and sending

```
behavior Bridge(i_tranceiver cfLink, i_tranceiver dspLink)
{
  void main(void) {
    struct Packet packet;

    while(true) {
      cfLink.receive(&packet, sizeof(packet));
      dspLink.send(&packet, sizeof(packet));
    }
  }
};
```

Listing 5.5: Transducer model.

them over one or more outgoing links (line 8). Transducers are allocated and empty templates or pre-designed IPs for behavioral transducer models are imported into the design out of the communication element (CE) database.

5.2.5.2 Link Model

Figure 5.7 shows the link model for the system design example. Generally, the link model contains additional implementations of the network layer inside each PE. In this example, however, explicit network layer implementations are not required. Since all data is routed statically in a pre-determined manner, the routing of the network layer can be modeled through proper static connectivity of link level channels.

In the link model, end-to-end channels have been replaced with point-to-point logical links between adjacent stations that will later be physically connected through wires. Since it will be impossible to directly connect the two processors in this example, an additional *Bridge* station has been inserted into the link model, connecting the two subsystems. As a result, the end-to-end channel between *ColdFire* and *DSP* processors has been split into two links. The bridge in between transparently forwards packets between the two links.

In general, channels representing the logical links between stations in the link model may or may not be reliable, synchronous, and buffered, i.e. the link model does not specify how links should be implemented. Again, if, on the other hand, information about their implementation is available, link channels can be chosen to model their actual, real behavior for feedback during simulation, for example. In the case of this example, all logical links in the design are known to be synchronous, reliable, and unbuffered. Therefore, the model contains channels with corresponding semantics.

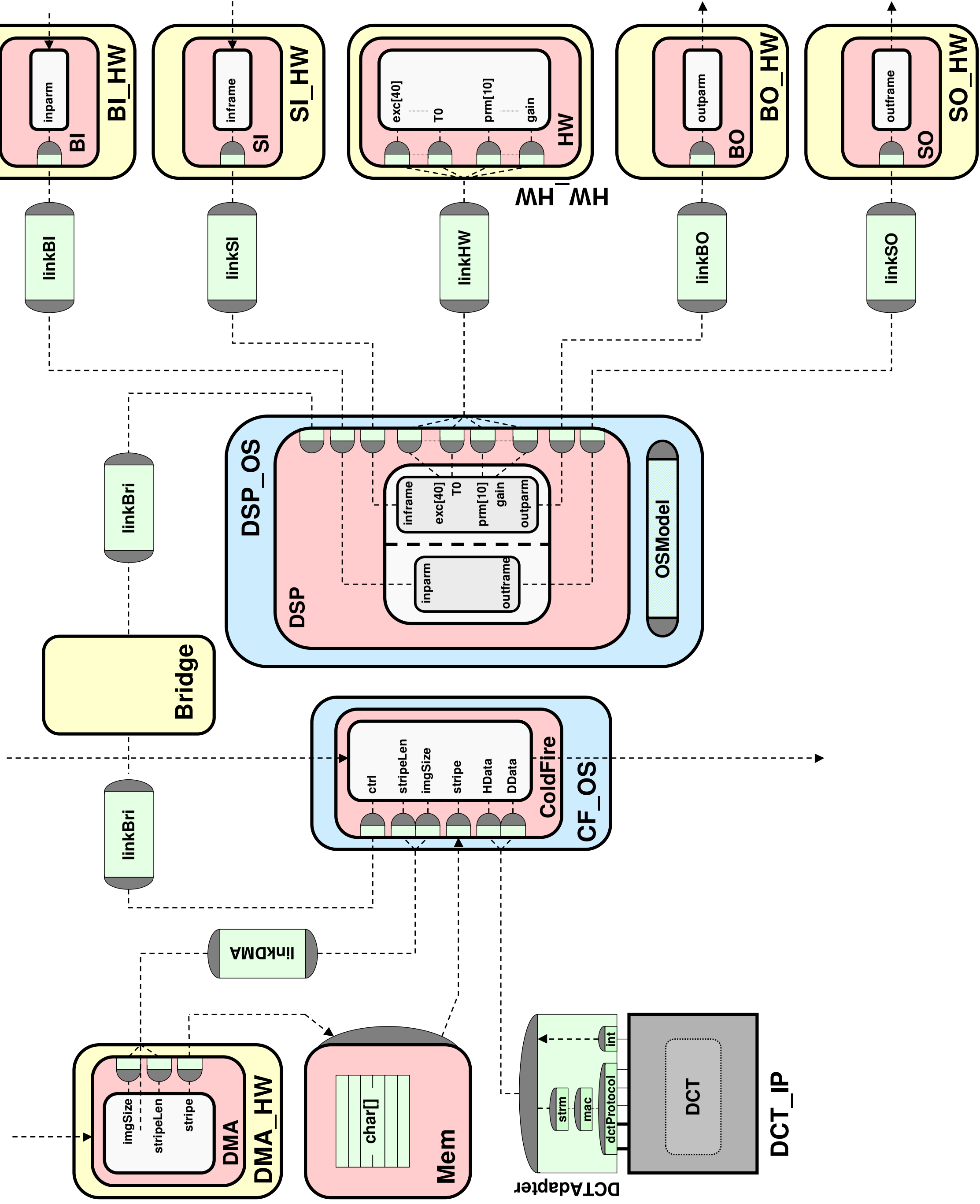


Figure 5.7: Link model.

5.3 Link Design

The link design task implements point-to-point logical links between stations over actual communication media. Links within each segment of the system communication network share the same communication medium. Hence, segments can be implemented separately. As a result of link design, the final physical model is generated. In the physical model, network stations are connected through physical pins and wires. Link design consists of link grouping and media interfacing. During link grouping, implementations for link and stream layers are inserted. Media interfacing, on the other hand, introduces media access and protocol layers on top of the actual physical layer.

5.3.1 Link Layer

The link layer provides services to establish logical links between adjacent (directly connected) stations and to exchange data packets in the form of uninterpreted byte blocks over those links. Depending on the lower layers, a number of named logical link channels can be established between pairs of stations. Furthermore, links may or may not be reliable and synchronous.

The link layer is the highest layer of drivers for external interfaces and peripherals in the operating system. It provides their interface to the rest of the OS kernel. The link layer defines the type of a station (e.g. master/slave) for each of its incoming or outgoing links. As a result, it implements any necessary synchronization between stations, for example by splitting each logical link into separate control (e.g. interrupt or acknowledgment) and data streams provided by lower layers.

5.3.1.1 Model Refinement

As part of link layer model refinement, implementations of link layer functionality in the form of adapter channels are inserted into the components of the system. Since the link layer is part of the operating system kernel, its implementation channels are inserted into the OS layers of the PEs (or the communication hardware layer for synthesizable PEs).

The link layer defines the types of each station and splits the packet streams into separate control and data streams as necessary. In a typical bus-based master/slave arrangement, each logical link is split into a data stream under the control of the master and a control stream from slave to master (Listing 5.6). On the master side (Listing 5.6(a)), the link layer implementation waits for a handshaking signal from the slave (line 5 and line 10) before initiating a write or read transfer

```
  channel MasterLink(IMasterStream stream, i_receive hndshk)
    implements i_tranceiver
  {
    void send(const void* data, unsigned long len) {
5     hndshk.receive();
      stream.masterWrite(data, len);
    }

    void receive(void* data, unsigned long len) {
10    hndshk.receive();
      stream.masterRead(data, len);
    }
  };
```

(a) master

```
  channel SlaveLink(ISlaveStream stream, i_send hndshk)
    implements i_tranceiver
  {
    void send(const void* data, unsigned long len) {
5     hndshk.send();
      stream.slaveWrite(data, len);
    }

    void receive(void* data, unsigned long len) {
10    hndshk.send();
      stream.slaveRead(data, len);
    }
  };
```

(b) slave

Listing 5.6: Link layer.

over the data stream (line 6 and line 11). On the slave side (Listing 5.6(b)), the link layer sends the handshaking signal (line 5 and line 10) and then enters the data stream to wait and answer the corresponding incoming write or read transfer (line 6 and line 11). By synchronizing master to slave through the control stream before a packet transfer can be initiated by the master, packet losses are avoided and reliable, unbuffered, and synchronous links are implemented.

In case of communication with shared memory PEs, memories are assumed to be always ready and no extra synchronization through control streams is necessary. Instead, a single data stream for memory slave accesses under the control of bus masters is sufficient. In contrast to normal message data streams, the memory data stream carries extra information about the offset of the byte block being accesses in each read or write transaction. Note that on the master side accessing the memory, the interface to the memory data stream channel is equivalent to the memory interface at higher levels and no refinement is necessary. In the process of explicitly modeling memory data streams, however, the memory model is refined into an active component listening and serving request that come in over its data stream (Listing 5.7). In an endless loop (line 7), the

```
  behavior Mem(ISlaveShm shm)
  {
    char mem[MEM_SIZE];    // storage

5   void main(void) {
      unsigned long ofs, len;
      while(true) {
        ofs = 0;
        len = MEM_SIZE;
10      if(shm.listen(&ofs, &len)) {
          shm.serveRead(ofs, mem + ofs, len);
        } else {
          shm.serveWrite(ofs, mem + ofs, len);
        }
15    }
    }
  };
```

Listing 5.7: Shared memory link model.

memory listens for memory accesses through the memory data stream (line 10) and serves them by reading or writing the requested data from/to its internal storage (line 11 and line 13, respectively)

5.3.1.2 Stream Model

Figure 5.8 shows the stream model for the design example. Implementations of the link layer in the form of adapter channels have been inserted into the PEs' OS layers for each logical link in the design. In this example, bus-based communication is chosen for implementation. The *ColdFire* and *DSP* processors are masters on their respective busses. Furthermore, the *DMA* component can act as both a master (for communication with the memory) or slave (for communication with the processor) on the *ColdFire* bus. All other PEs are bus slaves. Since communication with the *DCT_IP* component is compatible with this master/slave arrangement of stations, the corresponding link layer implementation could simply be taken from the IP wrapper and inlined into the *CF_OS* model as link layer adapter channel.

Semantics of channels in the stream model depend completely on the chosen implementation scheme and no general format can be defined. Channels for data streams have special semantics in the sense that they are synchronous and blocking on the slave side and asynchronous and non-blocking on the master side. Even though they are error-free, they are not reliable as packet losses can happen if transactions are not properly synchronized beforehand. Control channels, on the other hand, are simple handshake channels (queues of depth one for control messages that do not carry values).

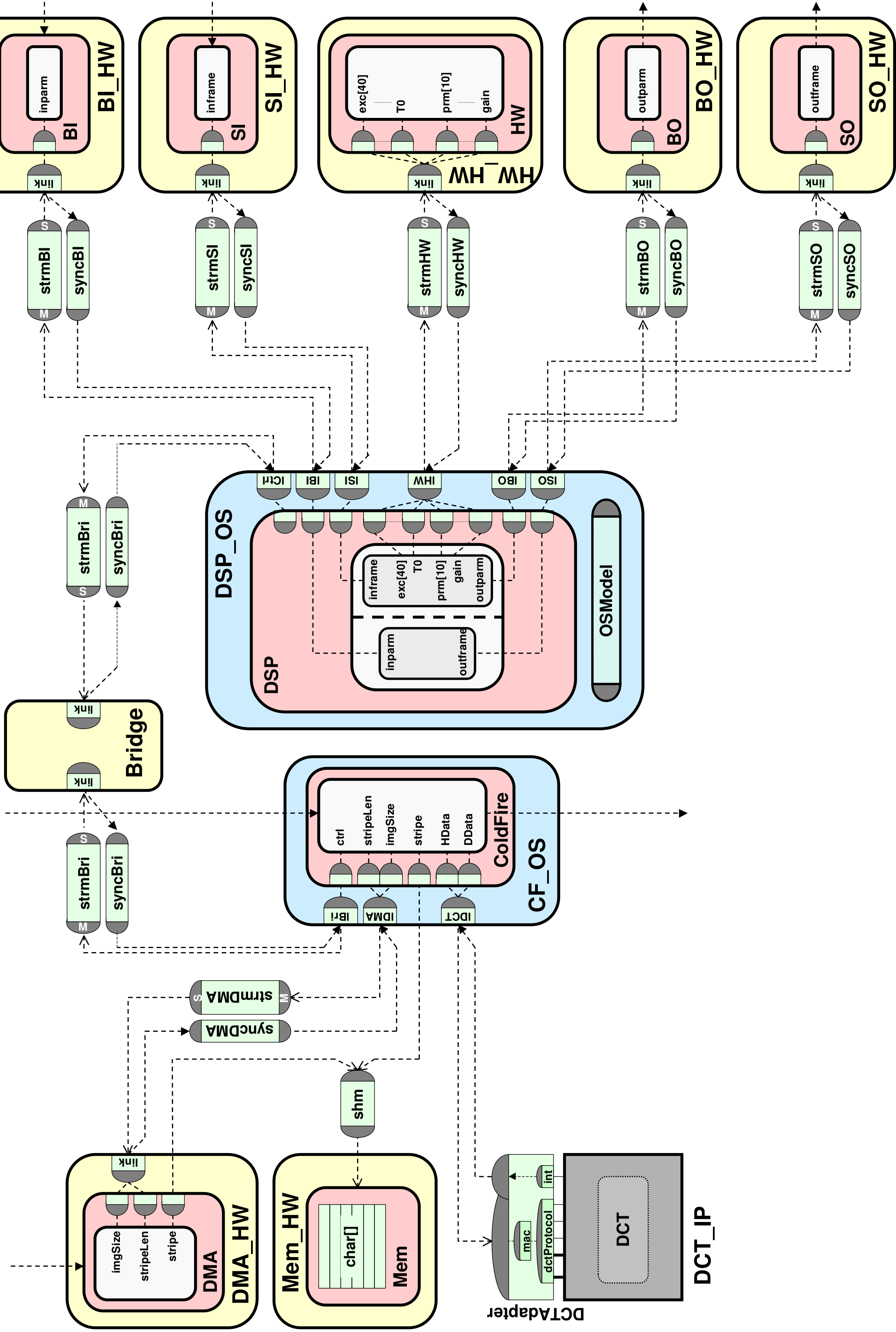


Figure 5.8: Stream model.

5.3.2 Stream Layer

The stream layer provides separate streams for transporting control and data messages from station to station. Data messages are arbitrary, uninterpreted byte blocks. The format of supported control messages, if any, is dependent on specific layer implementations (e.g. simple handshaking in the case of interrupt-driven synchronization). Stream channels are generally asynchronous and unreliable. Reliability of streams may depend on certain assumptions. For example, streams might guarantee reliability (or at least no data loss) with proper prior outside synchronization.

The stream layer is the bottom part of peripheral-specific drivers in the operating system. It is responsible for merging and implementing multiple control and data streams over a common, shared medium. As such, it multiplexes and de-multiplexes streams by separating them in space (but not time) through addressing, for example. Note that since control streams might require very specific access formats, merging them through simple appending of addresses to control messages might not be possible. Other schemes like polling might be required, for example in the case of interrupt sharing.

5.3.2.1 Model Refinement

Stream layer model refinement generally requires insertion of stream layer implementations for both control and data streams inside each station of the system.

Data Streams For each data stream, adapter channels with implementations of its stream layer are inserted into the corresponding PEs. The data stream layers implement multiplexing of data streams over a shared medium.

An example of an adapter channel for data streams is shown in Listing 5.8. For a typical bus based communication, the bus medium will generally support multiple concurrent and overlapping virtual connections through addressing. For each transaction, the bus address of the virtual connection it belongs to needs to be supplied to the bus medium by the data stream adapter (line 5 and line 9). In this case, multiple data streams over the same medium are separated through proper bus addressing where bus addresses are supplied as hardcoded parameters when instantiating the data stream adapters (line 1).

For shared memory communication, a base address and a corresponding range of addresses (base address plus size of memory) is assigned to each shared memory component. Data

```
channel MasterStream(IMasterAccess medium, in const int addr)
  implements IMasterStream
{
  void masterWrite(const void* data, unsigned long len) {
5   medium.masterWrite(addr, data, len);
  }

  void masterRead(void* data, unsigned long len) {
    link.masterRead(addr, data, len);
10 }
};
```

(a) master

```
channel SlaveStream(ISlaveAccess medium, in const int addr)
  implements ISlaveStream
{
  void slaveWrite(const void* data, unsigned long len) {
5   medium.slaveWrite(addr, data, len);
  }

  void slaveRead(void* data, unsigned long len) {
    medium.slaveRead(addr, data, len);
10 }
};
```

(b) slave

Listing 5.8: Data stream layer.

stream adapters inserted into master PEs accessing the memory (Listing 5.9(a)) translate each memory access into an appropriate media access by converting memory offsets into corresponding bus addresses (line 5 and line 9). Inside the memory, similar adapters are inserted (Listing 5.9(b)). The adapters implement listening of the memory to its assigned range of addresses on the medium (line 7) in addition to translation of memory offsets into memory addresses for each access (line 13 and line 16).

Control Streams Implementation of the stream layer of control streams in the media access model generally depends on the underlying medium. For example, control streams can be implemented through normal or specialized media channel transactions. In a typical bus-based communication, control streams are implemented through processor interrupts. For each control stream coming into the master processor, the processor's OS layer exports a method that implements the corresponding handshaking control transaction. Control channel calls inside the slave PEs are translated into calls to the corresponding master method through equivalent adapters (as shown in Listing 5.10 for the example of an adapter that translates *send* calls in the *DMA* slave to calls of the *DMAHandler* method exported by the *ColdFire* processor).

```
channel MasterShm(IMasterMemAccess medium, in const int addr)
  implements IShm
{
  void read(unsigned long ofs, void* data, unsigned long len) {
5   medium.masterMemRead(addr + ofs, data, len);
  }

  void write(unsigned long ofs, const void* data, unsigned long len) {
    medium.masterMemWrite(addr + ofs, data, len);
10 }
};
```

(a) master access

```
channel SlaveMbusShm(ISlaveMemAccess medium, in const int base_addr)
  implements ISlaveShm
{
  bool listen(unsigned long *ofs, unsigned long *len) {
5   bool type;
    unsigned long addr = base_addr;
    type = medium.listen(&addr, len);
    ofs = addr - base_addr;
    return type;
10 }

  void serveRead(unsigned long ofs, const void* data, unsigned long len) {
    medium.serveRead(base_addr + ofs, data, len);
  }
15 void serveWrite(unsigned long ofs, void* data, unsigned long len) {
    medium.serveWrite(base_addr + ofs, data, len);
  }
};
```

(b) slave memory

Listing 5.9: Memory stream layer.

Inside the processors, control handlers implement the handshaking through semaphores that connect to and signal the processor's link layer, replacing the previous control channel. In case the processor is not running an operating system (e.g. the *ColdFire* processor, Listing 5.11), the OS layer of the processor (Listing 5.11(c)) instantiates a minimal kernel as part of the processor's runtime environment (line 3) in addition to instances of data and memory stream adapters (line 5 through line 7 and line 9), link layer implementations (line 13 through line 15) and the instance of the actual software application layer (line 17). For implementation of handshaking, interrupt handlers (line 19, line 22 and line 25) communicate with link layer adapters via instances of special semaphores (line 11). Through *send* methods of handshaking semaphores (Listing 5.11(b)), interrupt handlers set a corresponding flag (line 13) in the runtime environment whenever an interrupt occurs. Receiving of control messages by the link layers is then implemented by polling the respective flag (line 7). In between polling cycles, the main application is suspended through a call

```
channel IntDMA(ICFIntService cf) implements i_send {
  void send(void) {
    cf.DMAHandler();
  }
};
```

Listing 5.10: Control stream translator.

to the processor's runtime kernel (line 8). On top of the processor's instruction set, the runtime kernel (Listing 5.11(a)) in turn blocks the application until the next interrupt is received (line 7). In order to model resuming of processor execution whenever an interrupt occurs, interrupt handlers call a corresponding method of the runtime kernel (line 13) at which point another polling cycle is triggered to check whether the flag has been updated by the interrupt handler. Note that in the actual implementation created during the backend design process, the runtime kernel will be translated into instructions that suspend and resume the processor accordingly[4].

In case the processor is running an operating system (*DSP* processor, Listing 5.12), handshaking through semaphores has to be under the control of the operating system. In addition to data stream implementations (line 5 through line 9) and link layer adapters (line 19), the processor's OS layer instantiates the OS model (line 3) introduced during dynamic scheduling (see Section 4.3.2, Chapter 4). For implementation of handshaking, control handlers (line 24 through line 28) spawn special interrupt handling tasks (line 13 through line 17) that communicate with link layer adapters through regular operating system semaphores (line 11). Interrupt handling tasks serve as the bottom halves of the control handlers and run on top of the operating system next to regular computation tasks (line 30 through line 45). Interrupt handling tasks are modeled (Listing 5.13), as high-priority, aperiodic tasks (line 11) that can be externally triggered by the interrupt handlers (line 15) and in each iteration implement the actual signaling of the semaphores on top of the OS model (line 23) before suspending themselves again (line 22).

5.3.2.2 Media Access Model

The media access model for the design example is shown in Figure 5.9. In the media access model, stations are connected through channels representing the underlying shared communication media. Media channels support transactions for exchanging data packets in the form of uninterpreted blocks of bytes. Due to the shared nature of normal media, media channels usually al-

[4]For example, most processors support an instruction that puts the processor into a sleep state. If the processor automatically leaves the sleep state and wakes up on every interrupt, the *wait_int* method will be translated into a single sleep instruction and the *ireturn* method will be empty.

```
channel OSNone
  implements IOSNone
{
  event intr;

  void wait_int(void) {
    wait(intr);
  }

  void ienter(void) { }

  void ireturn(void) {
    notify intr;
  }
};
```

(a) runtime kernel

```
channel OSNoneSema(IOSNone os)
  implements i_send, i_receive
{
  bool  f = false;

  void receive(void) {
    while (!f)
      os.wait_int();
    f = false;
  }

  void send(void) {
    f = true;
  }
};
```

(b) semaphore

```
behavior CF_OS(IMasterAccess medium) implements ICFIntService
{
  OSNone os;                                         // runtime kernel

  MasterStream DMA(medium, ADDR_DMA);                // data streams
  MasterStream DCT(medium, ADDR_DCT);
  MasterStream Bri(medium, ADDR_BRI);

  MasterShm   shm(medium, ADDR_MEM);                 // memory streams

  OSNoneSema sBri(os), sDMA(os), sDCT(os);           // semaphores

  MasterLink lDMA(DMA,  sDMA);                       // link layer
  MasterLink lDCT(DCT,  sDCT);
  MasterLink lBri(Bri, sBri);

  ColdFire cf(lDMA, lDCT, lBri, shm);                // application

  void DMAHandler(void) {                            // interrupt handlers
    os.ienter(); sDMA.send(); os.ireturn();
  }
  void DCTHandler(void) {
    os.ienter(); sDCT.send(); os.ireturn();
  }
  void BriHandler(void) {
    os.ienter(); sBri.send(); os.ireturn();
  }

  void main(void) { cf.main(); }
};
```

(c) PE OS layer

Listing 5.11: Stream layer for non-OS processor.

```
behavior DSP_OS(IMasterAccess medium) implements IDSPIntHandlers
{
  OS os;                                                    // OS model

  MasterStream SI(medium, ADDR_SPCHIN);                     // data streams
  MasterStream SO(medium, ADDR_SPCHOUT);
  MasterStream BO(medium, ADDR_SEROUT);
  MasterStream BI(medium, ADDR_SERIN);
  MasterStream HW(medium, ADDR_HW);

  OSSema sSI(os), sSO(os), sHW(os), sBI(os), sBO(os);       // semaphores

  IntHandlerTask SIHandlerTask(os, sSI);                    // interrupt tasks
  IntHandlerTask SOHandlerTask(os, sSO);
  IntHandlerTask BIHandlerTask(os, sBI);
  IntHandlerTask BOHandlerTask(os, sBO);
  IntHandlerTask HWHandlerTask(os, sHW);
                                                            // link layer
  MasterLink lSI(SI, sSI), lSO(SO, sSO), lBI(BI, sBI), lHW(HW, sHW);

  DSP dsp(os, lSI, lBI, lHW, lSO, lBO);                     // application

                                                            // interrupt handlers
  void spchInHandler(void)     { spchInHandlerTask.start();    }
  void serOutHandler(void)     { serOutHandlerTask.start();    }
  void serInHandler(void)      { serInHandlerTask.start();     }
  void spchOutHandler(void)    { spchOutHandlerTask.start();   }
  void hwHandler(void)         { hwHandlerTask.start();        }

  void main(void) {
    dsp.init();
    spchInHandlerTask.init();
    spchOutHandlerTask.init();
    serInHandlerTask.init();
    serOutHandlerTask.init();
    hwHandlerTask.init();
    par {
      dsp.main();
      spchInHandlerTask.main();
      spchOutHandlerTask.main();
      serInHandlerTask.main();
      serOutHandlerTask.main();
      hwHandlerTask.main();
    }
  }
};
```

Listing 5.12: Stream layer for processor with OS.

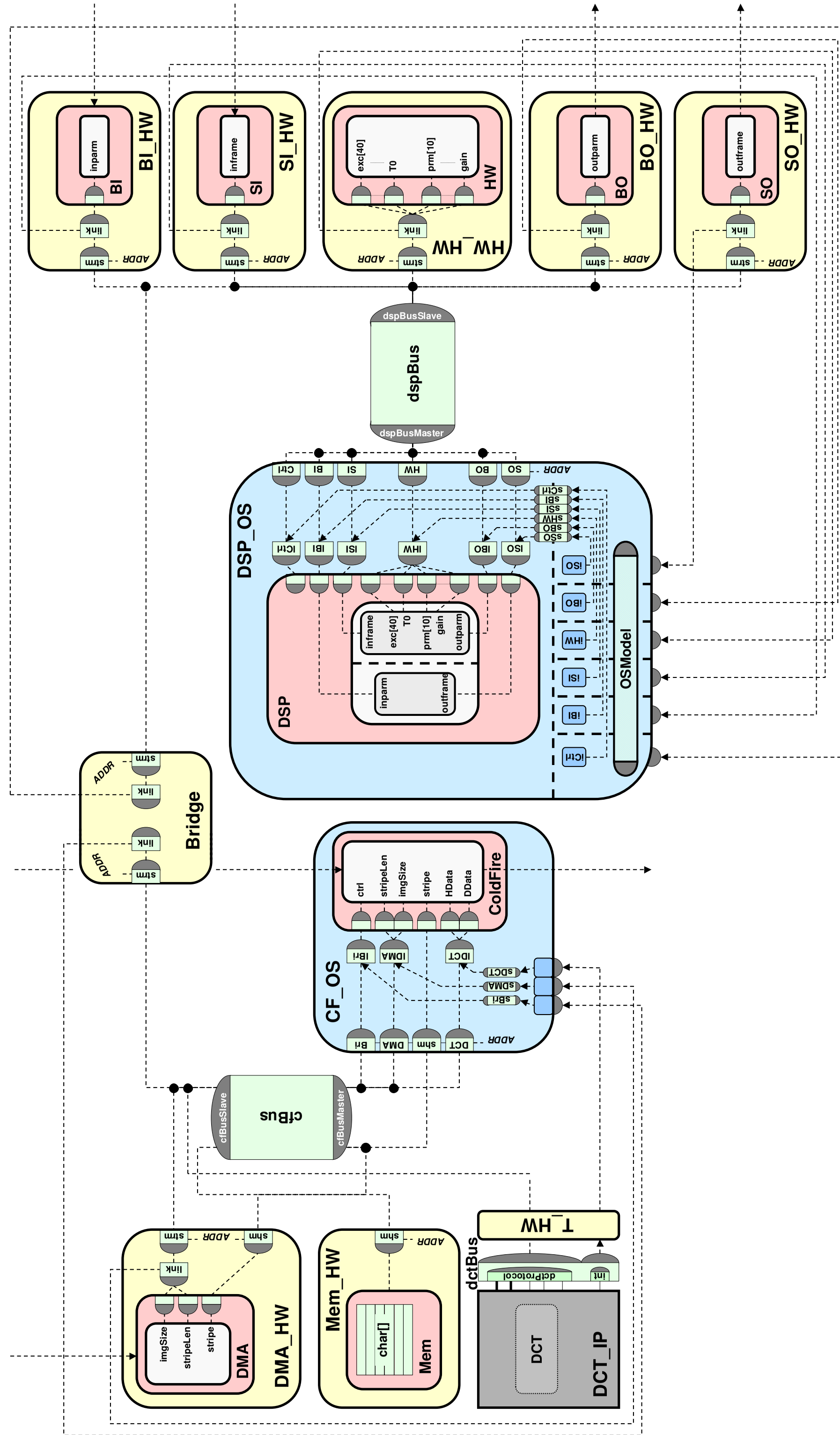


Figure 5.9: Media access model.

```
interface IIntHandlerTask {
  void start(void);
};

behavior IntHandlerTask(RTOS os, i_send s)
  implements Init
{
  Task h;

  void init(void) {
    h = os.task_create("Int", APERIODIC, 0);
  }

  void start(void) {
    os.task_resume(h);
  }

  void main(void)
  {
    os.task_activate(h);
    while(true) {
      os.task_sleep();
      s.send();
    }
  }
};
```

Listing 5.13: Interrupt handler task for control streams.

low multiple virtual multi-point connections over them that can occur concurrently, simultaneously and overlapping in time. In general, media channels are asynchronous and they may or may not be reliable and buffered. However, media channels reflect and model the underlying communication medium. Therefore, the semantics and exact format of the transactions they support depend on the behavior and capabilities of the chosen medium.

As defined previously, the example containts two busses, *cfBus* for the ColdFire subsystem and *dspBus* for the DSP subsystem. Mirroring the master/slave behavior of bus-based communication, both media channels are unbuffered, error-free, and asynchronous on the master side and synchronous on the slave side. Note that bus media channels are not reliable as packets can be lost if the slave is not waiting when the master initiates a transaction. For communication with shared memory components, the interfaces of the media channels supports special split transactions that allow the memory component to listen on the medium for a range of addresses and then serve the transaction request after proper address decoding.

Since both in general and in this example, the medium for communication with an IP component is proprietary and not 100% compatible with other general media in the system (as signified by different semantics and/or different media interfaces), additional transducer stations that translate

between different media interfaces have to be inserted in front of each IP in the media access model. In this example, transducer *T_DCT* performs the necessary address translations between *cfBus* and *dctBus*.

5.3.3 Media Access Layer

The media access layer provides services to transfer blocks of bytes over channels representing shared media between stations. Depending on the type of medium, different categories of transactions or different categories of information within a transaction might be supported by a channel (e.g. distinction of address, control, and data). In general, a medium is asynchronous and unreliable. It usually requires prior outside synchronization to avoid data loss. Furthermore, a medium may or may not be error-free.

The media access layer is responsible for slicing blocks of bytes into unit transfers available at the interface. In the process, it's implementation has to guarantee that the rates of successive transfers within a block match for all communication partners. Furthermore, the media access layer regulates and separates simultaneous accesses in time (e.g. through arbitration). Depending on the scheme chosen, additional arbitration stations are introduced into the system as part of the media access layer.

For programmable PEs, the media access layer is part of the PE's hardware abstraction layer (HAL). Through the media access layer, the HAL provides a canonical interface for accessing the PE's hardware communication features from the programmable computation inside. At its external interface, the HAL defines the boundary between hardware and software. Hence, it is the lowest layer of software in programmable PEs.

For each programmable PE, a HAL model is stored in the PE database and imported during refinement. HAL models in the database provide canonical communication services for stream I/O, memory I/O and interrupt handling. HAL services abstract the underlying PE hardware and define the PE's corresponding capabilities (number and type of external interfaces, amount and level of interrupts, etc.) [44]. During media access layer refinement, HAL models are inserted as an additional layer around programmable PEs. Database HAL models act as templates on top of which computation and communication of higher PE layers is implemented during refinement.

5.3.3.1 Model Refinement

The media access layer implements a medium's data, memory, control, and arbitration transactions on top of the medium's actual protocol. Therefore, media access layer refinement requires insertion of corresponding implementations inside the stations of the system network, generally in the form of corresponding media access adapters.

Media access adapters are usually imported out of the media database. For each communication medium in the database, the database contains models of all types of required media access adapters [44]. During refinement, adapter models are imported out of the database and instantiated and connected as needed (depending on the type of access and the type of media connection). In the case of programmable PEs, adapters are part of, pre-instantiated in, and imported into the design together with the PE's HAL model. In order to be able to support any type of transaction, database HAL models for programmable PEs contain instances of all types of media access adapters associated with the PE's interfaces.

Data Transactions For each data stream going in and out of a station, media access layer implementations are inserted in the form of media access layer adapters. Adapters slice abstract byte blocks of data packets at the interface of the media access layer into individual, bit-accurate data words or data frames supported by the underlying media protocol.

An example of a media access layer adapter for normal message-passing type data transactions over a typical bus medium is shown in Listing 5.14 (shown for the master side; the slave side is implemented in a similar manner). The data adapter slices the arbitrary-length packets of data supplied to the *read* and *write* methods into bus words. Using protocol primitives for byte and long word transfers, the methods loop over the data, transmitting as much data as possible in each bus cycle. In case of regular message-passing transfers, virtual connections at the protocol level are only needed to distinguish among different message streams. Hence, the same bus address is used for all successive transfers in a message packet.

Memory Transactions Similar to regular data transactions, memory access streams require insertion of corresponding media access adapters (Listing 5.15). Adapters perform slicing of high-level data blocks into individual data words supported by the medium and the memory.

