SystemC: From the Ground Up

Second Edition

David C. Black
Jack Donovan
Bill Bunton
Anna Keist

Springer

SystemC: From the Ground Up

David C. Black • Jack Donovan
Bill Bunton • Anna Keist

SystemC:
From the Ground Up

Springer

David C. Black
XtremeEDA
Austin, TX 78749
USA
dcblack@eslx.com

Jack Donovan
HighIP Design Company
Round Rock, TX 78764
USA
jack@donovanweb.org

Bill Bunton
LSI Corporation
Austin, TX 78749
USA

Anna Keist
XtremeEDA
Austin, TX 78749
USA

ISBN 978-0-387-69957-8 e-ISBN 978-0-387-69958-5
DOI 10.1007/978-0-387-69958-5
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2009933997

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

This book is dedicated to our spouses Pamela Black, Carol Donovan, Evelyn Bunton, and Rob Keist and to our children Christina, Loretta, & William Black, Chris, Karen, Jenny, & Becca Donovan, John Williams & Sylvia Headrick, Alex, Madeline, and Michael Keist

2^{nd} Edition Preface

Why the 2^{nd} Edition?

The reader (or prospective buyer of this book) might ask about the need for a second edition. The first edition was highly successful and progressed to a second and third life after being translated to Japanese and Korean.

There are three over-arching reasons for the second edition:

- A fast changing technical landscape
- Incorporation of additional topic suggestions from readers
- Fixing of errors and improvement of confusing text segments and chapters

To address the shifting technical landscape, we have significantly updated the chapters addressing Electronic System-Level design to reflect the refinements of ESL methodology thinking in the industry. Although this is not a complete discussion of ESL, it is an overview of the industry as currently viewed by the authors.

We have added a chapter on TLM, a standard that will enable interoperability of models and a model marketplace. Although this chapter discusses TLM 1.0, we think it imparts to the reader a basic understanding of TLM. Those of you who follow the industry will note that this is not TLM 2.0. This new standard was still emerging during the writing of this edition. But not to worry! Purchasers of this edition can download an additional chapter on TLM 2.0

Although SystemC is now IEEE 1666 it is not immune from the shifting technical landscape, so the authors have included material on some proposed extensions to the SystemC standard related to process control.

Readers have suggested several additional topics over the last several years and we have tried to address these with an additional chapter on the SystemC Verification (SCV) Library and an appendix on C++ fundamentals.

The chapter on the SCV library is a high level introduction and points the reader to additional resources. The authors have found that many organizations have started using the SCV library after becoming familiar with SystemC and ESL methodologies. For those readers, we have added this chapter.

The authors received several suggestions asking us to add examples and comparisons to HDL languages like Verilog and VHDL. The authors have respectfully declined, as we feel this actually impedes the reader from seeing the intended uses of SystemC. After exploring these suggestions, we have found that these readers were not entirely comfortable with C++, and because C++ is fundamental to an understanding of SystemC, this edition includes a special appendix that attempts to highlight those aspects of C++ that are important prerequisites, which is most of the language.

Writing a book of this type is very humbling, as most who have journeyed on similar endeavors will confirm. Despite our best efforts at eliminating errors from the first edition, the errata list had grown quite long. We have also received feedback that certain portions of the book were “confusing” or “not clear”. After reviewing many of these sections, we had to ask: What were we thinking? (a question asked by many developers when they revisit their “code” after several years)

In some cases we were obviously “not thinking”, so several chapters and sections of chapters have been significantly updated or completely rewritten. The topic of concurrency proved a more challenging concept to explain than the authors first thought. This edition effectively rewrote the chapters and sections dealing with the concept of concurrency.

The authors have been quite gratified at the acceptance of the first edition and the rapid adoption of SystemC. We hope we have played at least a small part in the resulting changes to our community. We wish you good luck with SystemC and your contributions to our community.

Jack Donovan, David Black, Bill Bunton, Anne Keist
4authors@scftgu.com

Preface

Jack Donovan, David Black, Bill Bunton, and Anne Keist

Why This Book

The first question any reader should ask is “Why this book?” We decided to write this book after learning SystemC using minimal documentation to help us through the quest to deeper understanding of SystemC. After teaching several SystemC classes, we were even more convinced that an introductory book focused on the SystemC language was needed. We decided to contribute such a book.

This book is about SystemC. It focuses on enabling the reader to master the language. The reader will be introduced to the syntax and structure of the language, and the reader will also learn a few techniques for use of SystemC as a tool to shorten the development cycle of large system designs.

We allude to system design techniques and methods by way of examples throughout the book. Several books that discuss system-level design methodology are available, and we believe that SystemC is ideally suited to implement many of these methods. After reading this resource, the reader should not only be adept at using SystemC constructs, but also should have an appreciation of how the constructs can be used to create high performance simulation models.

We believe there is enough necessary information about SystemC to learn the language that a stand-alone book is justified. We hope you agree. We also believe that there is enough material for a second book that focuses on using SystemC to implement these system-level design methods. With reader encouragement, the authors have started on a second book that delves deeper into the application of the language.

Prerequisites for This Book

As with every technical book, the authors must write the content assuming a basic level of understanding; this assumption avoids repeating most of an engineering undergraduate curriculum. For this book, we assumed that the reader has a working knowledge of C++ and minimal knowledge of hardware design.

For C++ skills, we do not assume that the reader is a “wizard”. Instead, we assumed that you have a good knowledge of the syntax, the object-oriented features, and the methods of using C++. The authors have found that this level of C++ knowledge is universal to current or recent graduates with a computer science or engineering degree from a four-year university.

