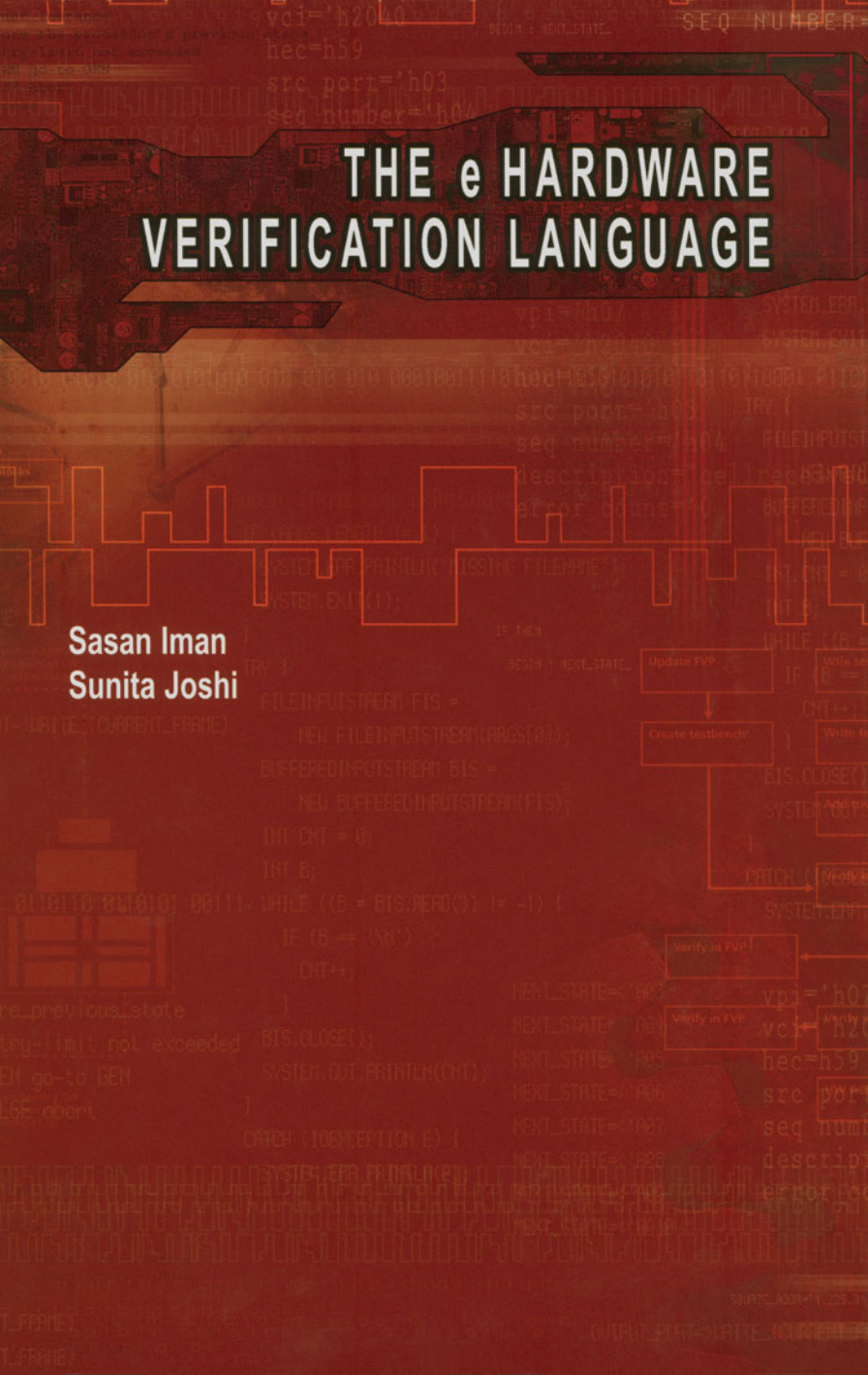


THE e HARDWARE VERIFICATION LANGUAGE

Sasan Iman
Sunita Joshi



The e Hardware Verification Language

This page intentionally left blank

The e Hardware Verification Language

Sasan Iman

SiMantis Inc.
Santa Clara, CA

Sunita Joshi

SiMantis Inc.
Santa Clara, CA

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 1-4020-8024-7
Print ISBN: 1-4020-8023-9

©2004 Springer Science + Business Media, Inc.

Print ©2004 Kluwer Academic Publishers
Boston

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Springer's eBookstore at:
and the Springer Global Website Online at:

<http://www.ebooks.kluweronline.com>
<http://www.springeronline.com>

To our loving families

This page intentionally left blank

Table of Contents

Table of Contents	vii
Reader Feedback	xvii
Foreword	xix
Preface	xxi
Chapter 1: Introduction.....	1
1.1 Design of e	1
1.2 Learning e	2
1.3 Programming in e	4
1.4 Book Structure	5
1.5 Book Conventions and Visual Cues	6
1.6 Summary	7
Part 1: <i>Verification Methodologies and Environment Architecture</i>	9
Chapter 2: Verification Methodologies	11
2.1 Functional Verification	12
2.1.1 Black-Box vs. White-Box Verification	13
2.1.2 Verification Challenges	14
2.1.3 Simulation Based Verification	16
2.1.4 Verification Terminology	18
2.2 Verification Metrics and Verification Quality	18
2.2.1 Granularity	19
2.2.2 Productivity	19

2.2.3 Effectiveness	20
2.2.4 Completeness	20
2.2.5 Verification Environment Reusability	20
2.2.6 Simulation Result Reusability	21
2.3 Directed Test Based Verification	22
2.3.1 Task Driven Verification Methodology	23
2.3.2 Verification Quality	23
2.4 Constrained Random Test Based Verification	24
2.4.1 Random Based Verification and Practical Considerations	26
2.5 Coverage Driven Verification	27
2.5.1 Verification Quality	29
2.6 The Enabling Technology: Hardware Verification Languages	29
2.7 Summary	29
Chapter 3: Anatomy of a Verification Environment	31
3.1 Verification Plan	32
3.1.1 Simulation Goals	33
3.1.2 Verification Views	33
3.1.3 Verification Checks	34
3.1.4 Debugging and the Verification Plan	35
3.2 Verification Environment Architecture	35
3.2.1 CPU Verification	37
3.2.2 Verification Bus Functional Model	38
3.2.2.1 BFM Features	39
3.2.2.2 VBFM User Interface	40
3.2.3 Verification Scenario Generation	41
3.2.3.1 Verification Environment Initialization	41
3.2.3.2 Verification Environment Configuration	41
3.2.3.3 Data and Scenario Generation	42
3.2.4 Monitors	43
3.2.5 Data Collector	45
3.2.6 Data Checking	46
3.3 Module Level vs. System Level Verification	49
3.4 Summary	50
<hr/>	
Part 2: <i>All About e</i>	51
Chapter 4: <i>e</i> as a Programming Language	53
4.1 <i>e</i> Programming Paradigm	53
4.1.1 Declarative Programming	54
4.1.2 Aspect-Oriented Programming	55
4.2 Struct and the Struct Instance Hierarchy	56
4.2.1 Data References	58
4.2.2 global and sys	58

4.3 Execution Flow	60
4.3.1 Merging User Code into the Implicit Execution Order	60
4.3.2 Steps to Writing an e Program	61
4.4 Structure of an e Program	63
4.4.1 Lexical Conventions	63
4.4.2 Code Segments	63
4.4.3 Comments	64
4.5 Statements	64
4.5.1 Import statements	65
4.5.1.1 Import Order Dependency	65
4.5.2 struct Declaration Statement	65
4.5.2.1 Struct Data Members	66
4.5.2.2 Methods	66
4.5.3 Type and Subtype Declaration Statements	67
4.5.3.1 Enumerated Type Declarations	67
4.5.3.2 Scalar Subtype Declarations	68
4.5.3.3 Struct Subtype Declarations	68
4.5.4 Extension Statements	69
4.6 Concurrency and Threads	71
4.6.1 Events and Temporal Expressions	72
4.6.1.1 Temporal Expressions	73
4.6.2 Time Consuming Methods (TCMs)	74
4.6.3 Thread Control	75
4.6.4 Semaphores	78
4.6.4.1 Mutual Exclusion	78
4.6.4.2 Thread Synchronization	79
4.7 Summary	80
Chapter 5: e as a Verification Language.....	81
5.1 Constrained Random Generation	82
5.1.1 Random Generation	83
5.1.2 Generation Constraints	85
5.1.3 Pre-Run vs. On-the-Fly Generation	87
5.2 HDL Simulator Interface	88
5.2.1 Multi-Valued Logic	89
5.3 HDL Simulator Synchronization	90
5.3.1 Notion of Time	90
5.4 Units	91
5.5 e-Ports	93
5.6 Packing and Unpacking	94
5.6.1 Packing	95
5.6.2 Unpacking	97
5.7 Coverage	98
5.8 Summary	101

Part 3: <i>Topology and Stimulus Generation</i>	103
Chapter 6: Generator Operation	105
6.1 Generator Execution Flow	105
6.1.1 pre_generate()	106
6.1.2 post_generate()	107
6.2 Constraint Types	108
6.3 Generation Steps and the Constraint Solver	109
6.3.1 Item Generation Order	109
6.3.2 Reduction	111
6.3.3 Constraint Evaluation	111
6.3.4 Set-Scalar	112
6.4 Controlling the Generation Order	112
6.4.1 when Blocks	114
6.4.2 Explicit Order Definition	114
6.4.3 value()	114
6.5 Generation and Program Execution Flow	115
6.5.1 Static Analysis	115
6.5.2 On-the-Fly Generation	116
6.5.2.1 Data Allocation: new	116
6.5.2.2 Data Generation: gen	118
6.6 Summary	120
Chapter 7: Data Modeling and Stimulus Generation.....	121
7.1 Data Model Fields	122
7.1.1 Physical Fields	122
7.1.2 Determinant Fields	123
7.1.3 Utility Fields	123
7.1.3.1 Avoiding Data Generation Inconsistencies	125
7.2 Data Model Subtypes	126
7.2.1 Field Value Customization	127
7.2.2 Conditional Fields	128
7.2.3 Conditional Fields and Generation Constraints	128
7.3 Data Abstraction Translation	130
7.3.1 Packing: Logical View to Physical View	130
7.3.2 Unpacking: Physical View to Logical View	131
7.3.2.1 Lists	131
7.3.2.2 Subtypes	133
7.4 Data Generation Constraints	134
7.4.1 Abstract Ranges	134
7.4.2 Coordinated Ranges	135
7.4.3 Default Ranges	135
7.5 Summary	136
Chapter 8: Sequence Generation	137

8.1 Verification Scenarios as Sequences	138
8.2 Sequence Generation Architecture	140
8.3 Homogeneous Sequences	142
8.3.1 Verification Environment Enclosing a Sequence Generator	143
8.3.2 Verification Item Definition	144
8.3.3 Driver and Sequence Creation	144
8.3.4 Verification Environment Attachment	145
8.3.5 User Defined Sequences	146
8.3.5.1 Flat Sequences	146
8.3.5.2 Hierarchical Sequences	147
8.3.6 Default Sequence Generation Starting Point	148
8.3.6.1 Merging New Sequence Kind with the Default Start Point	149
8.3.6.2 Over-riding the Default Start Point to a New Sequence Kind	149
8.3.7 Sequence Generator Flow Customization	149
8.4 Sequence Synchronization	151
8.4.1 Sequence and Sequence Driver Interaction	152
8.4.1.1 Push Mode	153
8.4.1.2 Pull Mode	154
8.4.2 Multiple Sequence Synchronization	154
8.5 Heterogeneous Sequences	155
8.5.1 Implementation Using Virtual BFMs	155
8.5.2 Implementation Using Virtual Drivers	156
8.6 Summary	161

Part 4: Response Collection, Data Checking, and Property Monitoring 163

Chapter 9: Temporal Expressions.....	165
9.1 Temporal Expression Basics	165
9.2 Temporal Expression Evaluation	167
9.2.1 Evaluation Abstract Model	167
9.2.2 Sequence Temporal Operator	169
9.2.3 Evaluation Threads and Program Context	170
9.2.4 Detach Operator	171
9.2.5 exec Construct	172
9.3 Temporal Operators	172
9.3.1 Base Temporal Operators	173
9.3.2 Atomic Temporal Operators	173
9.3.2.1 fail	173
9.3.2.2 and	173
9.3.2.3 or	174
9.3.2.4 First Match Variable Repeat [from..to]	175
9.3.3 Composite Temporal Operators	176
9.3.3.1 not	176
9.3.3.2 Fixed Repetition	176

9.3.3.3 True Match Variable Repeat	176
9.3.3.4 Yield	177
9.3.3.5 eventually	177
9.4 Temporal Operator Arithmetic	177
9.5 Temporals Dictionary	179
9.5.1 English Phrases and event Definitions	179
9.5.2 English Phrases and Property Checking:	183
9.6 Performance Issues	184
9.6.1 Over-sampling	185
9.6.2 Missing Sampling Event	186
9.6.3 Unanchored Sequences	186
9.6.4 Nested Sampling	187
9.7 Summary	188
Chapter 10: Messages	189
10.1 Messaging Strategy	190
10.2 Message Actions	191
10.2.1 Message Tags	192
10.2.2 Verbosity	193
10.2.3 Format Type	193
10.2.4 Action Block	193
10.3 Message Loggers	194
10.3.1 sys.logger	194
10.3.2 Message Handling Using Loggers	195
10.3.3 Configuring Loggers	196
10.3.3.1 Using Commands	196
10.3.3.2 Using Methods	197
10.3.3.3 Using Constraints	197
10.4 Summary	198
Chapter 11: Collectors and Monitors	199
11.1 Monitor Architecture	199
11.2 Protocol Checking	201
11.2.1 Protocol Checks	202
11.2.1.1 Monitor Events	203
11.2.1.2 Temporal Struct Members	203
11.2.1.3 Protocol Checking Reports	204
11.2.2 Protocol Checker Activation	204
11.2.2.1 Static Checker Activation	204
11.2.2.2 Dynamic Checker Activation	205
11.3 Collection and Reporting	205
11.3.1 Data and Transaction Collection	205
11.3.1.1 Sending collected data to scoreboard:	207
11.3.1.2 Sending to a Checker	208
11.3.1.3 Sending to a File	208

11.3.2 Event Extraction	209
11.3.3 Reporting	210
11.3.3.1 Messages	210
11.3.3.2 Message Loggers	210
11.4 Summary	211
Chapter 12: Scoreboarding	213
12.1 Scoreboard Implementation	214
12.2 Scoreboard Configuration Types	215
12.2.1 Driver/BFM Based Scoreboard	216
12.2.2 Monitor Based Scoreboard	216
12.3 Attaching Scoreboards to the Environment	217
12.3.1 Direct Method Call	217
12.3.2 Using Hook Methods	218
12.3.3 Using Events	219
12.4 Scoreboarding Strategies	220
12.4.1 End-to-end Scoreboarding	220
12.4.2 Multistep Scoreboarding	220
12.5 Summary	222
<hr/>	
Part 5: Coverage Modeling and Measurement	223
Chapter 13: Coverage Engine.....	225
13.1 Coverage Collection Steps	225
13.2 Coverage Terminologies	226
13.3 Scalar Coverage Constructs	227
13.3.1 Coverage Groups	227
13.3.2 Basic Coverage Items	229
13.3.3 Sampling Events	231
13.3.4 Coverage Buckets	232
13.3.4.1 Bucket Ranges	233
13.3.4.2 Default Buckets	235
13.3.4.3 Illegal Buckets	235
13.3.4.4 Ignored Buckets	236
13.3.4.5 Bucket Grading	236
13.4 Composite Coverage Items	236
13.4.1 Cross Coverage Items	237
13.4.2 Transition Coverage Items	238
13.5 Coverage Extension	239
13.5.1 Coverage Group Extension	240
13.5.2 Coverage Item Extension	240
13.6 Minimizing Coverage Collection Overhead	241
13.7 Summary	243
Chapter 14: Coverage Modeling	245