In case of memory accesses, however, random access has to be supported and data bytes in all memories attached to the bus have to be individually distinguishable. Therefore, protocol

```
channel MasterAccess(IMaster protocol) implements IMasterAccess
{
  void masterWrite(unsigned long addr, const void* data, unsigned long len) {
    unsigned char* p;

    for(p = (unsigned char*)data; len >= 4; len -= 4, p += 4)
      protocol.masterWriteLong(addr[31:2], (*p)[7:0] @ (*(p+1))[7:0] @
                                           (*(p+2))[7:0] @ (*(p+3))[7:0]);
    for( ; len; len--, p++)
      protocol.masterWriteByte(addr, *p);
  }

  void masterRead(unsigned long addr, void* data, unsigned long len) {
    unsigned char* p;
    bit[31:0] l;
    bit[7:0]  b;

    for(p = (unsigned char*)data; len >= 4; len -= 4, p += 4) {
      protocol.masterReadLong(addr[31:2], &l);
      *p = l[31:24]; *(p+1) = l[23:16]; *(p+2) = l[15:8]; *(p+2) = l[7:0];
    }
    for( ; len; len--, p++) {
      protocol.masterReadByte(addr, &b);
      *p = b;
    }
  }
};
```

Listing 5.14: Media access layer data transactions.

addresses have to be used to distinguish among individual addressable units (characters/words) in the memory. Each unit holds a certain amount of bytes as defined by the medium and consecutive bytes in memory are accessed as consecutive characters on the medium. Consequently, each memory access transaction spans a range of addresses. Starting with the base address supplied by higher layers, addresses are incremented accordingly for successive protocol transfers in a block. In addition, since base addresses supplied by higher layers can not be assumed to be properly aligned, the data adapter has to take care of proper alignment of data on the medium. Therefore, misaligned data at the beginning of the data block has to be transfered via transactions that do not utilize the full bus capacity.

On the memory side, adapters are inserted that translate abstract addresses and data byte blocks into actual, bit-accurate protocol addresses and words in a similar manner (Listing 5.16). A memory listens on the medium for any transfer that accesses its assigned range of addresses (line 7), and it simply serves the requested access by mapping it into a corresponding sequence of protocol transfers (line 13 and line 23).

```
channel MasterMemAccess(IMaster protocol) implements IMasterMemAccess
{
  void masterMemWrite(unsigned long addr, const void* data, unsigned long len) {
    unsigned char *p;

    for(p = (unsigned char*)data; len && (addr % 4); p++, len--)
      protocol.masterWriteByte(addr++, *p);
    for(; len >= 4; p += 4, len -= 4)
      protocol.masterWriteLong(addr[31:2]++, (*p)[7:0] @ (*(p+1))[7:0] @
                               (*(p+2))[7:0] @ (*(p+3))[7:0]);
    for(; len; p ++, len--)
      protocol.masterWriteByte(addr++, *p);
  }

  void masterMemRead(unsigned long addr, void* data, unsigned long len) {
    unsigned char *p;
    bit[31:0] l;
    bit[7:0]  b;

    for(p = (unsigned char*)data; len && (addr % 4); p++, len--){
      protocol.masterReadByte(addr++, &b);
      *p = b;
    }
    for(; len >= 4; p += 4, len -= 4) {
      protocol.masterReadLong(addr[31:2]++, &l);
      *p = l[31:24]; *(p+1) = l[23:16]; *(p+2) = l[15:8]; *(p+2) = l[7:0];
    }
    for(; len; p ++, len--){
      protocol.masterReadByte(addr++, &b);
      *p = b;
    }
  }
};
```

Listing 5.15: Media access layer for memory access in the master.

```
channel SlaveMemAccess(ISlave protocol) implements ISlaveMemAccess
{
  bool listen(unsigned long *addr, unsigned long *len) {
    unsigned bit[2] type;
    bit[31:0] a    = *addr;
    bit[31:0] mask = (len-1);
    type = protocol.slaveListen(&a, ~mask);
    *addr = a;
    *len = type[1];
    return type[0] == 0b;
  }

  void serveRead(unsigned long addr, const void* data, unsigned long len) {
    unsigned char *p;

    for(p = (unsigned char*)data; len >= 4; p += 4, len -= 4)
      protocol.serveReadLong(addr[31:2]++, (*p)[7:0] @ (*(p+1))[7:0] @
                                           (*(p+2))[7:0] @ (*(p+3))[7:0]);
    for(; len; p++, len--)
      protocol.serveReadByte(addr++, *p);
  }

  void serveWrite(unsigned long addr, void* data, unsigned long len) {
    unsigned char *p;
    bit[31:0] l;
    bit[7:0]  b;

    for(p = (unsigned char*)data; len >= 4; p += 4, len -= 4) {
      protocol.serveWriteLong(addr[31:2]++, &l);
      *p = l[31:24]; *(p+1) = l[23:16]; *(p+2) = l[15:8]; *(p+2) = l[7:0];
    for(; len; p++, len--){
      protocol.serveWriteByte(addr++, &b);
      *p = b;
    }
  }
};
```

Listing 5.16: Media access layer for memory slave.

```
channel MasterMAC(IMaster protocol, i_semaphore access) implements IMaster
{
  void masterReadByte(bit[31:0] addr, bit[7:0] &val) {
    access.acquire();
    protocol.masterReadByte(addr, val);
    access.release();
  }

  void masterReadLong(bit[31:2] addr, bit[31:0] &val) {
    access.acquire();
    val = protocol.masterReadLong(addr);
    access.release();
  }

  void masterWriteByte(bit[31:0] addr, bit[7:0] val) {
    access.acquire();
    protocol.masterWriteByte(addr, val);
    access.release();
  }

  void masterWriteLong(bit[31:2] addr, bit[31:0] val) {
    access.acquire();
    protocol.masterWriteLong(addr, val);
    access.release();
  }
};
```

Listing 5.17: Media access layer arbitration.

Arbitration In contrast to the semantics of media access layer transactions provided for higher layers, the underlying protocol only supports one active transaction at any given time, even when coming from different virtual connections. In addition to slicing of data packets into protocol units, therefore, media access layer implementation requires contention resolution according to the media's access protocol in order to regulate conflicting accesses to the underlying protocol.

In case of typical bus-based communication, contention can happen if multiple masters are accessing the same bus. In this case, contention is usually resolved using a distributed or centralized arbitration scheme. In both cases, bus masters implement arbitration around each transfer by requesting and releasing access to the bus through a separate arbitration protocol. As part of the media access layer, corresponding arbitration adapters are inserted in each master. An arbitration adapter re-implements protocol transfers by wrapping calls to primitives for acquiring and releasing the medium provided by the arbitration protocol around each data transfer protocol method (Listing 5.17).

Control Transactions In general, the media access layer can implement specialized control transactions based on control transfers provided by the underlying media protocol. Typical bus-based

```
behavior CF_HAL(IMaster protocol, i_semaphore access) implements ICFIntVectors
{
  MasterMAC        mac(protocol, access);      // arbitration
  MasterLinkAccess link(mac);                  // data transactions
5 MasterMemAccess  mem(mac);                   // memory transactions

  CF_OS cf_os(link, mem);                      // OS layer

                                               // interrupt handlers
10 void int0handler(void)     { cf_os.DMAHandler(); }
   void int1handler(void)     { cf_os.DCTHandler(); }
   void int2handler(void)     { cf_os.BriHandler(); }
   void int3handler(void)     { }
   void int4handler(void)     { }
15 void int5handler(void)     { }
   void int6handler(void)     { }
   void int7handler(void)     { }

   void main(void)            { cf_os.main(); }
20 };
```

Listing 5.18: PE hardware abstraction layer (HAL).

communication provides a control protocol for handshaking in the form of interrupts from bus slaves to bus masters. As part of the media access layer implementation, interrupt control handlers exported by the master are refined down to the level of actual hardware interrupts available in the processors.

The HAL models imported out of the PE database export empty methods that represent the processor's interrupt handlers and that correspond to the actual interrupt vectors and interrupt sources supported by the processor hardware. As part of media access refinement, interrupt handlers in the PE HAL models are filled and implemented with code that calls the appropriate slave handler in the processor's OS layer.

An example of a resulting processor HAL model is shown in Listing 5.18 for the example of the *ColdFire* processor. Apart from the processor's OS layer (line 7), the HAL model instantiates adapters for media access layer arbitration (line 3), data transactions (line 4) and memory accesses (line 5). In a straightforward implementation, each processor interrupt is assigned to one slave. Slaves call the assigned interrupt handler exported by the PE HAL model through adapters that translate control calls in the slaves to calls to interrupt handlers in the same manner as in stream layer refinement (see Listing 5.10). Implementation of interrupt handlers in the HAL then simply call the control handler for the corresponding slave in the OS layer (line 10 through line 12).

In case of interrupt sharing due to a limited number of interrupt sources in a processor, interrupt handlers in the HAL implement polling of slaves (shown in Listing 5.19 for the example

```
behavior DSP_HAL(IMaster protocol) implements IDSPIntVectors
{
  MasterAccess link(protocol);          // data transactions

  DSP_OS dsp_os(link);                  // OS layer

  void intAhandler(void) {              // interrupt handlers
    dsp_os.ctrlHandler();
  }

  void intBhandler(void) {
    bit[7:0] spchIn, serOut;

    protocol.masterReadByte(ADDR_POLL_SPCHIN, &spchIn);
    protocol.masterReadByte(ADDR_POLL_SEROUT, &serOut);

    if (spchIn)
      dsp_os.spchInHandler();
    if (serOut)
      dsp_os.serOutHandler();
  }

  void intChandler(void) {
    bit[7:0] serIn, spchOut;

    protocol.masterReadByte(ADDR_POLL_SERIN, &serIn);
    protocol.masterReadByte(ADDR_POLL_SPCHOUT, &spchOut);

    if (serIn)
      dsp_os.serInHandler();
    if (spchOut)
      dsp_os.spchOutHandler();
  }

  void intDhandler(void) {
    dsp_os.hwHandler();
  }

  void main(void)               { dsp_os.main(); }
};
```

Listing 5.19: PE hardware abstraction layer (HAL) with interrupt sharing.

```
channel SlavePolledInt(ISlave protocol, i_send intr, in const int poll_addr)
  implements i_send
{
  void send(void) {
    intr.send();
    protocol.slaveWriteByte(poll_addr, 1);
  }
};
```

Listing 5.20: Media access layer slave interrupt polling.

of the *DSP* PE HAL with shared I/O processor interrupts). Polling determines the actual interrupt source such that the right OS control handler can be called. Interrupt handlers for shared interrupts perform polling of slaves over the bus through appropriate media access layer transactions (line 14 and line 26) before calling the appropriate control handler in the OS layer. In the slaves participating in the polling, polling is implemented through additional adapter channels in the control path. An extra layer of adapter channels is inserted as part of media access layer refinement (Listing 5.20). After sending the actual interrupt (line 5), the adapter waits and answers matching polling requrests that arrive over the protocol (line 6).

5.3.3.2 Protocol Model

The protocol model of the design example is shown in Figure 5.10. The protocol model includes implementations of the media access layer in the form of adapter channels in all stations connected to a medium. For each media connection in a station, a corresponding media access adapter of the right type (e.g. master or slave access) is instantiated. For the software processors, the media access layer becomes part of the processor's hardware abstraction layer (HAL). Corresponding processor layers *CF_HAL* and *DSP_HAL* that instantiate the media access layer adapters are inserted for the ColdFire and DSP processors, respectively.

In the protocol model, media channels have been replaced with shared protocol channels connecting the stations. Protocol channels model and provide all the possible transactions supported over the actual physical medium. The media access layers use the different services and transfer models available in the protocol to efficiently implement media transfers, slicing data packets into actual data transfer units (words or frames) supported by the protocol on the physical medium. In the case of the example shown here, bus protocols support transactions for transfers of standard 24-bit bus words with 16-bit addresses in case of the *dspProtocol* or for byte, word, and long-word transfers with 32-bit addresses in case of the *cfProtocol*.

Since the media access layer does not implement any additional functionality, transaction semantics of protocol channels and media channels in the media access model are equivalent. Generally, protocol channels are asynchronous and may or may not be reliable and buffered. Their actual semantics and transaction format, however, are directly dependent on and a direct reflection of the behavior and capabilities of the protocol they represent. In the case of the example, bus protocols are asynchronous on the master side[5], synchronous on the slave side, error-free but not reliable

[5] Specifically, in order to enable polling, methods on the master side must not block even if no corresponding slave is available to complete the transfer.

Figure 5.10: Protocol model.

against data loss (in case of not properly synchronized transfers), and they support multiple virtual connections via bus addressing. Furthermore, for shared memory transfers, protocol channels provide split transactions that allow the shared memory to listen to and serve incoming accesses.

Since both *DMA* and *ColdFire* are masters on the *cfProtocol*, their media access layer adapters implement bus arbitration before each transfer by connecting to and communicating via an additional arbitration protocol channel *Arbiter*. Arbitration channels in the protocol model are standard channels with semaphore semantics. They provide an abstraction of the underlying arbitration protocol transactions for acquiring and releasing access to the medium. Depending on the type of arbitration protocol (centralized or distributed), arbitration channels represent communication of media masters among themselves or with additional arbiter components.

5.3.4 Protocol Layer

The protocol layer provides services to transfer words or frames (i.e. groups of bits) over a physical medium. Depending on the transfer modes supported by the medium, different types of transactions might be available. A protocol layer channel is asynchronous, unbuffered, lossy, and it may or may not be error-free. Since there is no buffering, it requires proper outside synchronization to provide lossless communication or any communication at all.

The protocol layer is responsible for driving and sampling the external pins according to the protocol timing diagrams and thereby matching the transmission timing on the sender and receiver sides. As part of the protocol layer, repeater stations are introduced as necessary. Repeaters connect physical wire segments with matching protocols in order to represent them as one common medium

The protocol layer is the implementation of the medium's transfer protocol in the hardware of a station's interface. As such, it is implemented as part of a station's hardware layer. For programmable PEs, an additional layer of hierarchy representing the PE's hardware is therefore inserted around the PE's HAL model. The protocol layer is inserted into this new hardware layer during refinement.

For all PEs and CEs with fixed, pre-defined interfaces and communication functionality, a bus-functional model of the component that accurately describes the PE/CE interface at the pin level and provides a model of the PE's or CE's communication aspects has to be stored in the corresponding database [44]. For such components, bus-functional models that include protocol layer implementations are imported out of the database and instantiated around the existing component

models during protocol layer refinement. In case of programmable PEs, bus-functional models in the database provide the PE's pre-defined hardware layer that wraps around and connects to the PE's previously imported HAL.

In case of IP components with fixed, pre-defined computation and communication functionality, a bus-functional model provides a timing- and data-accurate description of the complete IP functionality in terms of signals observed at the IP's pins. Bus-functional IP models with associated wrappers are imported at the beginning of the communication design process (see Section 5.2.1). During protocol layer refinement, the last level of the IP wrapper is inlined into connected stations and the IP's bus-functional model is exposed.

5.3.4.1 Model Refinement

During protocol layer refinement, implementations of the protocol layer in the form of protocol adapters are inserted into the network stations of the system. In general, for each transfer supported by the protocol, a protocol layer adapter drives and samples the media wires connected to it according to the corresponding timing diagram. Protocol adapters can be passive (channels) or active (behaviors). In the latter case, protocol adapters run in parallel to the rest of the component hardware in order to enable support for protocols that require a component to constantly listen and participate in the media protocol.

Protocol layer implementations in the form of adapters are stored in the media database for all protocols supported by each medium. The set of protocol adapters and the set of transfers supported by each adapter define the capabilities of a medium stored in the database [44]. During protocol layer refinement, appropriate protocol adapters are then imported out of the library, instantiated inside the stations, and connected as necessary.

A medium in general is associated with different, separate protocols for different types of transfers. Specifically, a medium can support a data transfer protocol for exchange of data among components and memories, a control transfer protocol for synchronization and handshaking, and an arbitration protocol for regulation of media accesses and resolution of media contention.

Data Transfer Protocol The data transfer protocol is the core of any medium. It describes bit-accurate primitives for transferring native words or frames distinguished by media addresses. A data transfer protocol provides methods for all atomic cycles available over the medium, including special types like burst mode, etc.

```
   channel Master(
       out signal bit[31:0] Addr,
           signal bit[31:0] Data,
       out signal bit[3]    Ctrl)
 5   implements IMaster
   {
     void masterReadLong(bit[31:2] A,
                         bit[31:0] *D)
     {
10     do {
         l1: Addr = A @ 00b;
             Ctrl = 111b;
             waitfor(10);
         l2: *D = Data;
15           Ctrl = 000b;
       } timing {
         range(l1; l2; 10; );
       }
     }
20
     void masterWriteLong(bit[31:2] A,
                          bit[31:0] D)
     {
       do {
25       l1: Addr = A @ 00b;
             Data = D;
             Ctrl = 011b;
             waitfor(10);
         l2: Ctrl = 000b;
30     } timing {
         range(l1; l2; 10; );
       }
     }

35   ...

   };
```

(a) master

```
   channel Slave(
       in  signal bit[31:0] Addr,
           signal bit[31:0] Data,
       in  signal bit[3]    Ctrl)
 5   implements ISlave
   {
     bit[2] slaveListen(bit[31:0] &A,
                        bit[31:0] M) {
       do {
10       wait(rising Ctrl);
       } while((Addr & M) != *A);
       *A = Addr;
       return Ctrl[2:1];
     }
15
     void serveReadLong(bit[31:2] A,
                        bit[31:0] D) {
       do {
         l1: waitfor(5);
20       l2: Data = D;
       } timing {
         range(l1; l2; ; 10);
       }
     }
25
     void serveWriteLong(bit[31:2] A,
                         bit[31:0] *D) {
       do {
         l1: waitfor(5);
30       l2: *D = Data;
       } timing {
         range(l1; l2; ; 10);
       }
     }
35
     ...
   };
```

(b) slave

Listing 5.21: Data transfer protocol layer.

Depending on the type of station, different data transfer protocol implementations might be available. An example of a typical bus protocol with two separate passive models for master and slave sides is shown in Listing 5.21. The data transfer protocol provides primitives for reading and writing data operands of different sizes from/to the bus in one cycle. Since bus addresses have to be aligned correctly, the slice of addresses accepted by each primitive depends on the size of the operand. On the slave side (Listing 5.21(b)), the protocol supports split transactions needed for implementation of memory slaves. Split transactions are separated into listening for a range of addresses on the bus (line 7) and serving the corresponding transfer (line 16, line 26, etc.).

Inside the body of the adapters, the sequence of statements inside the methods drive and sample the address, data, and control wires of the bus. Furthermore, the protocol state machines

```
channel MasterArbitration(out signal bit[1] request, in signal bit[1] grant)
  implements i_semaphore
{
  void acquire(void) {
5   request = 1;
    while (!grant) wait (rising grant);
  }

  void release(void) {
10  request = 0;
  }
};
```

Listing 5.22: Arbitration protocol layer.

are enclosed in `do-timing` constructs to specify the constraints on timing that have to be obeyed when implementing the protocol.

Arbitration Protocol A medium can support an arbitration protocol to regulate accesses to the shared medium and resolve any contention among multiple components trying to access the medium at the same time. During protocol layer refinement, corresponding arbitration protocol adapters are inserted inside each station participating in the arbitration protocol.

Arbitration adapters with standard semaphore semantics provide primitives for acquiring and releasing access to the medium (Listing 5.22, line 4 and line 9). Internally, they implement the arbitration protocol by appropriately driving and sampling the wires of the arbitration bus in the same manner as the data transfer protocol. In case of a typical bus medium, corresponding arbitration adapters are inserted next to the data transfer protocol into the hardware layers of each master connecting to the bus.

Arbitration protocols can be distributed or centralized. In a distributed scheme, (active) arbitration adapters in each media master regulate media accesses among themselves. In a centralized scheme, an additional arbiter component that connects to and implements the slave side of the arbitration bus is inserted into the system architecture as part of protocol layer refinement. Bus-functional arbiter models allocated and inserted during refinement are imported out of the CE database (if available as IP) or synthesized using arbitration protocol slave adapters from the media database. Arbitration slaves describe the arbiter side of the protocol similar to their corresponding master adapters (see Listing 5.22) [44].

Control Transfer Protocol A data transfer protocol generally only supplies limited inherent synchronization and handshaking functionality (e.g. only one-way synchronization from master to slave

```
channel IntDetect(
    in signal bit[1] intr
    )
  implements i_receive
{
  void receive(void)
  {
    wait(rising intr);
  }

};
```

(a) master

```
channel IntGenerate(
    out signal bit[1] intr
    )
  implements i_send
{
  void send(void) {
    intr = 1;
    waitfor(5);
    intr = 0;
  }
};
```

(b) slave

Listing 5.23: Interrupt protocol layer.

in a bus-based medium). However, in order to implement reliable communication with guaranteed data delivery, two-way synchronization between communication partners is required. Therefore, a medium can supply an optional, distinct synchronization protocol to efficiently send events. In a typical bus medium, this usually means an interrupt protocol and interrupt wires through which a slave can send interrupts to a master.

Control protocols provide primitives for sending and receiving events. In the case of an interrupt protocol, the control protocol adapters internally describe the logic required to detect interrupts on the master side and to generate interrupts on the slave side (Listing 5.23).

As part of protocol layer refinement for control protocols, interrupt handling in programmable PEs is refined down to a bit-, pin-, and timing-accurate level. Models of hardware layers for programmable PEs stored in the database and imported during refinement include a description of the PE's interrupt behavior of detecting interrupts, suspending regular computation, and executing the appropriate HAL interrupt handlers. Such hardware models provide a description of the functionality of the interrupt control logic implemented in the PE's real hardware.

An example of a processor hardware model for the *DSP* PE is shown in Listing 5.24. Inside its hardware layer (Listing 5.24(b)), the processor hardware model instantiates an interrupt service routine (ISR, line 8) in addition to the protocol layer adapter (line 6) and the processor's HAL model (line 7). Interrupt behavior is modeled by executing the HAL (line 12) under the control of the interrupt service routine (line 15). The ISR gets triggered and interrupts normal computation whenever an external interrupt condition signal *INTR* becomes true.

Internally (Listing 5.24(a)), the ISR processes pending interrupts as long as the interrupt conditions is true and the bus is not busy[6]. If there is a pending interrupt (line 6), the interrupt logic

[6] In order to maintain bus protocol timing, normal computation must not be interrupted in the middle of a bus transaction.

```
behavior DSP_ISR(IMaster protocol, IDSPIntVectors vectors,
                 in signal bit[1] INTR, in signal bit[1] busy)
{
  void main(void) {
5   unsigned bit[7:0] vec;
    while(INTR && !busy)          // process pending interrupts
    {                             // acknowledge interrupt, read vector
      protocol.masterReadByte(ADDR_PIC, &vec);
      switch(vec) {               // call corresponding interrupt handler
10      case 0: return handlers.intAhandler();
        case 1: return handlers.intBhandler();
        case 2: return handlers.intChandler();
        case 3: return handlers.intDhandler();
      }
15  }
  }
};
```

(a) interrupt control logic

```
behavior DSP_HW(out signal bit[31:0] Addr,
                    signal bit[31:0] Data,
                    signal bit[3]    Ctrl,
                in  signal bit[1]    INTR)
5 {
  Master  protocol(Addr, Data, Ctrl);           // data transfer protocol
  DSP_HAL hal(protocol);                        // hardware abstraction layer
  DSP_ISR isr(protocol, hal, INTR, Ctrl[0]);    // interrupt logic

10 void main(void) {
    try {
      hal.main();
    }
    interrupt(INTR, Ctrl) {
15    isr.main();
    }
  }
};
```

(b) hardware layer

Listing 5.24: Processor core hardware model.

communicates with the interrupt controller over the PE bus to acknowledge the interrupt and receive the interrupt vector (line 7). It then subsequently calls the corresponding interrupt handler in the HAL (line 10 through line 13) before eventually resuming normal processor execution.

In order to support more complex interrupt capabilities with more than one source of interrupts, different priorities, masking, etc., hardware models for programmable PEs in the PE database usually include models of interrupt controllers associated with the processor. Interrupt controllers are system components that sit in front of and are combined with the hardware model into a bus-functional PE model that encapsulates both through an additional layer of hierarchy (Listing 5.25).

```
behavior DSP_BF( signal bit[31:0] Addr,        // bus
                 signal bit[31:0] Data,
                 signal bit[3]    Ctrl,
              in signal bit[1]    intA,        // interrupts
5             in signal bit[1]    intB,
              in signal bit[1]    intC,
              in signal bit[1]    intD)
{
  signal bit[1] INTR = 0;                      // interrupt line
10
  DSP_HW  hw(Addr, Data, Ctrl, INTR);
  DSP_PIC pic(Addr, Data, Ctrl, INTR, intA, intB, intC, intD);

  void main(void) {
15  par {
      hw.main();
      pic.main();
    }
  }
20 };
```

Listing 5.25: Bus-functional processor model.

Bus-functional models of interrupt controllers that are part of the bus-functional PE models in the database perform interrupt detection on the actual interrupt wires, signal interrupt conditions to the processor hardware model, and deliver interrupt vectors to ISRs over the processor bus. Interrupt detection in the interrupt controllers is performed through instances of corresponding interrupt protocol master adapters (see Listing 5.23(a)). On the other hand, interrupt controllers connect to the processor bus wires as bus slaves through corresponding instances of the bus' data transfer protocol slave adapters (see Listing 5.21(b)).

5.3.4.2 Communication Model

The final bus-functional or physical model of the system design example is shown in Figure 5.11. As the end-result of the communication design flow, it is equivalent to the communication model in the overall design methodology. In the bus-functional model, components are connected and communicating through signals representing and modeling the actual physical wires of the chosen medium. Signals provide the associated driving and sampling semantics of wires. In the case of the example, components are connected through sets of wires forming the busses in the system.

Inside the components of the bus-functional model, implementations of the data transfer protocol in the form of adapter channels are inserted. For the two programmable processors, additional *CF_HW* and *DSP_HW* hardware layers that include protocol adapter instances have been imported from the database and instantiated around the previously imported HALs.

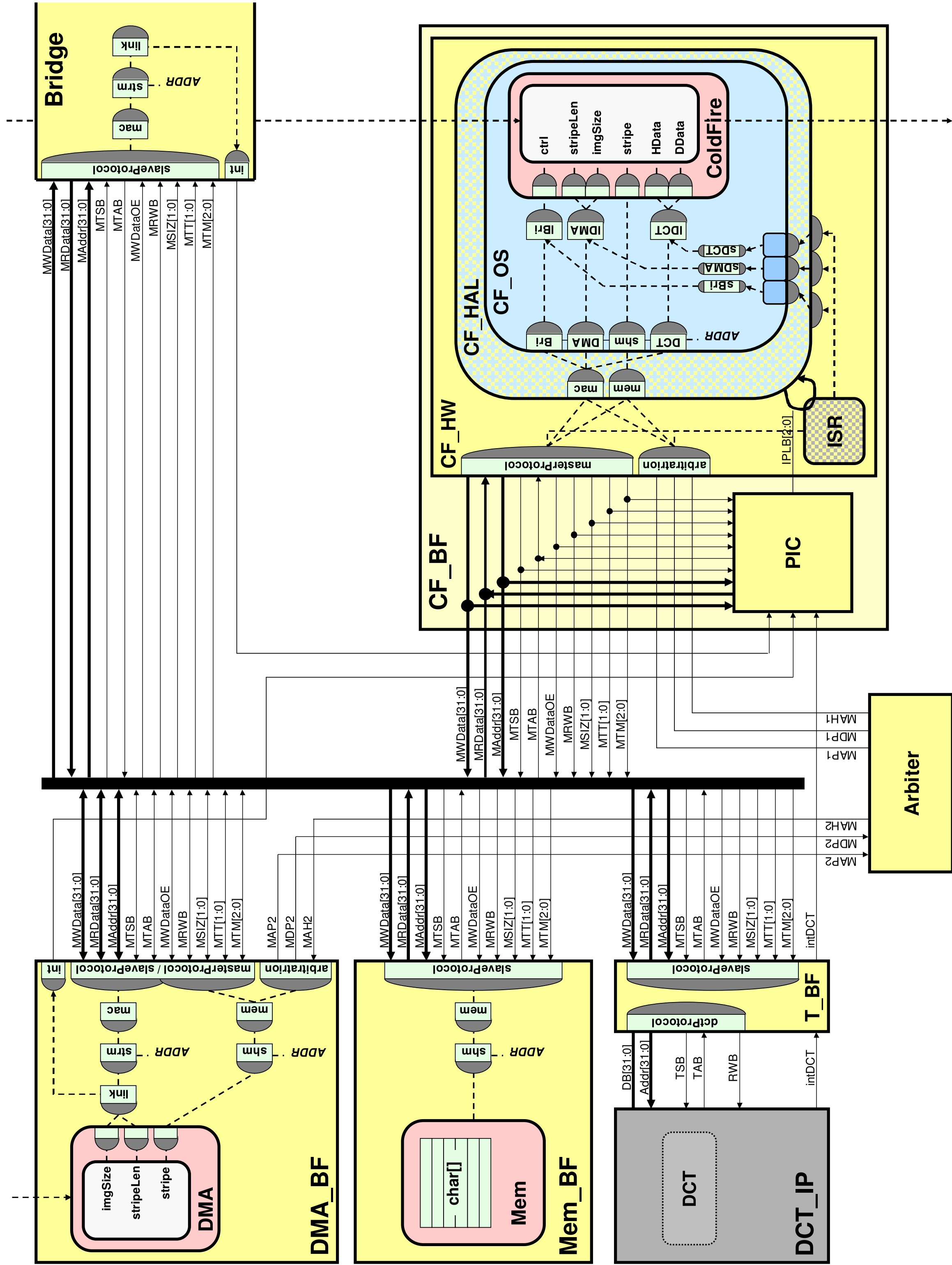


Figure 5.11: Communication model.

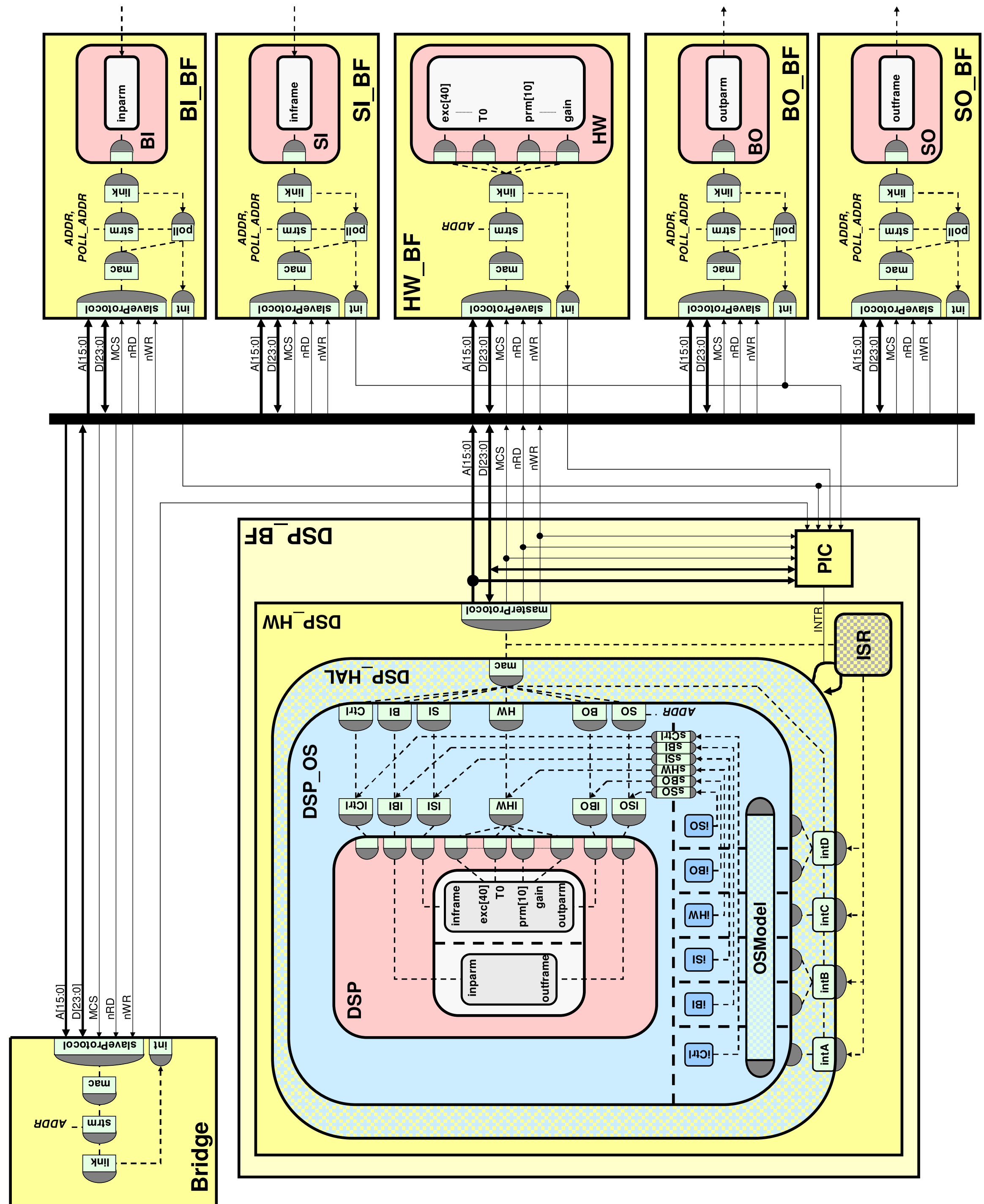


Figure 5.11: Communication model (continued).

