Hardware Verification with SystemVerilog

An Object-Oriented Framework
Mike Mintz
Robert Ekendahl

Hardware Verification with SystemVerilog

An Object-Oriented Framework

Cover art from the original painting “Dimentia #10” by John E. Bannon, johnebannon.com

Springer
For Joan, Alan, and Brian.  
Thanks again for your patience.  
   Mike

For Chantal.  
Thanks again for your understanding,  
love, and active support.  

And to Newton—and now Darwin.  
For many more missed walks.  
   Robert
Preface ......................................................... xix
Acknowledgments .............................................. xxi

Chapter 1: Introduction ................................. 1
  Background .................................................. 3
  What is Functional Verification? ....................... 4
  Why Focus on SystemVerilog? ......................... 5
  A Tour of the Handbook ................................. 5
  For Further Reading ...................................... 6

Part I:  
SystemVerilog and Verification 
(The Why and How) ......................................... 7

Chapter 2: Why SystemVerilog? ...................... 9
  Overview .................................................... 10
  SystemVerilog as a Verification Language .......... 11
  Main Benefits of Using SystemVerilog ............... 13
  Drawbacks of Using SystemVerilog ................. 13
  SystemVerilog Traps and Pitfalls ..................... 14
  SystemVerilog is not Verilog ......................... 14
  Errors and run-time crashes ......................... 15
Wiggling wires: the interface concept. 42
Building and using interfaces 44
Summary 46
For Further Reading 46

Chapter 4: A Layered Approach 47
Overview 48
A Whiteboard Drawing 50
An “ends-in” approach 51
Refining the whiteboard blocks 52
The “Common-Currency” Components 52
The Component Layer in Detail 53
The connection layer 54
The agent layer 56
The transaction layer 57
The Top-Layer Components 58
What is a Test? 60
The Test Component 62
The Test Irritator 64
A Complete Test 65
Summary 67
For Further Reading 67

Part II: An Open-Source Environment with SystemVerilog 69

Chapter 5: Teal Basics 71
Overview 72
Teal’s Main Components 72
Using Teal 74
A simple test 74
Logging Output 74
Using Test Parameters 77
Accessing Memory 79
Contents

A memory example ........................................ 80
Constrained Random Numbers ............................. 84
   Required initialization ................................. 84
   Using random numbers ................................. 85
Working with Simulation Events ............................ 86
Summary ..................................................... 87

Chapter 6: Truss: A Standard Verification Framework ........... 89
   Overview .................................................. 90
   General Considerations .................................. 91
      SystemVerilog considerations .......................... 91
      Keeping it simple ...................................... 92
   Major Classes and Their Roles ............................ 93
      Key test algorithm: The “dance” ...................... 94
   The verification_component Virtual Base Class ............ 97
   Detailed Responsibilities of the Major Components .......... 98
      The testbench class ................................... 99
      Watchdog timer ....................................... 101
      Test class ........................................... 102
   Test Component and Irritator Classes ..................... 106
      The test component virtual base class ................. 106
      An AHB example ....................................... 108
      Test-component housekeeping functionality .......... 109
      The irritator virtual base class ...................... 110
         Using the irritator ................................. 112
   Summary .................................................. 113
Chapter 7: Truss Flow ......................... 115
Overview ....................................... 116
About truss_verification_top.sv .............. 116
The Test Component Dance .................... 119
The Irritator Dance ............................ 121
Compiling and Running Tests .................. 122
  The truss run script ......................... 123
  Switches ...................................... 124
  Using “-f” files .............................. 125
The First Test: A Directed Test ............... 125
The Second Test:
  Adding Channels and Random Parameters .... 127
    The channel pseudo-templated classes ...... 128
    Building the second test ................... 129
    Building the second test’s test_component ... 131
    Adjusting the second test’s parameters .... 132
The Remaining Tests:
  Mix-and-Match Test Components ............ 135
  Summary ..................................... 136

Chapter 8: Truss Example ...................... 137
  Overview .................................... 138
  Directory Structure ........................ 138
  Theory of Operation ......................... 140
  Running the Simple ALU Example ............ 142
  Points of Interest .......................... 142
  Power-on Reset ................................ 143
  Driver and Monitor Protocol ................. 144
  The alu_test_component ..................... 145
  Checking the Chip .......................... 146
  Completing the Test ......................... 147
  Summary .................................... 149
## Part III: Using OOP for Verification (Best Practices)