Interestingly, the authors have also found that this level of knowledge is lacking for most ASIC designers with 10 or more years of experience. For those readers, assimilating this content will be quite a challenge but not an impossible one.

As an aid to understanding the C++ basics, this edition includes an appendix on C++. Those who have been exposed to C++ in the past are encouraged to quickly review this appendix. For a few novices, this appendix may also work as a quick introduction to the topics, but it is unlikely to be completely sufficient.

For readers who are C++ novices or for those who may be rusty, we recommend finding a good C++ class at a community college, taking advantage of many of the online tutorials, or finding a good consulting group offering an Intro to C++ class. For a list of sources, see Appendix A. We find (from our own experience) that those who have learned several procedural languages (like FORTRAN or PL/I) greatly underestimate the difficulty of learning a modern object-oriented language.

To understand the examples completely, the reader will need minimal understanding of digital electronics.

Book Conventions

Throughout this book, we include many syntax and code examples. We've chosen to use an example-based approach because most engineers have an easier time understanding examples rather than strict Backus-Naur form[1] (BNF) syntax or precise library declarations. Syntax examples illustrate the code in the manner it may be seen in real use with placeholders for user-specified items. For more complete library information, we refer the reader to the SystemC Language Reference IEEE 1666-2005, which you can obtain for free via www.systemc.org.

Code will appear in monotype Courier font. Keywords for both C/C++ and SystemC will be shown in Courier **bold**. User-selectable syntax items are in *italics* for emphasis. Repeated items may be indicated with an ellipsis (…) or subscripts. The following is an example:

```
wait(name.posedge_event()|eventi…);
if (name.posedge()) { //previous delta-cycle
  //ACTIONS…
```

Fig. 1 Example of sample code

[1]John Backus and Peter Naur first introduced BNF as a formal notation to describe the syntax of a given language. Naur, P. (1960, May). Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, *3*(5), 299-314.

When referring to a class within the text it will appear as **class_name** or **sc_class_name**. When referring to a templated class within the text, it will appear as **template_class_name<*T*>**. When referring to a member function or method from the text it will appear as **member_name(***args***)** or **sc_member_name**(*args*). Occasionally, we will refer to a member function or method without the arguments. It will appear in the text as **member_name**() or **sc_member_name**().

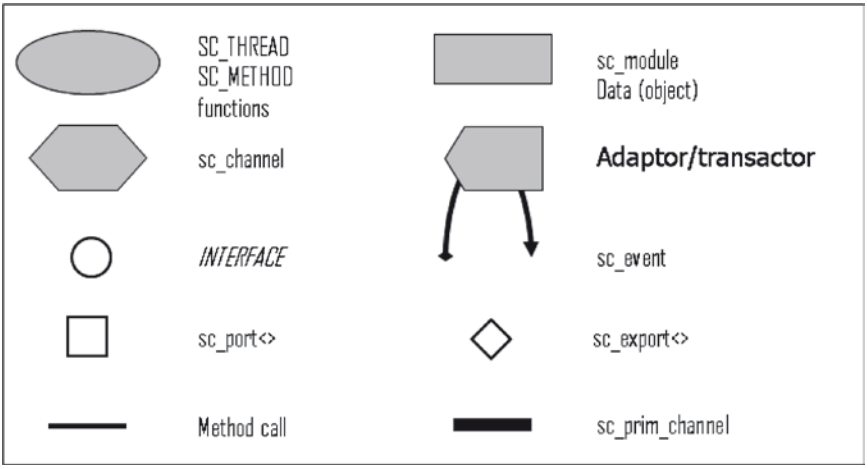

Fig. 2 Standard graphical notations

In addition, we have adopted standard graphical notations as shown in Fig 2. The terminology will be presented as the book progresses. Readers of the first edition will note that we changed the depiction of an **sc_export** from a dotted circle to a diamond. This change was the result of comments that the dotted circle was too hard to make out in some cases. We also removed arrows since in most cases, the meaning is not always clear[2].

SystemC uses a naming convention where most SystemC-specific identifiers are prefixed with **sc_** or **SC_**. This convention is reserved for the SystemC library, and you should not use it in end-user code (your code).

About the Examples

To introduce the syntax of SystemC and demonstrate its usage, we have filled this book with examples. Most examples are not real-world examples. Real examples become too cluttered too fast. The goal of these examples is to communicate

[2]Does an arrow convey calling direction (i.e., C++ function call) or direction of data flow? Since many interfaces contain a mixture of calls, some input and some output, showing data flow direction is not very useful.

concepts clearly; we hope that the reader can extend them into the real world. For the most part, we used a common theme of an automobile for the examples.

By convention, we show syntax examples stylistically as if SystemC is a special language, which it is not. We hope that this presentation style will help you apply SystemC on your first coding efforts. If you are looking for the C++ declarations, please browse the Language Reference Manual (LRM) or look directly into the SystemC Open SystemC Initiative reference source code (www.systemc.org).

It should also be noted that due to space limitations and to reduce clutter, we have omitted showing standard includes (i.e., **#include**) and standard namespace prefixes in most of the examples. You may assume something such as the following is implied in most of the examples:

```
#include <iostream>
#include <systemc>
#include <scv.h>
using namespace std;
using namespace sc_core;
using namespace sc_dt;
```

Fig. 3 Assumed code in examples

Please note that it is considered bad C++ form to include the standard namespace in header files (i.e., do not include "**using namespace** std;" in a header). We believe making the examples clear and brief warrants ignoring this common wisdom.