Hardware layers of processors from the database also include a model of the processor's interrupt behavior. Inside the hardware layer, the processor HAL model runs under control of an interrupt service routine (*ISR*) that, in connection with external interrupt controllers (*PIC*), interrupts normal HAL execution and calls corresponding interrupt handlers in the HAL whenever and external interrupt signal is triggered. The interrupt controllers are combined with processor hardware models *CF_HW* and *DSP_HW* into bus-functional models *CF_BF* and *DSP_BF*, respectively.

In the example shown here, a centralized arbitration scheme is defined for the bus on the ColdFire side. Therefore, the bus-functional model contains a centralized, shared *Arbiter* component that receives requests for bus access from the two masters (*DMA_BF* and *CF_BF*) and grants them according to the chosen arbitration protocol. Inside the hardware layers of the two bus masters, corresponding arbitration protocol adapters have been inserted and connected to the arbitration bus wires.

The bus-functional model is the end result of the communication design process. It is a structural representation of the complete system architecture including computation and communication, i.e. it defines the netlist of system components and their connectivity.

5.4 Summary

In this chapter, we presented the communication design flow with well-defined design steps and design models (as summarized in Table 5.2). Starting from a virtual architecture model with abstract message-passing communication, a design is brought down to a bus-functional implementation through network and link design tasks. In between design tasks, the link model defines the implementation of the end-to-end network on top of point-to-point logical links. Furthermore, two transaction-level models, namely media access and protocol models, are supported for providing accurate results above the pin level.

Communication design supports a wide range of target communication architectures with different media and protocols. It is structured along a layering of communication functionality where communication layers have been identified, defined and adapted based on specific requirements of SoCs. The flow includes customization of layers during synthesis for optimizations across merged layers and for application-specific optimizations of layers and target communication architectures.

The contributions of this chapter include the definition of communication layers and corresponding abstraction levels for SoC design, the division of the design flow into individual steps

<table>
<tr><th colspan="2">Design step</th><th>Design decisions</th><th>Model transformations</th></tr>
<tr><td rowspan="2">Network design</td><td>Channel streaming</td><td>(a) Network byte layout selection:
$\forall d \in$ set of data types D,
$l : D \mapsto \mathbb{Z}^* \times \mathbb{Z}^* \times \{0,1\}$,
$l(d) = (size, alignment, endianess)$
(b) Channel merging:
$S =$ set of streams,
set of system channels $C_s \mapsto S$</td><td>(a) Presentation layer insertion
(b) Memory layout refinement
(c) Session layer insertion</td></tr>
<tr><td>Network segmenting</td><td>(a) Transducer allocation:
$T =$ set of $(name, type)$ tuples
(b) Channel routing & packeting:
$\forall s \in$ set of stream channels S:
$R_s =$ ordered set of $r \in (PE \cup T)$
$p : S \mapsto \mathbb{Z}^+, p(s) = packet\ size$</td><td>(a) Transport layer insertion
(b) Transducer insertion
(c) Network layer insertion</td></tr>
<tr><td rowspan="2">Link design</td><td>Link grouping</td><td>(a) Bus allocation:
$BUS =$ set of $(name, type)$ tuples
(b) Connectivity relation:
$N \subseteq (PE \cup T) \times BUS$,
$if : N \mapsto$ set of interface types IF
(c) Link parameterization:
$\forall l \in$ set of link channels L,
$m : L \mapsto N \times N \times \mathbb{Z}^+ \times \mathbb{Z}^*$,
$m(l) = (src, dst, address, interrupt)$</td><td>(a) Link layer insertion
(b) Memory behavior refinement
(c) Stream layer insertion
(d) Interrupt task creation</td></tr>
<tr><td>Media interfacing</td><td>(a) Arbiter allocation:
$A =$ set of $(name, type)$ tuples,
$bus : A \mapsto BUS$
(b) Arbitration priority assignment:
$a : MASTER \subseteq N \mapsto \mathbb{Z}^*$
(c) Interrupt controller allocation:
$IC =$ set of $(name, type)$ tuples,
$pe : IC \mapsto PE$
(d) Interrupt assignment:
$i : SLAVE \subseteq N \mapsto$ set of interrupts</td><td>(a) HAL and MAC layer insertion
(b) Arbiter insertion
(c) Interrupt handler creation
(d) HW and protocol insertion
(e) Interrupt controller insertion</td></tr>
</table>

Table 5.2: Communication design steps.

with intermediate design models and the definition of intermediate and target design models. As experimental results (Chapter 8) using the automated system design environment (Chapter 7) will show, order and granularity of steps have been defined such that critical design decisions can be validated at early stages and models can be generated automatically through successive refinement.

Chapter 6

Backend

The backend design process follows the system design process. It derives the implementation model from the communication model. In the backend process, each component of the system architecture is separately brought down from its behavioral description of computation blocks and communication adapters in the communication model to a a cycle-accurate implementation on the RTL or instruction-set level through hardware and software design, respectively. From there, components can then be further sent off to manufacturing through traditional design flows for logic and physical design.

In this chapter, we will outline the different steps, intermediate design models and model transformations of the backend design process. In Section 6.1, a description of the design flow for behavioral or high-level synthesis of custom hardware will be given. Section 6.2, on the other hand, shows the process of synthesizing target-specific software from the behavioral component models.

6.1 Hardware Design

Custom hardware design derives structural descriptions of components' microarchitectures from the behavioral specifications of components' functionalities defined by the component models in the system communication model. For each custom hardware component in the communication model, the hierarchy of computation and communication behaviors and channels is gradually synthesized into a structural description in the form of an RTL netlist. For IP components, bus-functional IP models are replaced with corresponding soft, synthesizable hardware descriptions or with hard, pre-designed gate-level netlists taken out of the IP database.

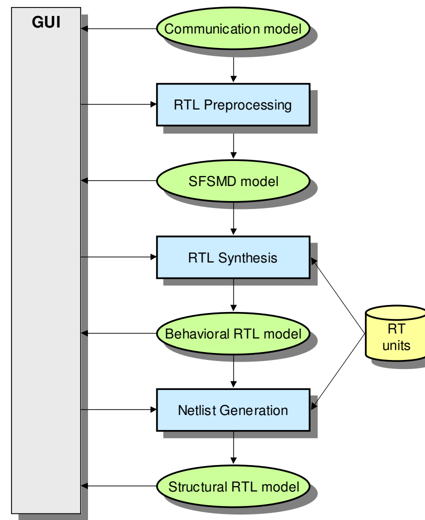


Figure 6.1: Hardware design flow.

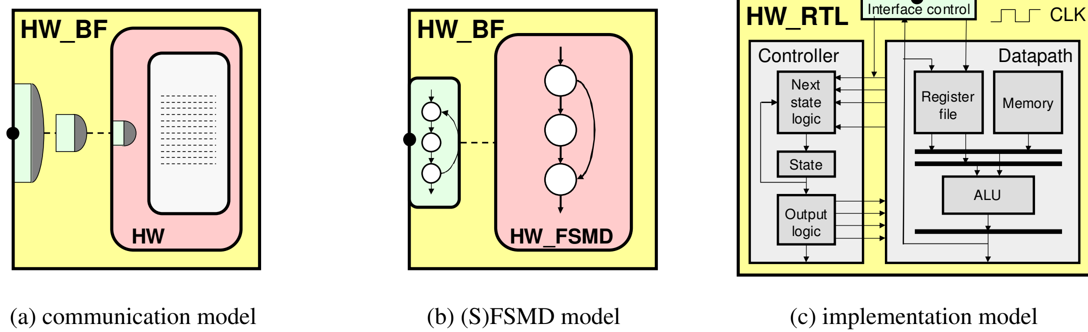
6.1.1 Overview

Custom hardware design follows a standard high-level or behavioral synthesis approach [34, 76] from C-based input descriptions [57, 100]. High-level synthesis of custom hardware generally requires allocation of RT units, scheduling of operations into clock cycles, and binding of operations, variables, and transfers to functional units, registers, and busses, respectively.

An overview of the custom hardware design flow is shown in Figure 6.1 [88, 110]. Hardware design starts from a behavioral description of the component in the communication model. Behavioral descriptions in the form of straight-line C code annotated with timing (and other) constraints are taken from the component's behaviors and channels.

In a first step, RTL preprocessing is performed. RTL preprocessing separates control and data flow by translating the C code into a superstate finite state machine with datapath (SFSMD [36]) model. In the process, some pre-synthesis, source-level optimizations (e.g. common subexpression and dead code elimination, constant propagation, etc.) are performed. In the resulting SFSMD model, C code is broken into superstates along control boundaries. Superstates contain original statements in the form of three-address code as a basis for the following scheduling and binding steps.

Following pre-processing, actual RTL synthesis on the SFSMD model is performed. Superstates are scheduled into individual, cycle-accurate states, and operations, variables, and transfers are bound to functional units, storage units (registers, register files and memories) and busses that



(a) communication model (b) (S)FSMD model (c) implementation model

Figure 6.2: Hardware refinement.

are each allocated out of the RT unit database. The result of RTL synthesis is a behavioral RTL model of the component in the form of a finite state machine with datapath (FSMD [33]) description. In the FSMD model, each state describes the operations to be performed in the corresponding clock cycle where operations are described as individual register transfers.

In a last step, the behavioral RTL model is converted into an equivalent structural RTL model through netlist generation. A structural RTL model describes the component's implementation on top of its microarchitecture as a netlist of RT units. Models of RT units in the required target hardware description languages (HDLs) are taken out of the RTL database. In the process of netlist generation, encoding of states and control words is defined and corresponding optimizations are applied. The structural RTL description is then the basis for further implementation of the component through logic and physical design tasks.

6.1.1.1 Model Refinement

With each step of the the custom hardware design flow, the behavior hierarchy representing the component in the system model is gradually refined from communication model down to implementation model (Figure 6.2). In the communication model at the input of hardware design (Figure 6.2(a)), the component is represented as a bus-functional model which accurately describes the desired component behavior as observed at its external pins. The bus-functional model is a hierarchy of sequentially composed subbehaviors with straight-line C code in the leaf behaviors. External communication functionality is described as a stack of adapter channels for different communication layers.

In the (S)FSMD model (Figure 6.2(b)), C code in the leaf behaviors of the component is refined into corresponding (S)FSMD models describing the leaf's state machine. Leaf behaviors in

the component are replaced with equivalent (S)FSMD models, optionally flatting parts or all of the behavior hierarchy. In terms of communication functionality, the top-most layers of the protocol stack are inlined into the caller's state machine whereas lower layers are refined into bus interface (S)FSMD behaviors that run in parallel to the computation behavior hierarchy.

Finally, in the implementation model (Figure 6.2(c)), the component is replaced with a refined model that represents the netlist of its target microarchitecture. The target architecture for hardware design is a standard RTL processor [3]. In general, the component implementation model consists of a controller, a datapath, and a bus interface control module. The datapath is composed as a netlist of storage units—such as registers, register files or memories—and functional units—such as ALUs, multipliers, shifters or comparators—connected via busses. The controller, on the other hand, consists of next state logic, state register, and output logic. It drives the datapath via its output logic and a set of control lines. Furthermore, it implements transitions of the state register through its next state logic based on a set of status lines received from the datapath.

6.1.1.2 Behavioral Model

A behavior model at the input of custom hardware design is a model that describes the functionality to be implemented in hardware in an abstract manner, independent of any implementation details. In the behavior model, functionality is described in the form of straight-line, sequential C code that specifies computation as operations on abstract data types.

An example of a behavioral hardware model is shown in Listing 6.1 using the example of the *BuildCode* leaf inside the custom *HW* coprocessor of the Vocoder subsystem. At the top level (Listing 6.1(a)), the coprocessor model is a sequential composition of its leaf behaviors (line 12 through line 14). Leaf behaviors communicate internally through a set of local variables (line 3 through line 4). In addition, leaf behaviors can communicate with other PEs in the system through calls to the external adapter channel interface (*session*, line 1). Inside the leaf behaviors (Listing 6.1(b)), computation is described in the form of plain C code with typical loops, branches, and expressions operating on local static and stack variables of standard, abstract types.

6.1.2 Superstate Model

A SFSMD model is a description of hardware functionality in the form of a state machine with states and transitions. In an SFSMD model, each state is a superstate that can contain any number of operations executing in any number of clock cycles. As a basis for RTL synthesis, C

```
  behavior HW(i_tranceiver session)
  {
    long int D[40];
    ...
5
    ...
    BuildCode bc(D, ...);
    ...

10  void main(void)
    {
      ...
      bc.main();
      ...
15  }
  };
```

(a) top level

```
  behavior BuildCode(in long int D[40], ...)
  {
    void main(void) {
      long int x, y, z;
5     ...

      ...
      x = y + z;
      if((x + D[5]) > 0) {
10      ...
      } else {
        ...
      }
      ...
15  }
  };
```

(b) leaf

Listing 6.1: Custom hardware behavioral model.

code is converted into a state machine by separating control flow from data flow into a corresponding control-data flow graph (CDFG [34]) representation that is captured in the form of the SFSMD model. The C code is split into superstates such that the control flow graph is explicitly captured through state transitions. In each superstate, a data flow graph in the form of three-address code describes the original C operations executed in that control state. As a result of the preprocessing step, the superstate model refines leaf behaviors and communication adapters in the custom hardware component behavior hierarchy into corresponding SFSMD descriptions. Optionally, parts or all of the behavior hierarchy is flattened out and code of channel methods is inlined into the caller's code.

In the SFSMD model of a custom hardware component (Listing 6.2), instances of leaf behaviors are replaced with instances of behaviors refined into corresponding SFSMD models. Otherwise, the top level of the component (Listing 6.2(a)) remains as a sequential composition of SFSMD subbehaviors that communicate via a set of local variables and external adapter channel interfaces. Inside the refined leaf behaviors (Listing 6.2(b)), the C code is replaced by a `fsmd` composition (line 9) with superstates (line 11) that contain the original operations (line 12 through (line 15) on the original local variables (line 4) and transitions of superstates replicating the original control flow (line 16, line 17). However, operations inside superstates are transformed into three-address code that requires additional local stack variables to hold temporary, intermediate values (line 6 and line 7).

Furthermore, note that the superstate model modifies the timing of the hardware component. In the SFSMD model, `waitfor()` statements in the C code representing estimated execution

```
behavior HW_SFSMD(
  i_tranceiver session
                  )
{
  long int D[40];
  ...

  ...
  BuildCode_SFSMD bc(D, ...);
  ...

  void main(void)
  {
    ...
    bc.main();
    ...
  }
};
```

(a) top level

```
behavior BuildCode_SFSMD(in long int D[40],
                         ...) {
  void main(void) {
    long int x, y, z;
    ...
    long int _tmp_1, _tmp_2;
    bool     _status_;

    fsmd(10u) {
      ...
      Si: {
        x        = y + z;
        _tmp_1   = D[5];
        _tmp_2   = x + _tmp_1;
        _status_ = (_tmp_2 > 0);
        if (_status_)
          goto Sj;
      }
      ...
    }
  }
};
```

(b) leaf

Listing 6.2: Custom hardware SFSMD model.

delays (see Section 4.2.1.1) have been removed and replaced with a unit delay assigned to each superstate in the `fsmd` composition (line 9). Therefore, in the SFSMD model, each superstate will take the same amount of time, independent of the actual amount of work performed in the state. As such, execution timing of superstate models is generally inaccurate. Therefore, SFSMD models only serve as input to the design process but can not be used for timing validation.

6.1.3 Behavioral RTL

Behavioral RTL models are the result of RTL synthesis of custom hardware components from their SFSMD models. In a behavioral RTL model, superstate machines in leaf behaviors and communication adapters are refined into cycle-accurate FSMD models that accurately describe the operations performed in each state and hence in each clock cycle. A behavioral RTL model represents splitting of superstates into clock cycles according to the scheduling decisions. Furthermore, it can represent binding information with the result that abstract C operations and data types are replaced with bit-accurate storage and functional unit models describing register transfers inside the states.

As defined by the Accellera RTL standard [3], different styles of behavioral RTL models are possible (Table 6.1). At a minimum, scheduling of operations into clock cycle has to be per-

Level	Binding
1	None
2	Storage
3	Functional unit, storage
4	Bus, functional unit, storage
5	Structural RTL

Table 6.1: Accellera RTL styles.

formed. In a style 1 model, superstates are split into individual states but operations inside states remain at an abstract C level. In models at styles 2 through 4, additional binding information is gradually introduced and explicitly represented in the FSMD model. In a fully bound model of style 4, operations and variables in each state are replaced with register transfers over explicity instantiated storage units, functional units, and busses. Finally, Accellera style 5 is defined as a netlist representation equivalent to the structural RTL model (see Section 6.1.4).

In the FSMD model of custom hardware (Listing 6.3), leaf behaviors are replaced with instances of behaviors refined into cycle-accurate FSMD models. At the top level (Listing 6.3(a)), allocated busses (line 3 and line 4), storage units (line 6 and line 7) and functional units (line 9) are explicitly instantiated where ports of functional units are connected to busses and other wires according to the selected interconnect structure. Instances of refined FSMD subbehaviors connect to RT unit instances as needed (line 12). Any local variables have been removed and are stored inside registers or memories instead. Furthermore, communication with refined adapter channel FSMD models that run in parallel to the main computation is explicitly represented as bit-accurate control, data, and status wires (line 1).

Inside the refined leaf behaviors (Listing 6.3(b)), superstates are split into cycle-accurate states (line 11 and line 17) that accurately represent the operations performed in each cycle. As a result, execution timing described as clock period delays associated with each each state transition in the `fsmd` composition is accurate (line 8). Inside the states, computation is described as transfers of values from storage units to busses, execution of functional units (that are connected to busses at the top level), and transfer of results from busses back to storage units.

6.1.4 Structural RTL

A structural RTL view of the PEs in the implementation model accurately reflects the microarchitecture internal to the system PEs. It is the result of netlist generation from the fully bound behavioral RTL model for a PE at the output of RTL synthesis. Instead of an implicit representation

```
behavior HW_FSMD( bit [2] if_ctrl , bit [31:0] if_data , bit [1] if_status )
{
  unsigned      bit [31:0] bus , bus1 , bus2 ;      // wires
  unsigned      bit [1]    status ;

  buffered [10u] bit [31:0] mem[256];               // storage
  buffered [10u] bit [31:0] rf [32];

  ALU alu (bus1 , bus2 , bus , status );             // functional units

  ...
  BuildCode_FSMD bc (bus , bus1 , bus2 , rf , mem, alu , status );
  ...

  void main (void ) {
    ...
    bc . main ( ) ;
    ...
  }
};
```

(a) top level

```
behavior BuildCode_FSMD(
  unsigned bit [31:0] bus , unsigned bit [31:0] bus1 , unsigned bit [31:0] bus2 ,
  buffered [10u] bit [31:0] mem[256] , buffered [10u] bit [31:0] rf [32] ,
  i_functional_unit alu , unsigned bit [1]    status )
{
  void main (void )
  {
    fsmd (10u ) {
      ...

      Si : {
        bus1  = rf [6];
        bus2  = rf [7];
        alu . main ( ) ;
        rf [5] = bus ;
      }
      Si_1 : {
        bus1  = rf [5];
        bus2  = mem[40];
        alu . main ( ) ;
        if ( status ) goto Si+1; else goto Sj ;
      }
      ...
    }
};
```

(b) leaf

Listing 6.3: Custom hardware FSMD model.

```
behavior HW_RTL(in   signal bit[31:0] Addr,     // bus
                     signal bit[31:0] Data,
                     signal bit[3]    Ctrl,
                 out signal bit[1]    Intr)     // interrupt
{
  event clk;                                  // clock
  signal bit[15:0]  status;                   // status lines
  signal bit[117:0] ctrl;                     // control lines
  signal bit[32:0]  d;                        // bus interface data

  HW_ClkGen   cg(clk);                        // clock generator
  HW_Control  ctrlr(clk, status, ctrl);       // controller
  HW_Datapath dp(clk, ctrl, status, d);       // datapath
  HW_IF       bus(clk, ctrl, status, d);      // bus interface

  void main(void) {
    par {
      cg.main();
      ctrl.main();
      dp.main();
      bus.main();
    }
  }
};
```

Listing 6.4: RTL netlist custom hardware model.

of register transfers performed in each state, the structural RTL model explicitly describes the PE as a netlist of RTL units connected by wires. Structural RTL is the basis for further implementation of PEs through traditional logic synthesis which in turn will derive a gate-level netlist from the netlist of units inside each PE. Note that for simulation purposes, behavioral and structural RTL models are equivalent. Since the structural RTL model does not provide more accuracy but introduces more overhead, it is usually sufficient to perform effective validation on the behavioral RTL model.

In a structural model, the component is implemented as a purely structural netlist of subcomponents (Listing 6.4). Subcomponents are represented by subbehaviors (line 11 through line 14). All subbehaviors operate in parallel (line 17 through line 22) and are connected via busses and/or wires (line 6 through line 9). In general, subcomponents themselves can be further decomposed hierarchically. At each level, however, the same purely structural netlist of behaviors running concurrently and being connected through wires is repeated. Therefore, if the hierarchy is flattened all the leaf behaviors will operate in parallel and communicate via busses and wires.

In general, a structural RTL model represents the component's RTL processor target architecture. At the top level (Listing 6.4), the component is composed out of clock generator (line 11), controller (line 12), datapath (line 13), and bus interface (line 14) subcomponents. The clock generator drives the internal clock event and triggers it periodically according to the component's clock

```
  behavior HW_Control(in  event          clk,
                      in  signal bit[15:0]  status,
                      out signal bit[117:0] ctrl)
  {
5   signal bit[21:0] state, nextstate;

    HW_SR  sr(clk, nextstate, state);     // state register
    HW_OL  ol(state, ctrl);               // output logic
    HW_NSL nsl(state, status, nextstate); // next state logic
10
    void main(void) {
      par {
        sr.main(); ol.main(); nsl.main();
      }
15  }
  };
```

Listing 6.5: Custom hardware controller.

period. The bus interface, which runs in parallel to, is controlled by, and communicates with the main controller and datapath, implements the state machine to execute bus transactions.

The component's master controller (Listing 6.5) is responsible for implementing the component's data and control flow for computation and communication by driving datapath and bus interface resources. The main controller is hierarchically decomposed into state register, next-state logic and output logic (line 7, line 9 and line 8). The state registers stores the current state value and updates it with the next state value in every clock cycle. The next stage logic generates the next state value from status and current state inputs. Finally, the output state logic generates the (datapath and other) control signals from the current state value.

The main component datapath (Listing 6.6) is hierarchically composed as a structural netlist of the different datapath components (line 8 through line 11) connected through internal busses (line 6). Similarly, the datapath's subcomponents are modeled following standard structural RTL design guidelines as outlined previously. In general, sub-components are register/storage units driven by the clock event, combinatorial logic blocks or a hierarchical composition thereof.

Leaf behaviors of the structural RTL hierarchy model registers and combinatorial logic between registers. Leaf behaviors are reactive, i.e. they continuously react to events on their inputs and create resulting events at their outputs. Registers are driven by a clock signal that is locally generated inside the component. On every clock event, a register takes over a new value from its input and updates its output accordingly. Combinatorial logic, on the other hand, reacts to changes on any of its inputs in order to recalculate its outputs. Structural RTL models describe hardware as a reactive system with a set of non-terminating processes operating concurrently [6].

```
  behavior HW_Datapath(in  event            clk,
                       in  signal bit[117:0] ctrl,
                       out signal bit[15:0]  status,
                           signal bit[31:0]  d)
5 {
    signal bit[31:0] bus, bus0, bus1, bus2;

    Mux mux(bus0, d, bus);
    Mem mem(clk, ctrl[117:94], bus0);
10  RF  rf (clk, ctrl[93:61], bus0, bus1, bus2);
    ALU alu(ctrl[60:0], bus, bus1, bus2, status);

    void main(void) {
      par {
15      mux.main();
        mem.main();
        rf.main();
        alu.main();
      }
20  }
  };
```

Listing 6.6: Custom hardware datapath.

As an example, the behavior hierarchy down to the leaves of the hardware controller is shown in Listing 6.7. The state register (Listing 6.7(a)) updates the current state value with the next state value (line 6) on every clock event (line 5). Output (Listing 6.7(b)) and next state (Listing 6.7(c)) logic are hierarchically composed out of subbehaviors mirroring the original behavior hierarchy of the behavioral RTL model. Leaf behavior state machines from the FSMD model are assigned a dedicated slice of the state register space each and the overall state space is the union over all leaf behavior states (line 5)[1]. At the leafs of the hierarchy, output and next stage logic (Listing 6.7(d) and Listing 6.7(c)) are combinatorial blocks that are senstive to changes on any of their input ports (line 9 and line 8) and, depending on input values (line 10 and line 9), perform control word and next state assignments to implement register transfers and state transitions of the behavioral RTL model (line 13 and line 12).

6.2 Software Design

Software design derives executable object code from behavioral component models of programmable processors in the system communication model. Object code in the processor's instruction set is running on top of a structural description of the processor's microarchitecture.

[1]Note that in the process of netlist generation, the state space can be optimized across leaf boundaries by flattening parts or all of the behavior hierarchy.

```
behavior HW_SR(in event clk, in signal bit[21:0] next, out signal bit[21:0] cur)
{
  void main(void) {
    while(true) {
5     wait(clk);
      next = cur;
    }
  }
};
```

(a) state register

```
behavior HW_OL(in signal bit[21:0] st,
              out signal bit[99:0] ctrl)
{
  ...
5 BuildCode_OL bc(st[7:3], ctrl);
  ...

  void main(void) {
    ...
10  bc.main();
    ...
  }
};
```

(b) output logic

```
behavior HW_NSL(in signal bit[21:0] st,
               in  signal bit[15:0] stat,
               out signal bit[21:0] nxt) {
  ...
5 BuildCode_NSL bc(st[7:3], stat, nxt);
  ...

  void main(void) {
    ...
10  bc.main();
    ...
  }
};
```

(c) next state logic

```
behavior Build_Code_OL(
  in  signal bit[4:0]  state,
  out signal bit[99:0] ctrl)
{
5 void main(void)
  {
    while(true)
    {
      wait(state);      // sensitivity
10    switch(state) {
        ...
        case Si_1:
          ctrl = "000...10b";
          break;
15      ...
      }
    }
  }
};
```

(d) output logic leaf

```
behavior Build_Code_NSL(
  in  signal bit[4:0]  state,
  in  signal bit[15:0] status,
  out signal bit[21:0] next)
5 {
  void main(void) {
    while(true) {
      wait(state, stat); // sensitivity
      switch(state)  {
10      ...
        case Si_1:
          next = Si+1;
          if( !status[7] ) nxt = Sj;
          break;
15      ...
      }
    }
  }
};
```

(e) next state logic leaf

Listing 6.7: Custom hardware controller behavior hierarchy.

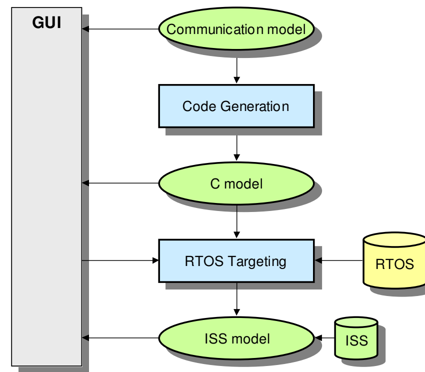


Figure 6.3: Software design flow.

For each programmable PE in the communication model, hardware and bus-functional component layers will be replaced with structural descriptions of the actual PE hardware taken out of the PE database. For simulation purposes, structural descriptions in the form of instruction set simulators (ISS) are plugged into the system design model. For manufacturing purposes, on the other hand, a gate-level netlist of the processor will be fed into the physical system design flow. Software in the form of a final executable targeted towards the PE's instruction set is then generated from the HAL and layers above through C code generation, compilation, and linking against target RTOS and processor libraries.

6.2.1 Overview

Figure 6.3 shows an overview of the software design flow [108, 107]. Software design starts from a specification of the desired functionality to be implemented on the target processor as given by the component models of programmable PEs in the communication model.

In a first code generation step, the behavior and channel hierarchy representing the software application layer of the PE is converted into an equivalent ANSI C code description. Behaviors and channels are translated into a corresponding C functional call hierarchy. The generated C code is then re-imported into the system design model as a refined component model by wrapping the C code into a behavior that replaces the original application layer model. In the resulting C model of the PE, the application software is represented by its C code that can thus be validated.

The generated C code is then compiled into the target PE's instruction set architecture (ISA) using the standard ANSI C compiler available for the processor. To generate the final executable, the compiled code is linked against customized RTOS and target processor HAL libraries. During this RTOS targeting step, an actual RTOS has to be selected out of the RTOS database to

implement necessary OS services on the processor. A customized version of the RTOS libraries is generated by extending the kernel of the selected target RTOS with necessary drivers for communication with other PEs in the system. The RTOS kernel binary for the selected target RTOS in the target processor's object format is taken out of the RTOS database. Drivers are synthesized by generating and compiling target-specific driver code from the adapter channel hierarchy representing the protocol stack in the communication model. Finally, a binary implementation of the hardware abstraction layer (HAL) for the given target processor is taken out of the PE database and linked against the code to supply the necessary HAL functionality.

The result of the software design process is the instruction set model of the PE. The specification of the desired software functionality has been converted into a binary executable running on the target processor hardware. In the system design model, an instruction set simulation (ISS) model of the processor is plugged into the system, replacing the behavioral PE model. The ISS model simulates the execution of the generated executable on top of the PE's hardware structure. As such, it describes the processor behavior in a cycle- and bit-accurate manner.

6.2.1.1 Implementation Layers

At the input of the software design flow, processor models as part of the system communication model (see Section 5.3.4.2) at the consist of several layers of functionality. During software design, different layers and different parts inside each layer are implemented in different ways depending on their characteristics and requirements.

Figure 6.4 shows the different parts of programmable processor models and their implementation on the target processor. At the bottom, a processor's bus-functional and hardware layers (HW) are implemented directly by the target processor's hardware description. Hardware models for processors taken out of the PE database come in the form of an ISS model for simulation and a synthesizable or hard-coded netlist for manufacturing. A processor's hardware implements its instruction set architecture (ISA), its I/O interfaces and its interrupt handling capabilities.

The hardware abstraction layer (HAL) is then implemented in software on top of the processor's hardware model. It provides a canonical abstraction of the processor hardware at its interface to higher software layers. A binary implementation of the HAL in the form of object code for the target processor is stored together with the processor in the PE database. The targeted, binary HAL implementation in the database is equivalent in its functionality and API to the abstract HAL model imported out of the database during communication design (see Section 5.3.3.1). As

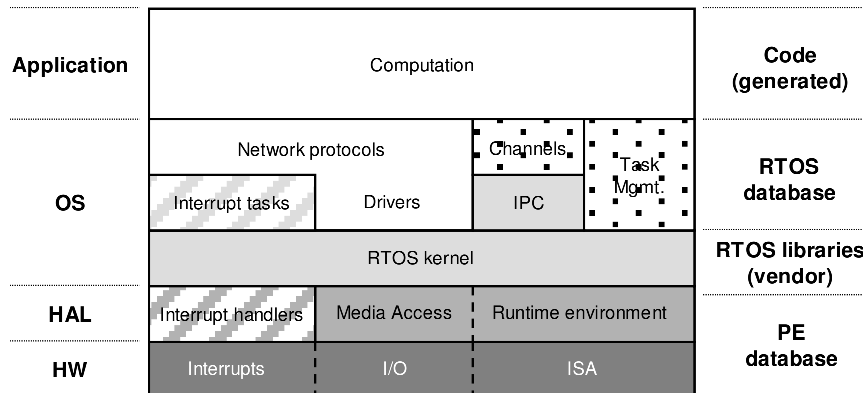


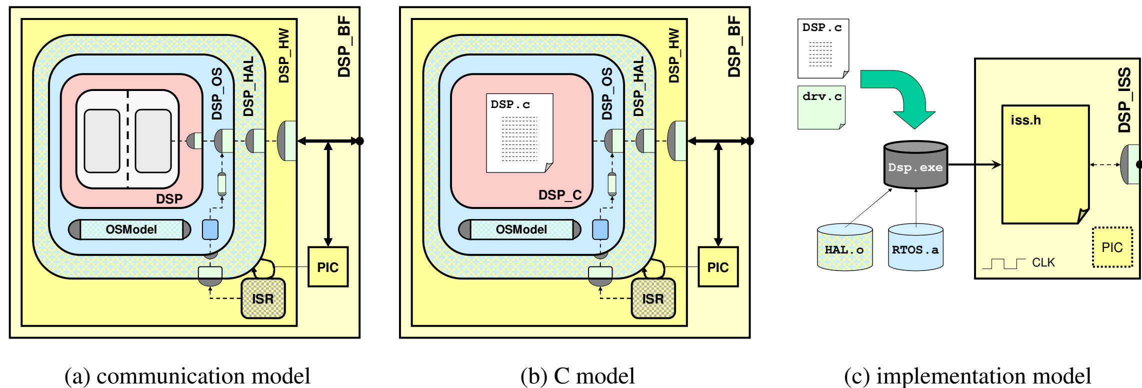
Figure 6.4: Software implementation layers.

such, it implements the assembly code for the media access layer and for the runtime environment to initialize the processor. Furthermore, the HAL implementation provides templates for setting up interrupt handlers. These templates are used and filled during RTOS targeting as part of generating code for implementation of communication drivers.