14.1 Coverage Planning and Design	246
14.2 Coverage Implementation	247
14.2.1 Coverage Model Organization	247
14.2.1.1 Hierarchical Coverage Models	248
14.2.1.2 Multi-dimensional Coverage Models	250
14.2.2 Coverage Data Source	251
14.2.2.1 DUV Signal Coverage	252
14.2.2.2 State Machine Coverage	253
14.2.2.3 Coverage of Generated Data	255
14.3 Coverage Grading	256
14.3.1 Changing Default Weights	258
14.3.2 Changing Default Goals	259
14.3.3 Ungradeable Items	259
14.3.4 Illegal and Ignored Items	261
14.4 Coverage Analysis	261
14.5 Summary	262

Part 6: *e Code Reuse* 265

Chapter 15: <i>e Reuse Methodology</i>	267
15.1 eVCs: <i>e Verification Components</i>	268
15.2 Packages and Package Libraries	269
15.2.1 Naming Conventions	270
15.2.2 Directory Structure	271
15.2.3 Accessing Files	272
15.2.4 LIBRARY_README.txt File	273
15.2.5 PACKAGE_README.txt File	273
15.2.6 any_env unit	274
15.3 Features	274
15.3.1 eVC Environment	276
15.3.2 eVC Agents	276
15.3.3 Configuration Settings	277
15.3.4 Sequence Generator and Driver	277
15.3.5 e-Port Interface	277
15.3.6 BFM	277
15.3.7 Monitor	277
15.4 Summary	278
Chapter 16: <i>si_util Package</i>	279
16.1 Stop-Run Controller	280
16.1.1 Stop-Run Controller and Stop-Run Interface	281
16.1.2 Migrating to Using Stop-Run Interfaces	281
16.1.3 Multiple Stop-Run Groups in the Same Module	283
16.1.4 Multiple Stop-Run Groups across the Hierarchy	285

16.1.5 Modular Stop-Run Control	286
16.2 Memory Package	288
16.2.1 si_util_mem_mgr Memory Manager	288
16.2.1.1 Memory Segment Placement Style	290
16.2.2 si_util_mem Sparse e Memory Core	291
16.3 Native e Time Manager	293
16.4 Signal Generator	295
16.5 Native e Float Arithmetic Package	299
16.6 Summary	301

Part 7: Appendices	303
---------------------------	------------

e BNF Grammar 305

e Reserved Keywords 331

eRM Compliance Checks 333

C.1 Packaging and Name Space Compliance Checks	334
C.2 Architecture Compliance Checks	336
C.3 Reset Compliance Checks	336
C.4 Checking Compliance Checks	338
C.5 Coverage Compliance Checks	338
C.6 Sequences Compliance Checks	339
C.7 Messaging Compliance Checks	339
C.8 Monitor Compliance Checks	340
C.9 Documentation Compliance Checks	340
C.10 General Deliverables Compliance Checks	342
C.11 End of Test Compliance Checks	342

Index	343
-------	-----

This page intentionally left blank

Reader Feedback

Functional verification is a dynamic and fast growing field where verification methodologies are continuously enhanced and improved. At the same time, the ϵ language is in the process of being adopted as IEEE standard 1647 where its syntactical and semantic features will be summarized and detailed in a language reference manual. In consideration of these dynamics, this book will continue to be updated to reflect the ongoing events in the verification community. We encourage you, as the reader, to help enhance this book by sending us your feedback on topics that you feel should be explained more concisely or more clearly; and we welcome comments on errors or inconsistencies that we may have overlooked in this first edition. Please send us your feedback via E-mail to **theehvl@simantis.com**.

This page intentionally left blank

Foreword

I am glad to see this new book on the *e* language and on verification. I am especially glad to see a description of the *e* Reuse Methodology (eRM). The main goal of verification is, after all, finding more bugs quicker using given resources, and verification reuse (module-to-system, old-system-to-new-system etc.) is a key enabling component.

This book offers a fresh approach in teaching the *e* hardware verification language within the context of coverage driven verification methodology. I hope it will help the reader understand the many important and interesting topics surrounding hardware verification.

Yoav Hollander

Founder and CTO, Verisity Inc.

This page intentionally left blank

Preface

This book provides a detailed coverage of the *e* hardware verification language (HVL), state of the art verification methodologies, and the use of *e* HVL as a facilitating verification tool in implementing a state of the art verification environment. It includes comprehensive descriptions of the new concepts introduced by the *e* language, *e* language syntax, and its associated semantics. This book also describes the architectural views and requirements of verification environments (randomly generated environments, coverage driven verification environments, etc.), verification blocks in the architectural views (i.e. generators, initiators, collectors, checkers, monitors, coverage definitions, etc.) and their implementations using the *e* HVL. Moreover, the *e* Reuse Methodology (eRM), the motivation for defining such a guideline, and step-by-step instructions for building an eRM compliant *e* Verification Component (eVC) are also discussed.

This book is intended for a wide range of users, including junior verification engineers looking to learn basic concepts and syntax for their project, to advance users looking to enhance the effectiveness and quality of a verification environment, to developers working to build eVCs, and also as a reference work for users seeking specific information about a verification concept and its implementation using the *e* HVL.

Acknowledgements

The *e* hardware verification language could not have existed without the creativity, hard work, and diligence of people at Verisity Inc. Verification methodologies as discussed in this book were either created or impacted by Verisity's continuous effort to address the functional verification challenge. We are grateful to Verisity and its employees for their hard work to create *e* and to keep it at the leading edge of verification technologies.

In the writing of this book, we have been fortunate to be supported by great engineers and technologists who have provided us with comments, feedbacks, and reviews of the material herein. We would like to acknowledge the following individuals for their active participation in the review process: Corey Goss, Michael McNamara, Kumar Malhotra, Michael McNamara, Patrick Oury, Andrew Piziali, Al Scalise, Efrat Shneydor, Mark Strickland, and Hari Tirumalai.

Sasan Iman

Sunita Joshi

Santa Clara, CA

1.1 Design of e

The *e* hardware verification language is designed to support the special requirements of functional verification. *e* provides abstractions that are specifically targeted to better implementations of functional verification concepts. Creating a language that provides native support for verification concepts gives the following benefits:

- Less code for engineers to write, which leads to higher productivity when implementing a verification environment
- Less code also means fewer errors in building the verification environment because the number of errors in a program is proportional to the lines of code in that program
- Greater runtime efficiency as verification abstractions are optimized in the language runtime engine
- Provides an easy interface with HDL simulators

The *e* programming language was created, supported, and enhanced by Verisity Inc. of Mountain View, CA. The Specman Elite® tool suite is also produced by Verisity and provides the runtime and development environments for the *e* language. Specman Elite provides the necessary utilities for writing, debugging, integrating, reporting, and configuring *e* programs. Although Specman Elite and the *e* language are closely tied, the *e* language exists independently of this runtime environment. Therefore, this book mainly focuses on verification tasks and their implementation using the *e* language independent of Specman Elite®. The implicit assumption is that an *e* program should be able to run in any runtime environment developed for the language and should not depend on specific implementations of any runtime environment. Consequently, this book does not describe Specman Elite in any detail.

As of this writing, *e* is in the process of standardization as IEEE P1647. The goal of this standardization process is to develop a standard verification language based on the *e* language, by clearly specifying:

- The *e* language constructs
- The *e* language interaction with standard simulation languages of interest
- The libraries currently used in conjunction with *e*
- New features of interest

Once this process is completed, the language grammar and a language reference manual will be produced by the working group for this standard. The latest status of this standardization effort can be obtained by visiting www.ieee1647.org.

1.2 Learning e

Programming languages are not the end but a means to more efficient implementations of concepts. We learn a new programming language not because we find new language constructs and syntaxes fascinating, but because they are useful in making us become better problem solvers. Thus, the first step in learning a new language is to understand the underlying concepts that first motivated the development of that language. These concepts are rooted in the problem that a language was created to address as well as the methodology that is the approach of choice for solving such problems as a conceptual level.

Functional verification is the specific problem *e* is designed to address. Methodologies for performing functional verification have evolved over the years in order to meet the verification demands of increasingly complex systems. Detailed understanding of the latest functional verification methodologies is essential to learning the *e* language.

Before a methodology can be implemented in a programming language, it has to be represented with an architectural view. This architectural view consists of components, modules, and tasks that have direct correspondence to constructs and abstractions provided in that language.

This book describes the evolution of functional verification methodologies culminating in the development of coverage driven verification methodology (chapter 2). The architectural view for effective application of this methodology and its components are described in chapter 3. As previously stated, in most cases, an architectural view is created with the specific facilities of a language in mind. In the case of *e*, language features and facilities were motivated by the architecture that leads to the best implementations of coverage driven verification methodology. Before learning the *e* language, it is important to gain a good understanding of coverage driven methodology and the architectural view for its verification environment. Such a detailed understanding promotes the learning to become an expert in using *e* to build a robust and complete verification environment rather than mere proficiency in the *e* language.

New languages are designed to support new, or a mix of new and existing programming paradigms. A programming paradigm describes the view that a programmer has of program execution. For example, in object-oriented programming, a program is viewed as a collection of communicating objects. In imperative programming, a program is viewed as a state and instructions that change this state. Learning how a new language is used is more involved than just learning the syntax of that language. A good part of becoming an expert in a programming language is understanding how to best implement the solution to a problem using the programming paradigms supported by that language. Programmers often learn the syntax of a new language only to use this new knowledge to implement a program using concepts from their previous programming experiences. For example: for someone already familiar with the BASIC programming language, an important step in learning C++ is to learn object-oriented programming. It is easy to learn C++ syntax and use it to write a program in the style of BASIC. But doing so would defeat the purpose behind the creation of C++ and would render learning C++ a futile effort.

Effective functional verification requires a mix of different programming paradigms. *e* supports imperative, object oriented, aspect oriented, and declarative programming paradigms. It is important to learn these programming paradigms and understand the proper use of utilities supporting these paradigms. Having a clear understanding of the usefulness of each paradigm to each verification task leads to a better verification program that takes full advantage of these paradigms. Chapter 4 describes *e* as a programming language and covers these programming paradigms.

Still, a language cannot be learned completely in a short time. Becoming an expert in a new language, its programming paradigms, and appropriate methodologies requires practice, and a conscious effort to expand beyond the subset that you know at any given time. The best approach to learning *e* is to focus on the verification tasks and language programming paradigms rather than the language constructs. You should start by building the smallest and most trivial verification environment on your own. I often come across verification engineers who in spite of having one or more years of experience in *e*, are unable to write the most basic program in *e* simply because their job only required them to learn a subset of the language and to work on an already existing program. Learning *e* is best accomplished by building a skeleton *e* program for a verification environment, and enhancing the components in this verification environment by learning more about how each component is designed and operated.

Functional verification has now become a major part of any design project consisting of multiple verification developers. Such developers will create code, but often also use code from other vendors or previous projects. As such, software programming concepts designed to facilitate code reuse and multiple team development efforts are an important part of building a verification environment. Solid understanding of the *e* Reuse Methodology (eRM) helps in building *e* verification components that can be reused and combined with components developed by other teams. eRM is therefore especially important in working within larger verification environment development projects.

A systematic approach to learning the *e* language includes the following steps:

- Learn functional verification methodologies especially coverage driven verification.
- Understand the recommended verification environment architecture for performing coverage driven verification.
- Understand programming paradigms supported by the *e* language, and their usefulness for different verification tasks.
- Learn eRM to understand how to structure software so that it is reusable and can be used combined other independently developed modules.
- Learn to build the most basic verification environment from scratch, even if all its components have no content.
- Learn the underlying operation of different language facilities (i.e. random generation, temporal expressions, sequences, etc.). Once you know how these facilities work and what operations they support, the appropriate syntax can be referenced and remembered as you become more familiar with the language.
- Focus first on just enough syntax so that you can build a syntactically correct program to create the desired module hierarchy and method calls.

1.3 Programming in e

Program development consists of 4 main stages:

- Logistic Planning
- Analysis
- Design
- Implementation

Logistic planning is concerned with software organization issues such as directory structure, naming conventions, future plans for reuse across projects generations, etc. In the analysis stage, establish a clear understanding of the problem that needs to be solved. During the design stage, key concepts for a solution are identified. During the implementation phase, the solution identified in the design stage is implemented in a program.

Issues discussed in the *e* Reuse Methodology guidelines (eRM) should be used to plan the logistics of a verification project. Issues that must be considered during this phase include:

- Software Directory Structure
- Packaging Information and Documentation
- Naming Conventions

Use the analysis phase to gain a good understanding of the problem. In the case of functional verification, the problem is usually solved using one of the verification methodologies discussed in chapter 2. Selecting the best methodology, however, depends on the specific requirements of the verification task and can be determined by comparing the advantages and disadvantages of the different methodologies discussed in this chapter. Given any non-trivial verification task, a coverage driven verification methodology is recommended.

The design phase corresponds to creating an architectural view of the selected methodology. The architectural view for a verification project is discussed in chapter 3. The implementation phase consists of building the modules in the verification environment architecture so that the desired behavior is achieved.

The following general guidelines should be used when building a verification environment in the *e* language:

- Break up the activities in the verification environment according to the architectural representation described in chapter 3.
- Model verification environment modules using the **unit** construct.
- Model the interface between these modules as abstract port definitions using **e-Ports**.
- Model abstract data types using the **struct** construct.
- Organize the program implementation into two dimensions: modules and aspects. Modules define the core implementation of each module. Aspects span modules and define properties or functions of these core implementations that can be changed or enhanced.
- Implement environment modules in the following order:
 - Stimulus Generators
 - Collectors and Monitors
 - Protocol and Data Checking
 - Coverage Collection
- Follow eRM guidelines for integrating existing *e* verification components (eVCs) into the verification environment.

1.4 Book Structure

This book consists of 7 parts:

- Part I: Describes functional verification methodologies and environment architecture independent of any programming language. Topics discussed will be used to motivate the features of the *e* language (chapters 2 and 3).
- Part II: Introduces *e* as a programming and verification language. The descriptions in these chapters are meant to give reader a comprehensive understanding of features of the *e* language (chapters 4 and 5).
- Part III: Describes the operation of the constrained random generation utility in *e*, and discusses details of creating the verification environment, stimulus generation, and verification scenario generation (chapters 6, 7, and 8).
- Part IV: Describes the details of temporal expressions and messages, and the architecture and implementation of monitors, collectors, data checkers, and protocol checkers (chapters 9, 10, 11, and 12).
- Part V: Describes the concepts of coverage collection and issues related to coverage collection and coverage analysis (chapters 13 and 14).
- Part VI: Describes the *e* Reuse Methodology (eRM) and covers in detail the contents of

si_util utility package (chapters 15 and 16).

- Appendices: Describes the grammar for the *e* language in the BNF format. Also gives the list of *e* keywords, and provides checklists for following the eRM guidelines (Appendices A, B, and C).