### Chapter 9: Thinking OOP

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>154</td>
</tr>
<tr>
<td>Sources of Complexity</td>
<td>155</td>
</tr>
<tr>
<td>Essential complexity vs. implementation complexity</td>
<td>155</td>
</tr>
<tr>
<td>Flexibility vs. complexity</td>
<td>156</td>
</tr>
<tr>
<td>Apparent simplicity vs. hiding inherent complexity</td>
<td>159</td>
</tr>
<tr>
<td>Example: How hiding complexity can create confusion</td>
<td>159</td>
</tr>
<tr>
<td>Example: How apparent simplicity leads to later problems</td>
<td>160</td>
</tr>
<tr>
<td>Team dynamics</td>
<td>162</td>
</tr>
<tr>
<td>Team roles</td>
<td>162</td>
</tr>
<tr>
<td>Using a “code buddy”</td>
<td>163</td>
</tr>
<tr>
<td>Creating Adaptable Code</td>
<td>163</td>
</tr>
<tr>
<td>Achieving adaptability</td>
<td>163</td>
</tr>
<tr>
<td>Why is adaptability tricky?</td>
<td>164</td>
</tr>
<tr>
<td>Architectural Considerations to Maximize Adaptability</td>
<td>165</td>
</tr>
<tr>
<td>Changes are easy—or just plain impossible.</td>
<td>166</td>
</tr>
<tr>
<td>Where is adaptation likely to happen?</td>
<td>167</td>
</tr>
<tr>
<td>Separating Interface from Implementation</td>
<td>168</td>
</tr>
<tr>
<td>Code Interface, Implementation, and Base Classes</td>
<td>169</td>
</tr>
<tr>
<td>Summary</td>
<td>170</td>
</tr>
<tr>
<td>For Further Reading</td>
<td>171</td>
</tr>
</tbody>
</table>
### Chapter 10: Designing with OOP

- Overview ........................................ 174
- Keeping the Abstraction Level Consistent ...... 174
- Using “Correct by Construction” ................. 176
- The Value of Packages .......................... 178
- Data Duplication—A Necessary Evil ............ 180
- Designing Well, Optimizing Only When Necessary .. 181
- Using the Protocol, Only the Protocol .......... 182
- Verification Close to the Programming Model .... 183
- The Three Parts of Checking .................. 184
- Separating the Test from the Testbench .......... 186
- Summary ........................................ 187
- For Further Reading ............................ 188

### Chapter 11: OOP Classes

- Overview ........................................ 190
- Defining Classes ................................ 191
- How Much Electricity? .......................... 191
  - Classes ......................................... 192
  - Packages ...................................... 192
  - Pointers and virtual functions ................. 192
- Global Services ................................. 193
  - Package it up! .................................. 193
  - Static methods ................................ 194
  - Singletons—A Special Case of Static Methods .. 194
  - Packages or static methods? .................. 195
  - Other considerations .......................... 196
- Class Instance Identifiers ....................... 197
  - Strings as identifiers ......................... 197
  - Static integers as identifiers .................. 197
  - Combination identifiers ....................... 198
- Class Inheritance for Reuse .................... 198
  - A BFM base-class example ...................... 199
Hardware Verification with SystemVerilog

Chapter 12: OOP Connections

Overview ............................................. 206
How Tight a Connection? ......................... 207
Types of Connections ................................. 209
Peer-to-peer connections ....................... 209
Master-to-slave and push-vs.-pull connections ...... 209
Two Tight Connection Techniques ............. 211
Using pointers ................................... 211
Using inheritance ................................ 212
Threads and Connections ....................... 214
Events—explicit blocking interconnects .... 214
Hiding the thread block in a method .......... 216
Fancier Connections ............................... 217
Listener or callback connections ............ 218
Channel connections ............................. 219
Action object connections .................... 220
Summary .......................................... 221
For Further Reading ............................. 222
Part IV: Examples (Putting It All Together) ................................. 247