At the core of the RTOS targeting task is the implementation of the OS layer in the processor model. A binary kernel of the selected RTOS for the target processor is taken out of the RTOS vendor database. The RTOS kernel implements basic multitasking and synchronization functionality corresponding to the services provided by the abstract OS model inserted during system scheduling (see Section 4.3.2). On top of the RTOS kernel, a compatibility layer implements the exact interface of the abstract OS model for use by higher layers. Specifically, a library that is stored together with the target RTOS in the RTOS database translates the API of the abstract OS model (task management) into the API of the target RTOS kernel. Furthermore, functionality of standard communication and synchronization channels is implemented through a library that maps channels down to inter-process communication (IPC) mechanisms that are part of the RTOS kernel.

On top of the media access layer provided by the HAL implementation, higher layers of the communication protocol stack are implemented by translating the adapter channels into C code and by compiling the code into the target processor instruction set. The RTOS database contains templates for setting up interrupt handling tasks on top of the target RTOS kernel. Together with interrupt handler templates in the HAL implementation, task templates are used to implement the necessary interrupt service functionality as part of communication driver synthesis.

Finally, as outlined previously, the application layer for the software in the processor model is implemented through code generation and compilation on top of the services provided



(a) communication model (b) C model (c) implementation model

Figure 6.5: Software refinement.

by the OS layer implementation and on top of the processor's ISA. In the final linking stage, the compiled code is then hooked into the runtime environment provided by the HAL and RTOS implementations to start execution of the `main()` routine after processor and OS initialization.

6.2.1.2 Model Refinement

Gradual, stepwise refinement of PE component models during software design is shown in Figure 6.5. As described previously, in the system communication model (Figure 6.5(a)), programmable PEs are represented by application, OS, hardware abstraction, hardware and bus-functional layers. The application layer contains the behavior and channel hierarchy describing the tasks of the application software and their interaction. The OS layer contains the abstract OS model managing and scheduling tasks, protocol adapter channels representing external interface drivers, and interrupt handling tasks that communicate with protocol adapters through OS semaphores as part of the bus drivers. The HAL contains low-level software for bus communication and interrupt handlers for servicing incoming synchronization exceptions. The hardware layer contains a model of the processor's interrupt control logic together with adapter channels that model bus interface protocol implementations in the processor's hardware. Finally, the bus-functional layer contains models of any additional, external interrupt controllers that are part of the processor.

During code generation, the SpecC behavior and channel hierarchy for the application layer in the communication model is converted into an equivalent C description. In the refined C model after code generation (Figure 6.5(b)), the generated C code is re-imported into the design, replacing the original SpecC application layer model. A refined application layer is created by wrapping and encapsulating the generated C code into a compatible SpecC behavior for re-import

into the C design model. Inside the C model of the design, the generated C code then executes on top of OS and outer layers using their services for task management, inter-task communication, synchronization, and external bus communication.

In the final implementation model (Figure 6.5(c)), the component model of a programmable processor is replaced with an instruction set simulation model of the target processor. The ISS runs the executable generated during RTOS targeting. The executable is generated by compilation of application and driver C code and linking against HAL and RTOS libraries. The ISS simulates execution of object code in a cycle-accurate manner on top of a structural model of the processor's microarchitecture. External instruction set simulators that provide a C level API are wrapped into component behaviors in order to plug them into the system design model. The wrapper implements proper cycle timing and translation of I/O instructions into simulated bus cycles via instances of adapter channels modeling bus protocols. Finally, the wrapper contains any necessary models of external interrupt controllers that are not part of the ISS itself.

6.2.2 C Model

The C model is the result of translating the application software behavior and channel hierarchy in the component model into an equivalent C description. In the generated application C code (Listing 6.8), behavior and channel class definitions are converted into C `struct` definitions. The behavior hierarchy is converted into a `struct` hierarchy (line 4 through line 19), child behavior (line 17, line 18) and variable (line 15) instances are represented as `struct` members, and ports are converted into `struct` pointer members (line 5 and line 10) [108, 107]. Methods of behaviors and channels are converted into global C functions that accept a pointer to the instance `struct` to operate on as their first parameter (line 21, line 27 and line 33). Finally, the corresponding instance hierarchy is created through appropriate initialization of the top-level `struct` where pointers are initialized to represent proper port mappings (line 40 through line 44).

In the C model, the generated C code is then plugged back into the system design model by wrapping it into a component behavior model (Listing 6.9). The wrapper imports the C code by including the generated C source file (line 2). Finally, it provides implementations of the external bus driver functions (line 4 and line 7) that are declared as prototypes in the C code (Listing 6.8, line 1 and line 2) by mapping them to calls of the corresponding adapter channel methods.

```
void session_send(const void* data, unsigned long len); // driver method
void session_receive(void* data, unsigned long len);    //   declarations

struct Coder {                        // behavior Coder
  int *v1;                            //   (out int v1)
  ...                                 // { ... }
}

struct Decoder {                      // behavior Decoder
  int *v1;                            //   (in int v1)
  ...                                 // { ... }
}

struct DSP {                          // behavior DSP() {
  int v1;                             //   int v1;
                                      //
  struct Coder   coder;               //   Coder coder(v1);
  struct Decoder decoder;             //   Decoder decoder(v1);
}

void Coder_main(struct Coder *self) {         // void Coder.main(void)
  ...
  *(self->v1) = ...;    // port access
  ...
}

void Decoder_main(struct Decoder *self) {     // void Decoder.main(void)
  ...
  ... = *(self->v1);    // port access
  ...
}

void DSP_main(struct DSP *self) {             // void DSP.main(void)
  Coder_main(&(self->coder));
  Decoder_main(&(self->decoder));
}

void main(void)
{
  struct DSP dsp = {                  // Instantiation:
    0,                                //   variable initialization
    { &(dsp.v1), ... },               //   Coder port mapping, init
    { &(dsp.v1), ... }                //   Decoder port mapping, init
  };

  DSP_main(&dsp);
}
```

Listing 6.8: Application software C code.

```
behavior DSP_C(i_tranceiver session) {
#include "DSP.c"

  void session_send(const void* data, unsigned long len) {
5   session.send(data, len);
  }
  void session_receive(void* data, unsigned long len) {
    session.receive(data, len);
  }
10 };
```

Listing 6.9: Software C model.

6.2.3 Instruction Set Simulation Model

After compilation of application and driver C code into the processor's instruction set, the final executable for the target processor is generated during RTOS targeting by linking against target processor and RTOS libraries. As a result, the system model is refined to insert an instruction set simulation model of the processor running the generated executable.

Different levels of instruction set simulation of the executable are possible. In a compiled instruction set simulation, each assembly instruction is translated into a set of C statements that perform updates of a simulated processor state cycle by cycle [112]. This C code is then wrapped into a behavior and plugged into the implementation model as a component model of the processor.

On the other hand, for interpreted instruction set simulation, the ISS model of the programmable PE consists of a behavior that reads and interprets the instruction stream. Any instruction-set simulator that supports a C-based API can be hooked into the simulation model (Listing 6.10). The external ISS is wrapped into a behavior (line 5) that calls the ISS routines via the ISS's API (line 2). The core of the processor behavior is a loop which simulates one clock cycle per iteration (line 20). The ISS's *exec()* function fetches and decodes instructions, performs the corresponding operations in each clock cycle, and updates the simulated processor state accordingly (line 25). In a normal cycle, time in the simulator advances by one clock period and the wrapper behavior synchronizes the ISS with the rest of the system by advancing SpecC logical time accordingly (line 34).

In both cases of compiled or interpreted simulation, the simulation model of the processor drives and samples the ports of the PE behavior based on the instruction stream executed. For each I/O instruction, the PE ports are updated from the processor state and vice versa. For example, interrupt input lines of the processor are updated in each cycle by sampling the corresponding input port of the PE behavior (line 22 and line 23).

```
  // ISS C/C++ interface
  #include "iss.h"

  // Instruction Set Simulator (ISS)
5 behavior Dsp_ISS(    signal bit[31:0] Addr,    // bus
                       signal bit[31:0] Data,
                       signal bit[3]    Ctrl,
                    in signal bit[1]    intA,    // interrupts
                    in signal bit[1]    intB,
10                  in signal bit[1]    intC,
                    in signal bit[1]    intD)
  {
    Master  protocol(Addr, Data, Ctrl);         // bus interface protocol

15  void main(void)
    {
      iss.startup();                      // initialize ISS, load program
      iss.load("a.out");

20    while(true)                         // run simulation
      {
        iss.intA = intA; iss.intB = intB; // drive ISS inputs
        iss.intC = intC; iss.intD = intD;

25      iss.exec();                       // run DSP cycle

        if ( iss.IR == MOVEM_RD ) {       // simulate bus cycle
          protocol.masterRead(iss.Addr, &iss.Data);
        }
30      else if ( iss.IR == MOVEM_WR ) {
          protocol.masterWrite(iss.Addr, iss.Data);
        }
        else {                            // advance time
          waitfor( DSP_CLOCK_PERIOD );
35      }
      }
    }
  };
```

Listing 6.10: Instruction set simulation (ISS) model.

Any special bus interface hardware of the processor is simulated through corresponding protocol adapters. The ISS model instantiates the necessary protocol adapter channels to simulate the processor's bus transactions (line 13). For every I/O instruction encountered in the instruction stream (line 27 and line 30), the corresponding method in the bus adapter is called (line 28 and line 31). For that I/O instruction, the protocol adapter simulates the timing-accurate driving and sampling of bus wires by the processor hardware. Note that protocol adapters are equivalent to the protocol layer in the the communication model.

	Design step	Design decisions	Model transformations
Hardware design	RTL preprocessing	None	(a) Source-level optimization (b) SFSMD generation
	RTL synthesis	(a) RT unit allocation: RT = set of (*name*,*type*) tuples (b) Scheduling function ϕ: set of statements $E \mapsto \mathbb{Z}^*, \phi(e) = step$ (c) Operation binding function β: $E \mapsto RT, \beta(e)$ = functional unit FU (d) Storage binding function γ: $E \mapsto RT \times RT \times RT$, $\gamma(e) = (dst, src1, src2)$ (e) Bus binding function δ: $E \times \{0,1,2\} \mapsto \mathbb{Z}^* \times RT \times \mathbb{Z}^*$, $\delta(e, transfer) = (outport, bus, inport)$	(a) State splitting (b) FU insertion (c) RF & memory insertion (d) Bus insertion (e) Netlist generation
Software design	Code generation	None	(a) C code generation (b) C code wrapping
	RTOS targeting	(a) RTOS selection: $rtos : PE \mapsto$ set of targets $RTOS$	(a) Driver code generation (b) HAL customization (c) Compilation & linking

Table 6.2: Backend design steps.

6.3 Summary

In this chapter, an overview and description of the backend design process has been given. Starting from behavioral component models in the system communication model, cycle-accurate, structural implementations of components on the register-transfer and instruction-set level are derived through hardware and software design. In the final implementation model, components are then ready for further implementation down to manufacturing through traditional logical and physical design processes.

The contributions of this chapter include a definition of steps and models for backend design, providing a path to implementation and connecting the system design flow to physical design and manufacturing. For each aspect of components and for each part of their models, clear design flow mapping desired functionality down to an implementation has been shown. Design steps, design decisions and model transformations (summarized in Table 6.2) have been defined to provide a flow that supports both automated and interactive design for rapid and efficient design space exploration under the control of the designer. Furthermore, software and hardware backend design

flows have been broken into individual steps and corresponding intermediate design models that enable early validation of critical issues, transparency of the design process, and automation of model generation and decision making.

Chapter 7

Design Environment

The system design flow and methodology presented in the previous chapters has been implemented in the form of the SoC Design Environment (SCE) that integrates a set of tools under a common framework and user interface. With the environment, the design flow has been automated to a large extent and aids have been provided that steer and guide the user through the design process. As part of developing the design flow within the scope of this work, we defined the overall framework of the design environment including architecture, organization, tool flow, design model management, interfaces, and databases. Furthermore, we developed textual and graphical user interfaces for model visualization and decision entry. As a result, the design environment proves the feasibility and effectiveness of a seamless, automated design flow from abstract specification down to implementation.

In the rest of this chapter, we will described the details of the implementation of the design environment framework, including its architecture, infrastructure, tool flow, and user interface. Following an overview of the design environment in Section 7.1, implementation of specification capture, profiling and estimation, computation design, communication design and backend design tasks will be shown in Section 7.2 through Section 7.7.

7.1 Overview

Figure 7.1 shows the overall architecture and tool flow of the SoC Design Environment (SCE) [1, 46]. The design environment integrates a set of tools under a common framework and user interface. Furthermore, SCE includes a set of databases that contain models and attributes of different components needed throughout the design flow.

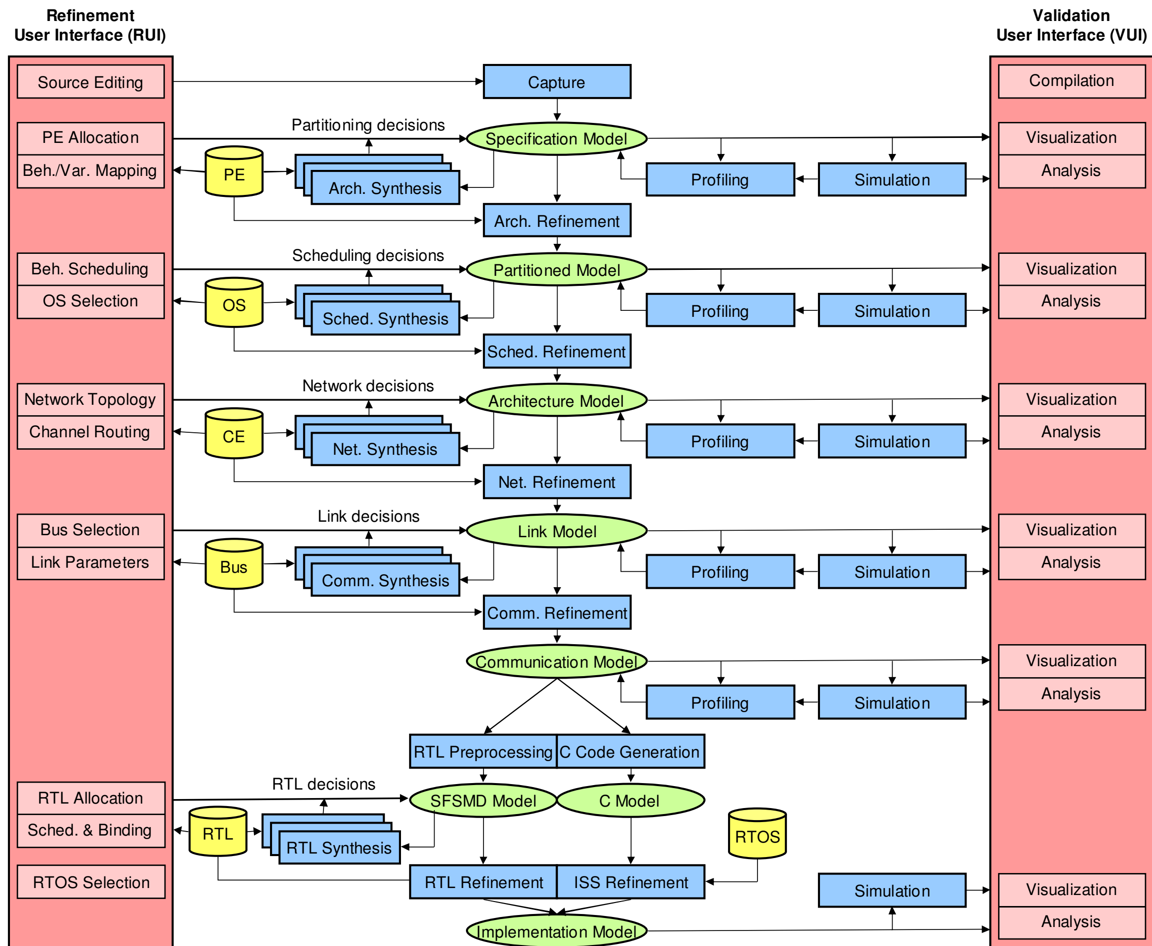


Figure 7.1: SCE tool flow and architecture.

Individual tools inside SCE are command line tools that are invoked by the framework under the control of the user. All tools operate on and exchange data solely by reading and writing system design models stored in the framework's model database, i.e. all design data is exchanged between tools inside SCE exclusively in the form of design models. Design models hold all the information available about a design at a certain stage of the design process. Inside SCE, design models are stored and exchanged in SIR (SpecC Internal Representation [24]) format as binary SIR files on disk. SIR files of different design models that are part of a common project are organized and managed through SCE's model database. Apart from the actual design models, tools exchange all necessary meta-data through persistent annotations attached to the design entities in the SIR files.

With support of the SpecC compiler [25, 26] integrated into the environment, SpecC source code for design models is translated into SIR files and back. Therefore, at any point of

the design flow source code can be imported into and exported out of SCE to allow source capture, browsing and modifications by the user. Specifically, at the specification level, source code captured by the designer is compiled into a specification model as the start of the design process.

7.1.1 Simulation

Through the SpecC compiler (`scc`), design models are compiled into executables for validation through simulation at any stage of the design process. The SpecC compiler translates SIR files into C++ code for simulation, compiles the code using a standard C++ compiler, and links the binaries against the SpecC simulation kernel [25, 26, 113]. In the environment, the compiled code is then executed on the host machine for simulation of design models.

7.1.2 Profiling

For feedback about design quality metrics, the environment integrates a system profiling and estimation tool (`scprof`, see Section 7.3). By combining dynamic profiling data obtained during simulation, abstract estimation models of target components taken out of the databases, and static analysis of design models, the profiler computes a variety of design quality metrics at every stage in the design process. Furthermore, to aid the designer in the design process, profiling and estimation derive both implementation-independent specification metrics and target-dependent implementation metrics. Within the environment, data and metrics obtained during profiling are back-annotated into and stored together with the design models. From there, selected metrics are extracted and prepared for visualization through the environment's user interface as requested by the designer.

7.1.3 Refinement

For automatic refinement of design models following the set of design steps described in the previous chapters, the environment integrates tools for architecture refinement (`scar` [82]) and RTOS refinement (`scos` [50, 109]), network and communication refinement (`scnr` and `sccr` [42, 2]), and hardware (`scrtl` [88]) and software (`sc2c` [108]) refinement. Within the environment, design decisions are stored and passed to the refinement tools as design model annotations. The refinement tools then read a design model including annotated design decisions, import the necessary component models out of the database and implement the given decisions to generate the refined system design model at the next lower level.

7.1.4 Synthesis

Via a plugin mechanism, an unlimited number of external synthesis tools can be integrated into the environment. The environment supports synthesis tools for automatic decision making at all stages of the design process. Through plugins, synthesis tools are dynamically loaded and integrated into the framework at runtime. Under the control of the user, synthesis plugins can be applied to all or parts of a design. The environment will then execute the synthesis tool on the chosen design and the tool will annotate the design decisions back into the design model. By plugging in additional new synthesis algorithms, the environment can easily be extended at any time, giving anybody the flexibility to adapt the environment to new technologies.

7.1.5 User Interface

The design environment integrates the different tools and databases under a common framework through both a textual (`scsh`) and graphical (`sce`) user interface. In both cases, designers can manipulate design models, make design decisions, browse databases, view simulation, profiling and estimation results, and finally apply refinement, exploration, and synthesis tools.

For scripting of the design process through a textual interface, the environment provides an interactive shell. The SCE shell reads and executes commands from a terminal or a script. Syntax and semantics of SCE shell commands are based on the the Python scripting language [97, 31]. At the core of the shell is a Python interpreter that has been extended with an API that provides access to the core functionality of the design environment. Therefore, the SCE shell allows full control of the core framework via an object-oriented Python API with the full power and flexibility of the Python language.

Finally, the design environment provides a unified graphical user interface (GUI) for refinement and validation. The goal of the GUI is to provide effective visualization of design data and decision entry to guide, steer, and aid the designer in rapid design space exploration [83, 5]. The GUI has been implemented in Python using the PyQt [85, 73] Python wrapper for the Qt toolkit and widget library [8, 61].

A screenshot of the core SCE GUI is shown in Figure 7.2. Apart from menubar, toolbar and statusbar, the basic GUI area is divided into three parts: a project window on the left, an output window at the bottom, and a main workspace. The project window shows and manages all the design models, imports, and source files in the current project. The output window logs diagnostic and error output from running the various external compilation, simulation, analysis, and refinement tools,

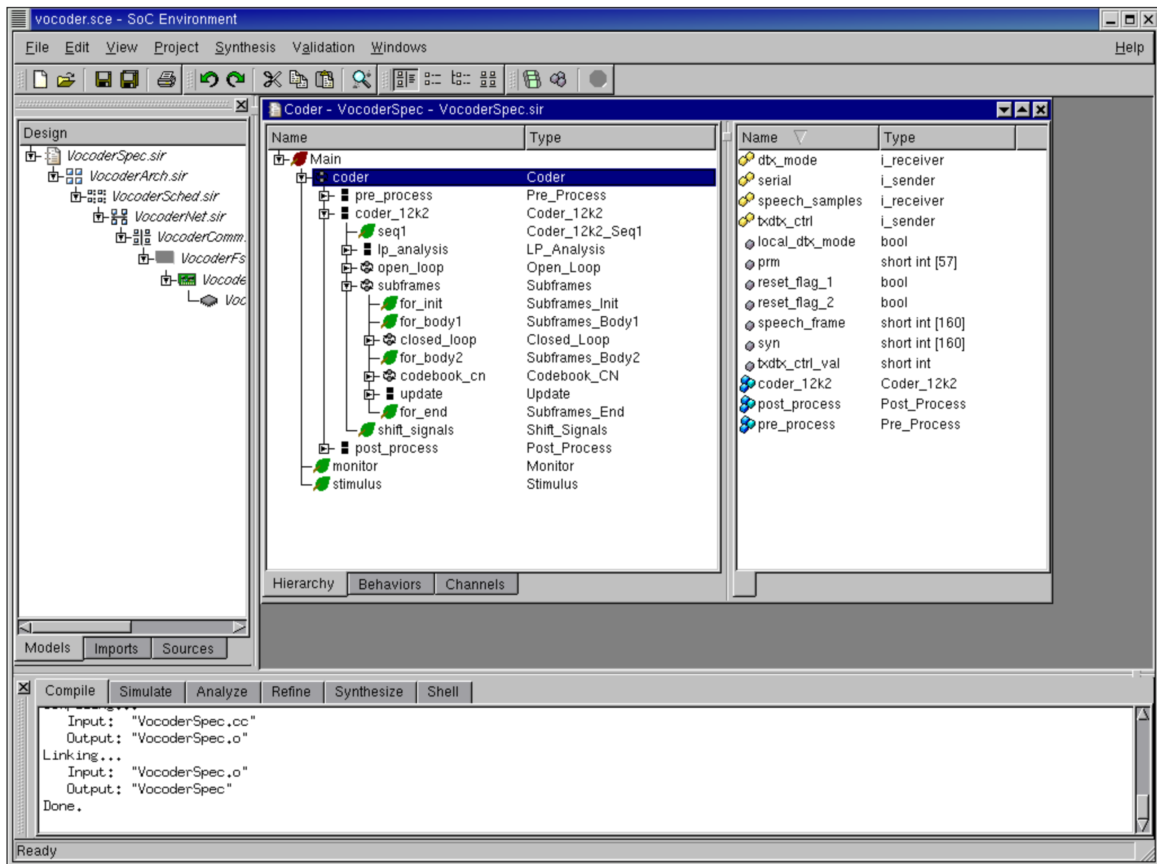


Figure 7.2: SCE graphical user interface (GUI).

and it provides access to an interactive SCE shell for issuing commands. Finally, the workspace acts as a container for design windows of various design models that are open at the same time. As shown in the screenshot, a design window provides a basic representation of the entities in a design for browsing and access. It is split into a sidebar and a viewpane. The sidebar shows the instance hierarchy of behaviors (and optionally variables and channels) plus a list of behavior and channel classes in the design. The viewpane then shows additional information about the currently selected design entity and its members (list of subbehavior, channel, and variable instances, ports, and methods).

In the following sections, additional aspects and screenshots of the SCE graphical user interface will be shown as they apply to the implementation of the various design tasks.

7.2 Specification Capture

Specification capture is the first task in the design process. At the input of the design flow, a specification model has to be developed and entered by the designer [45]. As the golden model for the whole system design, a specification has to describe the desired system behavior in a functionally correct manner. Furthermore, as design quality can depend to a large part on the characteristics of the specification (see Chapter 3), a specification has to optimized to balance and evaluate possible trade-offs in an iterative process.

The design environment supports the designer in this process by providing user interfaces for modeling of design functionality and for validation through simulation. Note that even though these parts of the environment are most important at the specification level, modeling and simulation features will be used for visualization and validation of all design models throughout the flow.

7.2.1 Modeling

The design environment supports design modeling by providing capabilities for editing of source code, manipulation of design entities, and visualization of design hierarchies. To that effect, the environment provides a variety of different views of data in a design model. As introduced in Section 7.1.5, the core view of a design model for basic browsing and manipulation is its design window with sidebar and viewpane. The design window allows browsing of the design hierarchy and access to information about all entities in the design. Furthermore, the design window supports editing of design models through (context or main) menu actions and drag-and-drop to instantiate, delete, rename, move, flatten, isolate, wrap, or change type (plug-and-play) of design entities and whole design hierarchies.

In addition to basic design windows, the environment integrates source code and graphical views of design models (Figure 7.3). Through the SpecC editor, source code of design models can be browsed and manipulated at any time. With the help of the SpecC compiler, the environment automatically translates the internal SIR format of design models into SpecC source code and back. Furthermore, whole design models can be read (opened) from or stored (saved) in SpecC source format for import or export of externally developed and exchanged designs, allowing specification capture outside of the environment or facilitating easy exchange of design models in a canonical format. In addition, parts or all of a design can be imported and exported in binary format for model reuse and IP exchange.

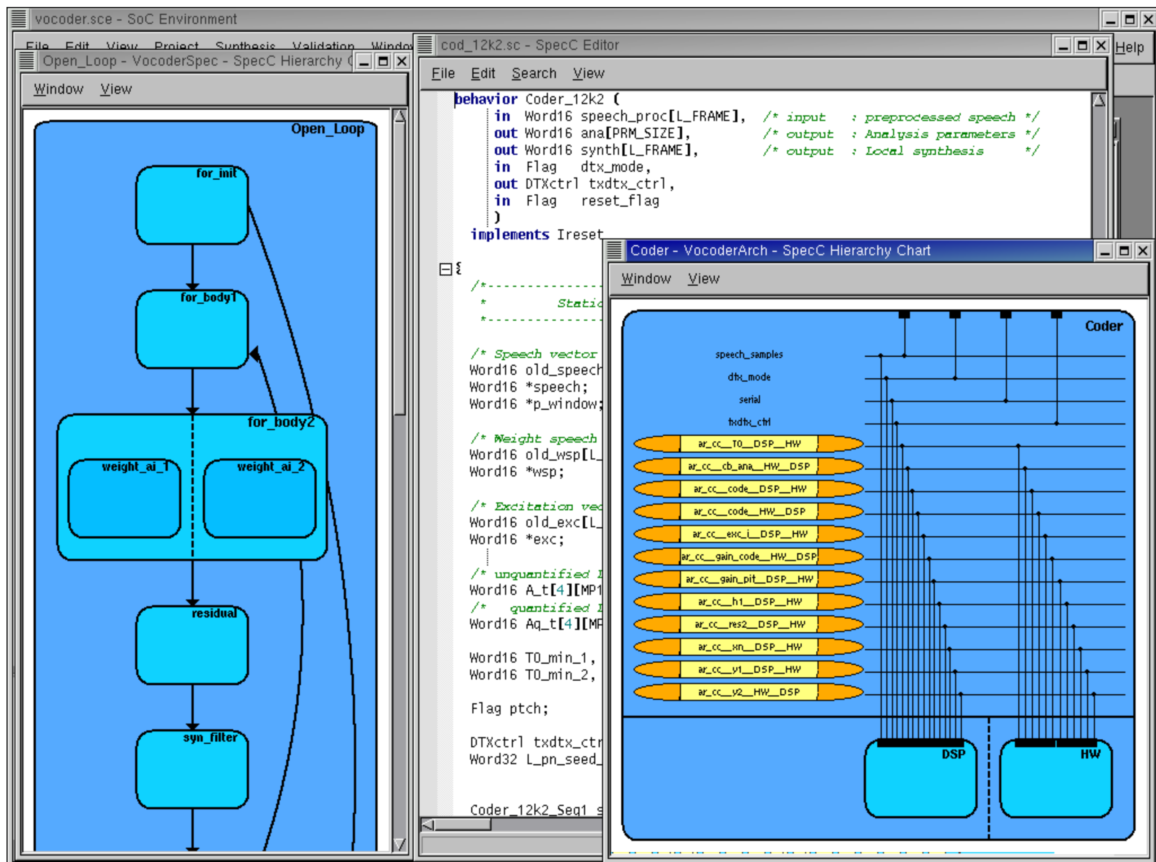


Figure 7.3: Model capture and browsing.

A graphical representation of design models is available in the environment in the form of SpecC hierarchy charts. SpecC charts show the behavioral hierarchy of a design, i.e. composition of a behavior in time out of subbehaviors and transitions between them (see Figure 7.3 on the left). Starting from a top behavior, chart views can show multiple levels of hierarchy where the user can selectively add or remove levels of nesting. In addition, charts can optionally display the structural hierarchy of behaviors, i.e. connectivity of subbehaviors and their ports via channels, variables, and ports of the parent (Figure 7.3 on the right). In summary, chart views can provide a complete graphical visualization of SpecC models which aid the designer in understanding and optimization of designs.

7.2.2 Simulation

After capture or refinement of designs, models need to be validated in order to ensure functional (and timing) correctness. The environment supports validation through simulation via

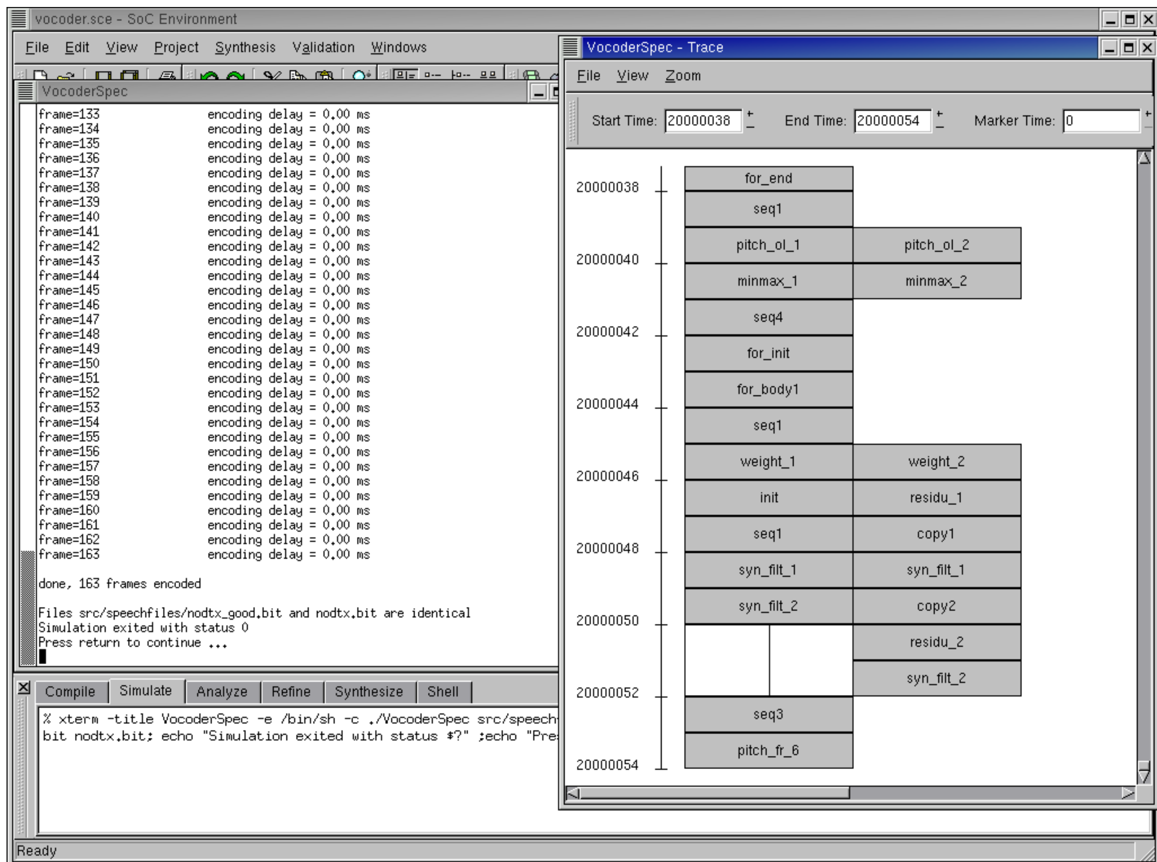


Figure 7.4: Simulation output and traces.

the SpecC compiler and simulator. At any time, design models can be compiled into simulation executables that can then be executed for simulation. As shown in Figure 7.4 on the left, executables can be run in an interactive terminal that captures simulation input and output. In addition, the environment supports background execution of simulation with optional logging of output in a file and log file viewer.

The SpecC simulator allows generation of traces of simulation runs during execution. Inside the environment, simulation traces can then be visualized (see Figure 7.4 on the right) to show the order of behavior and channel executions, values of variables and signals, and other simulation events over (logical) time. With the help of simulation traces, specifications and refined low-level design models can be debugged and analyzed to isolate problems and to optimize the design hierarchy. For example, to minimize slack and maximize utilization during scheduling, balancing of parallel behavior executions is easily achieved with the help of graphical trace views.