1.5 Book Conventions and Visual Cues

The visual cues in this book are shown in the following table. These conventions are used to enhance your understanding of the material. *e* keywords are shown using “times bold” typeset to prevent confusion when these terms are included in the main text flow. For example writing “the is also extension mechanism can be used” leads to ambiguities which are clarified when the appropriate keyword typeset is used as shown in the following: “the **is also** extension mechanism can also be used.”

New terms are indicated with the appropriate typeset wherever a definition for these terms are given. By providing a different typeset for new terms, these terms can easily be located and their definition found.

Visual Cue	Description
Times	Book text
Arial	<i>e</i> program text
Arial Bold	<i>e</i> program text in book descriptive text
Times Bold	<i>e</i> keywords
<i>Times Italic Underlined</i>	NewTerms

Sample *e* programs are shown either as *e* program fragments or as *e* programs. An *e* program refers to a code segment that can be compiled by an *e* compiler as shown in the text. The following is an example of an *e* program:

```
1 : <'
2 :   extend sys {
3 :     d: uint;
4 :   };
5 : >
```

e Programs are indicated by the solid lines before and after the example body. Code fragments refer to sample code that cannot be loaded into an *e* compiler without adding missing parts of the program. The following shows an example of a code fragment:

```

      :
      :
1   : struct data {
2   :     payload: uint;
3   :     is_legal: bool;
4   : };
      :
      :
```

1.6 Summary

This chapter provided an overview on how this book is best used for learning the *e* language, and how to use *e* to solve your verification problems. While reading this book, your first focus should be on understanding the methodologies and the programming paradigms that make *e* the language of choice for building your verification environment. Your knowledge of syntax will grow as you gain more experience in writing verification programs and building verification environments. As you learn *e*, keep in mind that the full power of a programming language is unleashed only when used as intended.

This page intentionally left blank

PART 1

Verification Methodologies and Environment Architecture

This page intentionally left blank

Verification Methodologies

In recent years, significant advances in chip and system fabrication technologies have afforded designers the ability to implement digital systems with ever increasing complexity. This trend has led to a productivity gap between design methodologies and implementation technologies where because of this gap, designs produced by existing methodologies do not yield enough gates per engineer per month to meet the strict time-to-market deadlines inherent in competitive markets. The realization that functional verification now consumes anywhere between 50% to 70% of the design cycle, has brought functional verification center stage in the effort led by the design community to close this productivity gap.

The focus on functional verification has consisted of a multi-faceted approach where verification productivity improvements are made possible through introduction of:

- New verification methodologies better suited for the verification of complex systems
- Reusable verification components
- Hardware Verification languages to facilitate the new verification paradigms

Verification methodologies and reusable components are abstract concepts that lead to verification productivity improvements, while new hardware verification languages are the enabling technology for these new concepts. This chapter presents the basics of functional verification and suggests metrics for measuring the quality of a verification methodology. It then presents the evolution of verification methodologies from task based verification to coverage driven constrained random based verification and beyond. The verification metrics introduced in this chapter will be used to motivate the introduction of new verification methodologies.

2.1 Functional Verification

The typical design flow and its associated design and verification tasks are shown in figure 2.1. All design activity starts from the design specification. The task of a design team is to create a hardware implementation through their interpretation of the design specification. This initial implementation is translated into the final implementation through a series of steps where appropriate design automation tools are used to move from one level of abstraction to the next at each step. For example in an ASIC design flow starting at the RT level, the initial RTL implementation is created by the design team from the design specification. The RTL implementation is then synthesized into a netlist targeting the target implementation technology. This netlist is then processed through physical design stages to prepare the final mask information used for semiconductor fabrication.

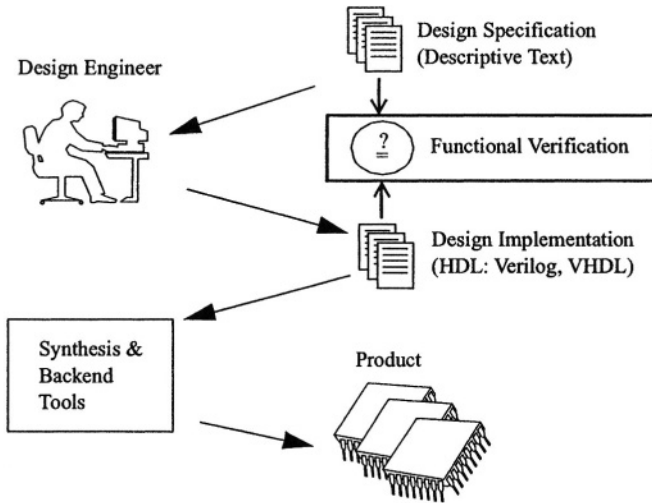


Figure 2.1 Product Design Flow

Perhaps the most tedious and error prone step in the design flow is the manual translation of the design specification into the initial design implementation. Generally speaking, other design tasks are less prone to functional errors because of the high degree of automation involved in performing these later steps. The main reasons for functional errors in a design are:

- Ambiguities in the design specification
- Misunderstandings by the designer even when the specification is clear
- Implementation errors even when the implementation goal is clear

The primary goal of *Functional Verification* is to verify that the initial design implementation is functionally equivalent to the design specification.

2.1.1 Black-Box vs. White-Box Verification

From a functional verification point of view, it is sufficient to verify a design implementation by checking its behavior on its boundary (i.e. input and output ports). If a property of a device cannot be verified through its ports, then that property is either not controllable (i.e. cannot be activated), or not observable. This verification approach is called *Black-Box verification*. The diagram in figure 2.2 shows the verification architecture for performing black-box verification. Note that in this approach, a reference model is required to check that the response generated by the device is in fact the expected response.

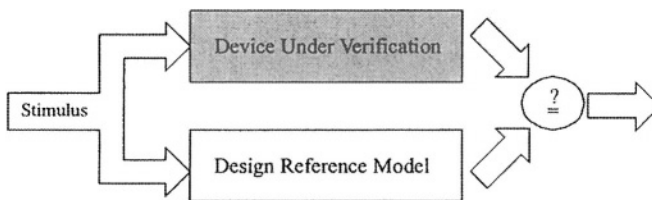


Figure 2.2 Black-Box Verification

The features that give a design implementation its performance edge are often not visible through device ports. For example, in an instruction accurate CPU verification environment, a CPU with an instruction pipeline would functionally seem to behave the same as a CPU without an instruction pipeline. To verify the correct operation of this instruction pipeline, it is necessary to examine the design and monitor the instruction pipeline behavior and verify that it is providing the performance improvement that the CPU designers intended. In another example, consider the settings for FIFO thresholds that are used to trigger FIFO read or write operations. Correct operation of such threshold settings would be very hard to verify without visibility into the device. Even though it is possible to create a cycle accurate reference model to check for such internal behaviors through the device ports, the effort associated with building such an accurate model would make black-box verification impractical in most instances.

Even in cases where a specific feature may be verified through device ports, the difference between the time the bug is activated and the time that bug is observed would make any causality analyses difficult to track. To enhance debugability, introduce monitors that track internal properties with the potential to become sources of device malfunctions.

White-Box and Gray-Box verification approaches are two alternative approaches that overcome limitations of black-box verification. *White-Box Verification* (figure 2.3) refers to verifying a design through checkers (assertion checkers and monitors) without using a reference design. *Gray-Box Verification* (figure 2.3) refers to verifying a design by using a reference model but using checkers (assertions checkers and monitors) to improve verification productivity.

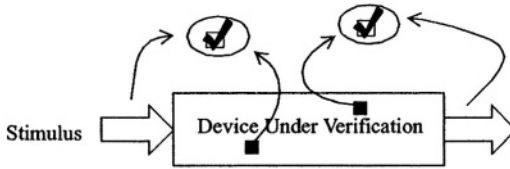


Figure 2.3 White-Box Verification

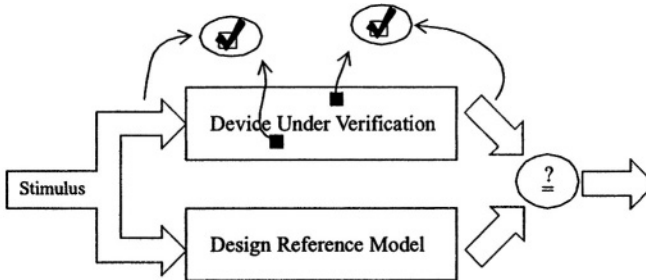


Figure 2.4 Gray-Box Verification

White-box functional verification is usually used for smaller modules in the early stages of the design process. This verification approach is hard to reuse and hard to manage. On the other hand, Black-box testing is easy to reuse as the verification project moves from module level to system level. However, the long latency for detecting bugs, the potential to miss critical internal state bugs, and the need to build a detailed reference model make Black-box verification difficult to use in practice. Gray-box testing allows the verification engineer to strike the right balance between design property checking and building a reference model. Therefore, gray-box verification is the approach that provides the most benefit throughout the verification flow.

2.1.2 Verification Challenges

As digital devices get more complex and time-to-market requirements become shorter, extra pressure is placed on verification engineers to complete exponentially more complex verification projects in shorter time periods. A number of challenges must be addressed in order to deal with the increasing complexity of successfully completing a functional verification project. These challenges are:

- Productivity
- Efficiency

- Reusability
- Completeness

Verification Productivity is defined as the ability to handle larger designs in a shorter time. For design engineers, these productivity gains have been made possible by moving from transistor level design to gate level design to RT and system level design methodologies. By designing at the RT level, design engineers can plan much larger circuits than if they were designing using discrete transistors. The same type of productivity gains must be afforded to verification engineers to allow them to deal with increasingly larger devices. Such productivity gains are achieved by moving to higher levels of abstraction both in terms of verification utilities and functional blocks that are being verified. New methodologies taking advantage of such new abstractions should be introduced so that the gap between design and verification productivity can be bridged.

Verification Efficiency is a measure of human intervention required to complete a verification task. With the increasing complexity of devices, it is desirable to reduce manual intervention or manual handling to as little as possible. Verification efficiency is increased by means of automation in the verification environment and through introduction of verification tool utilities, which with appropriate verification methodology, lead to a reduction of manual intervention.

Verification Reusability refers to the ability to reuse an existing verification environment, or pieces of an existing verification environment for new projects or later generations of the same project. Reusability is addressed by developing a modular architecture for the verification environment where modules boundaries are identified as pieces that are reuse-candidates in new projects. Additional gains can be made through better documentation of the verification architecture and the use of software programming and maintenance techniques available that simplify code enhancement

Verification Completeness is the goal to cover as much of the design functionality as possible. Improving verification productivity, efficiency, and reusability will provide more time to focus on improving verification completeness. It is also possible to further improve completeness by introducing verification methodologies that focus on this issue and facilities that give greater visibility to verification progress.

Clearly, addressing these challenges effectively is only possible through a comprehensive and multi-faceted approach. In short, these verification challenges must be addressed through:

- New verification methodologies
- Moving to higher levels of abstraction
- Using modular design techniques
- Measuring verification progress
- Reducing manual effort through automation
- Improved software development environment
- Better documentation
- Using software development techniques
- New verification language features (utilities) to facilitate these new methodology and

software development requirements

2.1.3 Simulation Based Verification

The fundamental operation in simulation based verification is the process of device state activation followed by device response checking (figure 2.5). In this operation, the device is placed in a specific state and correct device response at that state is checked. Note that in this representation, a device state may refer to a simple device state as defined by the contents of its registers and status of its state machines. A state in this representation may also represent an operating mode of the device where the device may go through many simple states while operating in that mode. For example, in a state machine, the device state in this diagram may refer to a specific state of the state machine, while for a bus interface module, a state may refer to write or read operation cycles for the interface module.



Figure 2.5 Simulation Based Verification

All functional verification requirements can be described as a series of such fundamental operations. Figure 2.6 shows a conceptual representation of the verification space for a given device. This representation describes the relationship between simulation steps in terms of the stimulus required to move the device into a given state.

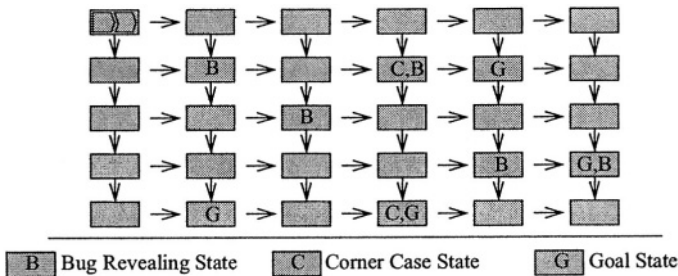


Figure 2.6 Device Verification Space

At the beginning of a verification project, the verification team establishes *Goal States* in the verification space that must be reached sometime during the simulation run. The aim of verification would be to generate the necessary stimulus to put the device into these goal states. As the device moves into new states on its way to the goal state, checkers guarantee the correct device response at each step. Because of the abstract definition of a device state where each state may also represent a device operating mode, it is possible to define a hierarchical verifica-

tion space where at the higher levels, such a diagram describes correct device operation as it moves from one mode of operation to another, while at lower levels, such a diagram represents correct device operation at the most detailed level.

Consider a state machine and its corresponding verification space shown in figure 2.7. In this representation, each state in the verification space corresponds to a state in the original state machine. In addition, the necessary check at each step is that the current state is valid based on previous state and the input values, and that the output generated at that state is as expected.

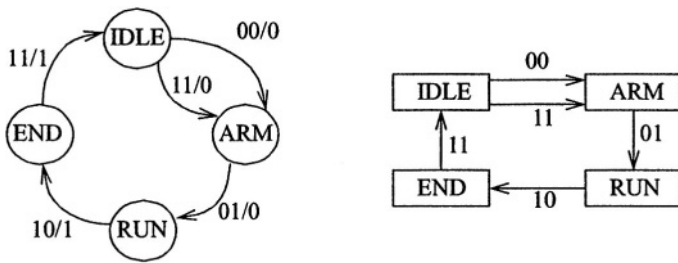


Figure 2.7 FSM Verification Space

The verification space for a bus interface module, shown in figure 2.8, gives verification targets in terms of modes of operation for the bus interface module.

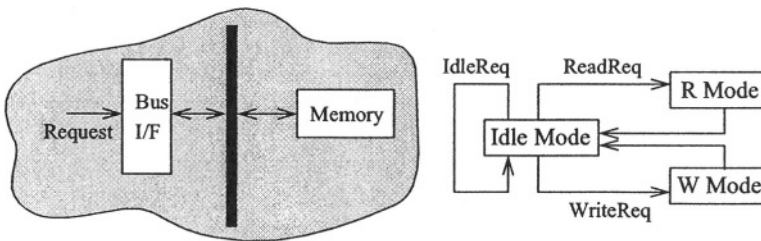


Figure 2.8 Bus Interface Verification Space

It should be noted that verification goals often depend not only on reaching a specific state, but also on how that verification target is reached. For example, in the finite state machine shown in figure 2.7, it is possible to reach state **ARM** through two different input combinations. Measuring verification progress therefore requires that not only target states are considered but also how these target states were reached.

2.1.4 Verification Terminology

The verification space as described in the previous section, is used to define the terminology for functional verification.

Each interesting scenario that should be verified is a *Verification Scenario (VS)*. A verification scenario is described in terms of a sequence of verification state traversals in the verification space. A *Verification Item* is the verification step (i.e. simulation run) that verifies the correctness of one or more verification scenarios. This can happen if a simulation run traverses a path in the verification space that spans multiple verification scenarios. The collection of verification scenarios form the *Verification Plan (VP)*. The *Verification Suite* is the collection of verification items that verifies all scenarios in a verification plan. The design implementation that is being verified is the *Device Under Verification (DUV)*. The *Verification Environment (VE)* (i.e. *Verification Bench*) is the collection of DUV and all verification related constructs.