Chapter 14: Block-Level Testing .................. 249

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>250</td>
</tr>
<tr>
<td>Theory of Operation</td>
<td>251</td>
</tr>
<tr>
<td>Verification environment</td>
<td>252</td>
</tr>
<tr>
<td>Verification IP</td>
<td>253</td>
</tr>
<tr>
<td>UART VIPs</td>
<td>253</td>
</tr>
<tr>
<td>Wishbone VIP</td>
<td>254</td>
</tr>
<tr>
<td>The verification dance</td>
<td>255</td>
</tr>
<tr>
<td>Running the UART Example</td>
<td>255</td>
</tr>
<tr>
<td>Points of Interest</td>
<td>256</td>
</tr>
<tr>
<td>Configuration</td>
<td>256</td>
</tr>
<tr>
<td>VIP UART package</td>
<td>257</td>
</tr>
<tr>
<td>VIP UART configuration class</td>
<td>258</td>
</tr>
<tr>
<td>Randomization of parameters</td>
<td>258</td>
</tr>
<tr>
<td>UART 16550 configuration class</td>
<td>260</td>
</tr>
<tr>
<td>Configuring the Chip</td>
<td>261</td>
</tr>
<tr>
<td>Register access</td>
<td>262</td>
</tr>
<tr>
<td>The wishbone_memory_bank and wishbone_driver</td>
<td>263</td>
</tr>
<tr>
<td>Traffic Generation</td>
<td>265</td>
</tr>
<tr>
<td>The generator_agent and uart_bfm_agent classes</td>
<td>265</td>
</tr>
<tr>
<td>The Checker</td>
<td>267</td>
</tr>
<tr>
<td>Checking the data</td>
<td>268</td>
</tr>
<tr>
<td>Connecting It All Together</td>
<td>270</td>
</tr>
<tr>
<td>The testbench</td>
<td>270</td>
</tr>
<tr>
<td>Building the channels</td>
<td>271</td>
</tr>
<tr>
<td>Building the configuration and interface port</td>
<td>271</td>
</tr>
<tr>
<td>Building the component-layer objects</td>
<td>273</td>
</tr>
<tr>
<td>The wishbone objects</td>
<td>274</td>
</tr>
<tr>
<td>The test component</td>
<td>275</td>
</tr>
</tbody>
</table>
Chapter 15: Chip-Level Testing ............ 281
Overview .................................. 282
Theory of Operation ....................... 282
  Verification environment ................. 283
Running the UART Example ............... 284
The quad_uart_test_components Test ....... 284
The quad_uart_irritators Test ............. 286
  UART irritator class .................... 286
  The test ................................ 288
The quad_uart_vectors Test ............... 292
The block_uart Test ..................... 293
Summary ................................ 293

Chapter 16: Things to Remember ............ 295
Part I: Use SystemVerilog and Layers! .... 296
Part II: An Open-Source Approach ......... 296
Part III: OOP—Best Practices ............. 297
Part IV: Examples—Copy and Adapt! ........ 298
Conclusion to the Conclusion ............. 298

Index ..................................... 301
Preface

This is the second of our books designed to help the professional verifier manage complexity. This time, we have responded to a growing interest not only in object-oriented programming but also in SystemVerilog. The writing of this second handbook has been just another step in an ongoing masochistic endeavor to make your professional lives as painfree as possible.

The authors are not special people. We have worked in several companies, large and small, made mistakes, and generally muddled through our work. There are many people in the industry who are smarter than we are, and many coworkers who are more experienced. However, we have a strong desire to help.

We have been in the lab when we bring up the chips fresh from the fab, with customers and sales breathing down our necks. We’ve been through software bring-up and worked on drivers that had to work around bugs\(^1\) in production chips.

What we feel makes us unique is our combined broad experience from both the software and hardware worlds. Mike has over 20 years of experience from the software world that he applies in this book to hardware verification. Robert has over 12 years of experience with hardware verification, with a focus on environments and methodology.

What we bring to the task of functional verification is over three decades of combined experience, from design, verification, software development, and management. It is our experiences that speak in this handbook. It is our desire that others might learn and benefit from these experiences.

We have had heated discussions over each line of code in this book and in our open-source libraries. We rarely agree at first, but by having to argue our cases we arrive at what we feel are smart, efficient, flexible, and simple solutions. Most of these we have “borrowed” from the software industry but have applied to the field of verification.