How to Use This Book

The authors designed this book primarily for the student or engineer new to SystemC. This book's structure is best appreciated by reading sequentially from beginning to end. A reader already familiar with SystemC will find this content to be a great reference.

Chapters 1 through 3 provide introductions and overviews of the language and its usage. Methodology is briefly discussed.

For the student or engineer new to SystemC, the authors present the language from the bottom up and explain the topics from a context of C++ with ties to hardware concepts. We provide exercises at the end of Chapters 4 through 16 to reinforce the concepts presented in the text. Chapters 16 through 18 strengthen the basic language concepts. In these chapters, readers will find discussions of areas to watch and understand when designing, writing, or using SystemC in a production environment.

For the student or engineer already familiar with SystemC, Chapters 4 through 13 will provide some interesting background and insights into the language. You can either skip or read these early chapters lightly to pick up more nuances of the language. The content here is not meant to be a complete description of the language.

For a thorough description, the reader is referred to the SystemC LRM. Chapters 14 through 18 provide intermediate to advanced material.

For the instructor, this book may provide part of an advanced undergraduate class on simulation or augment a class on systems design.

In most of the examples presented in the book, the authors show code fragments only so as to emphasize the points being made. Examples are designed to illustrate specific concepts, and as such are toy examples to simplify learning. Complete source code for all examples and exercises is available for download from www.scftgu.com as a compressed archive. You will need this book to make best use of these files.

SystemC Background

SystemC is a system design language based on C++. As with most design languages, SystemC has evolved. Many times a brief overview of the history of language will help answer the question "Why do it that way?" We include a brief history of SystemC and the Open SystemC Initiative to help answer these questions.

The Evolution of SystemC

SystemC is the result of the evolution of many concepts in the research and commercial EDA communities. Many research groups and EDA companies have contributed to the language. A timeline of SystemC is included in Table 1.

SystemC started out as a very restrictive cycle-based simulator and "yet another" RTL language. The language has evolved (and is evolving) to a true system design language that includes both software and hardware concepts. Although SystemC

Table 1 Timeline of development of SystemC

Date	Version	Notes
Sept 1999	0.9	First version; cycle-based
Feb 2000	0.91	Bug fixes
Mar2000	1.0	Widely accessed major release
Oct 2000	1.0.1	Bug fixes
Feb 2001	1.2	Various improvements
Aug 2002	2.0	Add channels & events; cleaner syntax
Apr 2002	2.0.1	Bug fixes; widely used
June 2003	2.0.1	LRM in review
Spring 2004	2.1	LRM submitted for IEEE standard
Dec 2005	2.1v1	IEEE 1666-2005 ratified
July 2006	2.2	Bug fixes to more closely implement IEEE 1666-2005

does not specifically support analog hardware or mechanical components, there is no reason why these aspects of a system cannot be modeled with SystemC constructs or with co-simulation techniques.

Open SystemC Initiative

Several of the organizations that contributed heavily to the language development efforts realized very early that any new design language must be open to the community and not be proprietary. As a result, the Open SystemC Initiative (OSCI) was formed in 1999. OSCI was formed to:

- Evolve and standardize the language
- Facilitate communication among the language users and tool vendors
- Enable adoption
- Provide the mechanics for open source development and maintenance

The SystemC Verification Library

As you will learn while reading this book, SystemC consists of the language and potential methodology-specific libraries. The authors view the SystemC Verification (SCV) library as the most significant of these libraries. This library adds support for modern high-level verification language concepts such as constrained randomization, introspection, and transaction recording. The first release of the SCV library occurred in December of 2003 after over a year of Beta testing. This edition includes a chapter devoted to the SCV from a syntactic point of view.

Current Activities with OSCI

At present, the OSCI has completed the SystemC LRM that has been ratified as a standard (IEEE 1666-2005) by the Institute of Electrical and Electronics Engineers (IEEE). Additionally, sub-committees are studying such topics as synthesis subsets and formalizing terminology concerning levels of abstraction for transaction-level modeling (TLM). This edition includes a chapter devoted to TLM and current activities.

Acknowledgments

- Our inspiration was provided by:
  Mike Baird, President of Willamette HDL, who provided
  the basic knowledge to get us started on our SystemC journey.
- Technical information was provided by:
  IEEE-1666-2005 Standard
  OSCI Proof-of-Concept Library associated information on systemc.org
  Andy Goodrich of Forte Design Systems, who provided technical insights.
- Our reviewers provided feedback that helped keep us on track:
  Chris Donovan, Cisco Systems Incorporated
  Ronald Goodstein, First Shot Logic Simulation and Design
  Mark Johnson, Rockwell-Collins Corporation
  Rob Keist, Freescale Corporation
  Miriam Leeser, Northeastern University
  Chris Macionski, Synopsys Inc.
  Nasib Naser, Synopsys Inc.
  Suhas Pai, Qualcomm Incorporated
  Charles Wilson, XtremeEDA Corporation
  Claude Cloutier, XtremeEDA Corporation
  David Jones, XtremeEDA Corporation
- The team who translated the first edition of the book into Japanese and asked us many thought provoking questions that have hopefully been answered in this edition:
  Masamichi Kawarabayashi (Kaba), NEC Electronics Corporation
  Sanae Nakanishi, NEC Electronics Corporation
  Takashi Hasegawa, Fujitsu Corporation
  Masaru Kakimoto, Sony Corporation
- The translator of the Korean version of the first edition who caught many detailed errors. We hope that we have corrected them all in this edition:
  Goodkook, Anslab Corporation
- Our Graphic Artist
  Felix Castillo
- Our Technical Editors helped us say what we meant to say:
  Kyle Smith, Smith Editing
  Richard Whitfield

- Our Readers from the First Edition:
  David Jones, Junyee Lee, Soheil Samii, Kazunari Sekigawa, Ando Ki,
  Jeff Clark, Russell Fredrickson, Mike Campin, Marek Tomczyk,
  Luke Lee, Adamoin Harald Devos, Massimo Iaculo, and many others
  who reported errata in the first edition.