In the context of verification environment development, *Physical Level* refers to signal descriptions at the bit and bit vector levels. Physical views are used to describe DUV at the hardware level. *Logical View* refers to any abstracted view in the design or environment. A logical view of data traffic may correspond to the data frame representation of physical level values. A logical view of a DUV may correspond to its user interface at a task level (i.e. write to device, read from device). *Physical Interface* refers to ports that are described at the physical level. *Logical Interface* refers to port interfaces that are described at a logical level.

2.2 Verification Metrics and Verification Quality

Verification methodologies are adopted according to their benefit. It is therefore necessary to identify a number of metrics that will be used to measure the effectiveness of a verification methodology. This section presents a number of verification metrics that will be used throughout this chapter to discuss the merits of different verification methodologies. The metrics presented in this section are closely related to the verification challenges discussed in section 2.1.2. Some of these metrics are measured quantitatively while others are used as qualitative guidelines for discussing merits of verification methodologies.

The following metrics are considered in measuring the value of a verification methodology:

- Granularity
- Productivity
- Effectiveness
- Completeness
- Reusability of the Verification Environment
- Reusability of the Simulation Data

Verification Quality refers to the combination of these verification metrics.

2.2.1 Granularity

Verification Granularity is a qualitative metric used to measure the degree of detail (i.e. granularity) that should be considered in crafting a verification plan, and subsequently completing the verification project. The ultimate goal is to allow verification engineers to deal with verification concepts at the highest level of abstraction possible. It is important to note that in functional verification, it is ultimately the traffic at the physical level (logic values on device signals) that should be verified. In this context, moving to a higher level of abstraction is only possible if the handling of the layer hidden under this newly introduced abstraction can be automated without loss of any detail at the physical level.

The move to a higher level of abstraction comes in two forms:

- Verification Goal Abstraction:
 - Deal with less detailed tasks
- Higher Level Verification Language Constructs
 - Write less code for same task
 - Make fewer mistakes in developing the environment

Verification goal abstraction allows the verification engineer to concentrate on higher level data constructs instead of logic values on device wires. For example, by providing an ethernet verification component, a verification engineer can define a verification goal as “injecting a valid ethernet packet” instead of having to describe the sequence of activity that will lead to such a packet at the device port. In this case, the ethernet verification component handles all the detail necessary to inject a valid packet. It is therefore important to pay careful attention to the design and usage of verification components used to architect a verification environment.

More powerful Language constructs allow the verification engineer to implement the same task in fewer lines of code and therefore a shorter time. As is commonly known in software development, the number of bugs in a software program is proportional to the number of lines of code. Thus, by reducing the verification code size, the potential for errors in verification code is also reduced.

2.2.2 Productivity

Verification Productivity is a measure of the amount of manual effort that is involved in a verification project. This manual effort consist of:

- Developing the verification environment
- Verifying all verification scenarios in the verification plan
- Maintaining the verification environment
- Migrating the verification environment for next project

Maintaining and migrating a verification environment includes measures that should be anticipated during the development of the verification environment. The main focus in developing a methodology for improving productivity is to minimize the amount of manual effort

required to develop the environment and create the set of items that verify all scenarios in the verification plan.

As in most engineering tasks, there is a fine balance between the effort spent on environment development and the effort required to create the set of all necessary verification items (verification suite). The more time is spent on development, the less time spent on creating the verification suite, and vice versa. For example, if the verification environment is complex enough to handle all corner cases automatically, then the manual effort required to test corner cases is very small. In general, considering all possible corner cases in a verification environment can be very time consuming and it is often best to verify such extreme corner cases through manually created verification items. The remainder of this chapter provides methodology guidelines that help in striking the right balance between these two efforts.

2.2.3 Effectiveness

During simulation runs, the goal is to verify all the scenarios described in the verification plan. *Verification Effectiveness* provides a measure for the contribution of a simulation run to the overall task of verifying all the scenarios in the verification plan. Ideally, all time spent in running any simulation should improve verification plan coverage.

2.2.4 Completeness

Verification Completeness is a measure of the portion of device function verified during the course of the verification project. Generally, no verification plan can completely specify all features and all possible corner cases. Although it is relatively easy to define a measure of completeness for a verification plan (i.e. measure how many scenarios have been covered), it is difficult to measure completeness when referring to all possible features of a design.

Since completing all verification scenarios in a verification plan is a required goal for the verification project, then verification completeness is concerned with the part of device function that may not be specified explicitly in a verification plan but is indirectly verified during the verification project.

In a verification environment where each scenario is explicitly verified (i.e. directed verification), completeness of device verification is hardly extended beyond the verification plan scenarios. However, in a verification methodology where verification scenarios are randomly generated and device response is automatically checked, it is highly likely that many scenarios beyond the initial verification plan are also verified.

2.2.5 Verification Environment Reusability

Verification Environment Reusability may be defined in two ways:

- Reusing verification environment for next generations of the same design

- Reusing verification environment modules and utilities while moving from module level verification to system level verification

The main assumption in re-using a verification environment across design generations is that design generations have similar profiles and features, though slightly modified or enhanced. The main challenge in achieving this target is to anticipate future design changes and to identify modules and features that are expected to change. By defining clear architectural boundaries between pieces that are expected to change and the pieces that are expected to remain the same, the environment migration tasks can be drastically reduced.

Reusing the verification environment through the project life-cycle is a more pressing requirement for projects where turnaround time is very important. As verification tasks move from module level verification to system level verification, verification is less focused on generation and more attention is placed on collection, monitoring, and coverage collection. A general guideline for achieving this target is to architect the verification environment such that the required pieces for system level verification (monitors, checkers, coverage collectors) can function independently from the generators.

2.2.6 Simulation Result Reusability

Simulation Results refers to the data that is collected during simulation runtime. It is often the case that after a long simulation run is completed, either the need to check some additional properties is realized, or it would be useful to find out if a certain scenario was in fact exercised during the simulation run. At one extreme, if a signal change dump of all signals during the simulation run is available, all such questions can be answered by analyzing the simulation data dump. At another extreme, if no data is stored, then the entire simulation has to be rerun for any question to be answered. The right approach is obviously to strike a good balance between the amount of collected data and the types of questions that may need to be answered after a simulation run is completed. *Simulation Result Reusability* refers to the ability to selectively define the information collected during simulation run so that post-simulation analysis can be performed without having to re-run the simulation.

Consider a multiplier design that multiplies two 16 bit numbers. If we collect a list of all pairs of numbers that have been multiplied during simulation, we can answer any question on whether the multiplier has been tested for a given pair of numbers without having to rerun the simulation. Not only is it important to collect the numbers seen on each input port, but the correlation between these input values should also be recorded.

Data reusability is usually considered as part of coverage collection strategy. During coverage collection, it is very important to collect information on specific coverage questions, and also to anticipate future questions that may arise and if possible, collect the necessary data to answer such questions without having to rerun the simulation.

2.3 Directed Test Based Verification

Directed Test based verification is a brute force technique for completing a verification project. In this approach, a specific verification item is created for each verification scenario and new facilities are implemented, or the existing infrastructure is enhanced to support the requirements for each new verification scenario.

The one and only potential advantage of a directed test based verification methodology is that project progress is almost linearly proportional to the amount of time spent on the project (figure 2.9). In this approach, verification progress is made one time-consuming and small step at a time. A directed test based verification methodology may be recommended for a verification environment that lacks modern verification tools and languages, or for projects where very little time is available to complete the verification phase and therefore insufficient time is devoted to target full verification. Studying directed test based verification is useful for two reasons: first, the need for more advanced verification methodologies is motivated by studying the shortcomings of this approach. Second, a directed test based approach may be used as part of other verification methodologies to cover very hard to reach corner case verification requirements.

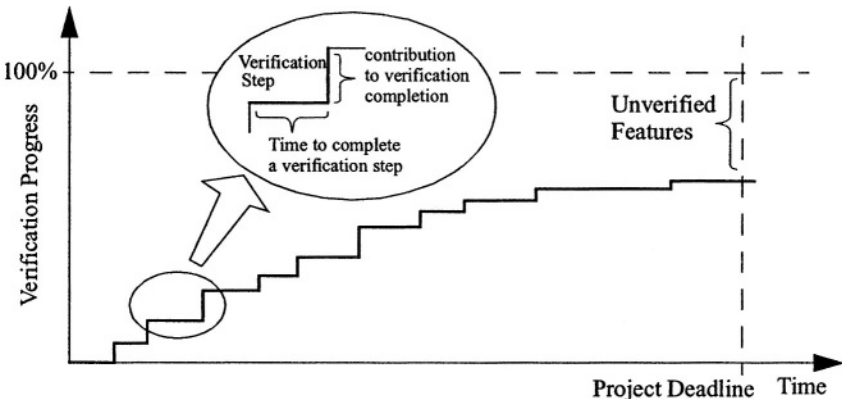


Figure 2.9 Verification Progress and Directed Test Based Verification

The fundamental steps in directed test based verification are very simplistic. These steps are:

- Complete the verification plan
- Sort the scenarios in the verification plan according to some priority considerations
- For each scenario, enhance the existing environment, or build new infrastructure to verify that specific scenario
- Add the completed verification item to the verification suite for regression

The sorting of verification scenarios in this approach may depend on multiple factors, including the importance of the features that are verified, the natural order of verification code development as the verification environment is being enhanced, and the order of development for the device under verification.

2.3.1 Task Driven Verification Methodology

Directed test based verification in its most fundamental form interacts with the DUV using physical level signals. This low level interaction and modeling introduces major inefficiencies in building the verification environment and verifying scenarios. *Task Driven Verification Methodology* is a variant of the directed test based verification methodology where logical views are used to improve verification productivity (figure 2.10). In task driven verification methodology:

- Traffic is defined at a higher level of abstraction (frames, packets, instructions etc.).
- Verification tasks are defined at a logical level (write to port, read from port, issue instruction, etc.)
- Verification utilities are developed to support these new data structure and task definitions
- Verification scenarios are created one scenario at a time using the available verification utilities

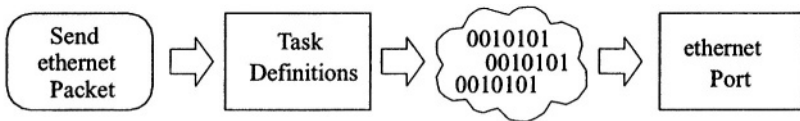


Figure 2.10 Task Driven Verification Methodology

2.3.2 Verification Quality

The brute nature of directed test based verification leads to significant loss of productivity. In such cases, the most fundamental shortcoming is that each verification scenario has to be considered independently and human interaction is required to set up the verification environment, generating the necessary traffic, and checking results for each verification scenario. At the same time, the verification plan may describe scenarios using ranges of acceptable values for each parameter. Using a directed test based approach, the values for these parameters have to be specifically decided, and often, additional scenarios created for corner cases consist of specific combinations of such parameters. The main problem with this approach is that it is difficult and time consuming to enumerate values for different parameters and come up with corner case conditions.

Additionally, because of the amount of hard-coded values in a directed test based approach, such a verification environment is hard to maintain and not easily portable even across very similar projects.

To summarize, the advantages of task driven verification methodology are:

- Improves productivity over the most basic task driven verification methodology by introducing tasks and abstract data types
- Useful as part of a more comprehensive verification methodology to cover very specific corner cases

The disadvantages, however, include:

- Requiring human interaction to create and check each scenario, leading to very low productivity
- Requiring detailed enumeration of all scenarios and corner cases (often too numerous to be possible) leading to verification incompleteness
- Difficult to use, tedious to maintain, and impossible to port across projects because of extreme customization of each verification item

Random generation of data and scenarios is used to improve on the shortcomings of task driven verification methodology. This verification methodology is described in the next section.

2.4 Constrained Random Test Based Verification

Randomization is a powerful technique, which, when used appropriately, can drastically improve verification quality. A closer look at the anatomy of a verification scenario can lead to a better understanding of the way randomization is leveraged to improve verification quality.

A verification scenario describes a well defined order of low level transactions in the verification environment where a set of data and parameter values are associated with each low level transaction¹. Ideally, a set of atomic transactions can be defined where any verification scenario is composed of a sequence of these atomic transactions. Each atomic transaction will require a set of data and parameter objects to complete its activity. Obviously, not all sequences of such atomic transactions lead to meaningful scenarios. At the same time, not all possible data and parameter combinations are allowed for a given atomic transaction. The legality of valid data and parameters may depend on the context (i.e. order of an atomic transaction within a sequence).

¹. A parameter modifies the behavior of a transaction while data does not. For example, for a bus read transaction, read burst size is a transaction parameter since it modifies the number of bus cycles, etc., while address to read from is considered transaction data since the transaction behavior is generally independent of its value.

Assume a list of such atomic transactions and their corresponding valid orderings, valid data content, and valid parameter content is available for a DUV. It is then straight forward to see that given enough time, all verification scenarios in the verification plan can be created by generating random sequences of these atomic transactions along with their corresponding data and parameters, while constraining the random generation to the subset of valid sequences, data values, and parameter settings.

Constrained Random Test Based leverages the concept of randomization to automatically generate constrained random sequences that by exercising the necessary device functions, verify the scenarios in the verification plan. This methodology leads to immediate and significant gains in verification productivity and completeness.

Using this methodology, it is no longer necessary to individually implement and verify each scenario in the verification plan (as is the case for directed test based verification) therefore improving verification productivity. At the same time, random generation of scenarios leads to far more scenario activations than could possibly be listed in a verification plan. This increase in the number of scenarios randomly generated leads to significant improvement in verification completeness and greater confidence in design correctness.

Practical deployment of constrained random based verification requires significant changes in verification environment implementation. The most fundamental requirement moving from directed test based verification to constrained random based verification is that the verification environment should be able to properly handle all types of activity generated by the random generation process. In directed test based verification, each scenario is built explicitly and the environment's response to that scenario could also be implemented while verifying the specific scenario. For constrained random based verification, any valid sequence of atomic transactions (including the ones that may not be in the verification plan) as well any valid data may be generated during simulation therefore the environment has to be implemented to handle all such activity.

Note that the verification environment supporting randomly generated data and scenarios should not only support the generation and successful completion of all possible scenarios, but also automatically check that correct system behavior was observed at every stage of randomly generated scenarios.

It is clear that since scenarios are generated randomly, verification progress can only be measured as each new scenario is generated, and verification progress is not known before simulation is run. This is in contrast to directed test based verification where verifying all directed tests guarantees complete coverage of the verification plan. This means that the verification environment supporting constrained random based generation should also include the necessary infrastructure to measure verification progress as simulation continues.

A finite state machine is a design style wherein the benefits of constrained random based verification are immediately obvious. In this design style, the verification plan simply consists of traversing all possible edges between states.

- Generator:

- Generate Random states constraining the state to the valid set of states
- Generate Random inputs
- Checks:
 - Check that the correct next state is reached after each clock cycle
 - Check that correct output is produced while in each state
- Coverage:
 - Collect the set of all states reached
 - Collect the set of all inputs applied while in a given state.

An important issue to consider is that complete verification of a finite state machine requires that the next state value is checked for all possible input combinations. This requirement is not practical, however, for any real size finite state machine description.