We believe that the verification industry can benefit from the lessons learned from the software domain. By using industry-standard languages, the verification domain can adapt techniques and code from over twenty calendar years

\(^1\) Features.
of software effort, the scope of which is nothing short of stunning. Many brilliant people have paved the way in the software field. Although the field of verification is much younger, we could benefit greatly from listening, learning, and adapting mature programming techniques to the production of products of the highest quality.

So why do we provide open-source software at our website, www.trusster.com? Open-source software is a key to uniting and increasing the productivity of our industry. There is almost no successful closed-source (“hard macro”) intellectual property (IP), for a good reason. Without the ability to look at the source and edit as necessary, the task is much more difficult and the chances for success are slim.

We hope that you enjoy this book—and better yet, find its principles increasingly useful in daily practice. We look forward to your comments. Please keep in touch with us at www.trusster.com.

Mike Mintz
Robert Ekendahl
Cambridge, Massachusetts, USA
March 2007
Acknowledgments

It takes a village to raise a child, and it takes a village to create a book. There is a core family, and a few relatives, and a whole lot of helpful neighbors and friends. Once again, the authors would like to bow humbly to our village—in particular, to the global verification village.

This, our second book, shares many of the same reviewers and adds some new ones. They provided great comments on almost every chapter, both detailed and “big picture,” helping to improve many sections substantially.

Michael Meyer was once again our main technical editor, turning our gibberish into English and making clear where we were unclear. This book would not have been readable without him.

We are truly grateful for all the reviewers, their time, and their suggestions during both the early and near final stages of the book. In particular, we thank Ed Arthur, Oswaldo Cadenas, Jesse Craig, Simon Curry, Thomas Franco, John Hoglund, Mark Goodnature, Tom Jones, James Keithan, Ajeetha Kumari, David Long, Bryan Morris, Nancy Pratt, Joe Pizzi, Dave Rich, Henrik Scheuer, Chris Spear, Peter Teng, Thomas Tessier, Greg Tierney, Igor Tsapenko, Gerry Ventura, Stephanie Waters, and Andrew Zoneball.

We are also grateful for the support and encouragement of the producers of the HDL simulators. In particular, we thank the following simulator companies—Cadence, Mentor Graphics, and Synopsys—for providing licenses to their products, so we could confirm that the examples in this handbook work.
Introduction

CHAPTER 1

_Coding is a human endeavor. Forget that and all is lost._

_Bjarne Stroustrup, father of C++_

There are several books about hardware verification, so what makes this book different? Put simply, this book is meant to be useful in your day-to-day work—which is why we refer to it throughout as a handbook. The authors are like you, cube dwellers, with battle scars from developing chips. We must cope with impossible schedules, a shortage of people to do the work, and constantly mutating hardware specifications.

We subtitled this book _An Object-Oriented Framework_ because a major theme of the book is how to use object-oriented programming (OOP) to do verification well. We focus on real-world examples, bloopers, and code snippets. Sure, we talk about programming theory, but the theme of this book is how to write simpler, adaptable, reusable code. We focus mainly on OOP techniques because we feel that this is the best way to manage the ever-increasing complexity of verification. We back this up with open-source Verification Intellectual Property (VIP), several complete test systems, and scripts to run them.
Chapter 1: Introduction

We cover the following topics:

- SystemVerilog as a verification language
- A tour of the features and real-world facts about SystemVerilog
- How to use OOP to build a flexible and adaptable verification system
- How to use specific OOP techniques to make verification code both simpler and more adaptable, with reference to actual situations (both good and bad) that the authors have encountered
- Useful SystemVerilog code, both as snippets, complete examples, and code libraries—all available as open source

This handbook is divided into four major sections:

- Part I provides an overview of OOP concepts, then walks through the transformation of a block-level view of a typical verification system into code and classes.
- Part II describes two free, open-source code libraries that can serve as a basis for a verification system—or as inspiration for your own environment. The first, called Teal, is a set of utility classes and functions. The second, called Truss, is a complete verification system framework. Both are available as open source and are available at www.trusster.com.
- Part III describes how to use OOP to make your team as productive as possible, how to communicate design intent better, and how to benefit from “lessons learned” in the software world.
- Part IV describes several complete real-world examples that illustrate the techniques described in the earlier parts of this book. In these examples we build complete verification environments with makefiles, scripts, and tests. These examples can serve as starting points for your own environment.