7.3 Profiling and Estimation

A critical aspect of any design process is the feedback about design quality metrics based on which designers can make decisions. In order to enable rapid design space exploration, estimation of metrics must be fast while providing accurate results in the sense that they are relevant and useful for evaluating and comparing alternatives. At high levels of abstraction, however, absolute accuracy is not of utmost importance but relative accuracy, so-called *fidelity* [68], is sufficient to prune the initial design space of infeasible alternatives.

Profiling that is part of the SoC design environment is based on a unique combination of dynamic profiling and static retargeting [12, 13, 11]. Initial profiling derives the characteristics of the application through simulation of the design specification. By then coupling application profiles with target characteristics based on the designer's application-architecture mapping, profiling is retargetable for static co-estimation of complete system designs in linear time without the need for time consuming re-simulation or re-profiling.

7.3.1 Profiling Flow

Figure 7.5 shows the flow of the profiling and estimation methodology in the design environment [12, 13, 11]. As design progress the design space is gradually trimmed and pruned of unsuitable design alternatives through profiling, retargeting, and simulation/estimation stages, until a final solution is reached.

In the dynamic profiling stage, the specification is instrumented and simulated to collect execution counts that capture the dynamic behavior of the application at the basic block level (N_{BB}). Using the counters collected during simulation together with a static analysis of the code, a profiling of the specification then computes the specification characteristics. Specification characteristics are implementation-independent and provide information about the inherent characteristics of the application. Based on these specification characteristics, the design space can be reduced to a large part. For example, if the specification does not contain any floating point operations, allocating dedicated floating point processors is counterproductive.

In the retargeting stage, designers allocate a target architecture of processing elements (PEs) connected via busses and map the computation and communication in the specification onto PEs and busses by matching specification characteristics and component attributes. A static retargeting of the specification then computes the implementation characteristics for computation and communication by coupling the design decisions and specification characteristics. These character-

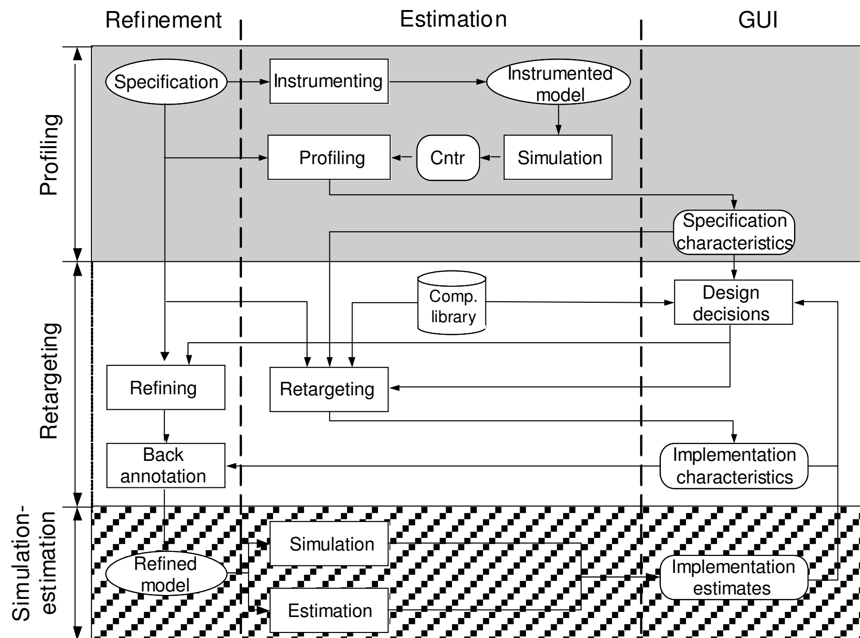


Figure 7.5: Profiling and estimation flow.

istics are implementation-dependent and represent the characteristics of the system design reflecting the designer's decisions. In an iterative process, the retargeting stage is executed repeatedly for different decisions in order to prune the design space of unpromising design alternatives. Because retargeting is a fast, purely static analysis, it enables designers to explore many alternatives and trim a large part of the design space in a short amount of time.

Finally, the most promising design alternatives remaining after the retargeting stage are then evaluated further in the simulation/estimation stage. Implementation characteristics are back-annotated into the design models created through refinement. Accurate implementation estimates for the design alternatives can then be obtained by simulating refined design models or through traditional estimation approaches. Implementation estimates provide the accuracy of traditional simulation- or estimation-based approaches at the expense of time-consuming simulation or analysis of each design alternative. However, as the design space has been reduced down to a few alternatives through profiling and retargeting, exhaustive simulation and analysis becomes feasible.

7.3.2 Metrics

The design environment compiles the raw data delivered by the profiler into a variety of quality metrics that allow evaluation of different aspects at different stages of the design process.

7.3.2.1 Profiling Data

The profiler computes the specification characteristics for each computation and communication entity in the specification. Raw profiling data from the profiler is a matrix of values $r_{i,d}$ over item types i and (standard and user-defined) data types d attached to each behavior, variable, and channel in the design. Characteristics are computed hierarchically by summation over the characteristics of an entity's children. Profiling supports three categories of data with different item types and with static and dynamic types in each category where static characteristics are derived directly from the code of the model whereas dynamic characteristics depend on data collected during simulation:

Operation Profiles Operation characteristics represent the complexity of the computation in the specification. They are attached to behaviors as the computational units of the system. Item types for operation profiles are standard and custom operators in the code where global functions can be designated as custom operations by the user. Data types for operation profiles are determined by the result type of the operation. Static operation data (code profile) relates to code size whereas dynamic data (computation profile) represents the amount of operations executed during simulation and is hence related to performance.

Traffic Profiles Traffic characteristics represent the complexity of the communication in the specification. They are attached to shared variables and channels connecting to ports of behaviors. Traffic profiles distinguish direction of traffic via corresponding input and output item types. Static traffic data (connection profile) relates to the static connectivity of entities whereas dynamic traffic data (traffic profile) represents the amount of data transfered during runtime.

Storage Profiles Storage characteristics represent the amount of memory required to hold the system's data. They are attached to variables and behaviors acting as containers for variables. Item types for storage profiles distinguish between local and global storage. Static storage data (data profile) represents memory allocated over the whole lifetime of the system whereas dynamic storage data (heap profile) relates to dynamically allocated stack and heap memory.

During retargeting, the profiler combines raw specification characteristics with target component attributes to derive implementation characteristics. Implementation characteristics

$$e_{i,d} = r_{i,d} \times w^t_{i,d}$$

are computed by multiplying specification characteristics $r_{i,d}$ with weights $w^t_{i,d}$ for a mapping of the design entity to target component t. Weight tables are defined for each component in the database.

DSP Weight Tables

Computation [cycles] | Code [instructions] | Variable Size [chars] | Variable Alignment [chars]

		*	/	%	+	-	<<	>>	<	>	<=	>=	==	!=
bool	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
char	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
unsigned char	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
short int	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
unsigned short int	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
int	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
unsigned int	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
long int	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
unsigned long int	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
long long int	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
unsigned long long int	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
float	4.0	8.0	8.0	8.0	8.0	4.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0
double	4.0	8.0	8.0	8.0	8.0	4.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0
long double	4.0	8.0	8.0	8.0	8.0	4.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0
void	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
event	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
void*	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

+ Type | - Type | + Operation | - Operation

Help | OK | Cancel

Figure 7.6: Weight table editor.

Depending on the component, they can be derived from the component's data sheet or from accurate simulations of selected, typical code kernels on the target component. In addition to the standard weights for basic data and item types stored in the database, the designer can manually tune weights for retargeting through a corresponding user interface (Figure 7.6). Furthermore, the designer can specify weights for custom data and item types collected during profiling instead of mapping them down to the basic data and item types they are composed of.

Similar to specification characteristics, implementation characteristics are computed hierarchically by adding implementation characteristics of children at each level. Retargeting supports two modes for hierarchical computation: analysis mode and estimation mode. The analysis mode provides mapping-independent results. It computes characteristics for each entity on each allocated component assuming that the whole entity (including children) is mapped to the target component. Results can be used by designers after allocation to select the most appropriate component to map each entity to. Estimation mode, on the other hand, computes characteristics based on both allocation and mapping decisions. For each entity, it generates characteristics on each target for those parts of the entity that are mapped onto this component. Results can therefore be used to evaluate mapping decisions.

7.3.2.2 Design Metrics

Given the raw profiling data, the design environment computes a set of design quality metrics for each category and type of data. Metrics are organized hierarchically where a hierarchical metric

$$M = \sum_n c_n$$

is computed as the sum over its child elements c_n. For a hierarchical metric, the metric is defined by the set of its child metrics. For metrics at the leaves of the hierarchy, the set of child elements is defined as a subset of specification or implementation characteristics. Leaf metrics

$$M = \sum_i \sum_d (m_{i,d} \times e_{i,d})$$

are computed as sums over subsets of characteristics $r_{i,d}$ or $e_{i,d}$ (resulting in specification or implementation metrics, respectively). Subsets and hence metrics are defined via masks $m_{i,d} \in 0,1$ applied during summation.

The design environment defines a hierarchy of standard metrics that cover all types and categories of profiling data. Standard metrics hierarchically organize metrics into useful and standard subcategories and subtypes. For example, computation is divided into ALU, memory access, and control operations where ALU operations are further subdivided according to data type (integer, floating point) and for each data type into arithmetic, logic, shift, and comparison operations. In addition to standard metrics, user-defined metrics are supported through a user interface that allows the designer to define custom masks for leaf metrics or custom sets of children for hierarchical metrics.

7.3.3 Visualization

In order to organize and present profiling and estimation data in an easily comprehensible manner, the design environment supports several different ways of displaying design data that can be used at different stages of the design process. Basic displays of individual design metrics are used throughout the design flow whereas special displays for connectivity and design quality provide a focused summary of critical information for selected steps in the design flow.

7.3.3.1 Design Metrics

The design environment provides a variety of ways to visualize individual design quality metrics (Figure 7.7). At the basic level, the viewpane of the design window shows metrics for the

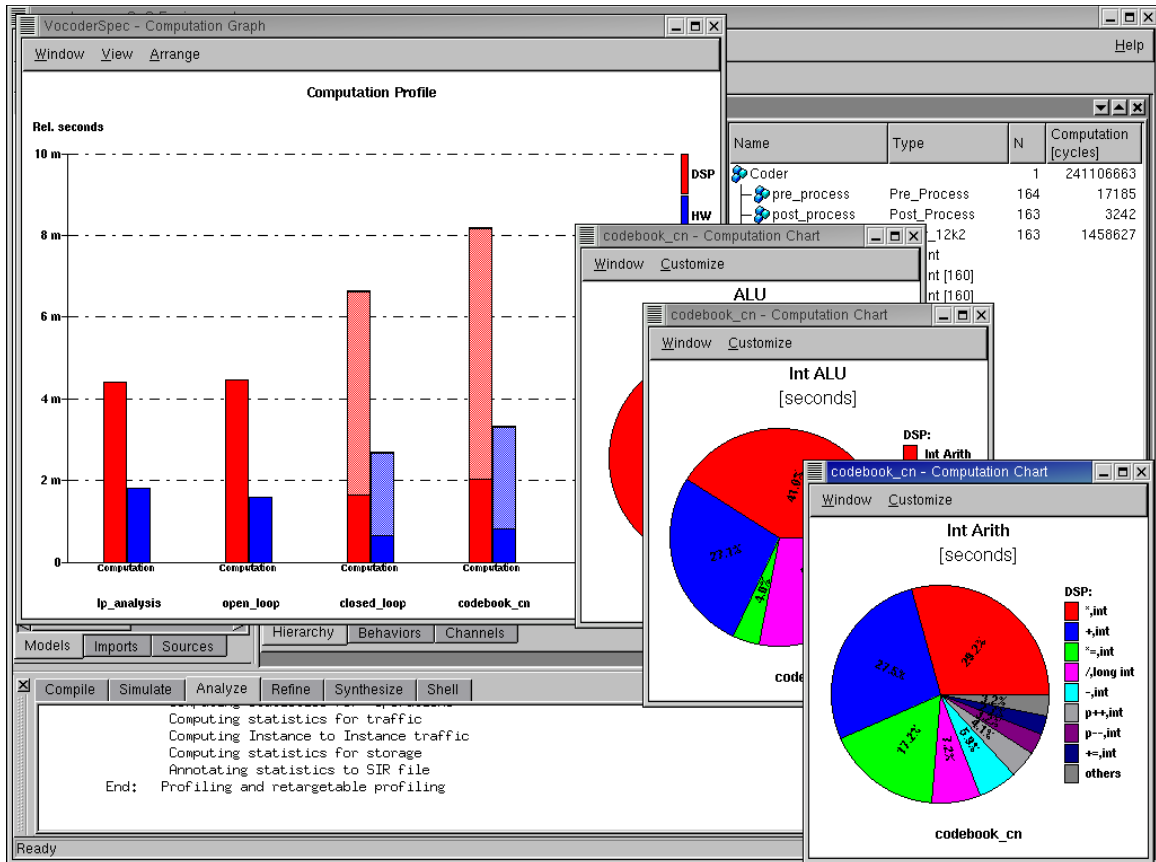


Figure 7.7: Visualization of design metrics.

currently selected design entitity and its members. The viewpane has columns for all currently active metrics where the user can customize the display by activating/deactivating metrics, reordering columns, and sorting viewpane rows by columns. If implementation characteristics are available after retargeting in either analysis or evaluation mode, the viewpane will have tabs for raw specification metrics and for all allocated target components. Target component tabs show implementation metrics for the currently selected entity on the respective target.

As shown in the screenshot, the environment supports visualization of metrics in the form of bar graphs for static and dynamic operation, traffic, and storage profiles. Bar graphs show the profile of metrics for the set of currently selected design entities where the user can select multiple design entities in the sidebar of the design window. A bar graph can chart either raw specification metrics or, if available, implementation metrics for selected design entities on all possible (allocated) target components. Hence, the graphs facilitate comparison between entities and their implementation on different targets.

Coder - VocoderSpec - Connectivity

Window View

	coder_12k2	post_process	pre_process	Totals
dtx_mode			0 B	0 B
serial		3 B		489 B
speech_samples			3 B	489 B
txdtx_ctrl		3 B		489 B
local_dtx_mode	3 B		0 B	489 B
prm	6 B	57 B		10269 B
reset_flag_1	7 B		1 B	1305 B
reset_flag_2		1 B	1 B	327 B
speech_frame	40 B		160 B	32858 B
syn	4 B			652 B
txdtx_ctrl_val	1 B	2 B		489 B
Totals	61 B	66 B	165 B	

Raw Bus0

Traffic

Figure 7.8: Connectivity display.

For hierarchical metrics, the environment can display the composition of the metric out of its children in the form of a pie chart. A pie chart shows the contribution of a child metric to its overall parent metric as either percentages, fractions, or actual values. The user can graphically navigate the metric hierarchy and open new pie chart windows by double-clicking on individual bars or pies in an already visible graph or chart.

7.3.3.2 Connectivity

The user interface of the design environment includes a connectivity display that focuses on showing data related to communication between different entities inside one level of hierarchy in the design (Figure 7.8). The connectivity display is arranged as a matrix of computational objects (behaviors) over communication objects (variables or channels). The matrix shows either simple connectivity information or raw or targeted traffic results from design profiling or estimation. In both cases, the connectivity display highlights the direction of data flow through appropriate coloring. In addition, the display can compress the data into a simplified matrix that only shows a summary of connectivity or traffic between behaviors.

Design Quality Metrics

PEs | Busses

PE	Utilization	Time	Program	Data	Power	Cost
DSP	83.1 %	2.68 s	8.5 kB (0.0 %)	15 kB (0.0 %)		2
HW	16.9 %	0.54 s	4.5 kB (0.0 %)	3 kB (0.0 %)		1
System	*50.0 %*	*3.23 s*	*13.0 kB*	*17 kB*		*3*

Help OK

Figure 7.9: Design quality dialog.

7.3.3.3 Design Quality

A summary of the most important metrics for a whole design is available through the design quality dialog in the environment's user interface (Figure 7.9). The design quality dialog shows overall attributes like utilization, size, delay, power, or cost for implementation of the system design on the chosen target architecture consisting of PEs and busses. Using the quality dialog, the designer can quickly evaluate the effect of design decisions on the overall quality of the design.

7.4 Databases

Part of the design environment are mandatory databases for processing elements (hardware and software processors, memories, IPs), communication elements (transducers, bridges, arbiters), busses (communication media in general) and RT units (register files, functional units, memories, interconnect) that are needed for partitioning, network, communication link and hardware design tasks throughout the design flow [43]. All databases inside SCE are based on a common, general, canonical format. Furthermore, a common, standard user interface for allocation and selection of components out of the databases is used throughout the design flow.

7.4.1 Database Format

SCE databases are stored as SIR files on disk. Therefore, the format of the databases is based on the SpecC syntax. The SpecC source code for each database must be compiled into SIR files using the SpecC compiler such that it can be used by SCE.

7.4.1.1 Database Organization

For each database, there is exactly one top-level SIR file. The top-level database SIR acts as a container for all components stored in the database. The database SIR includes components through import of individual component SIR files where component SIR files are stored in the same directory as or a sub-directory of the directory the database SIR is located in.

Each component in the database is stored in a separate SIR file. Component SIRs will be imported by SCE as needed throughout the design flow. Therefore, component SIRs must be self-contained. Furthermore, in case of parameterizable components, the component source code must be made available as part of the database, too.

In case of components with multiple models at different levels of abstraction, each model can be stored in a separate SIR file as long as the top-level database SIR contains all component models through direct or indirect import. In those cases, the basic component model used during allocation will contain pointers to other models (in the form of annotations) and SCE will import those models when needed in the design flow. The advantage of separate component files is that at any stage of the design flow, the system design will not contain any yet unused component models.

7.4.1.2 Component Format

Components are described by SpecC objects (behaviors or channels) in the SCE databases. Depending on the database, part or all of the functionality of a component is described through the SpecC code of the component object. As needed, component objects can be hierarchically composed out of other SpecC objects stored together with the top-level object in the database.

On top of SpecC code to describe functionality, additional meta-information about each component is stored in the database in the form of SpecC annotations attached to the component object. Apart from general annotations for database management, components generally have attributes, parameters, and profiling weight tables.

Attributes Component attributes describe characteristics or metrics for a component. Attributes of a component are stored as annotations attached to the component object under different keys or names as defined by the database. An attribute is either a simple annotation giving a fixed attribute value or a complex annotation describing a range of possible values for an adjustable attribute. For an adjustable attribute, the system designer will be allowed to tune the attribute value during allocation inside SCE within the range defined by the annotation.

Parameters All components in the SCE databases can be parameterizable. Components are made parameterizable by attaching a special annotation to them. The annotation specifies a list of parameters where each parameter is defined by its name, default value, range of possible values, unit and description. For a parameterized component, the system designer selects values for each of the component's parameters during allocation. The design environment will then supply the parameter values to the SpecC design generator to generate application-specific implementations of the component for use in the design as needed.

Weight Tables Component weight tables describe the special characteristics of a component which are used during the retargeting stage of profiling and estimation (see Section 7.3.2.1). Weight tables are attached to component objects as sets of annotations that define the rows and columns of the component's weight matrix. In general, a component can have static and dynamic operation, traffic, and storage weight tables corresponding to the calculation of respective implementation characteristics in the profiler. In all cases, a weight table is a matrix of weighting factors over data and item types as defined by the profiling data category they apply to.

In the case of storage weights, weight tables can be automatically calculated by the design environment from appropriate memory component attributes. If no storage weight tables are supplied by the database, the environment will compute storage weights based on size and alignment tables attached to the memory. Size and alignment tables in the database define the layout of standard data types in the given memory. Given the size and alignment information, storage requirements for all standard and custom data types in the design are computed as input to the profiler.

7.4.2 Allocation and Selection

The design environment supports interactive component allocation and selection by the designer via textual and graphical user interfaces. For the textual user interface, the environment provides corresponding APIs as part of the SCE shell. In addition, a pre-coded script that implements a comprehensive command line interface for allocation and selection is available.

Allocation and selection of components out of the databases through the environment's graphical user interface is performed via corresponding component allocation and database browser dialogs (Figure 7.10). An allocation dialog shows the list of currently allocated components. In the dialog, designers can rename components and change values of adjustable component attributes. Components can be added, copied, or removed via respective buttons.

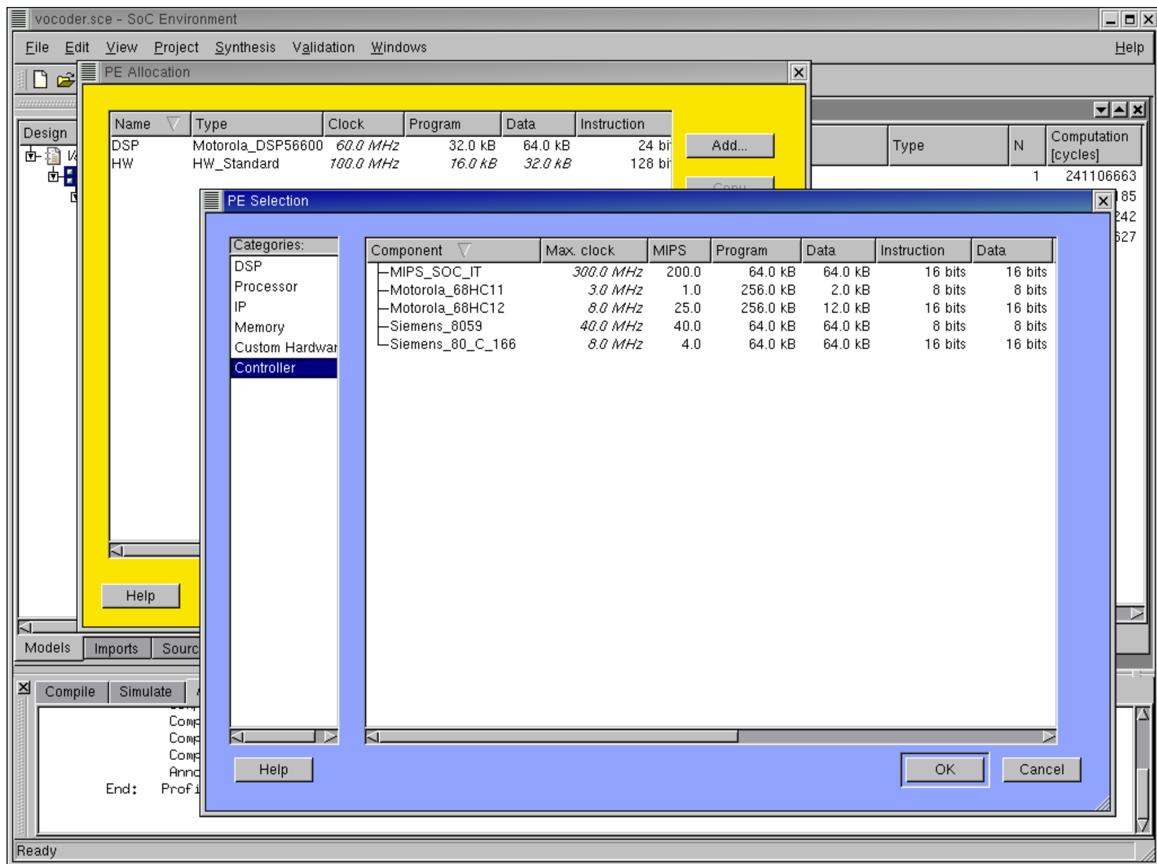


Figure 7.10: Database browser and component allocation.

Adding a component opens the database browser dialog. The database browser shows a list of all components in the database, organized by their categories. In order to support selection and searching, all relevant attributes of component are shown and lists can be sorted by any attribute. Selecting a component will add an instance of the component to the list of allocated components in the allocation dialog. In case of parameterizable components, a dialog will pop up where the designer can set component parameters used for allocation.

7.5 Computation Design

The design environment provides textual and graphical user interfaces for partitioning and scheduling within computation design (see Chapter 4). The textual interface consists of corresponding APIs inside the SCE shell and pre-coded, comprehensive command line scripts for partitioning and scheduling.

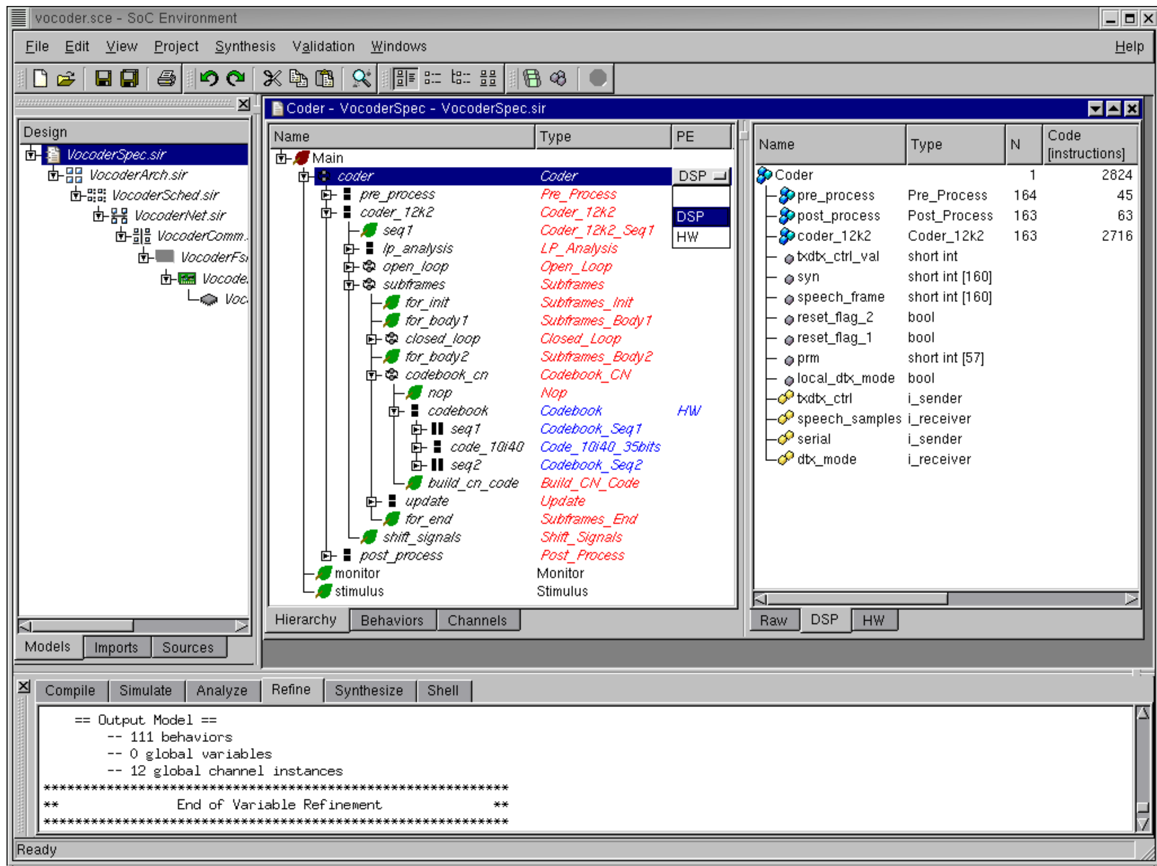


Figure 7.11: Behavior and variable mapping.

7.5.1 Partitioning

Partitioning requires mapping of behaviors and channels in the design onto previously allocated processor and memory PEs (Section 4.2). The allocation and mapping decisions are then annotated into the design as input to partitioning refinement. Allocation and selection of PEs out of the PE database is implemented based on the general database framework inside the environment described in Section 7.4.

After PE allocation and selection, mapping of behaviors and variables in the graphical user interface (Figure 7.11) is supported through the instance hierarchy tree in the sidebar of the design window. An additional column shows mapping of instances to PEs and allows the designer to select the target component for each behavior and variable instance. By default (no mapping), instances are mapped to the same target PE as their parent. The currently selected mapping is further highlighted by coloring behavior and variable types according their PE mapping.

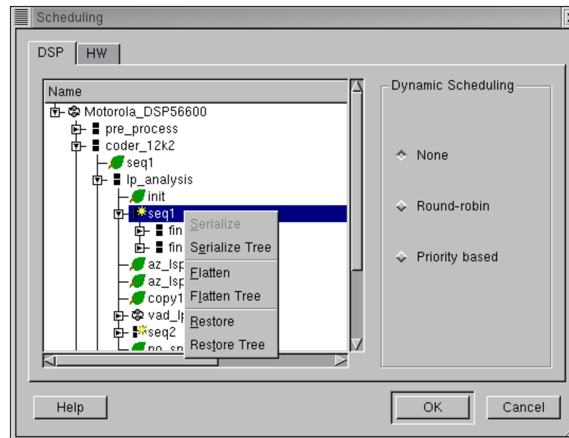


Figure 7.12: Scheduling dialog.

7.5.2 Scheduling

Scheduling requires serialization of behavior execution inside PEs. For software processors, static or dynamic scheduling is possible whereas custom hardware PEs have to be scheduled statically. Scheduling decisions about the order of subbehavior executions, task parameters, etc. are annotated to the behaviors inside PEs for input to scheduling refinement tools.

Scheduling through the graphical user interface is performed via a scheduling dialog (Figure 7.12). The scheduling dialog contains tabs for each PE in the system. Each tab shows the instance hierarchy tree of behaviors inside the corresponding PE where the designer can serialize, flatten, and reorder individual behaviors or whole behavior trees through context menus and drag-and-drop. In case of software processors, the designer can optionally select the dynamic scheduling strategy to be implemented on the PE. If dynamic scheduling is requested, task parameters can be assigned to concurrent behaviors through additional columns in the hierarchy tree.

7.6 Communication Design

Communication design consists of network and communication link design tasks (see Chapter 5) which are supported by the design environment through corresponding textual and graphical user interfaces. Similar to computation design, the textual user interface for communication design includes necessary APIs as part of the SCE shell and pre-coded, comprehensive command line scripts for bus allocation and channel mapping.

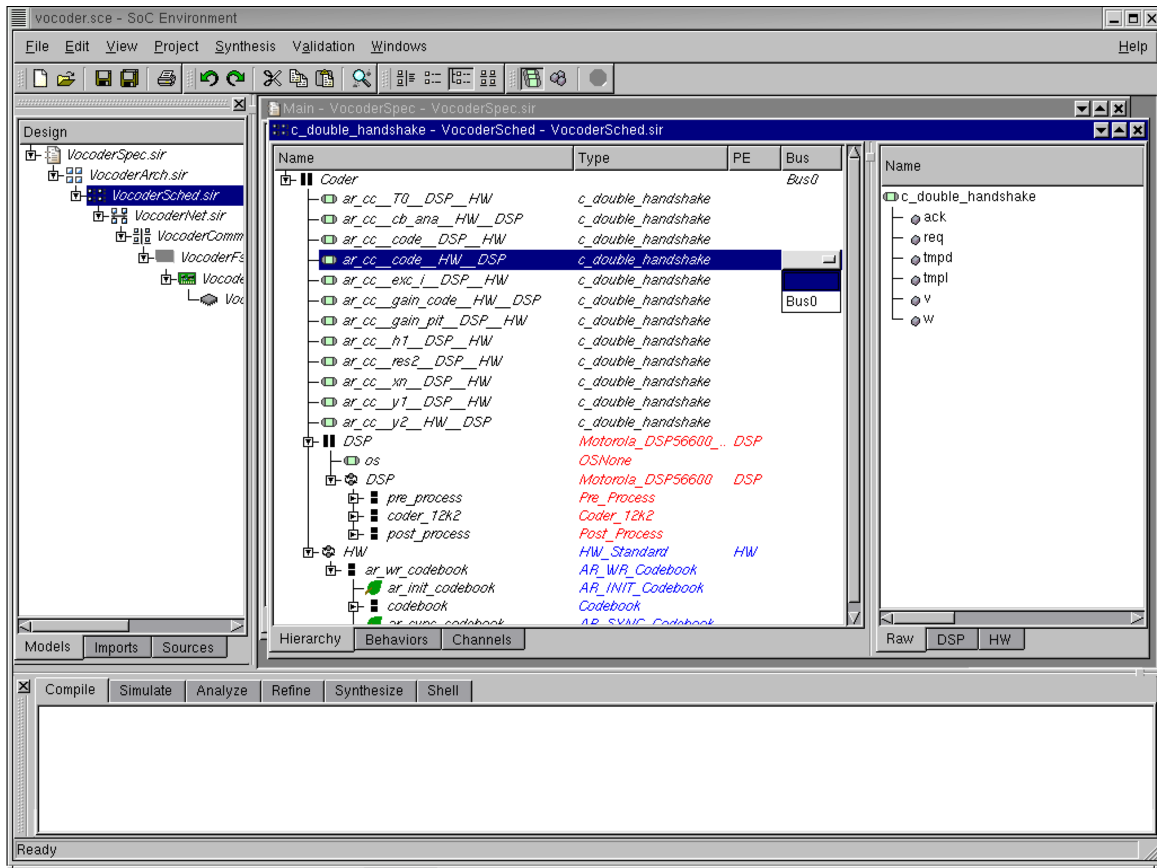


Figure 7.13: Channel routing.

7.6.1 Network Design

Network design requires definition of the overall system network topology and the routing of application channels over this network (Section 5.2). The network topology is defined by allocating communication media and additional transducer and bridge components out of the bus and CE databases, respectively. Allocation and selection of busses and CEs in the environment is accomplished using the general database framework introduced in Section 7.4. However, to allow allocation of both, the network allocation dialog contains separate tabs for bus and CE allocation. Furthermore, as part of bus allocation, connectivity of PEs and CEs to busses including the type of connection (master/slave) can be defined in the dialog.