2.4.1 Random Based Verification and Practical Considerations

Constrained random based verification, in its most ideal application (i.e. state machine testing), is very effective in improving the verification quality. But in real designs, a number of considerations make a fully randomized environment (i.e. verifying all possible device behavior in one simulation run) difficult to achieve. These factors include:

- The extensive effort required to build a verification infrastructure capable of supporting all scenarios
- Extremely low probability of reaching some corner case conditions without very specific constraint; even for very long simulation runs
- Not enough time to wait for eventual verification of all scenarios
- Randomly repeating the same verification scenario leading to low effectiveness

Sometimes during verification environment development it becomes clear that adding support for a randomly generated corner case condition would require extensive effort. Under such conditions it is advisable to build a directed test that verifies that specific corner case condition. Note that this corner case condition may in fact have its own randomly generated data and parameters but the assumption in running such a scenario is that a customized verification environment is required for handling that specific scenario.

Additionally, many parameters exist in a modern DUV and it is practically impossible to reach a specific corner case condition by a generic set of random generation constraints. Under such conditions, it is advisable to create a specific verification item with very specific constraints on random generation such that corner case condition occurs with very high likelihood. At the same time, it is generally the case that different sets of random generation constraints will lead to the generation of different groups of verification scenarios. Therefore, it is necessary to create multiple verification items where each verification item is focused on a group of verification scenarios.

To summarize, a constrained random based verification environment will in practice include:

- Verification environment infrastructure targeted to specific corner cases that require special handling
- Multiple Verification Items each having random generation constraints that lead to different types of verification scenarios or very specific corner case conditions.

2.5 Coverage Driven Verification

As described in the previous section, constrained random based verification methodology leads to significant improvements in verification completeness and productivity. However the potential gains promised by this approach may be difficult to harness without a clear strategy for measuring verification progress.

To that end, important questions that must be answered during constrained random based verification are:

- Did constrained random generation reach all verification goals?
- Did constrained random generation reach any interesting non-goal verification states?
- How much did each simulation run contribute to completing the verification plan?
- How should generation constraints be modified for the next simulation run?

Coverage Driven Verification methodology is the unification of coverage collection and constrained random based verification where the results of coverage collection are used to answer the above questions in order to guide random based verification methodology to a successful completion.

Coverage driven verification methodology is based on four fundamental concepts:

- Raising the level of abstraction for verification tasks
- Constrained random generation of verification data and scenarios
- Automatic checking of simulation results
- Coverage collection to measure verification progress and guide random generation to cover missing scenarios

Verification project life-cycle for coverage driven verification methodology is shown in figure 2.11. This project life-cycle is divided into four main phases:

- Phase 1: Verification Plan Development
- Phase 2: Verification Infrastructure Implementation
- Phase 3: Verification Environment Bring-up
- Phase 4: Constrained Random Verification
- Phase 5: Corner Case Verification

In the first phase, the verification plan is developed based on the DUV engineering specification and design engineer feedback. The goal should be to make the verification plan as complete as possible in this first phase as the implementation of the verification infrastructure

will depend on the requirements of the verification plan. The verification plan will be updated and extended during the project lifetime, however, to include missing scenarios or add new DUV features as the DUV is being completed by the design team.

In the second phase, the necessary infrastructure for supporting the verification plan is implemented. In phase three, some simulation runs using strict generation constraints are made to check for correct functionality of this implementation. Even though some verification progress is made in this phase, the goal is to complete the infrastructure and check for its correct operation. Phases two and three may require a few iteration before phase four is started.

In phase four, the least amount of constraint is used during random generation in order to cover as many verification scenarios as possible in any given simulation run. Results from coverage collection are used to monitor the effectiveness of each simulation run and to modify generation constraints to guide scenario generation towards missing verification scenarios. Each verification item corresponding to a simulation run will potentially cover many verification scenarios which will be measured by coverage collection facilities. During this phase, succeeding simulation runs will have stricter generation constraints to guide the generation towards the missing the scenarios. Consequently, the contribution to overall verification progress may be reduced with new verification runs.

In the phase five, corner case conditions requiring specific modifications to the verification infrastructure are completed. Ideally, any verification scenario that does not require a customized verification infrastructure should be covered in the fourth phase. Corner case conditions covered in this phase will look more like task based verification even though some parameters and data values of corner case conditions may be randomly generated.

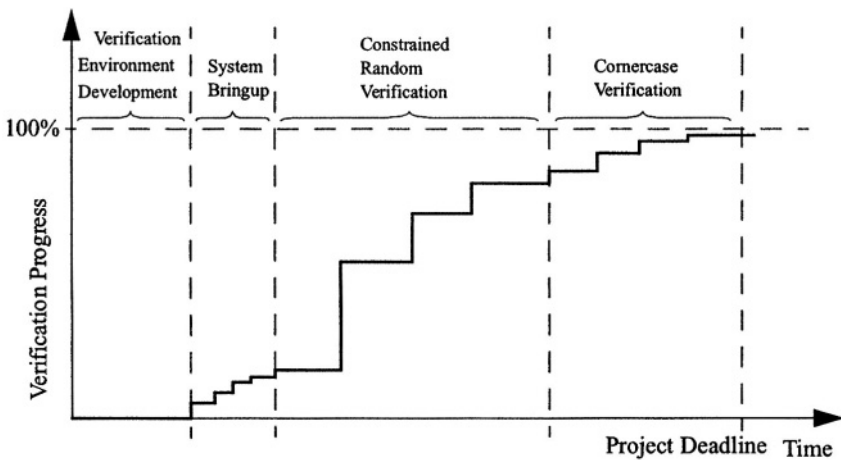


Figure 2.11 Coverage Driven Verification Methodology

2.5.1 Verification Quality

Coverage driven verification methodology provides the best quality of all verification approaches. Verification granularity is high since concepts from task driven verification are used to allow verification engineers to deal with more abstract verification tasks. Using the concept of constrained random generation provides very high verification productivity because human interaction is neither required for generating nor checking every individual verification scenario. Verification effectiveness is high as coverage collection is used to gain confidence in usefulness of every simulation run. At the same time, verification completeness is also high since coverage collection provides good confidence that all verification scenarios in the verification plan have been covered. Random generation of data and verification scenarios leads to covering more verification scenarios than the ones listed in the verification plan. In that regard, running a simulation beyond the point where the verification plan is fully covered leads to further confidence that additional scenarios not listed in the verification plan are also covered. Additionally, judicious collection of coverage allows for reusing simulation results to answer further questions about a simulation run after the run is completed.

2.6 The Enabling Technology: Hardware Verification Languages

The ability to successfully apply coverage driven verification methodology to a verification project depends on the availability of a verification language that supports the requirements for such a methodology. A hardware verification language should provide:

- A software development environment that:
 - provides a powerful debugging interface
 - promotes software development practices (i.e. object oriented programming)
- Simulation Related Facilities that support:
 - Concurrency
 - HDL Simulator Interface
- Verification related constructs for:
 - Random generation
 - Automatic checking
 - Coverage collection

2.7 Summary

This chapter introduced functional verification and motivated the need for an effective functional verification methodology. A set of verification metrics were introduced to provide a

means for contrasting and motivating the introduction of new verification methodologies. Directed test based verification, constrained random based verification, coverage driven verification, and autonomous verification environments were discussed and the introduction of each new methodology was motivated by the verification metrics discussed in this chapter.

This chapter strongly emphasized that the ever increasing complexity of verifying new digital systems can only be managed by increasing productivity; and this increase in productivity is only possible through automation, reuse, and by moving to higher levels of abstraction. Coverage driven verification provides the best combination of all available approaches for achieving a good balance between the effort required to build a verification infrastructure and completing the project on time and before the deadline.

Chapter 3 describes the verification environment architecture that is used for applying coverage driven verification.

Anatomy of a Verification Environment

The verification methodologies discussed in chapter 2 present verification metrics and approaches that lead to good verification quality. This chapter addresses the verification environment architecture and verification utilities needed to allow the application of the methodologies discussed in chapter 2. The discussion in this chapter is independent of any specific tool and will illustrate the architectural implementation of the methodologies introduced in chapter 2 as well as further motivate the requirements and features of a hardware verification language.

The steps in completing a verification project, at a high level, are:

- Extract the verification plan from the design specification
- Build the Verification Environment
- Perform Verification Activity (i.e. coverage driven verification in figure 2.11)

In a verification project, all activity is guided by and measured against the verification plan. The features of the verification environment are guided by the requirements of the verification plan, and the completion of the verification project is also measured against the scenarios listed in the verification plan. As such careful attention to verification plan development is essential.

This chapter discusses verification plan development and presents the verification environment architecture and components that facilitate the advanced verification methodologies discussed in chapter 2.

3.1 Verification Plan

The verification plan is the master document for all verification activity. The building of the verification plan is perhaps the most important step in completing functional verification. The verification plan is used as a guide to:

- Architect the verification environment
- Decide what scenarios to verify
- Define coverage metrics for measuring verification progress

Once the verification plan is available, the verification environment architecture is implemented to facilitate the verification of all scenarios in the verification plan. In addition, coverage metrics directly reflect the portion of the verification scenarios that have been verified. In this respect, once the verification plan is available, the quality of verification can be measured with respect to the verification plan.

Building a verification plan is somewhat subjective and requires collective decision making across the design and verification teams. Decisions must be made on what features are important to verify and what features can be left out or assumed verified as a consequence of other verifications. In other words, building the verification plan is subjective and depends on the business and technical requirements of the project. The distinction between creating the verification plan and other verification steps makes the verification plan the biggest potential pitfall in building a successful product, and can mean the difference between success and failure.

The verification space for a DUV is best described as a set of simulation states and the necessary conditions to take the DUV into a given state (figure 5.6). For a finite state machine design, this view of the verification space is in direct correlation with the physical implementation of the DUV where the view of the verification space is in one-to-one correspondence with the state machine description. For a more complex device, each state in this view may correspond to an operation state of the device for a given verification scenario.

Given such a verification space for a DUV, a set of simulation *Goal States* are defined such that correct device operation, while reaching all such goal states, verifies correct device implementation. Given the verification space and its corresponding simulation goals, a *Verification Plan* is the set of all verification scenarios that verify the correct operation of DUV for each goal state. A *Verification Scenario* in its most generic form describes the cause-and-effect sequence of taking the DUV to a desired goal state while automatic checks confirm correct device behavior. Using this abstraction, a verification scenario is described by:

- Goals: Simulation goal to be seen sometime during the simulation
- Input Sequences: Input sequences to apply to take the DUV to the desired simulation goal
- Checks: Checks to perform in goal state, and while reaching the goal state in order to verify correct DUV operation

Once simulation goals are defined and the necessary checks for each simulation goal identified, the specification of the input sequence naturally follows from the device specification. Thus, identifying the relevant simulation goals and the necessary checks are critical tasks in developing a verification plan.

3.1.1 Simulation Goals

A *Simulation Goal* defines a goal state to be observed sometime during DUV simulation. The simulation goal is defined at a given time instance and may include the following information:

- Valid and invalid values on key attributes
- Valid and invalid spatial relationships between key attributes

Verification progress in general can be measured in terms of the number of goal states reached during simulation. This assumption is valid when the verification criteria for a goal state is independent of the path that leads to that goal state. However, it is often the case that the goal state, as well as the path taken to a goal state, will be significant to the verification scenario. If verification coverage is to be measured in terms of goal states, then the goal state is duplicated for each relevant path leading to that goal state. For example, in figure 3.1, the original verification space has two goal states **A** and **B**. For goal state **B**, it is only important that state **B** is reached, For goal state **A**, however the path leading to that state is relevant to the verification task. The verification space is therefore modified to include two goal states **A1** and **A2**. In this new verification space, verification is complete (full verification coverage is achieved) when all goal states **A1**, **A2**, and **B** are reached.

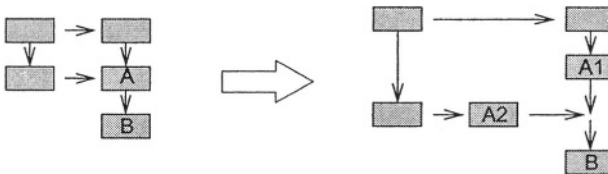


Figure 3.1 Goal State Duplication

3.1.2 Verification Views

Two views must be covered when defining attributes to be included in the definition of a goal state:

- Port-Based view
- Function-Based view

A *Port-Based Verification View* describes the DUV behavior as observed through its interface ports. This view is an integral part of verification for I/O modules such as memory interface modules, CPU bus attachment units, and standardized serial and parallel ports (i.e.

ethernet, USB, UART, etc.). In this verification view the following items are described in a verification item:

- Attributes defined by pin values
- Attributes defined by data structures passing through DUV pins
- Interaction between such attributes

A *Function-Based Verification View* describes verification scenarios for internal operation of the DUV as described in its specification. Even though a DUV behavior, once in use, is only visible through its interface ports, a function-based verification plan is necessary for two important reasons. First, most DUVs have special functionality that is intended to improve device performance. Examples of such functionality include FIFO threshold specifications, or cache coherency protocols in microprocessors. Even if the FIFO thresholds for FIFO read/writes are not working as expected, the device will seem to function normally through its interface ports. At the same time, even though a CPU's performance will be severely degraded without a memory cache, the bus behavior or CPU program will still function. In such cases special attention must be paid to device features that are critical to optimal performance but are not immediately visible through its port behavior. A second important consideration is the cause-and-effect relationships at DUV ports, which usually takes many cycles to complete, and any intermediate problems would be hard to track to its source. Therefore, by verifying the internal functional steps to implement port behavior, it is easier to locate the source of any problem that may be identified at the device ports.

Function based verification items should consider:

- DUV states that implement specific device features
- Simulation goals that describes such DUV states

In general it is not possible to verify all DUV features. It is therefore important to set some guidelines to assure a comprehensive and efficient verification plan. Some guidelines to remember while developing a verification plan are:

- Break down the verification plan into port-based and function-based parts.
- Include a simulation goal for each Normative Statement¹ in the function specification
- Exclude simulation goals that are checked as part of other verification scenarios

3.1.3 Verification Checks

Once the verification goals are defined, it is necessary to check that the device reaching that state, and at that state is performing as expected.

In general, a device behavior needs to be checked for two conditions:

- Device does not produce invalid outputs or behavior on its ports or internal signals

¹: Normative Statements are necessary requirements in a specification in order to properly allow a piece of equipment to attach to another piece of equipment.

- Device responds appropriately to all valid stimulus on its input ports

Note that in the context of verification, a valid stimulus is defined as all stimuli that the device is expected to handle properly. It is possible that in a higher abstraction, such stimulus may be considered “invalid input,” but if the device specification allows for special handling of such higher level protocol “invalid input” then, this special handling is considered part of the DUV operation and the input stimulus that activates this functionality is considered valid input in the context of DUV verification. For example, an ethernet device is expected to reject malformed ethernet packets received on its ports. From the perspective of ethernet packets, such packets are invalid, but such a malformed packet is still a valid verification input stimulus and should be applied as part of the verification effort.

In building a verification plan, it is important to list clearly all valid and invalid port conditions that a device is expected to handle properly. As will be described later in this chapter, it is this list that dictates the required features of the verification environment from the bus functional models to the monitors and collectors.

Checks fall into two categories: Syntax Checks and Timing Checks. Syntax Checks refer to checking attributes and signal values on DUV ports and internal signals to verify valid combinations. Timing Checks refer to temporal checks across DUV signals that check correct compliance with expected timing behavior.

3.1.4 Debugging and the Verification Plan

A verification plan is expected to expose potential problems in the DUV. It is important, however, to provide a means for tracking a problem to its source quickly. In developing a verification plan, the following considerations should be made to improve debugging features:

- Define verification items in terms of many small operations, as opposed to identifying the verification space in terms of a few simulation goals.
- For each verification scenario, define checks not only for the goal state, but also for all intermediate states that lead to the goal state.