For the curious, each of the chapters in Part I and Part III ends with a section called “For Further Reading,” which recommends relevant landmark papers and books from both the hardware and software domains.\(^1\)

---

\(^1\) The references in these sections, though not academically rigorous, should be sufficient to help you find the most recent versions of these works on the Internet.
Background

The silicon revolution\(^1\) has made computers, cell phones, wireless networks, and portable MP3 players not only ubiquitous but in a constant state of evolution. However, the major impediment to introducing new hardware is no longer the hardware design phase itself, but the verification of it.

Costs of $1M or more and delays of three to six months for new hardware revisions of a large and complex application-specific integrated circuits (ASICs) are common, providing plenty of incentive to get it right the first time. Even with field-programmable gate arrays (FPGAs), upgrades are costly, and debugging an FPGA in the lab is very complex for all but the simplest designs.

For these reasons, functional verification has emerged as a team effort to ensure that a chip or system works as intended. However, functional verification means different things to different people. At the 30,000-foot level, we write specifications, make schedules, and write test plans. Mainly, though, we code. This handbook focuses on the coding part.

White papers are published almost daily to document some new verification technique. Most of you probably have several papers on your desk that you want to read. Well, now you can throw away those papers! This handbook compresses the last ten years of verification techniques into a few hundred pages. Of course, we don’t actually cover that decade in detail (after all, this is not a history book), but we have picked the best techniques we found that actually worked, and reduced them to short paragraphs and examples.

Because of this compression, we cover a wide variety of topics. The handbook’s sections range from talking about SystemVerilog, to introducing OOP, to using OOP at a fairly sophisticated level.

\(^1\) Moore’s law of 1965 is still largely relevant. See “Cramming more components onto integrated circuits,” by Gordon Moore, *Electronics*, Volume 38, Number 8, April 19, 1965.
What is Functional Verification?

Asking “what is functional verification?” brings to mind the familiar poster, “A View of the World from Ninth Avenue,”\(^1\) in which the streets of New York City are predominant and everything beyond is tiny and insignificant. Every one of us has a different perspective, all of which are, of course, “correct.” Put simply, functional verification entails building and running software to make sure that a device under test (DUT, or in layman’s terms, the chip) operates as intended—before it is mass-produced and shipped.

We perform a whole range of tasks where the end goal is to create a high degree of confidence in the functionality of the chip. Mostly we try to find errors of logic, by subjecting the chip to a wide variety of conditions, including error cases (where we validate graceful error handling and ensure that the chip at least does not “lock up”). We also make sure that the chip meets performance goals, and functions in uncommon combinations of parameters (“corner cases”), and confirm that the chip’s features—such as the register, interrupt, and memory-map interfaces—work as specified.

As with the view of New York City, the perspectives of every company, indeed even of the design and test teams within a company, will naturally be slightly different. Nevertheless, as long as the chip works as a product, there are a number of ways to achieve success. That’s why this handbook does not focus on what the specific tasks are; you know what you have to do. Rather, we focus on how you can write your code as effectively as possible, to alleviate the inevitable pain of verification.

\(^1\) Saul Steinberg, cover of *The New Yorker*, March 29, 1976.
Why Focus on SystemVerilog?

A major development in the field of functional verification is the increasingly mainstream use of OOP techniques. Basically, those of us in the verification field need those techniques to handle increasingly complex tasks effectively. While most of the techniques presented in this handbook are adaptable to any number of languages such as Vera or C++, we focus on SystemVerilog—the marriage of the Verilog programming language with OOP.

At its core, OOP is designed to manage complexity. All other things being equal, simpler code is better. Because of the flexibility inherent in using OOP, we can write code that is simpler to use, and therefore more adaptable. In short, we can write reusable code that outlives its initial use.

This handbook is all about providing techniques, guidelines, and examples for using SystemVerilog in verification, allowing you to make more use of some “lessons learned” by software programmers. We distill the important bits of knowledge and techniques from the software world, and present them in the light of verification.

A Tour of the Handbook

The four parts of this handbook provide a variety of programming tips and techniques.

- **Part I** walks through the main concepts of OOP, introducing how to transform your high-level “whiteboard” idea for a verification system into separate roles and responsibilities. The goal is to build appropriately simple and adaptable verification systems.