In order to allow the definition of channel routing over allocated system network busses, channel mapping is supported in the graphical user interface similar to mapping of behaviors and variables to PEs. Hence, an additional column in the hierarchy tree of the design window sidebar

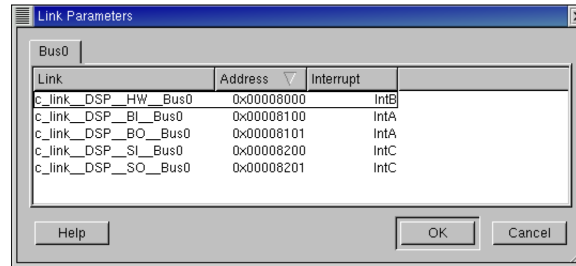


Figure 7.14: Link parameter dialog.

shows the mapping of channels to busses and allows the designer to define the routing of each channel (Figure 7.13). In this case, the designer defines mapping as an ordered list of busses over which all data of the channel will be routed.

7.6.2 Link Design

Communication link design requires selection of arbitration, addressing, and interrupt handling schemes on each medium in the system network (Section 5.3). For arbitration, corresponding arbitration protocols are part of and inserted together with communication media during bus allocation. As part of defining connectivity of PEs and CEs to busses, priorities of masters on the bus are assigned. In case of centralized media arbitration protocols, the designer has to allocate and connect arbiter components out of the CE database as part of network allocation (see Section 7.6.1). Similarly, as part of CE allocation, the designer can allocate and connect additional interrupt controller components for implementation of interrupt handling.

For implementation of communication links, addresses and interrupts have to be selected for all links on a medium. To this effect, a dialog is available in the graphical user interface that allows assignment of such link parameters (Figure 7.14). The dialog has tabs for each medium in the system. Each bus tab lists all logical communication links on that physical medium. Parameters for each link are shown and can be set in the respective columns.

7.7 Backend

The backend design process consists of hardware and software design tasks (see Chapter 6) where the environment provides corresponding textual and graphical user interfaces, analysis tools, refinement tools, and synthesis plugins.

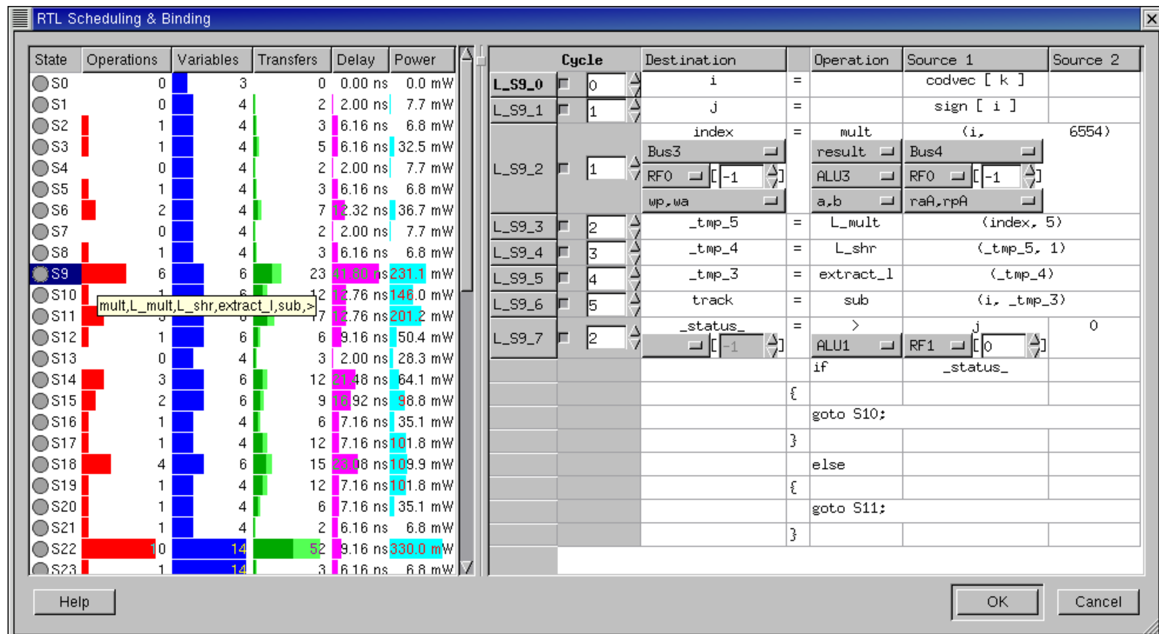


Figure 7.15: RTL scheduling and binding dialog.

7.7.1 Hardware Design

Hardware design generally requires allocation, scheduling and binding steps (Section 6.1). Allocation and selection of RT units out of the RTL database in the environment is based on the general database framework described in Section 7.4. In the case of RTL allocation, however, the allocation dialog contains tabs for each custom hardware PE in the system, allowing the definition of separate datapaths for each hardware PE.

The hardware design framework in the design environment includes an RTL analysis tool that computes hardware characteristics and implementation estimates. The RTL analysis tool can be applied to any (S)FSMD model. It produces metrics like number of operations and transfers in each state, variable lifetimes, critical path delays, and power estimates [88]. RTL metrics are used by the designer to make scheduling and binding decisions. For example, operations, transfers, and variables alive in each state determine functional unit, bus, and storage unit allocation in relation to scheduling. Furthermore, critical path delays and power metrics determine maximum clock frequency and power consumption of the component.

Scheduling and binding decisions for a selected behavior are visualized in the graphical user interface through the RTL scheduling and binding dialog (Figure 7.15). On the left, the scheduling and binding dialog shows the list of states in the behavior together with columns for

metrics computed by the RTL analysis tool. In addition to raw numbers, metrics are visualized as bar graphs for easy comparison. Furthermore, hovering the mouse over an operation or variable entry will pop-up a tooltip with details about the exact types of operations and variables in that state.

On the right, the scheduling and binding dialog shows a table with the three-address code for the currently selected state. Within that table, scheduling is performed by assigning statements to subcycles of the current state (within the constraints permitted by dependencies), splitting the state into individual cycles as necessary. Full or partial binding is supported for each entry in the table. If partial binding is enabled, the assignment of operators and variables to functional and storage units is shown and the designer can bind entries to units by selecting a target out of a drop-box containing a list of all allocated units. In addition, by enabling full binding, datapath connectivity can be shown and set. For each register transfer, the busses and functional or storage unit ports to be used can be selected. In summary, the dialog allows full control of scheduling and operator, variable, port, and bus binding.

Since manual scheduling and binding of complete custom hardware designs is tedious, time-consuming and error-prone, the default environment includes a set of RTL plugins that can be applied by the designer to automatically schedule and/or bind a set of hardware behaviors. In the dialog, the designer can make partial decisions and let the plugins fill in the rest. On the other hand, decisions made by the algorithms in the plugins can be verified and overridden by the designer at will.

7.7.2 Software Design

Software design (Section 6.2) only requires selection of a target operating system implementation out of the RTOS database for each dynamically scheduled software processor in the system. Selection of RTOS targets is implemented in the form of textual and graphical user interfaces of the environment using the general database framework outlined in Section 7.4.

All other necessary software design tasks are performed automatically by the respective backend tools, including code generation, RTOS targeting, compilation and linking. Outside of the system design environment, however, the designer has the option to employ external software development tool sets, e.g. to manually write, optimize, compile, and debug C or assembly code for the target processor based on and integrating with code automatically generated by SCE.

7.8 Summary

In this chapter, we described the implementation of the presented system design methodology in the form of the SoC Design Environment (SCE). SCE provides an environment for modeling, synthesis, exploration, and validation. It supports simulation, automatic model refinement, profiling, estimation, and synthesis for specification capture, computation, communication, and back-end design tasks. A set of databases that are part of SCE store component attributes and models needed for exploration and synthesis. Furthermore, user interfaces for refinement and validation enable designers to interactively make and enter decisions and to visualize design, analysis, and simulation results.

The contributions of this chapter include the definition of a complete design framework and comprehensive user interface for implementation of the complete system design flow from specification down to implementation. Software architecture, organization, tool flow, design model management, interfaces, and database formats of the design environment have been defined. Furthermore, textual and graphical user interfaces have been developed that support both automated and interactive design space exploration. Displays required for effective visualization and graphical organization of design data in order to aid designers in comprehension and in educated decisions making have been identified and implemented. In addition, interfaces for graphical capture and entry of design decisions have been developed. Finally, the environment allows the user to employ automated decision-making algorithms selectively on parts of the design through a plug-in mechanism.

In conclusion, the human-computer interface developed for SCE shows how to effectively visualize models and capture design decisions defined for each step in the flow. All in all, the SCE implementation proves the feasibility of an automated, interactive system design flow supporting rapid, efficient design space exploration.

Chapter 8

Experimental Results

The presented system-level design flow and methodology has been applied to the design of several industrial-strength system examples including the simplified mobile phone baseband system presented throughout this dissertation to illustrate the proposed design flow. The baseband system is a comprehensive system design example that covers a wide variety of application and target architecture requirements. Internally, it is composed out of separate voice encoding/decoding (vocoder) and JPEG encoding subsystems with specific requirements and features each. Therefore, in addition to the complete baseband system, both subsystems have been implemented separately. Experimental results have been obtained for each subsystem and for the overall baseband system as a combination of both.

In this chapter, we will present the results obtained from implementing the design examples following the steps of the developed design methodology. Section 8.1 gives an overview of the system design process as applied the baseband system and its subsystems. In Section 8.2, Section 8.3, and Section 8.4, results for the Vocoder subsystem, the JPEG encoder subsystem and the overall baseband system will be shown, respectively. For each step in the design flow, resulting system design models at corresponding abstraction levels will be analyzed, and tradeoffs between model complexity and accuracy as a result of introducing implementation detail in each step will be outlined. Furthermore, experimental results will show the benefits of each intermediate model and design step for rapid, early design space exploration.

8.1 Overview

Experimental results are derived from the implementation of the baseband system design introduced throughout this dissertation, The baseband system is an example of a simplified mobile phone baseband system design. The baseband system is composed hierarchically out of two separate subsystems for JPEG encoding and voice encoding/decoding (Vocoder). The design has been arranged and organized such that each subsystem can also be implemented and simulated independently. At each step of the design flow, the overall baseband system is then created as a simple, parallel composition of properly connected Vocoder and JPEG encoder subsystems[1].

8.1.1 Modeling and Simulation

Following the guidelines and rules for specification capture (see Chapter 3), the specification model of the example was written by manually converting available descriptions of the Vocoder and JPEG encoder algorithms in the form of C code into an equivalent SpecC description. At the lowest level, functions with basic algorithms in C were converted directly into SpecC leaf behaviors. At higher levels, the C function call hierarchy was converted into a clean, unambiguous, and explicit SpecC behavior hierarchy by analyzing dependencies and side-effects, exposing available parallelism, and encapsulating functionality and data hierarchically. Depending on the modularity and quality of the existing C code, effort for manual C to SpecC conversion can range from days up to months. In the case of the baseband system, quality of existing C code was low (communication through global variables, functions with side-effects, etc.) such that conversion took several months.

Given the clean, unambiguous, and precise specification model that is easy to understand for both humans and tools, effective and efficient system design, implementation, and design space exploration becomes possible. For the baseband system design example, SpecC models of the system at each step in the design flow and at all corresponding levels of abstraction were created through successive refinement of the initial specification model. Following the steps of the design methodology, all models of the baseband system were initially created through manual refinement over a period of 1-2 months. Later, with the help of the architecture, OS, network, communication, C, and RTL refinement tools, different models of the baseband system were also automatically generated where automatic model refinement was completed within seconds. Note that results for session and transport models are not shown here. Since session and transport layers are empty in

[1]Note that in order to simplify the illustration of the baseband system, this additional top level of concurrent hierarchy has been flattened out in the corresponding figures throughout this dissertation.

Behavior / Variable	Partitioning		Scheduling	
	PE	Memory	Child order	Algorithm
JPEG	ColdFire	-		
JPEGEncode			Default	
EncodeStripe			Default	
DCT	DCT_IP	-		
stripe[]	-	Mem	-	
ReceiveData	DMA	-		
Vocoder	DSP	-		Priority based
Decoder			Default	
Encoder			Default	
Codebook	HW	-		
SerIn	BI	-		
SpchOut	SO	-		
SpchIn	SI	-		
SerOut	BO	-		

Table 8.1: Baseband computation design parameters.

the case of the baseband implementation, corresponding models are equivalent to the link model in terms of complexity and accuracy.

All models of the baseband system in the design flow were simulated for validation and analysis. A common testbench was created and re-used for all models throughout the flow. The testbench exercises the baseband system by simultaneously encoding and decoding 163 voice frames (corresponding to 3.26 s of speech) while performing JPEG encoding of 50 pictures with 116×96 pixels. Models of the whole system and each subsystem were then simulated on a 360 MHz Sun Ultra 5 workstation using the QuickThreads version of the SpecC simulator.

8.1.2 System Design

The specification model of the baseband system has been introduced and described in detail in Section 3.3 of Chapter 3 (Figure 3.2). In Chapter 4, Chapter 5, and Chapter 6, the baseband system has then been taken down to an implementation through the individual steps and intermediate design models of computation, communication, and backend design tasks, respectively.

The design decisions made during computation design of the baseband system are summarized in Table 8.1. The system is partitioned onto nine PEs and one memory. In the resulting partitioned design model (see Figure 4.7, page 74), the JPEG encoder is running on a ColdFire processor assisted by a DCT IP, a DMA component, and a shared memory. The vocoder, on the other

<table>
<tr><th rowspan="2">Channel</th><th colspan="2">Network design</th><th colspan="4">Link design</th></tr>
<tr><th>Routing</th><th>Packeting</th><th>Address</th><th>Poll addr.</th><th>Intr.</th><th>Medium</th></tr>
<tr><td>stripeLen
imgSize</td><td>linkDMA</td><td>-</td><td>0x00010000</td><td>-</td><td>int1</td><td rowspan="4">cfBus</td></tr>
<tr><td>stripe[]</td><td>Mem</td><td>-</td><td>0x001000xx</td><td>-</td><td>-</td></tr>
<tr><td>HData
DData</td><td>linkDCT</td><td>-</td><td>0x00010010</td><td>-</td><td>int2</td></tr>
<tr><td rowspan="2">Ctrl</td><td>linkBri1</td><td>-</td><td>0x00010020</td><td>-</td><td>int0</td></tr>
<tr><td>linkBri2</td><td>-</td><td>0xB000</td><td>-</td><td>intA</td><td rowspan="7">dspBus</td></tr>
<tr><td>inparm</td><td>linkBI</td><td>-</td><td>0x8500</td><td>0x8501</td><td rowspan="2">intB</td></tr>
<tr><td>inframe</td><td>linkSI</td><td>-</td><td>0x8000</td><td>0x8001</td></tr>
<tr><td>exc[40]
⋮
T0</td><td rowspan="2">linkHW</td><td rowspan="2">-</td><td rowspan="2">0xA000</td><td rowspan="2">-</td><td rowspan="2">intC</td></tr>
<tr><td>prm[10]
⋮
gain</td></tr>
<tr><td>outframe</td><td>linkSO</td><td>-</td><td rowspan="2">0x9000
0x9500</td><td rowspan="2">0x9001
0x9501</td><td rowspan="2">intD</td></tr>
<tr><td>outparm</td><td>linkBO</td><td>-</td></tr>
</table>

Table 8.2: Baseband communication design parameters.

hand, is running on a DSP assisted by a hardware co-processor and four I/O processors. During scheduling, the two nested JPEG encoding pipelines and the encoding and decoding behaviors of the vocoder are statically scheduled in their default order. In addition, the vocoder is running encoding and decoding tasks dynamically scheduled under the control of a priority-based scheduling algorithm. The result of scheduling and computation design overall is the architecture model of the baseband system as shown in Figure 4.14, page 88.

In terms of communication design, Table 8.2 summarizes the design decisions for implementing the communication channels in the baseband system. During the network design process, the network is partitioned into one segment per subsystem with a bridge connecting the two segments. Individual point-to-point logical links connect each pair of stations in the resulting link model (see Figure 5.7, page 111). Application channels are routed over these links where the *Ctrl* channel spanning the two subsystems is routed over two links via the intermediate bridge. In all cases, no packeting of messages transfered over links is performed, and the bridge is implemented to be able to buffer whole *Ctrl* messages.

During link design, all links within each subsystem are implemented over a single shared medium. In both cases, the native ColdFire and DSP processor busses are selected as communi-

cation media. Within the segments, unique bus addresses and interrupts for synchronization are assigned to each link. On the ColdFire side, the memory is assigned a range of addresses with a base address plus offsets for each stored variable. On the DSP side, two of the four available interrupts are shared among the four I/O processors. In those cases, additional bus addresses for polling are assigned to each link (base address plus one). The result of communication link design is the communication model of the baseband system as shown in Figure 5.11, page 141. In addition, intermediate media access (see Figure 5.9, page 122) and protocol (see Figure 5.10, page 133) transaction-level models are created by the link design process.

Finally, in the backend design process the system is brought down to an implementation at the RTL/C level. For the software running on the ColdFire and DSP processors, C code is generated from the SpecC description of the software application and re-imported into the system design processor models. For the custom hardware co-processor on the DSP side, a 16-bit datapath with three ALUs (for general, saturated, and long arithmetic), three register files (32 entries each), one 256-word memory and six busses is allocated. Based on datapath allocation, scheduling and binding decisions, the SpecC description of the codebook search is synthesized into a fully bound and scheduled FSMD (style 4) model running at 100 MHz. Finally, for the DCT behavior, a predesigned, fully bound and scheduled FSMD (style 4) model of the IP including a transducer that translates between the DCT and ColdFire bus protocols was inserted out of the component database.

8.2 Vocoder System

The Vocoder is a voice encoding/decoding application implementing the so-called Enhanced Full Rate (EFR) speech transcoding standard that is part of the Global System for Mobile communications (GSM) set of standards employed worldwide for mobile telephony networks [30]. It is a lossy speech codec that encodes speech at 104 kbit/s into a bit stream with a rate of 12.2 kbit/s and vice versa. The codec is based on a speech synthesis model that emulates the way speech is generated in the human vocal tract [51]. Speech is processed in frames of $4 \times 5\,\text{ms} = 20\,\text{ms}$, and the standard specifies a timing constraint of 20 ms on the transcoding latency, i.e. on the delay when operating encoder and decoder in back-to-back mode.

As part of a previous project, the Vocoder design had been manually implemented on a Motorola DSP56600 processor running at 60 MHz assisted by a custom hardware co-processor running at 100 MHz [51]. For that implementation, assembly code for the Motorola DSP had been created by compiling the C code for the DSP into the DSP's instruction set using the (retargeted `gcc`)

compiler supplied by Motorola, by manually optimizing the assembly code generated by the compiler, and by linking the object code against a small custom RTOS kernel that uses a non-preemptive scheduling algorithm in which the decoder has higher priority than the encoder. A gate-level implementation of the custom hardware co-processor, on the other hand, had been obtained by manually designing an RTL implementation of the codebook search C algorithm and by synthesizing the resulting RTL VHDL code using Synopsys DesignCompiler. As a result, this final implementation consisted of approximately 70,500 lines of assembly and 45,000 lines of VHDL code.

The design flow as applied to the Vocoder subsystem in this dissertation directly replicates this previous implementation. At the input of the design flow, the specification model of the Vocoder in SpecC was developed based on a bit-exact C reference implementation included in the original Vocoder standard published by the European Telecommunication Standards Institute (ETSI)[99]. In addition to the RTL/C model at the output of the design flow, a fully cycle-accurate implementation model was created that replaces the C model of the DSP with an instruction-set simulator (ISS) running the existing DSP assembly code. For this purpose, the ISS supplied by Motorola for their DSP566k family of processors was plugged into the Vocoder model by wrapping a SpecC DSP model around the ISS' C-level API and by linking the model against the external ISS libraries.

8.2.1 Modeling and Simulation

The results for modeling the Vocoder subsystem at different levels of abstraction are shown in Table 8.3. The table lists complexities and characteristics of each Vocoder model in the design flow. For each model, the total number of lines of code including testbench, the total number of behaviors and channels, the maximal depth of the behavior hierarchy, and the number of behavior, channel, and variable instances at the top, system level of the Vocoder design are shown. In addition to totals, behaviors are further broken down into leaf behaviors and parallel behavior compositions. Major modeling results are summarized in Figure 8.1, showing both number of lines of code and number of design objects as the sum of behaviors and channels in each design.

Vocoder simulation results are shown in Table 8.4. For each model in the design flow, the table lists runtimes of Vocoder simulations and feedback about minimum and maximum Vocoder delays obtained during simulation. Simulation times are given for running the whole testbench of encoding and decoding 163 frames of speech. Simulated delays are shown for encoding, decoding and transcoding a single frame of speech. In general, simulated delays vary between minimum and maximum values since the decoder execution is data dependent. Vocoder simulation results are sum-

Model	Lines of code	Behaviors Total	Behaviors Leaf	Behaviors Parallel	Chnl Total	Max. depth	Top-level Beh	Top-level Chnl	Top-level Var
Specification	11,755	132	92 (69.7%)	12 (9.1%)	2	9	1	0	0
PE	12,244	144	96 (66.7%)	12 (8.3%)	2	10	6	7	11
Partitioned	12,601	144	96 (66.7%)	12 (8.3%)	2	10	6	17	0
Scheduled	12,585	144	96 (66.7%)	2 (1.4%)	2	10	6	17	0
Architecture	13,939	145	96 (69.2%)	1 (0.7%)	9	11	6	17	0
Link	13,590	145	96 (66.2%)	1 (0.7%)	13	11	7	6	0
Stream	13,663	145	96 (66.2%)	1 (0.7%)	16	11	7	12	0
Media Access	13,873	146	97 (66.4%)	1 (0.7%)	19	11	7	7	0
Protocol	14,058	148	99 (66.9%)	1 (0.7%)	18	12	9	3	0
Communication	14,311	152	100 (65.8%)	3 (2.0%)	18	14	7	0	9
RTL / C	21,387	39	21 (53.8%)	3 (7.7%)	18	8	7	0	9
Implementation	10,203	32	18 (56.2%)	2 (6.2%)	16	7	7	0	9

Table 8.3: Vocoder modeling results.

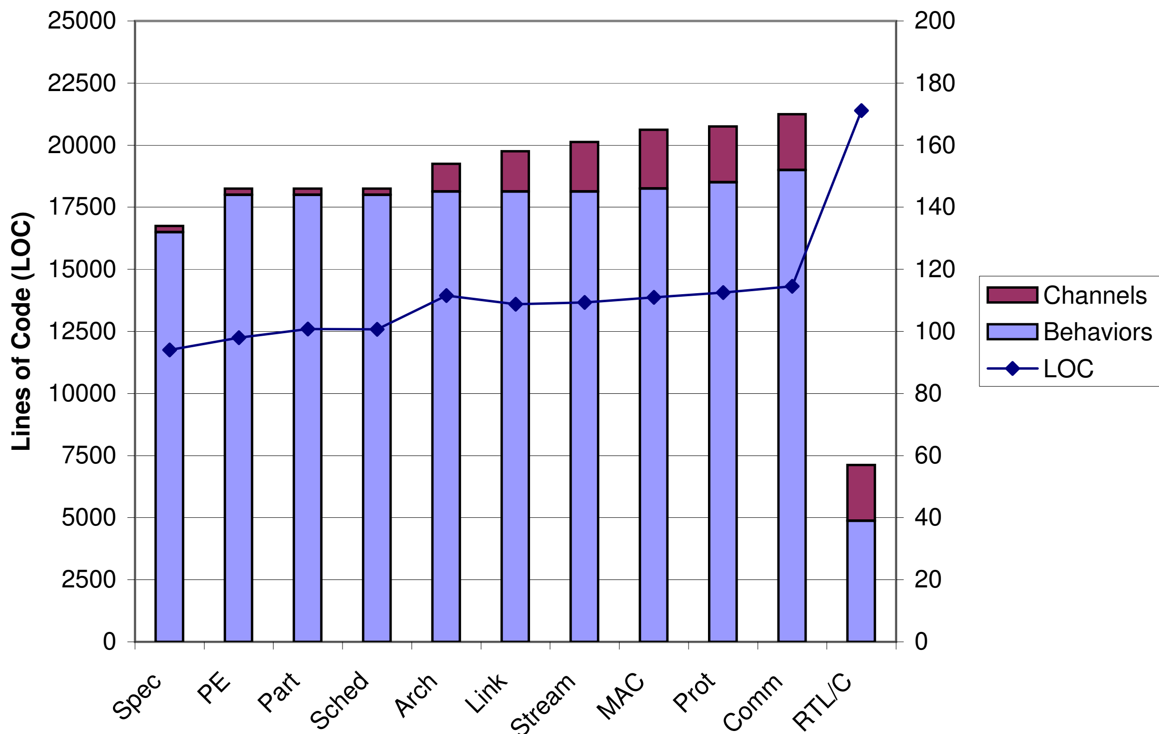


Figure 8.1: Vocoder modeling results.

Model	Simulation time	Simulated delays		
		Encoding	Decoding	Transcoding
Specification	16.3 s	0.00 - 0.00 ms	0.00 - 0.00 ms	0.00 - 0.00 ms
PE	16.9 s	7.14 - 7.14 ms	1.01 - 1.14 ms	8.15 - 8.28 ms
Partitioned	16.7 s	7.14 - 7.14 ms	1.01 - 1.14 ms	8.15 - 8.28 ms
Scheduled	16.6 s	8.29 - 8.29 ms	1.02 - 1.14 ms	9.31 - 9.43 ms
Architecture	17.8 s	9.09 - 9.22 ms	1.02 - 1.14 ms	10.11 -10.36 ms
Link	18.7 s	9.09 - 9.22 ms	1.02 - 1.14 ms	10.11 -10.36 ms
Stream	18.8 s	9.31 - 9.44 ms	1.10 - 1.23 ms	10.41 -10.67 ms
Media Access	25.2 s	9.86 - 9.99 ms	1.24 - 1.37 ms	11.10 -11.36 ms
Protocol	56.1 s	10.09 -10.22 ms	1.37 - 1.50 ms	11.46 -11.72 ms
Communication	178 s	10.22 -10.35 ms	1.44 - 1.56 ms	11.66 -11.91 ms
RTL / C	1270 s	10.44 -10.56 ms	1.44 - 1.56 ms	11.88 -12.12 ms
Implementation	≈ 5 h	9.22 - 9.22 ms	1.49 - 1.49 ms	10.71 -10.71 ms

Table 8.4: Vocoder simulation results.

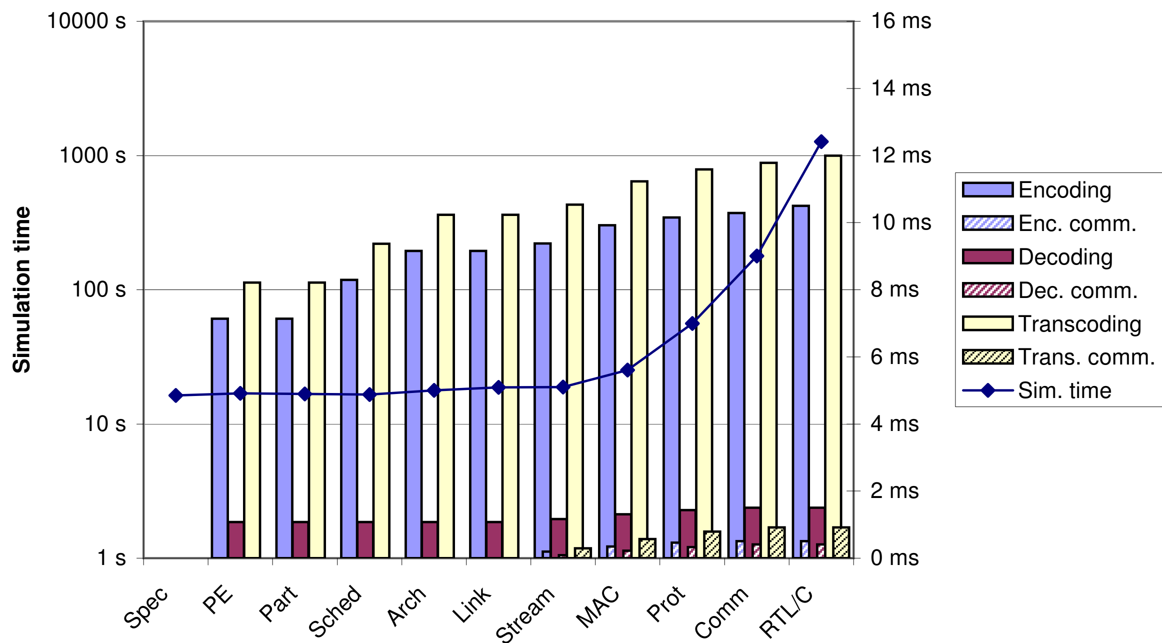


Figure 8.2: Vocoder simulation results.

marized in Figure 8.2. The graph plots both simulation runtimes and simulated encoding, decoding, and transcoding delays. Note that runtimes are plotted on a logarithmic scale. Furthermore, the graph also includes contributions of communication to overall simulated delays (see Section 8.2.4).

At the specification level, the design is a single behavior at the top level that executes the Vocoder algorithm and that defines the functionality to be implemented as a concurrent composition of four I/O behaviors plus the vocoder itself. The vocoder block is then internally composed of parallel encoder and decoder blocks which in turn are a hierarchical composition of different behavior types. The specification model is untimed and simulated delays are zero. Furthermore, events in the system are limited to modeling of pure causality only and simulation is fast.

In the PE model, an additional PE level of hierarchy is inserted and the depth of the behavior tree increases by one. In addition to newly added PE behaviors, additional leaf behaviors and top-level channel instances are inserted for inter-PE synchronization. In the process of behavior grouping, variables shared between behaviors on different PEs become global variables instantiated at the top. At the top level, the system becomes a composition of 6 PE behaviors connected via 17 communication channels and 11 shared variables where the five concurrent blocks from the original specification are mapped onto 6 concurrent PEs[2]. Timing in the form of estimated execution delays for behaviors mapped onto PEs is added. Finally, due to additional synchronization and timing events, simulation runtimes increase slightly.

In the partitioned model, global, top-level variables shared between PEs are distributed into local PE memories and replaced with additional channels for message-passing communication of updated variable values. Synchronization behaviors inside PEs are extended to exchange data over message-passing channels when necessary. At the top level, the partitioned model is a composition of the 6 PE behaviors that communication via 17 channels. Delays and simulation runtimes are not affected.

In the scheduled model, all parallelism expect for the concurrent composition of encoder and decoder behaviors has been statically scheduled away. Therefore, only one concurrent behavior is left in addition to the concurrent composition at the top level. Note that the top level of the design is not affected by scheduling. Due to scheduling, simulated delays increase and become more accurate as computation inside PEs is serialized. The overall decrease in parallelism reduces the number of simulated events and hence the simulation overhead. As a result, simulation runtimes decrease slightly.

[2]Hence, no increase or decrease in overall concurrency.

At the architecture level, an OS layer is added around the DSP PE as a result of dynamic scheduling. Consequently, the depth of the behavior hierarchy increases by one and there is one additional behavior. Due to insertion of the abstract RTOS model into the DSP OS layer, additional code in the form of channels needed for OS modeling is part of the design. Conversion of concurrent behaviors into tasks on top of the RTOS model finally removes all concurrency inside the inherently sequential PEs. Based on the chosen priority-based dynamic scheduling, the encoder delays increase as the high-priority decoder interrupts ongoing encoding whenever ready. Note that some parallelism is exploited and the decoding task gets interleaved into encoder wait times when blocking on results from the hardware. Therefore, encoding delays do not increase by the full decoding delay amount. Furthermore, note that variations in decoding delays lead to variations in the encoder delay as dynamic scheduling of both creates interdependencies between the tasks. In terms of simulation overhead, the additional context switching overhead for dynamic scheduling of tasks at fine timing granularity introduced together with the OS model results in a corresponding increase in simulation runtimes.

At the link level, top-level channel instances have been merged and multiplexed over the six remaining links for communication of the DSP with HW, SI, BI, BO, SO, and bridge components. As a result, additional channels for links and adapters inside PEs have been created. Furthermore, an additional behavior instance for the bridge connecting the Vocoder system with the testbench has been inserted at the top level of the design. Simulated delays are not affected by merging and routing of communication over yet untimed links and transducers. On the other hand, simulation overhead increases slightly.

In the stream model, adapter channels with link layer implementations are inserted into the PEs of the design. At the top level, the six communication links are split into 6 data and 6 control streams. Data and control stream channels are annotated with estimated delays for media latencies and interrupt handling overhead. Therefore, simulated delays are refined to include additional communication overhead. Since modeled delays are integrated into the synchronization inherent in the channels, simulation runtimes are not affected.

In the media access model, additional channels implementing stream layers are inserted into the design in the form of communication adapters inside PEs. At the top level, the 6 data streams are multiplexed over a single medium. As a result, the top level contains instances for the medium and the 6 separate control stream channels. In addition to adapter, control stream and media channels, the design contains an additional behavior for modeling of interrupt handling tasks. Therefore, in addition to estimated delays for communication over the medium as part of the medium channel,

simulated delays are refined to include additional overhead for interrupt processing through interrupt tasks on top of the operating system. As a consequence of these additional implementation details, simulation runtimes increase accordingly.