Most important of all, we acknowledge our spouses, Pamela Black, Carol Donovan, Rob Keist, and Evelyn Bunton. These wonderful life partners (despite misgivings about four wild-eyed engineers) supported us cheerfully as we spent many hours researching, typing, discussing, and talking to ourselves while pacing around the house as we struggled to write this book over the past year.

We also acknowledge our parents who gave us the foundation for both our family and professional life.

Contents

1 Why SYSTEMC: ESL and TLM 1

1.1 Introduction 1
1.2 ESL Overview 2
1.2.1 Design Complexity 2
1.2.2 Shortened Design Cycle = Need For Concurrent Design 3
1.3 Transaction-Level Modeling 7
1.3.1 Abstraction Models 7
1.3.2 An Informal Look at TLM 8
1.3.3 TLM Methodology 10
1.4 A Language for ESL and TLM: SystemC 14
1.4.1 Language Comparisons and Levels of Abstraction 15
1.4.2 SystemC: IEEE 1666 16
1.4.3 Common Skill Set 16
1.4.4 Proper Simulation Performance and Features 16
1.4.5 Productivity Tool Support 17
1.4.6 TLM Concept Support 17
1.5 Conclusion 18

2 Overview of SystemC 19

2.1 C++ Mechanics for SystemC 20
2.2 SystemC Class Concepts for Hardware 22
2.2.1 Time Model 22
2.2.2 Hardware Data Types 23
2.2.3 Hierarchy and Structure 23
2.2.4 Communications Management 23
2.2.5 Concurrency 24
2.2.6 Summary of SystemC Features for Hardware Modeling 24
2.3 Overview of SystemC Components 25
2.3.1 Modules and Hierarchy 25
2.3.2 SystemC Threads and Methods 25
2.3.3 Events, Sensitivity, and Notification 26

2.3.4 SystemC Data Types ... 27
2.3.5 Ports, Interfaces, and Channels ... 27
2.3.6 Summary of SystemC Components ... 28
2.4 SystemC Simulation Kernel ... 29

3 Data Types ... 31

3.1 Native C++ Data Types ... 31
3.2 SystemC Data Types Overview ... 32
3.3 SystemC Logic Vector Data Types ... 33
3.3.1 sc_bv<W> ... 33
3.3.2 sc_logic and sc_lv**<*W*>** ... 34
3.4 SystemC Integer Types ... 35
3.4.1 sc_int<W> and sc_uint<W> ... 35
3.4.2 sc_bigint**<*W*>** and sc_biguint**<*W*>** ... 35
3.5 SystemC Fixed-Point Types ... 36
3.6 SystemC Literal and String ... 39
3.6.1 SystemC String Literals Representations ... 39
3.6.2 String Input and Output ... 40
3.7 Operators for SystemC Data Types ... 41
3.8 Higher Levels of Abstraction and the STL ... 43
3.9 Choosing the Right Data Type ... 44
3.10 Exercises ... 44

4 Modules ... 47

4.1 A Starting Point: sc_main ... 47
4.2 The Basic Unit of Design: SC_MODULE ... 49
4.3 The SC_MODULE Class Constructor: SC_CTOR ... 50
4.4 The Basic Unit of Execution: Simulation Process ... 51
4.5 Registering the Basic Process: SC_THREAD ... 52
4.6 Completing the Simple Design: main.cpp ... 53
4.7 Alternative Constructors: SC_HAS_PROCESS ... 53
4.8 Two Styles Using SystemC Macros ... 55
4.8.1 The Traditional Coding Style ... 55
4.8.2 Recommended Alternate Style ... 56
4.9 Exercises ... 57

5 A Notion of Time ... 59

5.1 sc_time ... 59
5.1.1 SystemC Time Resolution ... 60
5.1.2 Working with sc_time ... 61
5.2 sc_time_stamp() ... 61
5.3 sc_start() ... 62
5.4 wait(sc_time) ... 63
5.5 Exercises ... 64

6 Concurrency ... 65
6.1 Understanding Concurrency ... 65
6.2 Simplified Simulation Engine ... 68
6.3 Another Look at Concurrency and Time ... 70
6.4 The SystemC Thread Process ... 71
6.5 SystemC Events ... 72
6.5.1 Causing Events ... 73
6.6 Catching Events for Thread Processes ... 74
6.7 Zero-Time and Immediate Notifications ... 75
6.8 Understanding Events by Way of Example ... 78
6.9 The SystemC Method Process ... 81
6.10 Catching Events for Method Processes ... 83
6.11 Static Sensitivity for Processes ... 83
6.12 Altering Initialization ... 86
6.13 The SystemC Event Queue ... 87
6.14 Exercises ... 88

7 Dynamic Processes ... 89
7.1 Introduction ... 89
7.2 sc_spawn ... 89
7.3 Spawn Options ... 91
7.4 A Spawned Process Example ... 92
7.5 SC_FORK/SC_JOIN ... 93
7.6 Process Control Methods ... 96
7.7 Exercises ... 97