- **Part II** uses these techniques and presents two open-source code libraries for verification, called Teal and Truss. Teal is a utility package that is lightweight and framework agnostic. Truss is a verification framework that encourages the use of the canonical form described in Part I. Both are used by several companies and run under most simulators.
Part III introduces the OOP landscape in a fair amount of detail. OOP thinking, design, and coding are illustrated by means of code snippets representative of problems that verification engineers commonly have to solve.

Part IV provides several complete examples of verification test systems, providing real-world examples and more details on how the OOP techniques discussed are actually used. Part IV is all about code. While a handbook may not be the best vehicle for describing code, it can be a good reference tool. We show a relatively simple example of how the verification of a single block of the ubiquitous UART\(^1\) can be done. Then we show how this block-level environment can be expanded to a larger system.

The authors sincerely hope that, by reading this handbook, you will find useful ideas, techniques, and examples that you can use in your day-to-day verification coding efforts.

For Further Reading

- On the topic of coding well, *Writing Solid Code*, by Steve McGuire, is a good tour of the lessons Microsoft has learned.
- *Principles of Functional Verification*, by Andreas Meyer, provides an introduction to the broad topic of chip verification.

\(^1\) Universal asynchronous receiver-transmitter.
Part I: SystemVerilog and Verification (The Why and How)

This part of the handbook explores the use of SystemVerilog for verification and then look at the benefits and drawbacks of using SystemVerilog. In the next chapter we take a brief tour of the features of SystemVerilog.

Next, we weave three different themes together: the evolution of programming in general, the creation of object-oriented programming (OOP) techniques, and the evolution of functional verification. The reason we chose to look at these three themes is to show why OOP exists and how it can be harnessed to benefit verification.

A major theme of this handbook is to build a verification system in layers. OOP techniques are well-suited to this approach. In the last chapter of this section, we’ll look at a canonical verification system by using a standard approach to building verification components.
Why SystemVerilog?

If you want to do buzzword-oriented programming, you must use a strongly hyped language.

Mike Johns

We, in the functional verification trade, write code for a living. Well, we do that, and also puzzle over code that has been written and that has yet to be written. Because functional verification is a task that only gets more complex as designs become more complex, the language we work in determines how well we can cope with this increasing complexity.

The authors believe that SystemVerilog is an appropriate choice for functional verification, but as with any choice, there are trade-offs. This chapter discusses the advantages and disadvantages of using SystemVerilog for functional verification. We’ll look at the following topics:

- An abbreviated comparison of the languages and libraries available for functional verification
- Why SystemVerilog is an appropriate choice for verification
Chapter 2: Why SystemVerilog?

- The disadvantages of using SystemVerilog

Overview

Coding for functional verification can be separated into two parts. One is the generic programming part, and the other is the chip testing part. The generic part includes writing structures, functions, and interactions, using techniques such as OOP to manage complexity. The chip testing part includes connecting to the chip, running many threads, and managing random variables.

The generic programming part becomes more and more crucial as the complexity of the hardware to be tested grows. While the problem of connecting to a more complex chip tends to grow only linearly, the overall problem of dealing with this increased complexity grows exponentially.

The authors believe the generic part of programming is served reasonably by SystemVerilog. The language’s features and expressive capabilities make it usable for functional verification. As will be discussed in detail in later sections, the downside is that the language is immature, and compliance from one simulator to the next is inconsistent.

While SystemVerilog might be a little rough around the edges, it is a good way for those who are mainly hardware oriented to learn OOP. As with Verilog, threading is built in, and connection to the chip is relatively well thought out. Realize though, that the actual percentage of code devoted to these tasks is small.

These tasks of HDL connection and parallel execution generally increase linearly with the complexity of the chip. In other words, there are more wires to connect, more independent threads to run, more variables to constrain, and so on.

By contrast, it is much more difficult to make the complexity of a chip increase only linearly. So, as a verification system gets bigger, things tend to get out of hand quickly. Our ability to understand a complex verification system is often more important than how we actually connect to the hardware description language (HDL) wires.
So this handbook concentrates on the “How to make the code reasonable” part of programming. Sure, our examples are multithreaded and use virtual interfaces,¹ but the bulk of this handbook is about how to write understandable code.