3.2 Verification Environment Architecture

Verification tasks move in lockstep with the design effort. Therefore, a verification environment architecture should reflect the requirements of the design project phase. Two such phases of a verification project are the module level and system level verification phases. Having a reusable module level verification environment for system level verification is an important consideration when defining a verification environment architecture. This section on verification environment architecture focuses on module level verification. Section 3.3 contrasts system level and module level verification and shows how the relevant parts of a module’s verification environment is migrated to its system level verification environment.

The architectural view of a module level verification environment is shown in figure 3.2. Note that in this view, the attachment of the verification environment is shown for only one DUV port.

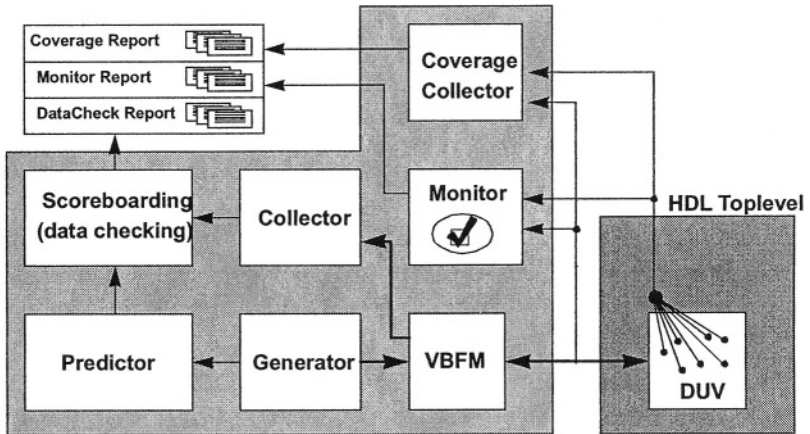


Figure 3.2 Verification Environment Architecture

The prominent features of this verification architecture are:

- Device Under Verification (DUV)
- HDL Top-level
- Verification Bus Functional Model (VBFM)
- Generator
- Collector
- Monitor
- Predictor
- Data Checker
- Coverage Collector

In this figure, DUV is the device that is being verified. Top-level HDL creates a layer where the DUV is connected to other HDL modules that are necessary for its operation. Clock and reset generation modules, if in HDL, are also placed in this layer.

VBFM is the module that provides an abstract view of the DUV to the verification environment and also contains the necessary features for supporting verification operations. The generator creates the necessary stimulus and environment settings to produce the specific scenario that is to be verified. The monitor checks for properties that should be maintained during the simulation runtime. The collector extracts output generated by DUV and forms actual data items that are then checked against expected data during data checking phase. The predictor acts as the reference model for the DUV and generates the expected results for data checking operations. The data checker uses the results of the predictor and the collector to compare

expected and actual data values. Throughout the simulation, coverage collectors record information that indicate which scenarios have been verified.

Note that data checking module connectivity (i.e.collector, predictor, Scoreboard) depends on the points at which data checking is performed. Therefore the connectivity shown in figure 3.2 may change depending on data checking strategy.

The remainder of this section presents a discussion on issues related to VBFM, generation, and checking tasks.

3.2.1 CPU Verification

The architecture shown in figure 3.3 is best suited for data communication devices where device ports act as ports for data traffic. CPU verification would require additional considerations. The main goal in verifying a CPU implementation is verifying that the CPU executes instructions correctly and that system busses comply with the bus specifications. This means that CPUs are usually verified by executing a program that is residing in its memory. In other words, the stimulus for CPU verification is the program that resides in memory. Often, such programs are self-checking programs where the correct execution of the program is verified by checking for specific conditions in CPU registers and memory locations. Because of this approach, the stimulus generation for CPU verification is usually a pre-verification step where programs are generated before verification starts. Each assembly program in this case verifies one or many scenarios in the verification plan. To complete the verification, each assembly program is loaded into the CPU memory and the simulation is run for that specific program.

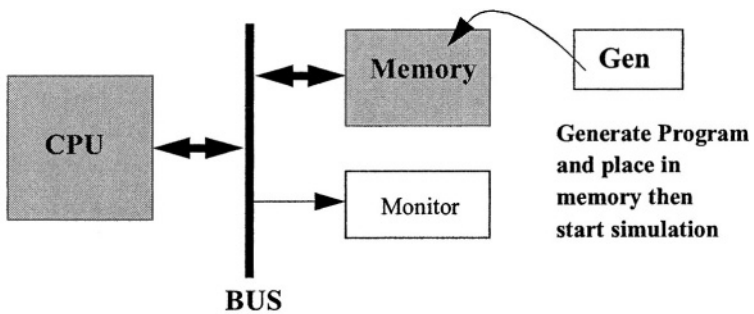


Figure 3.3 CPU Verification Environment Architecture

After a program is loaded into the CPU memory, the CPU verification environment becomes a self-contained verification environment. This means that no input is injected and no output is collected from the environment². Therefore, monitors are key for a CPU verification

² CPUs have additional features such as interrupts that require external stimulus.

environment to provide visibility into the state of device operation. The location of these checkers is shown in figure 3.3.

It is possible to create a CPU verification environment where the CPU instructions are dynamically created during the simulation process. In this approach, new instructions are generated and returned in response to each memory instruction fetch cycle. The advantage of this approach is that the mix of instructions may be modified to guide the verification to cover different parts of the verification plan.

3.2.2 Verification Bus Functional Model

A device uses complex protocols at its ports to communicate with outside devices. Hiding this physical level port complexity from the verification environment allows the verification environment to interact with the DUV at a higher level of abstraction. Such abstraction of a physical port interface forms a more productive easy to use verification environment. A *Bus Functional Model* (BFM) is a module that provides the verification environment with a logical view of the physical bit level activity at the DUV ports. The abstraction provided by a BFM facilitates task driven verification methodology (figure 2.10).

A BFM, in its most basic form, is very similar to a module that would be attached to a device port in its final application environment. This means that the goal in building a BFM is to duplicate the port functionality of a module that will connect to the DUV, and at the same time provide a friendlier user interface to simplify interactions with the DUV. Examples include a BFM attached to CPU system bus providing a simplified interface to the complex handshaking on the bus, and an ethernet BFM that translates an abstract ethernet packet into serial bit stream corresponding to ethernet traffic.

Even though the function of a BFM described above is sufficient for simple checking of device behavior, these functions must still be extended to include features needed by verification requirements. Such enhancements depend on the verification plan and the conditions that should be generated to cover all verification items.

A *Verification Bus Functional Models* should provide:

- A simplified logical interface for interacting with the device
- Handling bit level interaction with DUV ports for all allowed modes of DUV port operation
- Means to inject errors that the DUV is expected to handle gracefully
- Fault tolerant features for graceful recovery from faulty device operation
- Means to set and query configuration settings
- Status collection and reporting as related to BFM interaction with the DUV

A VBFM should be able to generate error conditions as part of its feature set. This is necessary to make sure the device correctly handles error conditions at its ports. In addition, a VBFM should be configurable so that it can accommodate different verification scenarios. A VBFM should also not fail (i.e. it should either terminate gracefully or continue to search for

next valid activity) when detecting a bug in the device, so that automated simulation runs can progress without interruption and continue to look for further errors.

Two important considerations in building a VBFM are:

- Feature Set
- User Interface

These topics are discussed in later subsections.

3.2.2.1 BFM Features

An *Active VBFM* is a VBFM that interacts with the DUV by both applying input and reacting to outputs. An active VBFM, in its simplest form, should be able to generate all valid input sequences to the DUV and be able to react to all sequences that may be observed on the DUV output ports. It should also support handshaking sequence required at the DUV port to communicate with the DUV. This specification should be immediately identifiable from the DUV functional specification.

As previously mentioned, VBFM includes additional features beyond the basic function of a BFM to facilitate verification operations. The specifics of these additional features is not as clear, however, as the specification for the basic functionality of the BFM. As a trivial example, assume a DUV port signal is expected to be low for only one clock cycle at a time, and different errors are detected by DUV when this signal stays low for two, three, or four clock cycles. One approach to adding this feature to the BFM is to define 3 BFM transactions that keep this signal low for two, three, or four clock cycles. Generating error conditions would then require a request to the BFM by specifying the desired transaction. A different approach would be to allow for a parameterized transaction where the number of cycles that this signal is low is defined along with the transaction request. This trivial example shows that there are different ways of defining the feature set of a BFM that achieve the same effect. Some guidelines to consider while defining the enhanced set of features for a BFM are:

- Analyze verification sequences defined in the verification plan to identify the minimum set of BFM features that can be combined in different ways to construct the required verification sequences.
- In creating a logical abstraction, a BFM hides details of its interaction with the DUV port (i.e. hides handshaking mechanisms). Identifying the types of errors that should be created in this physical interaction as part of the verification requirement. Identifying the types of information that should be recorded when errors are observed in this lost detail. Add capability to the logical interface to create such errors and to query error status.
- Choose a parameterized BFM feature over multiple BFM features that are essentially the same but different in a way that can be parameterized (i.e. error definition in above example).

3.2.2.2 VBFM User Interface

A VBFM should have the following user interface:

- User Request Interface (types of request and parameters for each request)
- Configuration interface (set configuration and get configuration)
- Status Interface (get status)
- Interrupt Interface (using software events)

As discussed, a VBFM's feature set should be defined to allow for implementation of all verification sequences defined in the verification plan. To that end, the verification sequences are divided into a set of actions that define an interaction between the VBFM and the DUV, and a set of questions about the result of interactions between the VBFM and the DUV. These actions are combined in different ways to construct verification scenarios, and the answer to these questions are used in performing checks on successful completion of a verification sequence.

Parameters passed to a VBFM request include user data as well as other information required for each type of action. For example a memory write action to a system bus VBFM requires address and data value, control information such as burst size, and error information indicating if any error should be injected on the bus while performing the write operation.

Parameters common across multiple requests to VBFM should be defined as configuration settings for the VBFM. A configuration interface should be defined to allow for setting and querying the status of VBFM configuration. For example, in a bus write action, the burst size is usually the same for consecutive write operations so it can be set as a configuration setting for the BFM. However, the error injection feature however is more closely tied to each action request and is therefore a parameter that is passed to the BFM with each action request.

The set of questions that need to be answered by a VBFM are included in the status setting of the BFM and a status interface is defined for accessing this information.

A VBFM should generally be able to provide event information at its user interface, such as passing the DUV clocking information to the verification environment. Note that this clock may be generated inside the VBFM and then used in the DUV and the verification environment, or it may be taken from the DUV environment. Regardless, the VBFM should pass such clocking information to the verification environment as needed by the verification requirements. Moreover, an active VBFM may need to indicate to the verification environment it has completed its latest interaction with the DUV. An event is used to pass such indications to the verification environment.

General guideline for defining the interface for a VBFM are:

- Identify all requests to VBFM from the set of actions that must be supported by VBFM
- Identify the required parameters for each request
- Define the *Configuration Interface* as the set of request parameters that do not change often across multiple requests
- Define the *Status Interface* so that VBFM related questions during checking can be

answered

- Define events that will be useful for interaction with the BFM (i.e. clk, reset, different events for different types of activity, etc.)

Figure 3.4 shows an example user interface for a CPU bus VBFM.

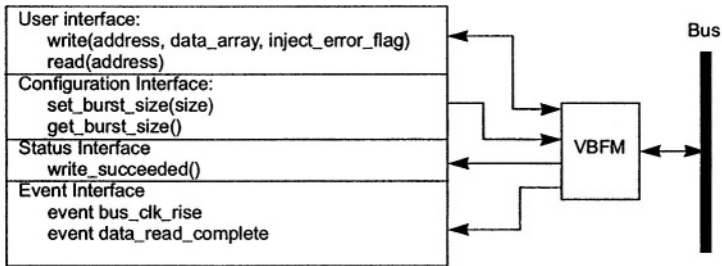


Figure 3.4 CPU Bus VBFM User Interface

3.2.3 Verification Scenario Generation

Generation is the process of creating the necessary stimulus to activate a given verification scenario. In order to generate a verification scenario, the following items should be considered:

- Verification Environment Initialization
- DUV and Verification Environment Configuration
- Data Generation
- Sequence of Activities that comprise the Verification Scenario

3.2.3.1 Verification Environment Initialization

Verification Environment Initialization refers to all data initialization that takes place before simulation starts. Environment initialization includes:

- Memory pre-loading
- Setting values to registers in the DUV that can only change before simulation is started
- Assigning values to DUV pins that can only change before simulation is started
- Verification environment settings

3.2.3.2 Verification Environment Configuration

Verification Environment Configuration refers to setting of all DUV and verification environment parameters that can change during simulation runtime. These parameters include:

- Runtime configurable DUV parameters
- VBFM configuration parameters as described in section 3.2.2.

The goal in setting these parameters is to direct the simulation to a desired verification scenario. Note that if there is only one verification scenario in a simulation run, then initialization and configuration steps can be folded into the same step. However, for a simulation run that includes multiple scenarios, configuration can potentially be changed to guide the simulation toward different scenarios during the simulation runtime.

3.2.3.3 Data and Scenario Generation

Generating a verification scenario involves generating the traffic that flows through the DUV ports, indicating how this data is treated by the verification environment as it flows through the ports, in what order the data is applied, and what configuration changes (DUV, or verification environment configuration settings) are required to move the simulation toward the goal verification state. The information the generator needs to produce is divided into two categories:

- Data information: traffic and how the verification environment treats it
- Sequence information: how the generated data is used to form a verification scenario

In a verification context, data is not simply the traffic that flows through DUV ports, but also the information that describes how a given data item is handled by the verification environment (i.e. injecting errors while sending a packet). Note that information that identifies a given verification scenario (i.e. how many packets to send) is not considered as part of data items. In a verification context, data items include:

- Data packets flowing through DUV ports.
- Information on errors that the generated data should contain. Such errors relate to the data abstraction that is being generated (i.e. generate an ethernet packet with checksum error).
- Information on errors to be injected by the VBFM while interacting with DUV to send a generated data packet.

A verification scenario consists of a series of steps in the verification space that directs the DUV into a goal state. In this view, generating a verification scenario consists of generating the sequence of operations that leads the DUV into the goal state.

Requirements for generating such a sequence of operations leading to a verification scenario are:

- Such sequences should be parameterizable.
- Simultaneous interaction with different DUV ports is required to create most verification scenarios. Synchronization between multiple sequences at different DUV ports is required to facilitate complex verification scenarios
- Complex verification sequences are usually a combination of less complex verification scenarios. Ability to generate sequences that are composed of smaller sequences is important in generating complex sequences.
- Complex verification scenarios require interactions with the DUV where output generated by the DUV affects the generated sequence. It should be possible to define a sequence of operations that depend on DUV output.

To complete the verification, a set of verification sequences is generated where each verification sequence covers a set of verification scenarios. The goal is to come up with a set of verification sequences that completely covers all verification scenarios in the verification plan.

3.2.4 Monitors

Monitors, also known as *Protocol Checkers* and *Protocol Analyzers*, verify that the traffic at DUV ports or DUV internal signals follow the requirements of the design specification. Monitors are passive components, which means that they do not interact with the DUV. Instead, they only sample DUV signals and check the protocol according to rules that are built into the monitor.

Monitors are usually placed at DUV ports or internal signals where device operation is defined based on well defined protocols (busses, standard ports, etc.), or where specific properties should be maintained (i.e. FIFO operations). Monitors are defined based on a set of properties that should be maintained throughout the simulation. Note that monitors do not require a reference model as the properties that should be maintained is built into the monitor.