8 Basic Channels ... 99
8.1 Primitive Channels ... 100
8.2 sc_mutex ... 100
8.3 sc_semaphore ... 102
8.4 sc_fifo ... 104
8.5 Exercises ... 106

9 Evaluate-Update Channels ... 107
9.1 Completed Simulation Engine ... 108
9.2 SystemC Signal Channels ... 110
9.3 Resolved Signal Channels ... 113
9.4 Template Specializations of sc_signal Channels ... 115
9.5 Exercises ... 116

10 Structure ... 117
10.1 Module Hierarchy ... 117
10.2 Direct Top-Level Implementation ... 119

10.3 Indirect Top-Level Implementation 119
10.4 Direct Submodule Header-Only Implementation 120
10.5 Direct Submodule Implementation 120
10.6 Indirect Submodule Header-Only Implementation 121
10.7 Indirect Submodule Implementation 122
10.8 Contrasting Implementation Approaches 123
10.9 Exercises 123

11 Communication 125

11.1 Communication: The Need for Ports 125
11.2 Interfaces: C++ and SystemC 126
11.3 Simple SystemC Port Declarations 129
11.4 Many Ways to Connect 130
11.5 Port Connection Mechanics 132
11.6 Accessing Ports From Within a Process 134
11.7 Exercises 135

12 More on Ports & Interfaces 137

12.1 Standard Interfaces 137
12.1.1 SystemC FIFO Interfaces 137
12.1.2 SystemC Signal Interfaces 139
12.1.3 sc_mutex and sc_semaphore Interfaces 140
12.2 Sensitivity Revisited: Event Finders and Default Events 140
12.3 Specialized Ports 142
12.4 The SystemC Port Array and Port Policy 145
12.5 SystemC Exports 148
12.6 Connectivity Revisited 153
12.7 Exercises 155

13 Custom Channels and Data 157

13.1 A Review of SystemC Channels and Interfaces 157
13.2 The Interrupt, a Custom Primitive Channel 158
13.3 The Packet, a Custom Data Type for SystemC 159
13.4 The Heartbeat, a Custom Hierarchical Channel 162
13.5 The Adaptor, a Custom Primitive Channel 164
13.6 The Transactor, a Custom Hierarchical Channel 166
13.7 Exercises 170

14 Additional Topics 171

14.1 Error and Message Reporting 171
14.2 Elaboration and Simulation Callbacks 174
14.3 Configuration 175
14.4 Programmable Structure 177
14.5 sc_clock, Predefined Processes 181

14.6 Clocked Threads, the SC_CTHREAD 182
14.7 Debugging and Signal Tracing 185
14.8 Other Libraries: SCV, ArchC, and Boost 187
14.9 Exercises 187

15 SCV 189

15.1 Introduction 189
15.2 Data Introspection 189
15.2.1 Components for scv_extension Interface 190
15.2.2 Built-In scv_extensions 192
15.2.3 User-Defined Extensions 193
15.3 scv_smart_ptr Template 193
15.4 Randomization 194
15.4.1 Global Configuration 194
15.4.2 Basic Randomization 196
15.4.3 Constrained Randomization 197
15.4.4 Weighted Randomization 198
15.5 Callbacks 200
15.6 Sparse Arrays 201
15.7 Transaction Sequences 202
15.8 Transaction Recording 203
15.9 SCV Tips 204
15.10 Exercises 204

16 OSCI TLM 207

16.1 Introduction 207
16.2 Architecture 208
16.3 TLM Interfaces 210
16.3.1 Unidirectional Blocking Interfaces 211
16.3.2 Unidirectional Non-Blocking Interfaces 211
16.3.3 Bidirectional Blocking Interface 213
16.4 TLM Channels 213
16.5 Auxiliary Components 214
16.5.1 TLM Master 215
16.5.2 TLM Slave 215
16.5.3 Router and Arbiter 216
16.6 A TLM Example 217
16.7 Summary 220
16.8 Exercises 220

17 Odds & Ends 223

17.1 Determinants in Simulation Performance 223
17.1.1 Saving Time and Clocks 224
17.1.2 Moving Large Amounts of Data 225

17.1.3 Too Many Channels 226
17.1.4 Effects of Over Specification 227
17.1.5 Keep it Native 227
17.1.6 C++ Compiler Optimizations 227
17.1.7 C++ Compilers 227
17.1.8 Better Libraries 227
17.1.9 Better and More Simulation Computers 228
17.2 Features of the SystemC Landscape 228
17.2.1 Things You Wish Would Just Go Away 228
17.2.2 Development Environment 230
17.2.3 Conventions and Coding Style 230
17.3 Next Steps 231
17.3.1 Guidelines for Adopting SystemC 231
17.3.2 Resources for Learning More 231

Appendix A 235
A.1 Background of C++ 236
A.2 Structure of a C Program 236
A.3 Comments 237
A.4 Streams (I/O) 237
A.4.1 Streaming vs. printf 238
A.5 Basic C Statements 238
A.5.1 Expressions and Operators 238
A.5.2 Conditional 240
A.5.3 Looping 241
A.5.4 Altering Flow 242
A.6 Data Types 242
A.6.1 Built-In Data Types 243
A.6.2 User-Defined Data Types 243
A.6.3 Constants 246
A.6.4 Declaration vs. Definition 246
A.7 Functions 247
A.7.1 Pass By Value and Return 248
A.7.2 Pass by Reference 248
A.7.3 Overloading 249
A.7.4 Constant Arguments 249
A.7.5 Defaults for Arguments 250
A.7.6 Operators as Functions 250
A.8 Classes 251
A.8.1 Member Data and Member Functions 251
A.8.2 Constructors and Destructors 252
A.8.3 Destructors 255
A.8.4 Inheritance 256
A.8.5 Public, Private and Protected Access 258
A.8.6 Polymorphism 258