**SystemVerilog as a Verification Language**

Several attempts have been made to move verification away from HDLs, such as Verilog or VHDL.² An HDL does a good job of spanning design concepts (called the *register transfer level*, or RTL) down to a few primitives that are used in great numbers to implement a design (called the *gate level*). However, HDLs are not adept at “moving up” in abstraction level to handle modern programming techniques. HDLs are concerned with creating silicon, not with programming. Specifically, HDLs do not provide for object-oriented concepts.

SystemVerilog makes a step in this direction, and can be used to verify a chip. However, it is not clear that such a large span of concepts as SystemVerilog tries to cover can be integrated well into a single language. This handbook provides advice and examples that the authors believe will maximize the programming features of the language, while minimizing the “clunky” parts.

Not surprisingly, there are many choices and trade-offs when you choose a verification language. The table on the following page briefly lists the pros and cons of various languages suitable for verification.

---

¹ We talk about virtual interfaces in the next chapter, but for now just know that they are the way to connect HDL wires with testbench OOP code.

² VHSIC (Very High-Speed Integrated Circuit) HDL.
### Chapter 2: Why SystemVerilog?

As we stress repeatedly through this handbook, the team must decide what features of which languages to use, and how. This handbook will show how best to use SystemVerilog’s OOP features.

<table>
<thead>
<tr>
<th>Language</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog, VHDL</td>
<td>Simple, no extra license required</td>
<td>No class concept, no separation of verification and chip concerns</td>
</tr>
<tr>
<td>Cadence Specman “e”</td>
<td>Rich feature set</td>
<td>Effectively proprietary, nonorthogonal language design</td>
</tr>
<tr>
<td>OpenVera</td>
<td>OOP-“like”, better feature set than HDL</td>
<td>Effectively proprietary, interpreted, lacking full OOP support</td>
</tr>
<tr>
<td>SystemVerilog</td>
<td>IEEE standard, OOP features, one simulator does HDL and HVL, C interface</td>
<td>Covers all aspects from gates to OOP, implementation compliance is weak, language is large, yet lacking full OOP support</td>
</tr>
<tr>
<td>SystemC (C++)</td>
<td>Mature language, open source, most often does not need a simulator</td>
<td>Big footprint, focus is on modeling, heavy use of templating, coverage and constraint system dominates coding, long compile times, clumsy connection to HDL</td>
</tr>
<tr>
<td>Teal/Truss (C++ form)</td>
<td>Mature language, good use of C++, open source, few source files</td>
<td>Not a product, no inherent automatic garbage collection</td>
</tr>
<tr>
<td>Homegrown PLI/C</td>
<td>Free, well known</td>
<td>Not usually multithreaded, usually called from HDL as a utility function</td>
</tr>
</tbody>
</table>
Main Benefits of Using SystemVerilog

A major benefit of SystemVerilog is that it provides a relatively painless introduction to OOP, allowing you to use as little or much of OOP as you feel comfortable with. To this end, SystemVerilog allows the concept of “code interface” versus “implementation,” allowing someone reusing code to concentrate on the features the code provides, not on how the code is actually implemented.

SystemVerilog is well-marketed, with several books and experts. (A quick web search for “SystemVerilog” yielded over 365,000 references.) The language is a good stepping stone from Verilog to OOP, reusing a fair amount of the Verilog syntax.

Furthermore, SystemVerilog vendors are developing useful debugging tools, and because SystemVerilog can coexist with Verilog and VHDL, existing HDL code can be integrated easily.

Many companies have behavioral c-models of their core algorithms. For models with a simple integral interface, the DPI\(^1\) can be used to run the code in SystemVerilog. Note that the current compliance and feature set are spotty, so be prepared that you may have to rewrite the code in SystemVerilog.

SystemVerilog allows a clean separation between HDL and OOP concerns. As will be explained further in the next chapter, the use of the virtual interface feature, along with new keywords such as class and local, can be used to support the OOP concerns.

Drawbacks of Using SystemVerilog

While there are many benefits to using SystemVerilog, there are naturally drawbacks as in any language. One drawback is that, by itself, SystemVerilog is not a solution. Even with the open-source verification libraries of Teal and Truss, you have to write code in a new language.

\(^1\) Direct Programming Interface—SystemVerilog’s API for connecting to C, and by extension to C++.