The protocol model adds a hardware abstraction layer (HAL) behavior around the DSP PE's existing OS layer. As such, the depth of the behavior hierarchy increases one additional level. Furthermore, additional adapter channels are inserted into the PEs for implementation of the media access layer. At the top level, control stream and media channels are replaced with a single protocol channel. In addition to the protocol channel and PE and bridge behaviors, the top level contains instances of adapter channels and behaviors that are needed for connectivity and routing of interrupts in the protocol model. Implementation of the media access layer in the Vocoder protocol model includes additional detail for data slicing and polling as part of low-level interrupt handling. Consequently, simulated delays increase mainly due to the additional interrupt handling overhead. On the other hand, slicing of packets into individual protocol transfers increases the number of simulated events and hence simulation overhead significantly, more than doubling simulation runtimes.

At the communication level, the bus-functional hardware layer of the DSP is inserted and the top-level protocol channel, adapter channel and adapter behavior instances are replaced with signal variables modeling the wires of the bus. As part of the DSP hardware model, an interrupt controller model running concurrently with the DSP core is inserted. Hence, the new model contains an additional interrupt controller behavior hierarchy with additional parallelism both inside the controller and between controller and DSP. Furthermore, the depth of the DSP hierarchy increases by two levels for the hardware layer and for the bus-functional layer that encapsulates DSP hardware core and interrupt controller. As communication and processor models get refined, simulated delays become more accurate. Particularly, processor hardware layers in the communication model now accurately describe suspension of regular computation during interrupt handling[3]. However, accurate modeling of protocols and protocol timing results in a large increase in simulation overhead as driving and sampling of individual wires within each protocol transaction incurs an additional simulation event each.

After backend design, the RTL/C model replaces the complete DSP software behavior hierarchy with a single behavior wrapping the generated C code. As such, the number of behaviors and the depth of the hierarchy drops significantly. On the hardware side, leaf behaviors are simply replaced with corresponding FSMD models. At the top level, the design remains unchanged from

[3]In previous models, low-level interrupt handlers are executed in parallel to regular computation such that their delays do not accumulate into the computation delays.

the communication model. Implementing custom hardware down to an RTL description refines the hardware execution delays to a cycle-accurate level. However, simulation overhead becomes huge as hardware simulation requires one additional event per simulated clock cycle. Aside from the SpecC RTL/C model, the backend process for the Vocoder produced 10,817 lines of C code for the application software running on the DSP and 96,930 lines of Verilog code for the synthesizable RTL netlist of the codebook search in hardware.

Finally, the implementation model replicating the existing Vocoder design replaces the complete DSP model with a SpecC wrapper around the DSP's instruction set simulator. Since the ISS code is not included in the SpecC model but linked against externally, lines of code, number of behaviors and depth of the hierarchy are reduced by the difference between behavioral and ISS DSP models. Cycle-accurate simulation of the software and hardware in the implementation model provides accurate delays equivalent to the actual, physical Vocoder design at the expense of an orders of magnitude increase in simulation time. Note that the difference in simulated delays between RTL/C and implementation models is due to inaccuracies in estimated DSP delays (in this case mainly due to inaccurate estimation of encoding software execution delays).

8.2.2 PE Modeling

A crucial aspect of most system designs is partitioning of functionality onto components. In a computation-dominated design like the Vocoder, therefore, insertion of PEs during behavior partitioning becomes a critical decision. In order to demonstrate exploration of the PE design space, we allocated alternative system architectures with three PEs: in addition to the (Motorola) DSP and the custom hardware co-processor running at 60 MHz and 100 MHz, respectively, we selected a general-purpose processor (Motorola ColdFire at 60 MHz) to explore performance versus cost tradeoffs. Mappings of eight top-level behaviors (five top-level encoder behaviors plus three levels of hierarchy of behaviors inside *Codebook*) to every PE were evaluated.

Using the scripting capabilities of the design environment together with simulation of each alternative, we ran an automated, exhaustive search of all $3^8 = 6561$ design alternatives. Running on a Pentium IV Linux PC at 2.4 GHz, each iteration required 4.7 s (1.7 s per simulation and 3 s per model refinement) and the complete search was finished in 85:04 h. Figure 8.3 shows the transcoding delay vs. cost for all design alternatives. For both ColdFire and DSP a fixed cost of 20 each was assigned for the manufacturing cost. For custom hardware, a linear cost function with a base cost of 20 and an additional cost of $1 per 10 static operations of code complexity was assumed

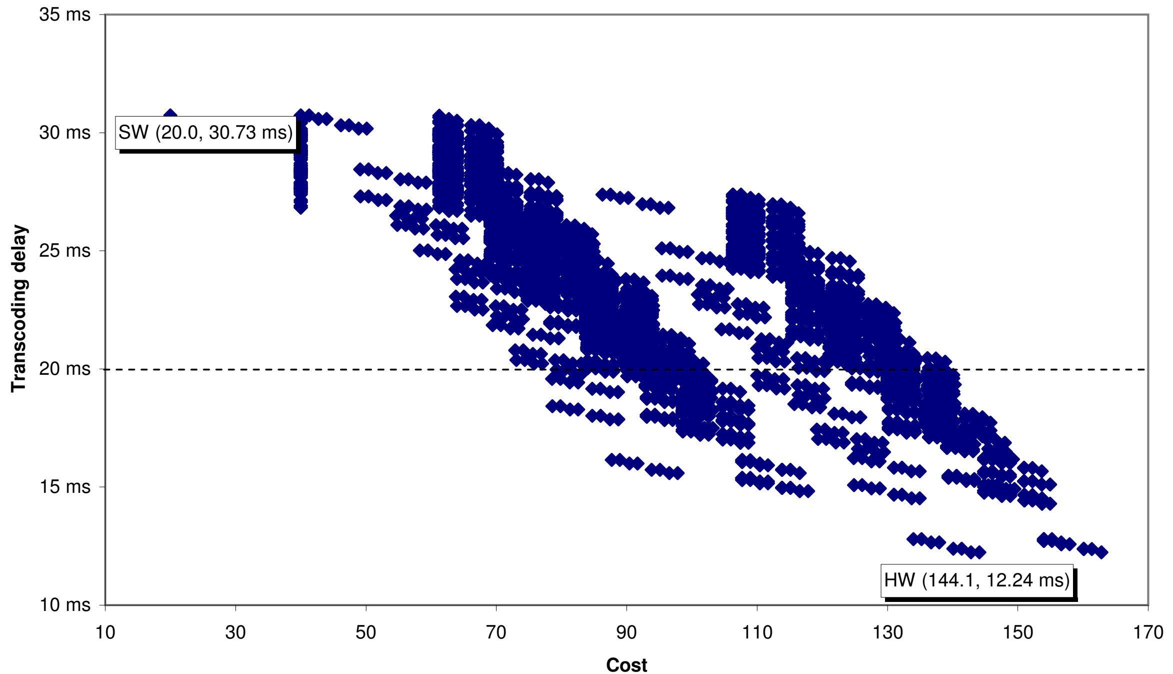


Figure 8.3: Vocoder PE exploration.

to estimate costs of control logic and design overhead. If no behavior is mapped to a PE, its cost is assumed to be zero.

Exploration results show that a pure software solution running on the ColdFire processor (upper left) is the cheapest design but has the largest delay. A pure hardware solution (lower right), on the other hand, is the fastest design at one of the highest costs. Optimal solutions in the sense of the cheapest design that meets the transcoding timing constraint of 20 ms can easily be identified.

Based on the exploration results, a large part of the design space can be pruned, infeasible design alternatives can be ruled out at this early stage, and a Pareto-optimal region of the design space can be identified for further investigation through following design stages. In summary, high-level modeling of PEs enables rapid, early validation, exploration and evaluation of design alternatives in terms of the system computation architecture.

8.2.3 OS Modeling

As part of the system architecture model, an RTOS layer which contains an abstract model of the underlying operating system scheduler gets introduced into the system design models (see Section 4.3.2 in Chapter 4). In order to demonstrate the effectiveness of the RTOS layer for design space exploration, we evaluated different architectural alternatives using our RTOS model.

	Simulated delays			OS switches	
	Encoding	Decoding	Transcoding	Task	Context
Unscheduled	8.29 - 8.29 ms	1.02 - 1.14 ms	9.31 - 9.43 ms	0	0
Round-robin	8.78 - 8.91 ms	1.83 - 2.33 ms	10.61 -11.24 ms	1771	64
Encoder > Decoder	8.34 - 8.37 ms	2.92 - 3.05 ms	11.26 -11.42 ms	1712	8
Decoder > Encoder	9.09 - 9.22 ms	1.02 - 1.14 ms	10.11 -10.36 ms	1711	2
RTL / C	10.44 -10.56 ms	1.44 - 1.56 ms	11.88 -12.12 ms	1711	1711

Table 8.5: Vocoder OS simulation results.

We created three scheduled models of the Vocoder with varying scheduling strategies: round-robin scheduling and priority-based scheduling with alternating relative priorities of encoding and decoding tasks.

Depending on the type of the chosen scheduling strategy, the RTOS model imported out of the RTOS model database adds about 1000 lines of code to the system design model. Furthermore, the template for the processor's RTOS layer imported out of the PE database adds approximately another 100 lines of code. Actual refinement of the template and the system model by converting relevant SpecC statements into RTOS interface calls following the steps described in Section 4.3.2.2, Chapter 4 requires changing or adding only 104 lines or less than 1% of code. All of these steps have been integrated into the automatic refinement tool which automatically generates RTOS-based architecture models from unscheduled partitioned models. With automatic refinement, all three models could be created within seconds, enabling rapid exploration of the RTOS design alternatives.

Table 8.5 shows the simulation results of this RTOS exploration. For exploration, the models were annotated to deliver feedback about the number of RTOS task and context switches in addition to results about response times as reflected by the transcoding delays encountered during simulation. A task switch occurs whenever the OS model is selecting a new task to dispatch. Context switches, on the other hand, only count the number of times CPU state is actually changed. Due to timing granularity, multiple task switches can happen in the same simulated time period. However, only the last task switch will possibly lead to a real context switch.

At high levels of abstraction, task switches can provide an estimate about expected context switches at finer timing granularity in the implementation. On the other hand, only context switch counts provide feedback about real CPU state changes and associated overheads, especially at lower levels of abstraction. In the Vocoder, for example, the OS model has to switch from decoder to encoder task every time the decoder communicates. In the untimed case at high levels, however, decoder communication finishes in zero time and the OS model can switch back to the decoder in

the same time period without ever executing the encoder (resulting in two task switches and zero context switches). In the final Vocoder implementation, task switches measured at higher levels translate into real context switches as each decoder communication then requires two actual context switches for execution of the corresponding interrupt handling task.

Simulation results show that the simulation overhead introduced by the RTOS model is negligible while providing accurate results. As explained by the fact that both tasks alternate with every time slice, round-robin scheduling causes the largest number of task and context switches while providing a low response time. Note that context switch delays in the RTOS are not modeled, i.e. the larger number of context switches would introduce additional delays. In priority-based scheduling, it is of advantage to give the decoder the higher relative priority. Since the encoder execution time dominates the decoder execution time this is equivalent to a shortest-job-first scheduling which minimizes wait times and hence overall response time. Furthermore, the number of task and context switches is lower since the RTOS does not have to switch back and forth between encoder and decoder whenever the encoder waits for results from the hardware co-processor. Therefore, priority-based scheduling with a high-priority decoder was chosen for the final implementation as mentioned previously during description of the design process.

In summary, compared to the huge complexity required for the implementation model, results prove that the RTOS model enables early and efficient evaluation of dynamic scheduling implementations.

8.2.4 Communication Modeling

In order to evaluate accuracy of communication modeling in the different Vocoder design models, we obtained simulation results that separate overall delays into communication overhead and computation delays. To measure the contributions of communication overhead to overall simulated delays, we simulated all Vocoder design models with (estimated) computation delays for application code inside the PEs set to zero. The resulting simulated delays therefore provide feedback about the pure communication overhead in each model. More specifically, these simulated communication delays do not include (estimated or other) delays for implementation of communication (i.e. adapter channels) inside the PEs. They only reflect unavoidable overhead due to multiplexing, arbitration or interrupt handling, for example.

Table 8.6 lists the simulated contributions of communication overhead to overall encoding, decoding and transcoding delays. For resulting communication delays, both absolute and rela-

Model	Encoding		Decoding		Transcoding	
Specification	0.00 ms	(0.0%)	0.00 ms	(0.0%)	0.00 ms	(0.0%)
PE	0.00 ms	(0.0%)	0.00 ms	(0.0%)	0.00 ms	(0.0%)
Partitioned	0.00 ms	(0.0%)	0.00 ms	(0.0%)	0.00 ms	(0.0%)
Scheduled	0.00 ms	(0.0%)	0.00 ms	(0.0%)	0.00 ms	(0.0%)
Architecture	0.00 ms	(0.0%)	0.00 ms	(0.0%)	0.00 ms	(0.0%)
Link	0.00 ms	(0.0%)	0.00 ms	(0.0%)	0.00 ms	(0.0%)
Stream	0.20 ms	(2.1%)	0.09 ms	(7.7%)	0.29 ms	(2.8%)
Media Access	0.35 ms	(3.5%)	0.22 ms	(16.9%)	0.57 ms	(5.1%)
Protocol	0.46 ms	(4.5%)	0.33 ms	(23.0%)	0.79 ms	(6.8%)
Communication	0.51 ms	(5.0%)	0.41 ms	(27.3%)	0.92 ms	(7.8%)
RTL / C	0.51 ms	(4.9%)	0.41 ms	(27.3%)	0.92 ms	(7.7%)
Implementation	0.44 ms	(4.7%)	0.26 ms	(17.4%)	0.70 ms	(6.5%)

Table 8.6: Vocoder simulated communication delays.

tive values (as percentage of overall delays) are shown. Note that communication delay results are included in the graph of Vocoder simulation results shown in Figure 8.2.

In the Vocoder design, communication delays on the encoder side are generally higher due to the additional overhead of communication with the hardware co-processor. On the other hand, due to the low computational complexity of the decoder, communication delays have a relatively large contribution to the overall delay on the decoder side, i.e. the decoder is more communication-dominated than the encoder.

Results confirm that communication delays remain zero throughout the computation design process. During communication design, delays are then gradually refined down to their final implementation values. In the link model, delays remain zero since channel merging and insertion of yet untimed transducers does not introduce any communication overhead. Beginning with the stream model, nonzero communication delays are observed as channels are annotated with estimated media latencies. In the media access model, additional communication delays are introduced largely due to high-level interrupt handling through interrupt tasks running on top of the RTOS. In the protocol model, interrupt handling delays are further increased due to the necessity of slave polling inside the added low-level interrupt handlers. Finally, in the communication model, protocol transactions are refined down to accurate protocol timing, and processor interrupt behavior is refined by inserting interrupt controllers and processor hardware layers that accurately model suspension of regular computation during processing of interrupts.

Communication delays in the RTL/C and communication model are the same as they are not affected by synthesis of computation inside the PEs. Compared to the actual implementation

model, however, communication delays in higher-level models are generally overestimated. On the DSP side, the actual Vocoder implementation employs a small, custom operating system kernel with minimal overhead in terms of interrupt handling. In the implementation, interrupt tasks disappear as they are inlined into computation tasks and interrupt handlers. Consequently, interrupt handling delays are lower which reduces overall delays especially in the communication-intensive decoder.

In summary, different levels of communication modeling allow exploration of trade-offs between complexity, speed and accuracy of design models. Depending on the application and the chosen target communication architecture, abstracted transaction-level models can provide accurate results in shorter time. Furthermore, intermediate models allow stepwise implementation and evaluation of different communication designs aspects.

8.3 JPEG Encoder Subsystem

JPEG is a lossy image compression standard originally developed by the Joint Photographic Experts Group committee [7] and now published by the International Telecommunication Union (ITU) [63]. It is designed to compress color or gray-scale photographic images by exploiting known limitations of the human eye. For the JPEG encoder subsystem that is part of the baseband system, we implemented a so-called baseline sequential JPEG encoding process. Based on an existing, public C reference implementation, the initial SpecC specification model of the JPEG encoder was developed [14, 106]. Furthermore, as part of previous projects, cycle-accurate RTL implementations of the discrete cosine transform (DCT) block [14] and of a transducer between DCT and ColdFire busses [106] had been developed. For the design of the JPEG encoder shown here, these pre-designed DCT and transducer components were added to the IP database for import into the design during the design process.

8.3.1 Modeling and Simulation

Results of modeling the JPEG encoder at different levels of abstraction are shown in Table 8.7 and Figure 8.4. Simulation results of the JPEG encoder models, on the other hand, are listed in Table 8.8 and summarized in Figure 8.5.

At the specification level, the JPEG encoder is a single behavior internally composed out of untimed behaviors and channels including the two nested, concurrent pipelines. In the PE model, the specification is mapped onto 4 concurrent PEs communication 7 synchronization channels and

Model	Lines of code	Behaviors Total	Behaviors Leaf	Behaviors Parallel	Chnl Total	Max. depth	Top-level Beh	Top-level Chnl	Top-level Var
Specification	2,684	31	21 (67.7%)	3 (9.7%)	5	6	1	0	0
PE	3,243	53	32 (60.4%)	5 (9.4%)	5	7	3	7	4
Partitioned	3,243	51	28 (54.9%)	7 (13.7%)	5	7	4	7	0
Scheduled	3,236	51	28 (54.9%)	1 (2.0%)	5	7	4	7	0
Architecture	3,420	52	28 (53.8%)	1 (1.9%)	7	8	4	7	0
Link	3,633	55	32 (58.2%)	1 (1.8%)	18	8	5	3	0
Stream	3,788	55	32 (58.2%)	1 (1.8%)	21	8	5	7	0
Media Access	4,010	56	33 (58.9%)	1 (1.8%)	27	8	5	4	0
Protocol	5,007	61	36 (59.0%)	1 (1.6%)	38	9	6	6	0
Communication	5,620	65	38 (58.5%)	3 (4.6%)	41	11	7	0	25
RTL / C	7,089	35	18 (51.4%)	3 (7.7%)	41	7	7	0	25

Table 8.7: JPEG encoder modeling results.

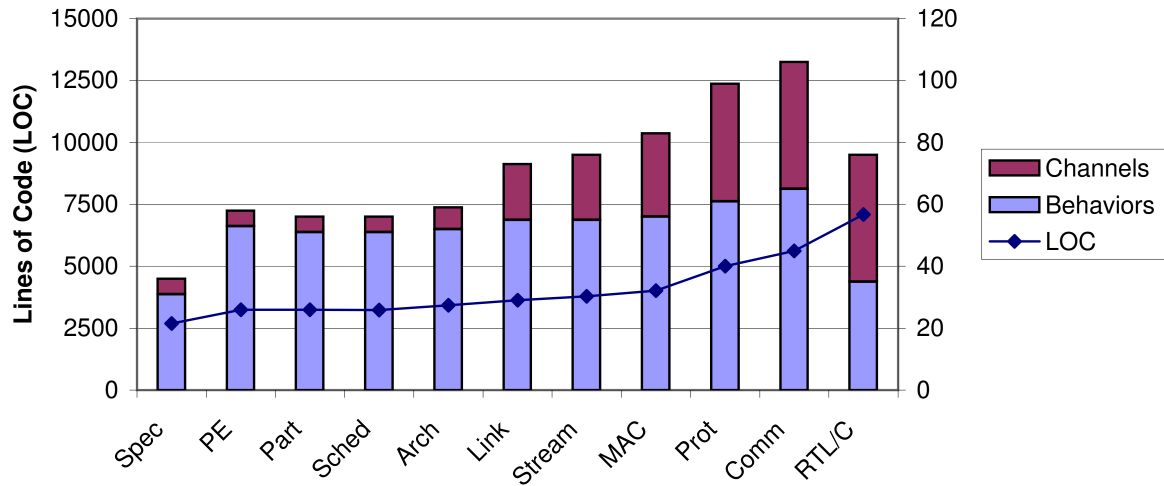


Figure 8.4: JPEG encoder modeling results.

Model	Simulation time	Simulated delays		
		Encoding	Communication	
Specification	0.30 s	0.00 ms	0.00 ms	(0.0%)
PE	0.28 s	33.21 ms	0.00 ms	(0.0%)
Partitioned	0.27 s	33.21 ms	0.00 ms	(0.0%)
Scheduled	0.29 s	74.13 ms	0.00 ms	(0.0%)
Architecture	0.29 s	74.13 ms	0.00 ms	(0.0%)
Link	0.30 s	74.13 ms	0.00 ms	(0.0%)
Stream	0.62 s	74.41 ms	0.28 ms	(0.4%)
Media Access	0.99 s	74.53 ms	0.40 ms	(0.6%)
Protocol	8.66 s	75.67 ms	1.18 ms	(1.6%)
Communication	20.6 s	75.63 ms	1.50 ms	(2.0%)
RTL / C	23.4 s	74.78 ms	1.50 ms	(2.0%)

Table 8.8: JPEG encoder simulation results.

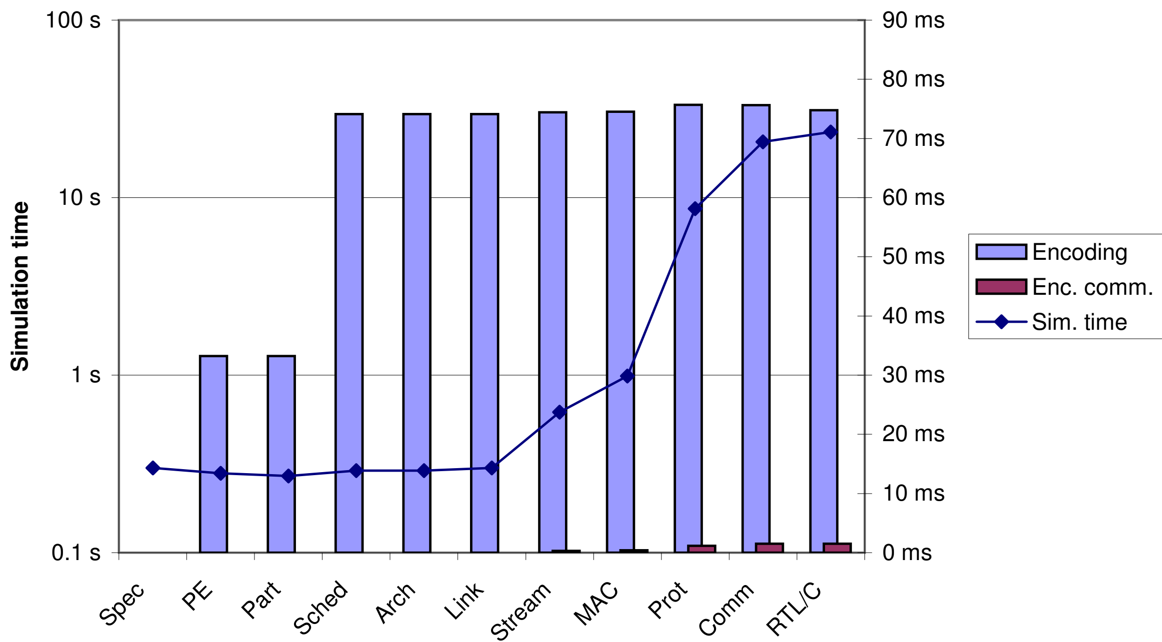


Figure 8.5: JPEG encoder simulation results.

4 shared, global variables instantiated at the top level. In the partitioned model, an additional shared memory component is inserted, top-level variables are moved into local and shared memories and synchronization behaviors and channels are refined to communicate data values. Additional concurrent synchronization behaviors and channels are inserted into the hierarchy of PE and partitioned models as necessary. In the scheduled model, pipelines and synchronization/communication behaviors have been statically scheduled. Serialization of behaviors results in an according increase in computation delays. At the architecture level, an empty OS layer is then inserted around the ColdFire processor model. Since no dynamic scheduling overhead is required and since static scheduling decreases the number of fork and join events during simulation, simulation runtimes decrease down to a level below the original specification in the scheduled and architecture models.

At the link level, adapter channels are inserted into the PEs and top-level channels are merged over three links (DMA, DCT, and bridge) and an additional bridge behavior. In the stream model, the three top-level links are split into three control and three data streams. In addition, a top-level channel for memory communication is inserted. In the media access model, the three top-level data streams and the memory channel are multiplexed over one data transfer medium. High-level interrupt handling behaviors are inserted into PEs and stream and media channels are annotated with estimated latencies, resulting in increased communication delays. In the protocol model, the top-level medium channel is replaced with two protocol channels for ColdFire and DCT bus protocols connected via one transducer component behavior. In addition, a semaphore channel for arbitration on the ColdFire bus is instantiated. For the ColdFire PE, a hardware abstraction layer is inserted and media access layer implementation channels are added to each PE. Due to data slicing and arbitration, simulated communication delays and associated simulation overhead increase drastically. Especially the detailed modeling of arbitration behavior in the protocol model results in significantly more accurate delay results at the expense of a corresponding increase in simulation runtimes.

The communication model finally refines protocol channels down to wires modeled by signal-type variables, replaces the arbitration channel with an instance of the actual arbiter component behavior, and inserts the bus-functional and hardware processor model including its interrupt controller. As expected, modeling of protocol timing and interrupt handling increases accuracy of results even further while incurring an order-of-magnitude increase in simulation overhead.

As a result of the backend process, the RTL/C model replaces the ColdFire application software behavior hierarchy with a single SpecC behavior that encapsulates the generated C code. Furthermore, DCT and transducer component models are replaced with their pre-designed, cycle-

accurate RTL descriptions taken out of the IP database. Cycle-accurate simulation of synthesized hardware results in a corresponding increase in simulation overhead while providing exact timing for hardware PEs. Results show that estimated hardware computation delays at higher levels were too conservative.

Results generally confirm observations made in the case of the Vocoder design. Compared to the Vocoder, the JPEG encoder requires less communication on its critical path. As such, overall delays are influenced by communication overhead to a much lesser extent and the output of computation design provides already accurate results.

On the other hand, the JPEG encoder overall is a more communication-intensive design than the Vocoder. Since DMA communication is overlapped with encoding computation in the processor (exploiting the available pipeline parallelism), it leaves overall delays largely unaffected while contributing to the overall communication complexity. In addition, implementation of communication inside each PE is more complex since the JPEG encoder uses more advanced communication features (e.g. bigger variety in terms of message data types exchanged between PEs compared to only a single data type in the Vocoder). Furthermore, overall computational complexity of the JPEG encoder as expressed by the complexity of the specification model is smaller than the Vocoder application complexity. Therefore, implementation details inserted during communication design as measured, for example, by the number of channels, contribute to a larger relative growth of overall model complexities and simulation runtimes when compared to the Vocoder.

Finally, comparing Vocoder and JPEG encoder backend design, the relative size of custom hardware in the JPEG encoder is smaller. Hence, hardware design in the JPEG encoder has a smaller effect on overall complexity and simulation overhead of its resulting RTL/C model.

8.3.2 PE Modeling

Similar to the PE design space exploration in the Vocoder case (Section 8.2.2), we explored several behavior partitioning design alternatives for the JPEG encoder in order to demonstrate effectiveness and usefulness of the high-level design models. At the core of the JPEG encoder, the inner loop consists of four behaviors running in a pipelined fashion: *HandleData* (HD), *DCT* (D), *Quantization* (Q), and *HuffmanEncode* (HE). For exploration of behavior partitioning, we allocated a system architecture with two PEs, a Motorola DSP56600 (SW) running at 60 MHz and a custom hardware (HW) processor running at 80.8 MHz. By mapping four behaviors to two PEs in different permutations, we derived 16 (2^4) design alternatives.

Design				Encoding delay		
HD	D	Q	HE	Impl.	Arch.	Diff.
SW	SW	SW	SW	205.00 ms	199.44 ms	2.71%
SW	SW	SW	HW	184.77 ms	177.18 ms	4.11%
SW	SW	HW	SW	189.00 ms	180.87 ms	4.30%
SW	SW	HW	HW	168.77 ms	158.61 ms	6.02%
SW	HW	SW	SW	73.35 ms	76.79 ms	-4.69%
SW	HW	SW	HW	53.12 ms	54.53 ms	-2.65%
SW	HW	HW	SW	57.35 ms	58.22 ms	-1.52%
SW	HW	HW	HW	37.12 ms	35.96 ms	3.12%
HW	SW	SW	SW	183.23 ms	176.92 ms	3.44%
HW	SW	SW	HW	163.00 ms	154.66 ms	5.12%
HW	SW	HW	SW	167.23 ms	158.35 ms	5.31%
HW	SW	HW	HW	147.00 ms	136.09 ms	7.42%
HW	HW	SW	SW	51.58 ms	54.27 ms	-5.22%
HW	HW	SW	HW	32.01 ms	29.93 ms	-2.11%
HW	HW	HW	SW	35.70 ms	38.84 ms	-0.34%
HW	HW	HW	HW	15.35 ms	13.44 ms	12.44%

Table 8.9: JPEG encoder PE exploration.

For each design alternative, we generated a SpecC architecture model using the scripting capabilities of the design environment together with the architecture refinement tool. Each architecture model was then simulated to obtain feedback about estimated encoding delays. Results were compared against delays of an actual implementation of the design alternative. Implementation delays for software were obtained by converting SpecC code to C code, compiling the C code into assembly code, and running the assembly code on a customized, annotated version of the DSP56600 instruction set simulator [14]. Implementation delays for hardware were obtained by simulating manually written RTL models.

Results of the PE exploration for the JPEG encoder are shown in Table 8.9. For each design alternative, the tables lists implementation delays and simulated delays at the architecture level. Results show that the simulated delays at the high level are accurate to within 12.5% of the actual implementation. It should be noted, however, that the JPEG encoder is not communication-intensive. Hence, it is possible to achieve a relatively high absolute accuracy at this high level. In more communication-oriented designs, communication delays which are not included at the architecture level will have a bigger impact on the overall, final implementation delays.

An important metric at high levels is the so-called *fidelity* [68, 36]. The fidelity is defined as the percentage of correctly predicted comparisons between design alternatives. If the estimated

values of a design metric for two design alternatives bear the same comparative relationship to each other as do the measured values of the metric, then the estimate correctly compares the two alternatives. Based on the results, we compute the fidelity for the JPEG encoder PE exploration by comparing implementation and simulated architecture delays. For JPEG encoder example, the fidelity of PE exploration is 100%. Therefore, results confirm that high-level models allow the designer to make educated design decisions by providing relevant feedback.

8.4 Baseband System

The overall baseband system is a combination of Vocoder and JPEG encoder subsystems. In general, the baseband system is the parallel composition of the two subsystems both at the specification and at architecture and lower levels. As such, complexities of baseband design models are generally the sum of corresponding Vocoder and JPEG encoder model complexities. Note, however, that at architecture and lower levels, the two subsystem architectures are combined at the top level into one overall architecture with ten PEs. Table 8.10 lists the results of modeling the complete baseband system at different levels of abstraction.

As design progresses, more and more implementation detail gets added to the design models and model complexities grow. Figure 8.6 compares the two subsystems and the overall system in terms of model complexity growth. The graphs plot the number of lines of code added to each design model when compared to the the complexity of the original specification model. Graphs are shown for both absolute number of added lines of code (Figure 8.6(a)) and added lines of code normalized against the number of lines of code in the respective specification model (Figure 8.6(b)).

Modeling results show that implementation detail added to the models generally grows linearly with lower levels of abstraction. Note, however, that complexity increase is largely independent of the complexity of the original specification. Rather, added implementation detail depends mainly on the complexity of the selected target architecture as measured by the number of PEs and busses, for example. Since Vocoder and JPEG encoder target architectures have similar complexity, their model complexities increase by approximately the same absolute amount. In the baseband system as the combination of both subsystems, model complexity increase is then the sum of complexity increases in each subsystem.

Figure 8.7 and Figure 8.8 summarize the simulation results for the two subsystems and the overall system in terms of simulation runtimes and simulated delays, respectively. For simulation runtimes, both absolute times (Figure 8.7(a)) and runtimes normalized against the simulation run-

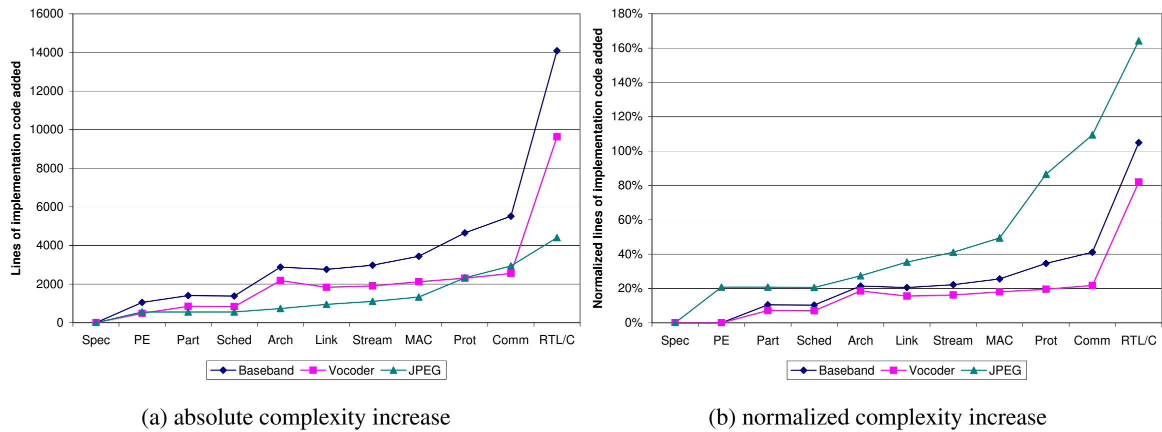
Model	Lines of code	Behaviors Total	Leaf	Parallel	Chnl Total	Max. depth	Top-level Beh	Chnl	Var
Specification	13,428	161	111 (68.9%)	16 (9.7%)	6	10	2	1	0
PE	14,476	193	124 (64.2%)	18 (9.3%)	6	11	9	15	15
Partitioned	14,833	193	122 (63.2%)	20 (10.4%)	6	11	10	25	0
Scheduled	14,810	193	122 (63.2%)	3 (1.6%)	6	12	10	25	0
Architecture	16,307	195	122 (62.6%)	3 (1.5%)	15	12	10	25	0
Link	16,193	199	127 (63.8%)	3 (1.5%)	30	12	11	9	0
Stream	16,403	199	127 (63.8%)	3 (1.5%)	31	12	11	19	0
Media Access	16,866	201	129 (64.2%)	3 (1.5%)	45	12	11	11	0
Protocol	18,078	208	134 (64.4%)	3 (1.4%)	56	13	14	9	0
Communication	18,946	216	137 (63.4%)	7 (3.2%)	59	15	13	0	34
RTL / C	27,511	73	38 (52.1%)	7 (9.6%)	59	9	13	0	34

Table 8.10: Baseband system modeling results.

time of the respective specification model (Figure 8.7(b)) are plotted. For simulated transcoding and JPEG encoding delays, overall delays (Figure 8.8(a)) and communication overheads (Figure 8.8(b)) are shown, both normalized against the corresponding delay in the RTL/C model.