A.8.7 Constant Members 260
A.8.8 Static Members 260
A.9 Templates 261
A.9.1 Defining Template Functions 261
A.9.2 Using Template Functions 261
A.9.3 Defining Template Classes 262
A.9.4 Using Template Classes 262
A.9.5 Template Considerations 262
A.10 Names and Namespaces 263
A.10.1 Meaningful Names 263
A.10.2 Ordinary Scope 263
A.10.3 Defining Namespaces 264
A.10.4 Using Names and Namespaces 264
A.10.5 Anonymous Namespaces 264
A.11 Exceptions 265
A.11.1 Watching for and Catching Exceptions 265
A.11.2 Throwing Exceptions 266
A.11.3 Functions that Throw 267
A.12 Standard Library Tidbits 268
A.12.1 Strings 268
A.12.2 File I/O 268
A.12.3 Standard Template Library 270
A.13 Closing Thoughts 270
A.14 References 271

Index 273

Chapter 1
Why SYSTEMC: ESL and TLM

1.1 Introduction

The goal of this chapter is to explain why it is important for you to learn SystemC. If you already know why you are studying SystemC, then you can jump ahead to Chapter 2. If you are learning SystemC for a college course or because your boss says you must, then you may benefit from this chapter. If your boss doesn't know why you need to spend your time learning SystemC, then you may want to show your boss this chapter.

SystemC is a system design and modeling language. This language evolved to meet a system designer's requirements for designing and integrating today's complex electronic systems very quickly while assuring that the final system will meet performance expectations.

Typically, today's systems contain both application-specific hardware and software. Furthermore, the hardware and software are usually co-developed on a tight schedule with tight real-time performance constraints and stringent requirements for low power. Thorough functional (and architectural) verification is required to avoid expensive and sometimes catastrophic failures in the device. In some cases, these failures result in the demise of the company or organization designing the errant system. The prevailing name for this concurrent and multidisciplinary approach to the design of complex systems is electronic system-level design or ESL.

The drive for concurrent engineering through ESL has side effects that affect more than the design organizations of a company. ESL affects the basic business model of a company and how companies interact with their customers and with their suppliers.

ESL happens by modeling systems at higher levels of abstraction than traditional methods used in the past. Portions of the system model are subsequently iterated and refined, as needed. A set of techniques has evolved called Transaction-Level Modeling or TLM to aide with this task.

ESL and TLM impose a set of requirements on a language that is different than the requirements for hardware description languages (HDLs) or the requirements

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_1,

for traditional software languages like C, C++[1], or Java. The authors believe that SystemC is uniquely positioned to meet these requirements.

We will discuss all these topics in more detail in the following sections.

1.2 ESL Overview

ESL techniques evolved in response to increasing design complexity and increasingly shortened design cycles in many industries. Systems, Integrated Circuits (ICs), and Field Programmable Gate Arrays (FPGAs) are becoming large. Many more multi-disciplinary trade-offs are required to optimize the hardware, the software, and the overall system performance.

1.2.1 Design Complexity

The primary driver for an ESL methodology is the same driver that drove the evolution of previous design methodologies: increasing design complexity.

Modern electronic systems consist of many subsystems and components. ESL focuses primarily on hardware, software, and algorithms at the architectural level. In modern systems, each of these disciplines has become more complex. Likewise, the interaction has become increasingly complex.

Interactions imply that trade-offs between the domains are becoming more important for meeting customer requirements. System development teams find themselves asking questions like:

- Should this function be implemented in hardware, software, or with a better algorithm?
- Does this function use less power in hardware or software?
- Do we have enough interconnect bandwidth for our algorithm?
- What is the minimum precision required for our algorithm to work?

These questions are not trivial and the list could go on and on. Systems are so complex, just deriving specifications from customer requirements has become a daunting task. Hence, this task brings the need for higher levels of abstraction and executable specifications or virtual system prototypes.

Figure 1.1 illustrates the complexity issues for just the hardware design in a large system-on-a-chip (SoC) design. The figure shows three sample designs from three generations: yesterday, today, and tomorrow. In reality, tomorrow's systems are being designed today. The bars for each generation imply the code complexity for four common levels of abstraction associated with system hardware design:

- Architecture
- Behavioral
- RTL
- Gates

[1] We will see later that SystemC is actually a C++ class library that "sits on top" of C++.