Protocols are defined as a set of properties for signal attributes of DUV signals. *Signal Attributes* specify the allowed values on a DUV signal or a bus. Protocols define two types of properties:

- Syntax Properties: Description of valid attributes and combinations of valid attributes that can appear on DUV signals
- Timing Properties: Description of valid attribute sequences across multiple clock cycles

Syntax checks are in general easy to make. All that needs to be done for syntax checks are:

- When to check syntax
- What are the values allowed at the time and place of checking

Timing checks are more difficult to because of complex inter-relationships between signal values across multiple clock cycles. Steps to define timing checks are:

- Break down the protocol into a number of timing properties that should be maintained throughout the simulation
- Define simple timing check operations that can be used to implement all timing properties.
- Implement all properties in terms of these basic properties.

The following example shows basic protocol definition for a memory bus interface and the description for its protocol checking module. Example of read and write bus cycles for this example are shown in figure 3.5.

Example: Memory I/O module Interface Monitor:

Signals are:

ToMemory: **Addr, Size, WriteEnable, ReadEnable**
From Memory: **MemReady**
Bidir: **Data**

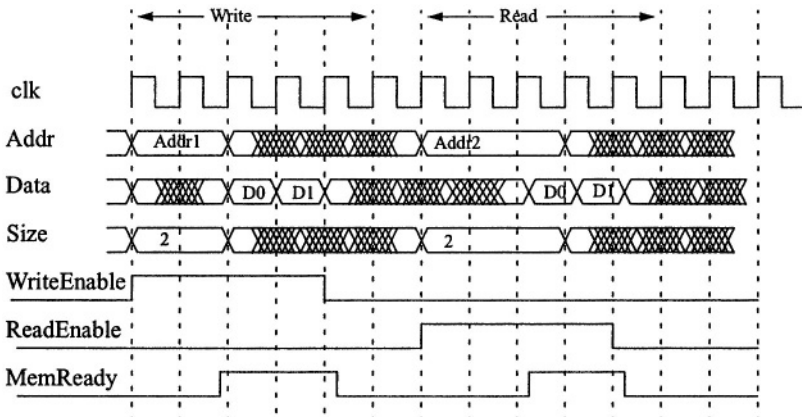


Figure 3.5 Protocol Checking for CPU Bus Read/Write Cycles

Device Operation Definition:

Write:

- CPU asserts **Addr**, **Size**, **WriteEnable**
- Memory should assert **MemReady** within 4 clock cycles
- CPU can de-assert **Addr** and **Size** once **MemReady** is asserted
- CPU continues to assert **WriteEnable** during write operation
- When CPU samples **MemReady** at asserted value, it provides data in next cycles
- Memory continues to assert **MemReady** during write operation
- At the end of an operation, both CPU and memory de-assert all signals

Read:

- CPU asserts **Addr**, **Size**, **ReadEnable**
- Memory asserts **MemReady**, and **Data [Addr]** within 8 clock cycles
- For next **Size** clock cycles, memory provides following data values on **Data**
- Memory continues to assert **MemReady** throughout the read operation
- CPU can de-assert **Addr**, **Size**, once **MemReady** is asserted

Protocol Checking steps:

Define a sampling time:

- Use the **clock** rising edge common to both CPU and memory

Define basic events:

- **WriteEnableAsserted**: **WriteEnable** asserted at rising edge of **clock**
- **ReadEnableAsserted**: **ReadEnable** asserted at rising edge of **clock**
- **cpuEnableAsserted**: one of **ReadEnable** or **WriteEnable** asserted
- **MemReadyAsserted**: **MemReady** asserted at rising edge of **clock**
- **AddrChanged**: value of **Addr** changed during the previous clock cycle
- **SizeChanged**: value of **Size** changed during the previous clock cycle

Define Value Holders:

SizeValue: Value of **Size** at time of **cpuEnableAsserted**

Monitoring Properties:

Syntax Checks:

- **SizeValue** is always in [1,2,4]

Temporal checks:

- If **cpuEnableAsserted**, then no **AddrChanged** or **SizeChanged** before **MemReadyAsserted**
- If **readEnableAsserted**, then **MemReadyAsserted** happens within 8 clock cycles
- If **writeEnableAsserted**, then **MemReadyAsserted** happens within 4 clock cycles
- If **cpuEnableAsserted**, then no **cpuEnableDeasserted** until **MemReadyAsserted** and then **MemReadyDeasserted**
- If **MemReadyAsserted**, then no **MemReadyDeasserted** for **Size Value** clock cycles, followed by **MemReadyDeasserted** at next clock cycle.

If these checks are performed during the simulation process and observe all valid scenarios at this interface, then correct operation of the memory interface is completely verified. Note however that verifying the correct operation of the interface does not mean that the memory device is working correctly. An important step in checking the function of the memory is to check that the data was written and read back correctly from the memory, and that memory contains the expected data based on the history of write operations performed during the simulation. Data collecting and checking are discussed in the next sections.

3.2.5 Data Collector

Data Collection is the step to extract data from the simulation environment while removing handshaking and timing information used to move or manipulate the data in the DUV. For example, in a memory write operation, the collected data from the memory bus is the address, the data to be written, and the size of this operation. All handshaking and control signal information are removed while collecting such data.

Data collection can be done as part of the monitor or as part of the BFM. This positioning of the collector depends on the following factors:

- BFMs are needed for module level verification, but not necessarily for system level verification. Therefore it is better to place data collectors in the monitor module so that they can be used as part of the system level simulation environment.
- Data collection may be done at the DUV ports or at some internal signals of the DUV. For internal signals, then the collector is best implemented within the monitor. For port signals, the collector may be a part of the BFM functionality.

As mentioned before, a data collector extracts data from the environment according to a data abstraction level. Case in point: for the memory bus, the collected data is in the form of memory transaction operations. The following guidelines can be used to decide the type of information that should be extracted by a data collector module:

- Control timing information is usually a part of protocol definition. All such timing information that is checked by the monitor can be dropped from collected data.
- What is the minimum amount of data that should be collected to check for correct data movement and manipulation? This decision depends on the data checking strategy.

For example, one strategy for verifying correct memory module operation, is to extract all write transactions from the bus and update a model of the memory with the data write transaction. This data model is then used to compute the expected result when a read operation is encountered. To follow this strategy, this is the information that should be collected:

- Transaction Type (read/write)
- Transaction Address
- Transaction Size
- Data values (read/write)

3.2.6 Data Checking

Data Checking operation is done to verify that DUV handles and moves data according to its specification. At its most accurate level, Data Checking consists of a cycle accurate reference model where device output are compared to reference values.

It is usually not necessary to perform cycle accurate checking since protocol checking already verifies cycle accurate behavior of the design. In the majority of cases, transaction level checking is sufficient. At the same time, data checking may only be needed across a few DUV ports or even between internal signals in the DUV. In such cases, a reference model that describes DUV behavior across data checking points is sufficient. A data checking strategy is defined by:

- Ports or internal signals across which data checking is performed (i.e. switch ports, internal CPU bus to memory interface)
- Data abstraction for data movement through these ports
- How the abstracted data is expected to change and move through DUV (requiring a reference model)

Figure 3.6 shows a pictorial view of data checking environment. Note that in this diagram, packet source is not indicated. The components of this architecture are:

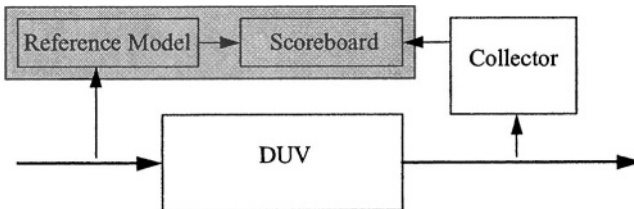


Figure 3.6 Data Checking Architecture

- Reference Model
- Collector
- Scoreboard

The reference model need only be as accurate as the type of checking that is being done. For example, if the data checking is being done at every cycle, then the reference model has to be cycle accurate. Also the reference model only need to describe the DUV behavior across the points where data checking is being performed.

The scoreboard is basically a list that is used to hold the expected data until DUV produces the data that will match that specific data item. The reference model is often encapsulated with the scoreboard. A scoreboard has the following features:

- Allow for ordered and unordered matching of data/transactions
- For ordered matching:
 - Flag skipped expected transactions as error
- For unordered matching:
 - Have a time-out feature for unmatched expected transactions
- Provide for initial ignore of mismatched data (in case of devices that require to sync up before operating correctly)
- Provide end of simulation check to make sure all expected transactions have found a match

The design of the collector should also reflect the abstraction used in comparison. For example, if the data is being compared at every cycle, then the collector simply samples signal values at every cycle, but if the comparison is done at a higher abstraction, then the collector must take the necessary steps to extract the necessary data at that level of abstraction.

Questions to answer in designing a data checking strategy are:

- At what abstraction level does the data movement and modification need to be checked?
- Where is the scoreboard placed (the source and destination of data)?
- How do you handle initial and ending conditions (first match, last match)?
- Is the order of transactions important in scoreboarding?

The following examples show the scoreboarding strategy for a memory subsystem.

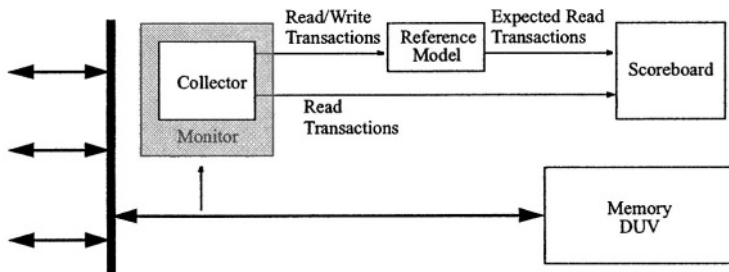


Figure 3.7 Memory Subsystem Scoreboarding

Figure 3.7 shows the architecture for verifying the functionality of the memory system described in the previous section. This architecture consists of:

- A reference model: This reference model includes a memory model. The function of reference model is:
 - Upon a write transaction, it updates its internal memory content
 - Upon a read transaction, it places a read transaction inside the scoreboard. The expected value of the read transaction are extracted from its internal memory model.
- A Monitor: To check that all relevant protocols are correctly followed.
- A collector: extracts memory transactions from the memory interface. Note that in this model, the collector is implemented as part of the monitor.

Figure 3.8 shows an example where the same memory subsystem is used in a system with multiple CPU attachments. The bus fabric, including its arbitration module is now a part of the DUV. In this case, the data checking strategy should check that all transactions generated at the CPU interfaces do in fact arrive at the memory interface port. For this new data checking requirement:

- No reference model is required since transactions are transferred from CPU interface to the memory interface without any modifications.
- Transactions may arrive in different order than the order they were issued because of arbitration in the bus fabric
- Read and Write transactions are checked for both data and address.

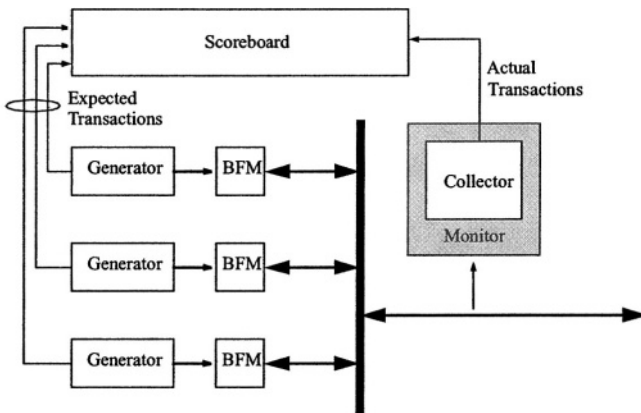


Figure 3.8 System Bus Scoreboarding

3.3 Module Level vs. System Level Verification

During the design flow, modules are initially designed based on input/output specifications. Systems are then constructed by combining these modules where each module port is connected to ports of other modules. During module design phase, most of the errors are in the function of the module. During system level verification, however, the majority of the errors are detected in port interfaces and in the communication between modules.

During module level verification, other modules that drive ports of the module under verification are not present. Therefore such stimulus should be generated by the verification environment while monitors and data checkers operate to verify correct functionality of the module. During system level verification, all or most of the traffic is generated by system modules and the verification environment is mostly responsible for protocol and data checking across module boundaries. It is therefore important to architect a module level verification so that its monitors and data checkers can easily be migrated to the verification environment of the system containing this module.

Figure 3.9 shows a module level verification environment architecture that is readily reusable during system level verification. In this architecture, the DUV module has one input port and one output port. Monitors are attached to these ports to monitor protocols and to collect data from these ports. The collected data from the input port is passed to a predictor that will predict the expected output on the output port. The expected output produced by the predictor is then compared against the data item collected on the output port. In this architecture, input data is applied to the input port by the generator. Therefore it is easier from an implementation perspective to simply have the generator pass the input data to the predictor. However, if extra effort is made to add a collector to the input port to pass the output of this collector to the predictor, then the complete protocol and data checking environment for this module can be reused during system level verification. As shown in this figure, during system level verification, even though the generator module is replaced by another DUV module, the monitors, predictors, and scoreboards are directly reused.

The configuration shown in figure 3.9 is an ideal case where the DUV module implements a simple input/output function. In complex modules, building a scoreboarding strategy that can completely function based on a module's ports requires significant effort. A good strategy for defining module level verification environment features is to first decide which features of this environment should be reused during system level verification and then build the environment so that the features (i.e. monitoring, data checking) are independent of the stimulus generation modules.

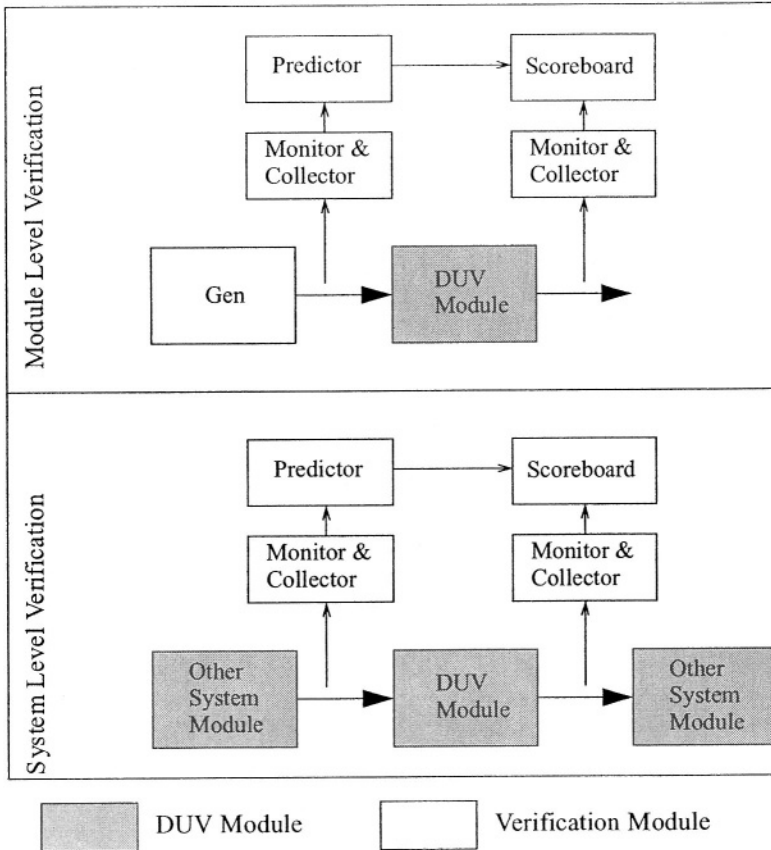


Figure 3.9 Module vs. System Level Verification

3.4 Summary

This chapter presented the architecture of a verification environment and outlined important issues that need to be considered for verification blocks of this architecture. The structure and responsibilities of the generator, the collector, the monitor, and data checking modules were introduced and issues related to the design of these components were enumerated.