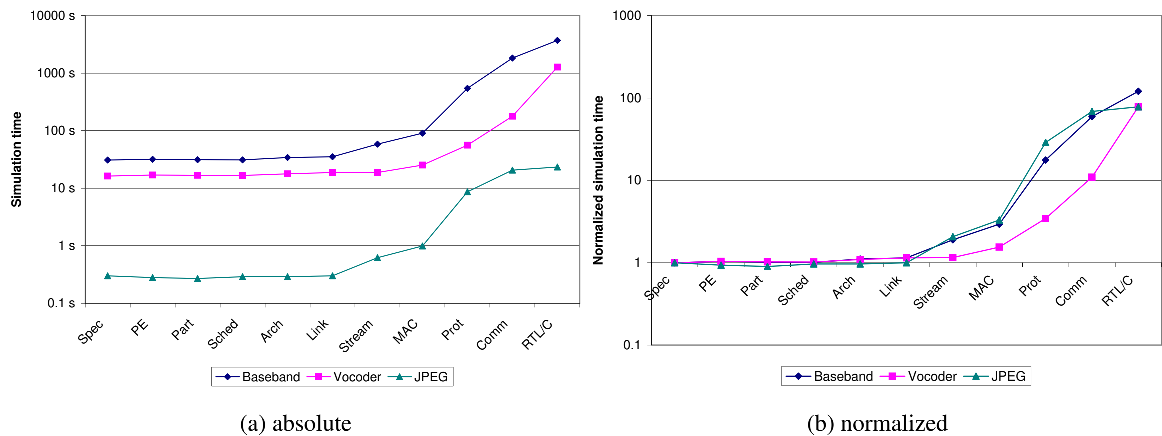
In general, simulation overhead grows exponentially with lower levels of abstraction as more and more implementation detail gets added. Especially during communication and backend design, simulation overhead increases dramatically. Simulation overhead added during computation design, on the other hand, is negligible. As mentioned previously, due to shared memory and DMA communication the JPEG encoder is a more communication-intensive design compared to the Vocoder. Therefore, overhead added during communication design in the JPEG encoder has a bigger impact on simulation runtimes than in the Vocoder. On the other hand, the hardware part in the JPEG encoder is smaller than in the Vocoder. As a result, simulation overhead for cycle-accurate RTL simulation is comparatively less. Note that for the overall baseband system, absolute simulation runtimes are not equal to the sum of Vocoder and JPEG encoder runtimes. Rather, results show that combining subsystems in such a way that their simulated events overlap in logical time leads to an exponential increase in simulation runtimes for the whole system.

Along with model complexities, model accuracy increases in a linear fashion as more implementation is added with each new abstraction level. Simulated transcoding and JPEG encoding delays show that the architecture model at the output of computation design can deliver almost 100% accurate results. Depending on the amount of parallelism in each PE, scheduling is required to provide accurate results. In other cases, the partitioned model can already return useful feedback.



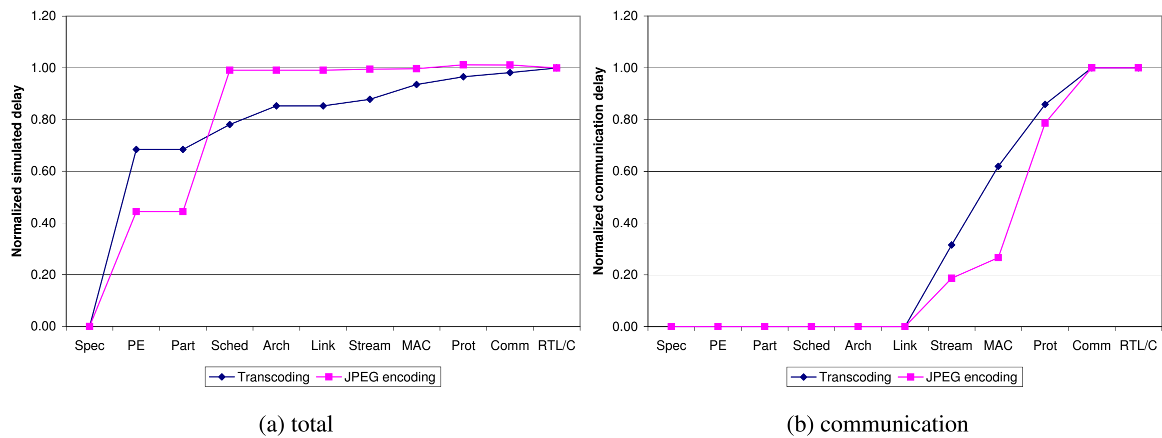
(a) absolute complexity increase (b) normalized complexity increase

Figure 8.6: Baseband implementation detail added during model refinement .



(a) absolute (b) normalized

Figure 8.7: Baseband simulation times.



(a) total (b) communication

Figure 8.8: Baseband simulated delays.

Accuracy of the architecture level generally depends on the amount of non-overlapping communication in the design.

Compared to the JPEG encoder, the Vocoder requires more communication on the critical path. Therefore, delay estimates after computation design are less accurate. On the other hand, since there is only a single bus master and no bus contention in the Vocoder design, its media access model provides already relatively accurate results for communication and hence overall delays. Protocol and communication models add further delays for slave interrupt polling and actual suspension of computation, respectively[4]. If, however, arbitration for resolution of bus contention is required as is the case for the JPEG encoder, the media access model does not provide accurate feedback and only detailed modeling of arbitration and slicing in the protocol model results in precise measurements.

In summary, results show that model complexities and model accuracies increase linearly as design moves down in the level of abstraction. However, simulation runtimes grow exponentially with lower levels of abstraction. With each step in the design flow, more implementation information is added in the form of additional code and additional behaviors and channels. Additional implementation information increases the accuracy of simulation results with each new model down to the final RTL/C model. Specifically, as communication gets more and more refined, additional channels are inserted into the design and simulated delays include additional overhead for communication and synchronization. On the other hand, additional implementation information dramatically increases simulation overhead as the number of events simulated during execution of the system grows exponentially with each step.

Overall, experimental results for the baseband system including its two subsystems confirm that moving to higher levels of abstraction enables more rapid design space exploration while being able to provide accurate results. Depending on the application and design characteristics, intermediate high-level models can provide early feedback about critical design decisions where each model focuses on a different, separate design aspect.

At the specification level, functionality of the application is validated. PE and partitioned design models provide the same accuracy at the same speed. However, only the partitioned model provides additional feedback about actual organization, sizes, and utilization of memories. Depending on the amount of required scheduling, partitioned or architecture models allow evaluation of different computation architectures with little or no additional overhead. If extensive static or dynamic scheduling is required, the architecture model together with any RTOS models provides

[4]If neither arbitration nor slave polling is required, the media access model will provide even more accurate results.

feedback for exploration of different scheduling strategies. Therefore, the architecture model as a representation of the overall computation architecture allows reliable exploration of the computation design space at simulation runtimes comparable to the abstract C-level specification.

With little or no additional overhead, the link model defines and validates the overall network topology while serving as the specification for implementation of individual bus segments. The stream model introduces communication delays in the form of estimated media latencies but is not accurate enough for reliable exploration. Depending on the communication architecture, media access or protocol transaction-level models provide rapid, accurate feedback about overall results including communication overhead. If there is no bus contention and slave polling in the system, the media access model returns exact results at high simulation speeds. On the other hand, in the presence of arbitration, slicing of packets into protocol words/frames has to be modeled in order to get accurate delays that include effects of interleaved media accesses at the protocol level[5]. In these cases, only the protocol model can provide correct estimates at the expense of significantly reduced simulation speed.

Finally, at the communication level, final, pin- and timing-accurate results are available at the expense of vastly increased simulation runtimes compared to transaction-level models above. Delay results in the communication model only depend on the accuracy of software and hardware estimations results for all PEs. Once cycle-accurate descriptions for hardware and/or software are included as part of RTL/C or implementation models, precise results can be obtained. However, simulation performance at the RTL/C and implementation levels is prohibitive for exhaustive simulations of complete system designs.

8.5 Summary

In this chapter, we presented the results of implementing several industrial-strength system design examples following the steps of the developed design methodology. The design flow has been applied to the example of a mobile phone baseband platform consisting of separate vocoder and JPEG encoder subsystems. The results show the tradeoffs in terms of complexity, overhead, and accuracy of system design models as design moves down in the level of abstraction. In general, with increasing implementation detail at lower levels of abstraction, accuracy gradually improves while model complexities grow exponentially. Thereby, results provide insights into the relationship between critical design issues and the represented implementation detail at each abstraction level.

[5] Similar to the effects of scheduling of computation on PEs.

The results presented in this chapter results confirm the choice of intermediate models for system design and design space exploration. The specification model is the starting point of the design process and defines the desired functionality at native C simulation performance. The architecture model allows evaluation of different computation architectures with negligible overhead. Depending on the communication architecture, media access or protocol models enable rapid, reliable communication architecture exploration. The communication model then provides pin- and timing-accurate results for further investigation of actual communication implementations. Finally, implementation models and/or mixed-level communication/implementation models are necessary for cycle-accurate validation and sign-off.

In addition to the analysis of model tradeoffs and benefits, contributions of this chapter include a demonstration of the effectiveness of the proposed design methodology using the design examples. The examples verify that the design flow supports a broad range of target implementations as used in real SoC designs. Therefore, with the help of the automated tools in the SoC design environment (see Chapter 7), the methodology can achieve the required productivity gains while being able to create designs of comparable quality. Furthermore, rapid design space exploration of large numbers of design alternatives allows for global optimization of overall system design parameters with potentially significant improvements in design quality compared to locally optimal solutions.

Chapter 9

Summary and Conclusions

In this dissertation, we presented a well-defined flow of design steps and design models for embedded system and system-on-chip (SoC) design. To our knowledge, this is the first approach to provide a rigorous structuring of the system design process for the design of complete systems from abstract specification down to a cycle-accurate implementation. System design thus far has been mostly done in an ad-hoc fashion based on the experience of designers. Design decisions are made in no particular order and without any systematic organization. However, in order to achieve the required productivity gains, system design has to be structured such that all important design issues are addressed in the proper order.

We identified, classified and organized implementation decisions required for such system design into a set of successive design steps based on grouping of highly related decisions, breaking of the design gap into independent, manageable steps and ordering of steps according to dependencies. Models of the design after each step have been developed that accurately represent the corresponding implementation decisions while abstracting unnecessary or unknown implementation detail. The resulting sequence of design steps and design models enables rapid and reliable exploration of critical design aspects at early stages in the design process. Furthermore, design decisions and model transformations for each design step have been defined such that model refinement and decision making can be automated while supporting a wide range of realistic applications and target implementations.

Based on a separation of concerns and orthogonality of concepts, system design is split into computation, communication and backend design. Within computation and communication design, the component architecture is defined in a first task before the order of events on each inherently sequential structural components can be determined. Computation design therefore consists

of partitioning and scheduling tasks. Behavior and variable partitioning define the number and type of processing elements (PEs) and memories in the computation architecture and the mapping of behaviors and variables onto PEs and memories. Since behavior partitioning determines the set of global variables, it is performed before variable partitioning. On the other hand, component allocation and mapping are highly dependent and are performed together in a single step each. Scheduling during computation design consists of static and dynamic scheduling steps which define the order of behavior execution on the sequential PEs in a pre-defined, fixed manner or dynamically at runtime under the control of a chosen scheduling algorithm, respectively. In the case of dynamic scheduling, an abstract OS model which accurately represents and reflects dynamic scheduling behavior without the overhead of a complete operating system is introduced. The result of partitioning and scheduling is the architecture model of the design. The architecture model provides accurate feedback about the computation architecture critical to most designs. On the other hand, it introduces only a negligible overhead, enabling rapid architecture exploration.

Communication design consists of network and link design tasks. Network design defines the overall network topology and the implementation of abstract channels over the network. In a first step, channels are converted and merged into untyped byte streams. Then, streams are broken down into packets and routed over links connecting PEs and additional transducers. After the communication architecture has been defined, link design determines the implementation of groups of links within each network segment over a selected medium and corresponding media interfaces for each station connected to the network. First, interface types, addresses and interrupts for each link are defined and packet transfers are implemented over media transactions. Then, connection of arbiters and interrupt controllers to busses and processors is defined and pin- and timing accurate protocol implementations are inserted. Intermediate transaction-level models in between steps of the link design task enable rapid communication exploration while providing accurate feedback. On the other hand, since arbitration depends on the slicing of data packets into bus words/frames, the link design task is broken into individual design steps and steps are ordered accordingly. The result of communication design is the communication model which provides an accurate description of the system communication structure and timing.

Backend design consists of separate, independent hardware and software design tasks. During hardware design, behavioral code is converted into state machine descriptions and state machines are implemented on RTL processors with datapaths consisting of functional units, memories, register files and busses through scheduling and binding. During software design, C code for application software and network protocol stacks is generated, code is customized and targeted

for implementation of application tasks and bus drivers on a selected RTOS, and resulting code is compiled and linked against target processor and RTOS libraries. The result of backend design is the implementation model which provides a cycle-accurate description of the whole system. On the other hand, by exchanging components between implementation and communication models through simple plug-and-play, mixed-level models can be easily created.

Based on a separation of synthesis into decision making and model refinement, the design flow has been implemented in the form of the system-on-chip design environment (SCE). As part of this work, we defined architecture, organization, tool flow, design model management, interfaces, and databases of an environment that enables integration and interoperability of various tools for automation of the design process under a common framework. In addition, a graphical user interface (GUI) has been developed that aids and steers the designer in the decision making process, provides visualization of design model characteristics and metrics and supports interactive, graphical decision entry for each design step. Overall, the design environment shows the feasibility and benefits of a seamless interactive, automated design flow based on the proposed design steps and models. The environment proves that the flow enables automation of tedious, error-prone tasks of model rewriting. Models in the flow have been defined such that they can be automatically generated and corresponding model refinement tools have been integrated into the environment. On the other hand, design steps and design models support an interactive flow that effectively draws from human knowledge and insight by keeping the designer in the loop at all times. Intermediate models provide a direct representation of implementation decisions for immediate transparency of results and observability of design decisions. In addition, well-defined design decisions enable decision entry under control of the designer assisted by the design environment. Furthermore, the user can employ automated decision-making algorithms selectively on parts of the design at any stage of the design process. However, at any time decisions can be fully or partially overwritten, replaced or predefined, resulting in the necessary controllability of the process.

The design flow has been applied to the design of several industrial-size system examples. Experimental results show the tradeoffs and benefits of intermediate design models at high levels of abstraction. Results confirm the choice of design steps and intermediate design models for rapid, early design space exploration. Furthermore, they prove that critical design issues can be made quickly and reliably at early stages of the design flow. Finally, experiments show that designs of comparable quality can be created for realistic applications and target architectures. With the help of automated tools integrated into the design environment, designs of significant complexity can be completed and optimized within days. Therefore, a design flow based on the steps and

models presented in this dissertation can achieve the required productivity gains for the design and exploration of complete systems from specification down to implementation.

Bibliography

[1] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel Gajski. System-on-chip environment (SCE version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-41, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[2] Samar Abdi, Dongwan Shin, and Daniel D. Gajski. Automatic communication refinement for system level design. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, CA, June 2003.

[3] Accellera, C/C++ Class Library Standardization Working Group. *RTL Semantics*, February 2001. Draft Specification, Version 0.8.

[4] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1997.

[5] David Berner, Dirk Jansen, and Daniel D. Gajski. Development of a visual refinement and exploration tool for SpecC. Technical Report ICS-TR-01-12, Information and Computer Science, University of California, Irvine, March 2001.

[6] Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, second edition, 1997.

[8] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 3*. Prentice Hall, February 2004.

[9] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4(2):155–182, April 1994.

[10] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1999.

[11] Lucai Cai. *Estimation and Exploration Automation of System-Level Design*. PhD thesis, Information and Computer Science, University of California, Irvine, January 2004.

[12] Lucai Cai, Andreas Gerstlauer, and Daniel D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. In *Proceedings of the Design Automation Conference (DAC)*, San Diego, CA, June 2004.

[13] Lucai Cai, Andreas Gerstlauer, and Daniel D. Gajski. Retargetable profiling for rapid, early system-level design space exploration. Technical Report CECS-TR-04-04, Center for Embedded Computer Systems, University of California, Irvine, March 2004.

[14] Lucai Cai, Junyu Peng, Chun Chang, Andreas Gerstlauer, Hongxing Li, Anand Selka, Chuck Siska, Lingling Sun, Shuqing Zhao, and Daniel D. Gajski. Design of a JPEG encoding system. Technical Report ICS-TR-99-54, Information and Computer Science, University of California, Irvine, November 1999.

[15] Marco Caldari, Massimo Conti, Marcello Coppola, Stephane Curaba, Lorenzo Pieralisi, and Claudio Turchetti. Transaction-level models for AMBA bus architecture using SystemC 2.0. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[16] Francky Catthoor, Sven Wuytack, Eddy De Greef, Florin Balasa, Lode Nachtergaele, and Arnout Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, 1998.

[17] Wander O. Cesário, Damien Lyonnard, Gabriela Nicolescu, Yanick Paviot, Sungjoo Yoo, Ahmed A. Jerraya, Lovic Gauthier, and Mario Diaz-Nava. Multiprocessor SoC platforms: A component-based design approach. *IEEE Design and Test of Computers*, 19(6), November/December 2002.

[18] Marcello Coppola, Stephane Curaba, Miltos Grammatikakis, and Giuseppe Maruccia. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[19] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, Marc Massot, Sandra Moral, Claudio Passerone, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Task generation and compile time scheduling for mixed data-control embedded software. In *Proceedings of the Design Automation Conference (DAC)*, Los Angeles, CA, June 2000.

[20] Pascal Coste, F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and Ahmed A. Jerraya. Multilanguage design of heterogeneous systems. In *Proceedings of the International Symposium on Hardware-Software Codesign (CODES)*, Rome, Italy, May 1999.

[21] Ali Dasdan, Dinesh Ramanathan, and Rajesh K. Gupta. A timing-driven design and validation methodology for embedded real-time systems. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):533–553, October 1998.

[22] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. COSYN: Hardware-software co-synthesis of embedded systems. In *Proceedings of the Design Automation Conference (DAC)*, Anaheim, CA, June 1997.

[23] Dirk Desmet, Diederick Verkest, and Hugo De Man. Operating system based software generation for system-on-chip. In *Proceedings of the Design Automation Conference (DAC)*, Los Angeles, CA, June 2000.

[24] Rainer Dömer. The SpecC internal representation. Technical report, Information and Computer Science, University of California, Irvine, January 1999. SpecC V 2.0.3.

[25] Rainer Dömer. *System-Level Modeling and Design with the SpecC Language*. PhD thesis, University of Dortmund, Germany, April 2000.

[26] Rainer Dömer. SpecC reference compiler SCRC 1.1, software architecture and implementation. Technical report, Information and Computer Science, University of California, Irvine, September 2001.

[27] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, http://www.specc.org, December 2002.

[28] Petru Eles, Krzysztof Kuchcinski, and Zebo Peng. *System Synthesis with VHDL*. Kluwer Academic Publishers, December 1997.

[29] Rolf Ernst, Jörg Henkel, and Thomas Brenner. Hardware-software co-synthesis for micro-controllers. *IEEE Design and Test of Computers*, 10(4):64–75, December 1993.

[30] European Telecommunication Standards Institute (ETSI). *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding*, final draft edition, 1996. GSM 06.60.

[31] Python Software Foundation. Python Programming Language. http://www.python.org.

[32] Daniel Gajski, Junyu Peng, Andreas Gerstlauer, Haobo Yu, and Dongwan Shin. System design methodology and tools. Technical Report CECS-TR-03-02, Center for Embedded Computer Systems, University of California, Irvine, January 2003.

[33] Daniel D. Gajski. *Principles of Digital Design*. Prentice Hall, 1997.

[34] Daniel D. Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[35] Daniel D. Gajski and R. Kuhn. Guest editors' introduction: New VLSI tools. *IEEE Computer*, 16(12):11–14, 1983.

[36] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Ji Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[37] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. The SpecC methodology. Technical Report ICS-TR-99-56, Information and Computer Science, University of California, Irvine, December 1999.

[38] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[39] Lovic Gauthier and Sungjoo Yo Ahmed A. Jerraya. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 20(11), November 2001.

[40] Patrice Gerin, Sungjoo Yoo, Gabriela Nicolescu, and Ahmed A. Jerraya. Scalable and flexible cosimulation of SoC designs with heterogeneous multi-processor target architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2001.

[41] Andreas Gerstlauer. SpecC modeling guidelines. Technical Report CECS-TR-02-16, Center for Embedded Computer Systems, University of California, Irvine, April 2002.

[42] Andreas Gerstlauer. Communication abstractions for system-level design and synthesis. Technical Report CECS-TR-03-30, Center for Embedded Computer Systems, University of California, Irvine, October 2003.

[43] Andreas Gerstlauer, Lukai Cai, Dongwan Shin, Haobo Yu, Junyu Peng, and Rainer Dömer. *SCE Database Reference Manual, Version 2.2.0 beta*. Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[44] Andreas Gerstlauer, Lukai Cai, Dongwan Shin, Haobo Yu, Junyu Peng, and Rainer Dömer. System-on-chip component models. Technical Report CECS-TR-03-26, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[45] Andreas Gerstlauer and Rainer Dömer. *SCE Specification Model Reference Manual, Version 2.2.0 beta*. Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[46] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[47] Andreas Gerstlauer and Daniel D. Gajski. System-level abstraction semantics. In *Proceedings of the International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[48] Andreas Gerstlauer, Kiran Ramineni, Rainer Dömer, and Daniel D. Gajski. System-on-chip specification style guide. Technical Report CECS-TR-03-21, Center for Embedded Computer Systems, University of California, Irvine, June 2003.

[49] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. RTOS modeling for system level design. In Ahmed A. Jerraya, Sungjoo Yoo, Norbert Wehn, and Diedrik Verkest, editors, *Embedded Software for SoC*. Kluwer Academic Publishers, 2003.

[50] Andreas Gerstlauer, Haobo Yu, and Daniel D. Gajski. Rtos modeling for system level design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[51] Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski, and Arkady M. Horak. Design of a GSM vocoder using SpecC methodology. Technical Report ICS-TR-99-11, Information and Computer Science, University of California, Irvine, March 1999.

[52] Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski, and Arkady M. Horak. SpecC system-level design methodology applied to the design of a GSM vocoder. In *Proceedings of the Workshop of Synthesis and System Integration of Mixed Information Technologies*, Kyoto, Japan, April 2000.

[53] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[54] Peter Grun, Nikil Dutt, and Alex Nicolau. *Memory Architecture Exploration for Programmable Embedded Systems*. Kluwer Academic Publishers, 2003.

[55] Rajesh K. Gupta and Giovanni De Michelli. Hardware-software co-synthesis for digital systems. *IEEE Design and Test of Computers*, pages 29–41, September 1993.

[56] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14(2):72–80, April-June 1997.

[57] Sumit Gupta. *Coordinated Coarse-Grain and Fine-Grain Optimizations for High-Level Synthesis*. PhD thesis, Information and Computer Science, University of California, Irvine, June 2003.

[58] Wolfram Hardt, Achim Rettberg, and Bernd Kleinjohann. The PARADISE design environment. In *Proceedings of the Embedded System Conference (ESC)*, Auckland, New Zealand, 1999.

[59] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[60] SEMATECH Inc. International technology roadmap for semiconductors (ITRS). http://public.itrs.net, 2001.

[61] Trolltech Inc. Qt application development framework. http://www.trolltech.com/products/qt/.

[62] Internation Organization for Standardization (ISO). *Reference Model of Open System Interconnection (OSI)*, second edition, 1994. ISO/IEC 7498 Standard.

[63] International Telecommunication Union (ITU). *Digital Compression and Coding of Continous-Tone Still Images*, September 1992. ITU Recommendation T.81.

[64] International Telecommunication Union (ITU). *Specification and Description Language (SDL)*, November 1999. ITU-T Recommendation Z.100.

[65] Axel Jantsch, Shashi Kumar, and Ahmed Hemani. The Rugby model: A conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 1999.

[66] Ahmed A. Jerraya and Kevin O'Brien. SOLAR: An intermediate format for systemlevel modelling and synthesis. In Jerzy Rozenblit and Klaus Buchenrieder, editors, *Computer Aided Software/Hardware Engineering*. IEEE Press, 1994.

[67] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Applications*. Kluwer Academic Publishers, 1997.

[68] Fadi J. Kurdahi, Daniel D. Gajski, Champaka Ramachandran, and Viraphol Chaiyakul. Linking register transfer and physical levels of design. *IEICE Transactions on Information and Systems*, 76(9):991–1002, September 1993.

[69] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), January 1987.

[70] Heiko Lehr and Daniel D. Gajski. Modeling custom hardware in VHDL. Technical Report ICS-TR-99-29, Information and Computer Science, University of California, Irvine, July 1999.

[71] C. L. Liu and James W. Leyland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[72] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[73] Riverbank Computing Ltd. PyQt. http://www.riverbankcomputing.co.uk/pyqt/.

[74] Damien Lyonnard, Sungjoo Yoo, Amer Baghdadi, and Ahmed A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the Design Automation Conference (DAC)*, Las Vegas, NV, June 2001.

[75] Peter Marwedel. *Embedded Systems Design*. Kluwer Academic Publishers, 2003.

[76] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[77] Wolfgang Mueller, Rainer Dömer, and Andreas Gerstlauer. The formal execution semantics of SpecC. In *Proceedings of the International Symposium on System Synthesis*, Kyoto, Japan, October 2002.

[78] Open SystemC Initiative, http://www.systemc.org. *Functional Specification for SystemC 2.0*, 2000.

[79] Open SystemC Initiative. http://www.systemc.org.

[80] Achim Österling, Thomas Brenner, Rolf Ernst, Dirk Herrmann, Thomas Scholz, and Wei Ye. The COSYMA system. In Jorgen Staunstrup and Wayne Wolf, editors, *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.

[81] Preeti R. Panda, Nikil D. Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.

[82] Junyu Peng, Samar Abdi, and Daniel D. Gajski. Automatic model refinement for fast architecture exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Bangalore, India, January 2002.

[83] Junyu Peng, Lucai Cai, Andreas Gerstlauer, and Daniel Gajski. Interactive system design flow. Technical Report CECS-TR-02-15, Center for Embedded Computer Systems, University of California, Irvine, April 2002.

[84] Shiv Prakash and Alice Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16:338–351, December 1992.

[85] Boudewijn Rempt. *GUI Programming with Python: Qt Edition*. Opendocs Llc., January 2002.

[86] Kai Richter, Dirk Ziegenbein, Marek Jersak, and Rolf Ernst. Model composition for scheduling analysis in platform design. In *Proceedings of the Design Automation Conference (DAC)*, New Orleans, LA, June 2002.

[87] Patrick Schaumont, Serge Vernalde, Luc Rijnders, Marc Engels, and Ivo Bolsens. A programming environment for the design of complex high speed ASICs. In *Proceedings of the Design Automation Conference (DAC)*, San Francisco, CA, June 1998.

[88] Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. C-based interactive RTL design environment. Technical Report CECS-TR-03-42, Center for Embedded Computer Systems, University of California, Irvine, December 2003.

[89] Robert Siegmund and Dietmar Müller. SystemC^{SV}: An extension of SystemC for mixed multi-level communication modeling and interface-based system design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2001.

[90] David Stepner, Nagarajan Rajan, and David Hui. Embedded application design using a real-time OS. In *Proceedings of the Design Automation Conference (DAC)*, New Orleans, LA, June 1999.

[91] SpecC Technology Open Consortium. http://www.specc.org.

[92] Kjetil Svarstad, Nezih Ben-Fredj, Gabriela Nicolescu, and Ahmed A. Jerraya. A higher level system communication model for object-oriented specification and design of embedded systems. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2001.

[93] Hiroyuki Tomiyama, Yun Cao, and Kazuaki Murakami. Modeling fixed-priority preemptive multi-task systems in SpecC. In *Proceedings of the Workshop of Synthesis and System Integration of Mixed Information Technologies*, Nara, Japan, October 2001.

[94] Frank Vahid, Sanjiv Narayan, and Daniel D. Gajski. SpecCharts: A VHDL frontend for embedded systems. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 14(6):694–706, June 1995.

[95] Carlos A. Valderrama, Mohamed Romdhani, Jean-Marc Daveau, Gilberto F. Marchioro, Adel Changuel, and Ahmed A. Jerraya. Cosmos: A transformational co-design tool for multiprocessor architectures. In Jorgen Staunstrup and Wayne Wolf, editors, *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.

[96] Karl van Rompaey, Diederick Verkest Ivo Bolsens, and Hugo De Man. CoWare: A design environment for heterogeneous hardware/software systems. In *Proceedings of the European Design Automation Conference (Euro-DAC)*, Geneva, Switzerland, September 1996.

[97] Guido van Rossum and Fred L. Drake, Jr. (Editor). *The Python Language Reference Manual*. Network Theory Ltd., September 2003.

[98] Geert Vanmeerbeeck, Patrick Schaumont, Serge Vernalde, Marc Engels, and Ivo Bolsens. Hardware/software partitioning for embedded systems in OCAPI-xl. In *Proceedings of the International Symposium on Hardware-Software Codesign (CODES)*, Copenhagen, Denmark, April 2001.

[99] Martin von Weymarn. Development of a specification model of the EFR vocoder. Technical Report ICS-TR-01-35, Information and Computer Science, University of California, Irvine, July 2001.

[100] Kazutoshi Wakabayashi and Takumi Okamoto. C-based SoC design flow and EDA tools: An ASIC and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 19(12):1507–1522, December 200.

[101] Wayne Wolf. An architectural co-synthesis algorithm for distributed, embedded computing systems. *IEEE Transactions on VLSI Systems*, 5(2):218–229, June 1997.

[102] Wayne Wolf. Hardware/software co-synthesis algorithms. In Ahmed A. Jerraya and Jean Mermet, editors, *System-Level Synthesis*. Kluwer Academic Publishers, 1998.

[103] Chun Wong, Paul Marchal, and Peng Yang. Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform. In *Proceedings of the International Symposium on Hardware-Software Codesign (CODES)*, Copenhagen, Denmark, April 2001.

[104] Ti-Yen Yen and Wayne Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, San Jose, CA, November 1995.

[105] Ti-Yen Yen and Wayne Wolf. Performance estimation for distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, November 1998.

[106] Hanyu Yin, Haito Du, Tzu-Chia Lee, and Daniel D. Gajski. Design of a JPEG encoder using SpecC methodology. Technical Report ICS-TR-00-23, Information and Computer Science, University of California, Irvine, July 2000.

[107] Haobo Yu, Rainer Dömer, and Daniel Gajski. Automatic software generation for system level design. Technical Report CECS-TR-03-18, Center for Embedded Computer Systems, University of California, Irvine, May 2003.

[108] Haobo Yu, Rainer Dömer, and Daniel Gajski. Embedded software generation from system level design languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2004.

[109] Haobo Yu and Daniel D. Gajski. RTOS modeling in system level synthesis. Technical Report CECS-TR-02-25, Center for Embedded Computer Systems, University of California, Irvine, August 2002.

[110] Peng Zhang, Dongwan Shin, Haobo Yu, Qiang Xie, and Daniel D. Gajski. SpecC RTL design methodology. Technical Report ICS-TR-00-44, Information and Computer Science, University of California, Irvine, December 2000.

[111] Jianwen Zhu, Rainer Dömer, and Daniel D. Gajski. Syntax and semantics of the SpecC language. In *Proceedings of the International Symposium on System Synthesis*, Osaka, Japan, December 1997.

[112] Jianwen Zhu and Daniel D. Gajski. A retargatable, ultra-fast instruction set simulator. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 1999.

[113] Jianwen Zhu and Daniel D. Gajski. Compiling SpecC for simulation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2001.

[114] Dirk Ziegenbein, Kai Richter, Rolf Ernst, Lothar Thiele, and Jürgen Teich. SPI — a system model for heterogeneously specified embedded systems. *IEEE Transactions on VLSI Systems*, 2002.