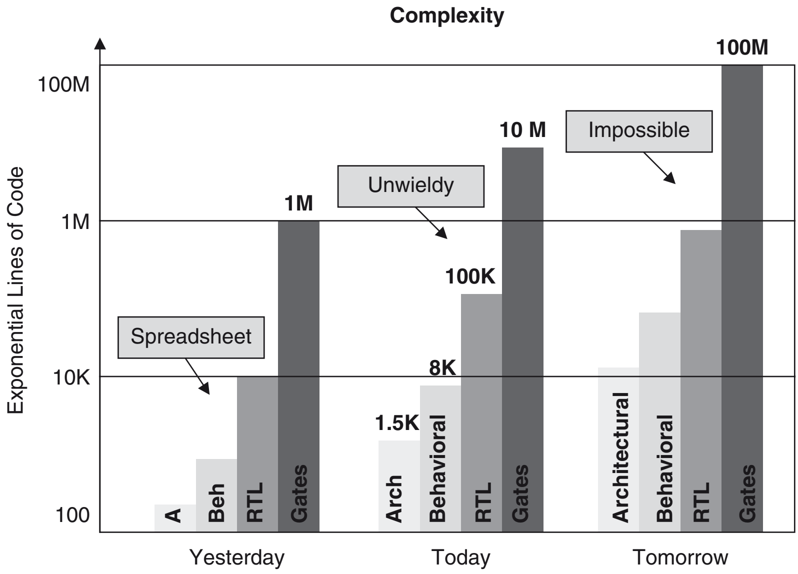

Fig. 1.1 Code complexity for four levels of abstraction

Today's integrated circuits often exceed 10 million gates, which conservatively translates to one hundred thousand lines of RTL code. Today's designs are practical because of the methodologies that apply RTL synthesis for automated generation of gates. Tomorrow's integrated circuits, which are being designed today, will exceed one hundred million gates. This size equates to roughly one million lines of RTL code, if written using today's methodologies.

Notice that Figure 1.1 considers only a single integrated circuit. It does not reflect the greater complexity of a system with several large chips (integrated circuits or FPGAs) and gigabytes of application software. Many stop-gap approaches are being applied, but the requirement for a fundamentally new approach is clear.

1.2.2 Shortened Design Cycle = Need For Concurrent Design

Anyone who has been part of a system design realizes that typical design cycles are experiencing more and more schedule pressure. Part of the reason for the drive for a shortened design cycle is the perceived need for a very fast product cycle in the marketplace. Anyone who has attempted to find a cell phone or a laptop "just like my last one", even just nine months after buying the "latest and greatest" model, will find themselves looking long and hard throughout the web for a replacement[2].

[2] This scenario describes a recent experience by one of the authors.

Many are under the misguided assumption that shorter design cycles imply reduced development expenses. If the scope of the new system is not reduced and the schedule is reduced, then additional resources are required. In this scenario, a shorter design cycle actually requires more resources (expenses). The project requires more communication between development groups (because of concurrent design), more tools, more people, and more of everything. ESL and TLM are an approach to reduce the cost of development through more efficient communication and through reduced rework.

1.2.2.1 Traditional System Design Approach

In the past, when many systems were a more manageable size, a system could be grasped by one person. This person was known by a variety of titles such as system architect, chief engineer, lead engineer, or project engineer. This guru may have been a software engineer, hardware engineer, or algorithm expert depending on the primary technology leveraged for the system. The complexity was such that this person could keep most or all of the details in his or her head. This technical leader was able to use spreadsheets and paper-based methods to communicate thoughts and concepts to the rest of the team.

The guru's background usually dictated his or her success in communicating requirements to each of the communities involved in the design of the system. The guru's past experiences also controlled the quality of the multi-disciplinary trade-offs such as hardware implementation versus software implementation versus algorithm improvements.

In most cases, these trade-offs resulted in conceptual disconnects among the three groups. For example, cellular telephone systems consist of very complex algorithms, software, and hardware. Teams working on them have traditionally leveraged more rigorous but still ad-hoc methods.

The ad-hoc methods usually consist of a software-based model. This model is sometimes called a system architectural model (SAM), written in C, Java, or a similar language. The SAM is a communication vehicle between algorithm, hardware, and software groups. The model can be used for algorithmic refinement or used as basis for deriving hardware and software subsystem specifications. The exact parameters modeled are specific to the system type and application, but the model is typically un-timed (more on this topic in the following section). Typically, each team then uses a different language to refine the design for their portion of the system. The teams leave behind the original multi-discipline system model and in many cases, any informal communication among the groups.

The traditional approach often resulted in each design group working serially with a series of paper specifications being tossed "over the wall" to the other organization. This approach also resulted in a fairly serial process that is many times described as a "waterfall schedule" or "transom engineering" by many program managers and is illustrated in Fig. 1.2.

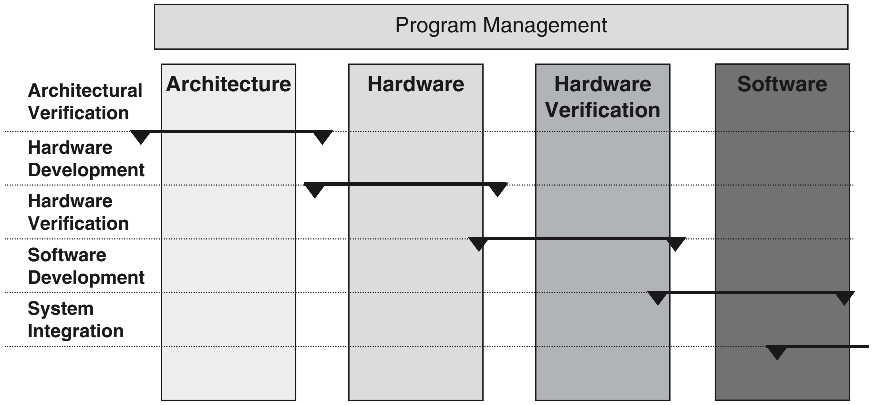

Fig. 1.2 The traditional approach of waterfall scheduling

To minimize the serialization of the design process, many techniques have been used to create some design concurrency. These techniques include processor development boards using a single-chip implementation of the processor. These implementations were used on the SoC or embedded system, FPGA prototypes of algorithms, and hardware emulation systems, just to name a few. These techniques were focused on early development of software, usually the last thing completed before a system is deployed.

The ESL approach uses these existing techniques. ESL also leverages a virtual system prototype or a TLM model of the system to enable all the system design disciplines to work in parallel. This virtual system prototype is the common specification among the groups. The resulting Gantt chart is illustrated next in Fig. 1.3.

Even though all of the electronic system design organizations will finish their tasks earlier, the primary reason for ESL is earlier development of software. Even getting a product to market a month earlier can mean tens of millions of dollars of business to a company.

Not using ESL methods will likely result in the under-design or over-design of the system. Both of these results are not good. Under-design is obviously not good. The product may be bug-free, but it doesn't necessarily meet the customer's requirements. The product may not operate fast enough, may not have long enough battery life, or just may not have the features required by the customer.

Over-design is not as obvious, but it is not good either. Over-design takes significantly more resources and time to achieve, and it adds a heavy cost to an organization. In addition, over-designed products usually are more complex, more expensive to manufacture, and are not as reliable.

The authors have significant anecdotal stories of under-design and over-design of systems. One company built an ASIC with multi-processors that made "timing closure" and paired those processors with software that made the "timing budget."

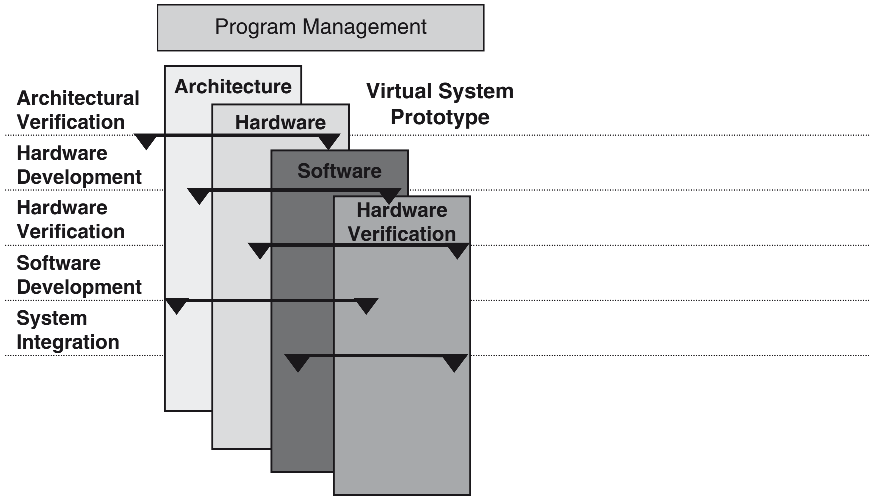

Fig. 1.3 The ESL approach of parallel schedule

Unfortunately, the ASIC didn't meet the customers requirements because of "on chip" bottlenecks. Another company related how a significant function on a chip caused weeks of schedule slip for design and verification. However, the function was later found not to be used by the software.

Things become even more interesting if a system, say a cell phone, are really a subsystem for the customer, who is a mobile phone and infrastructure provider. Now, the customer needs models very early in their design process and will be making system design trade-offs based on a model provided by the subsystem provider (previously system). In addition, the subsystem provider likely relies on third-party intellectual property. The subsystem cell phone supplier will then need a model of the intellectual property used in their subsystem very early in the development cycle to enable their design trade-offs. Customers up and down the value chain may now be making business decisions based on the type and quality of the model provided by their supplier. This hierarchy of reliance is fundamentally a different way of doing business.

The virtual system prototype may have different blocks (or components or subsystems) at different levels of abstraction for a particular task to be performed by one of the system disciplines. Initially, most of the system may be very abstract for software development until the software team is reasonably sure of the functionality. At this point, a more detailed model of the blocks that closely interact with the software can be introduced into the model.

The technique that allows this "mixing and matching" of blocks at different levels of abstraction is called Transaction-Level Modeling or TLM. We will discuss TLM in much greater detail in the following section.

1.3 Transaction-Level Modeling

TLM and the associated methodologies are the basic techniques that enable ESL and make it practical. To understand TLM, one must first have a terminology for describing abstraction levels. Secondly, one must understand the ways the models will probably be used (the use cases).

1.3.1 Abstraction Models

Several years ago, Professor Gajski from UC Irvine proposed a taxonomy for describing abstraction levels of models. The basic concept states that refinement of the interface or communication of a logical block can be refined independently of functionality or algorithm of the logical block [3]. We have significantly modified his concept and presented the result in the figure below.

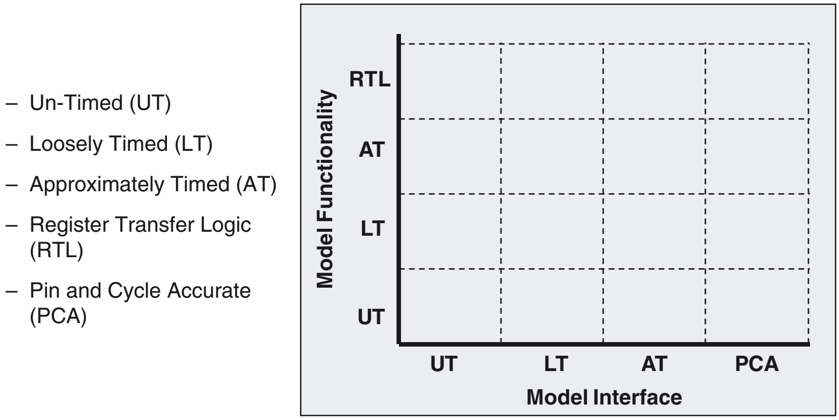

Fig. 1.4 Decoupling of abstraction refinement

[3] Gajski and L. Cai, "Transaction Level Modeling," First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2003), Newport Beach, CA, October 1, 2003