PART 2

All About **e**

This page intentionally left blank

e as a Programming Language

The *e* hardware verification language is a high level programming language specifically architected for hardware verification projects. The *e* language is a powerful and productive verification tool not only for its high level programming constructs and features, but also because of its suitability for modeling projects.

Programming languages are identified by the following properties:

- Programming Paradigm
- Execution Flow
- Data Model
- Program Structure

A detailed understanding of these properties is the key to learning any new language. This chapter introduces *e* as a programming language and focuses on its programming properties. The discussion will provide an overview of *e* and describe the basic language constructs necessary for a beginner to write an *e* program. Features of *e* are covered in detail in chapter 5. Verification abstractions and features of the *e* language are discussed in chapter 3.

4.1 e Programming Paradigm

A programming paradigm is a methodology for how software systems are constructed. A programming paradigm provides the programmer with a view of the program execution. A partial list of programming paradigms includes:

- *Imperative Programming*: Traditional view of programming, consisting of a program
-

- state (i.e. data variables) and a sequence of instructions that change this program state.
- *Declarative Programming*: Describes relationships between data variables in terms of fixed rules and produces results during the program runtime using these rules.
 - *Object-Oriented Programming*: Runtime environment is viewed as a collection of communicating objects, each having data members and methods that operate on its data.
 - *Aspect-Oriented Programming*: Runtime environment is viewed as potentially having different views or aspects that require the core implementation to be modified or enhanced based on a specific aspect's requirements.

High level languages are defined and architected to support one or more of these programming paradigms, according to their intended application. To support these paradigms, a language provides specific constructs that closely match the imposed view of the preferred paradigm. For example, the **class** construct in C++ is introduced to support the notion of object-oriented programming. Note that these programming paradigms are generally not contradictory concepts and a programming language may use or support multiple programming paradigms. For example, C++ is an object-oriented imperative programming language.

e is an imperative, declarative, object oriented, aspect oriented programming language. Imperative and object-oriented programming paradigms are commonly used and extensively covered in software programming literature. Declarative and aspect-oriented programming are powerful paradigms supported by the *e* language that can provide great benefits when used appropriately. These paradigms are discussed in more details in the following sections.

4.1.1 Declarative Programming

Declarative programs view the runtime environment as a collection of data variables and a set of fixed rules maintained between these variables. It is the responsibility of the program compiler to use built-in inference rules to declare relationships between variables and use the declarations to perform operations during program runtime.

Declarative programming style is used extensively in solving *Constraint Satisfaction Problems*¹. Constraint satisfaction problems are used in the random generation feature of *e* when constraint declarations are used to specify relationships between fields that are to be generated. The important observation is that these constraint declarations are processed by the *e* compiler and are taken into account during the program runtime every time any of the involved data objects are generated. Consider the following *e* code fragment:

```
┌  
  :  
  :  
  :  
1 : struct data_pair {  
2 :     data1:uint;  
3 :     data2: uint;
```

¹. Constraint-satisfaction problems are mathematical problems where one must find states or objects in a system that satisfy a number of constraints or criteria.

```

4 :      keep data1 > data2;  -- constraint declaration
5 :  };
    :
    :
    :

```

By specifying the constraint declaration describing the relationship between **data1** and **data2** in object **data_pair**, anytime an object of type **data_pair** is generated during the program runtime, the required relationship between **data1** and **data2** is automatically maintained without having to re-specify it again. At the same time, any time **data1** is generated for an object of type **data_pair**, it is always set to a larger value than that of **data2**.

Temporal expressions are another powerful feature of the *e* language. Temporal expressions use events to represent functions of events and data variables at different times during program execution. It is possible to use a declarative statement to define an event to be a function of another event, as shown in the following *e* code fragment:

```

    :
    :
1 :  struct event_holder
2 :      event A is @B; -- event declaration using a temporal expression
3 :  };
    :
    :

```

The result of this declaration is that at anytime during program execution, event A is emitted when event B is executed.

Random generation and temporal expressions are discussed later in their corresponding sections. However the important point in this discussion is that the declarative approach for specifying program properties provides great flexibility in defining program properties that should be maintained throughout the program execution. This flexibility will prove specially useful for verification tasks.

4.1.2 Aspect-Oriented Programming

In programming terminology, a *concern* is defined as the problem a program is trying to solve. *Separation of concerns* is an important goal in program design where a program is broken up into distinct features that overlap as little as possible. The main problem that a program is trying to solve is called the *core concern* of that program. An *aspect* is defined as part of a program function that *cross-cuts* its core-concerns, therefore violating the separation of concerns requirement.

Another possible view of aspect oriented programming is that every major feature of a program is an aspect. A program is then created by weaving these aspects into the implementation of that program's core concern.

Consider an ethernet port interface module that provides an internal DMA-based bus interface for its local interface. In the terminology of software programming, the core concern of this module is to provide local bus to ethernet port interface. The core concern can be broken into two distinct concerns consisting of the DMA controller and the ethernet Media Access Controller (MAC). The separation of concerns is maintained for these two since the operation of these two modules are essentially independent and communicate only through their respective ports. Reporting and coverage collection are two verification related aspects of this module. Verification status reporting requires that both these modules be enhanced to print messages when necessary during simulation runtime, and coverage collection requires that each module be enhanced to collect coverage data during the simulation runtime. An aspect that relates to the function of these modules may require that the range of supported packet sizes be reduced for specific applications in which case, both sub-modules must be modified to reduce the range of supported packet sizes.

In concrete terms, aspect-oriented programming allows for redefining and customizing objects representing modules or data, for the specific requirements of their intended usage. In this type of programming, a core implementation is created that solves the main problem, models the main functionality of a system, or represent a data item. Language constructs that support aspect oriented programming are then used to customize this core implementation without modifying the original program that models the core implementation.

Aspect-oriented style of programming is supported extensively in the *e* language. The definitions for data types, objects, and methods can all be extended after the core implementation is completed. This feature of the *e* language is particularly powerful in a verification project where the verification scenario can be viewed as an aspect of the verification environment, where the core *e* program is extended to create the aspect that represents that specific verification scenario.

4.2 *Struct and the Struct Instance Hierarchy*

In *e*, objects are modeled using the **struct** construct. Structs may have different types of members including data, events, coverage groups, methods, etc. However, struct members that contribute to the struct instance hierarchy consist of a single or a list of predefined scalar types, user defined scalar types, or user defined object types. For example, a data packet containing a payload field of eight bits, a parity bit field, and a list of 4 flag bits is modeled by the following *e* code fragment:

```
1 : struct data_payload {  
2 :     data: uint(bits:8);  
3 :     parity: bit;
```

```

4 :     flag_bits[4]: list of bit;
5 : };
      :
      :

```

In this example, the **struct** keyword is used to define the **data_payload** object. A **struct** may also include other previously declared structs as one of its members. To define a data packet that includes a data payload object:

```

      :
      :
1 : struct data_packet {
2 :     header: uint(bits:4);
3 :     payload: data_payload;
4 : };
      :
      :

```

The *struct instance hierarchy* for a struct definition is a tree of struct instances whose leaves are scalar types and whose nodes are data objects instantiated in its parent struct. For example, the struct hierarchy for the **data_packet** object is shown in figure 4.1.

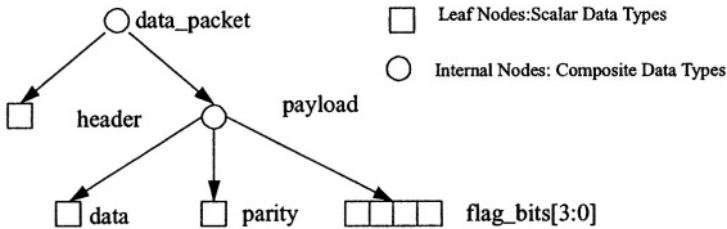


Figure 4.1 Struct Hierarchy for data_packet

As with all object oriented programming languages, methods can be defined for a **struct**. For example:

```

      :
      :
1 : struct data_payload {
2 :     data: uint(bits:8);
3 :     parity: bit
4 :
5 :     set_parity(p: bit) is {
6 :         parity = p;
7 :     };
8 : };
      :
      :

```


In *e*, every struct object has a number of predefined methods. These predefined methods are the mechanism used to control the order of execution in the runtime environment, as will be shown in section 4.3.

4.2.1 Data References

In an *e* program, *Data References* are used to access data objects². The usage of references in *e* contrasts with using both pointers and references in programs like C++. Reference accesses have multiple advantages over pointer access:

- Dereference operators are not needed with references, while they have to be used with pointers. This leads to cleaner code.
- The memory location pointed at by a reference is implicitly managed by the program and can even change during the program runtime during garbage collection. Therefore the programmer does not need to explicitly manage memory addresses.

4.2.2 global and sys

All data objects in *e* reside in the global name space **global**. It is important to keep this organization in mind for a number of reasons.

global is the root of all name resolution scopes. This means that global is the last scope checked for a data reference that is not declared in a local scope (i.e. within a struct, or a method body). As a consequence, any declaration made in **global** can be used in any context in an *e* program, assuming that a local declaration does not hide the declaration at the global level.

global also contains many data structures that are used by the *e* runtime environment to monitor and control program execution. At times it is useful to peek into these data structures to learn about the runtime environment of the program.

global contains the predefined struct **sys**³. **sys** plays an important role both in instantiating user defined data and also in the order of program execution. All user data should be instantiated under **sys**. In that sense, **sys** is the root of the struct hierarchy containing all user data. Although it is possible for the user to instantiate data objects under global, this method is not generally recommended because of the special handling of **sys** during program execution.

As mentioned, **sys** is a predefined object and therefore adding user defined data instantiations to **sys** is only possible through extension of its base definition. This is only possible through an aspect oriented utilities provided in *e* that allow the definition of structs⁴ to be changed without modifying the base definition. For example, an instance of **data_packet**, as

² In an *e* program, pointers can be used but only for parameters for method calls.

³ **sys** is in fact a **unit** which is a special form of a **struct**. Units are described later in this book.

⁴ **struct** is only one of extendible constructs in *e*. Other extendible language constructs are discussed in section 4.5.4.

defined in section 4.2, is added to the struct hierarchy using the following *e* code fragment. In addition, the definition for `data_payload` struct is extended in this example to include the new member `checksum`.

```

      :
1  :  extend data_packet {
2  :      checksum: uint(bits:8);
3  :  };
4  :
5  :  extend sys {
6  :      dp: data_payload;
7  :  };
      :

```

Note that extension to a **struct** may in fact exist in either the same file or one separate from the file containing the original definition, but the final effect after reading all files in the compiler is the collective result of all extensions to a struct definition. The **global** struct hierarchy for the above example is shown in figure 4.2. This figure also shows other members of **global** struct hierarchy that is created by the *e* program runtime environment. Only the struct hierarchy below `sys` is shown in expanded form in this figure.

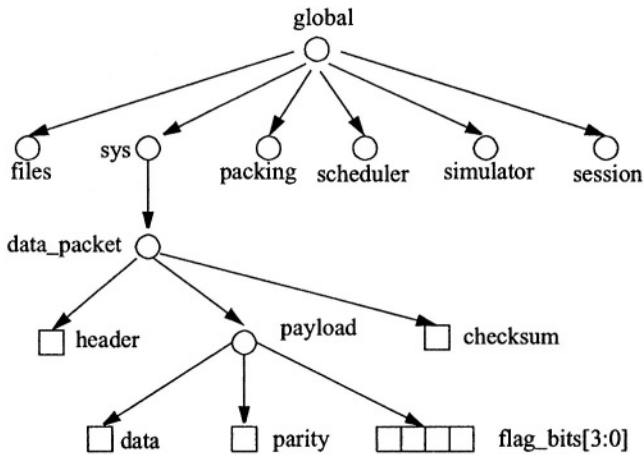


Figure 4.2 **global** Struct Hierarchy

4.3 Execution Flow

The *e* language is architected for the specific requirements of verification projects. As such, *e* uses a predefined execution flow that is more appropriate for verification tasks. Clear understanding of this flow is essential to learning and programming in *e*. This section describes the execution flow of an *e* program.

In most programming languages, program execution starts from a predefined procedure call (i.e. `main()` in C or C++) and follows a path dictated by the user's software program. The execution flow in *e* however, follows a predefined order, which is built into the runtime engine that executes an *e* program. As mentioned earlier, this predefined order of execution is organized to fit the requirements of a verification project.

Verification tasks are generally divided into a number of phases. These phases directly reflect the verification requirements of the methodologies used for verifying complex digital systems. These phases are broadly divided into:

- Initialization
- Pre-Run Generation
- Simulation Run
- Post-Run Result Checking
- Finalization

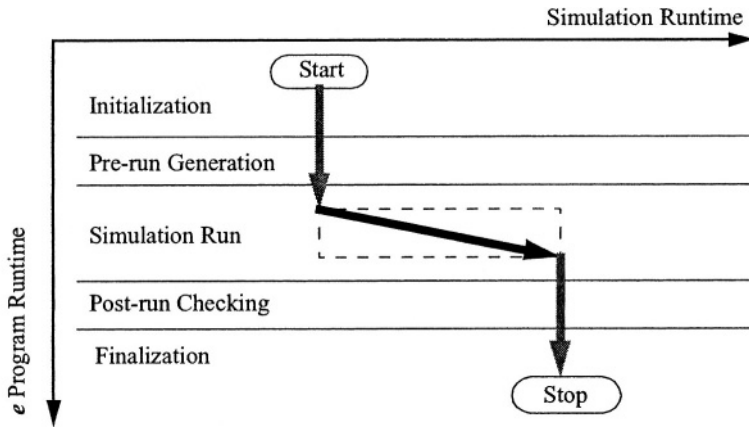
Initialization is the beginning phase where the environment is initialized with values that define the specific verification behavior for the current simulation. During the generation phase, the stimulus and environment settings are generated. Actual design simulation takes place during the simulation run phase. During the post-run result checking phase, simulation results are checked. During finalization phase, verification activity is summarized and finalized. Note that it is only during simulation runtime that simulation time is advanced. Other verification phases are performed in zero simulation time.

The *e* runtime environment follows an implicit order of execution that mirrors these verification phases. The implicit execution flow for an *e* program is shown in figure 4.3. Note that in this diagram, simulation time is advanced only in the simulation runtime phase. The notion of advancing time is discussed in more detail in section 5.3.1.

4.3.1 Merging User Code into the Implicit Execution Order

In an *e* program, every **struct** has predefined methods that correspond directly to the implicit program execution order. Predefined methods of a **struct** that relate to this execution order are:

- `init()`
- `pre_generate()`
- `post_generate()`
- `run()`
- `check()`

Figure 4.3 *e* Program Runtime Phases

- `finalize()`

Because of its special role in the execution flow, `sys` has additional predefined methods that can be extended. One important predefined `sys` method is `setup()` which is called during the initialization phase.

The runtime engine for an *e* program executes the predefined methods of every struct in a predefined order. The `do_test()` pseudo code in figure 4.4 shows this predefined order. Upon running an *e* program, the execution flow follows the order shown in `do_test()`. The user program is merged into the main program execution flow by extending the predefined methods marked in figure 4.4 with bold typeset⁵. Method extension will be discussed as part of extension mechanisms.

Note that in the actual implementation of the runtime environment, predefined global methods are used to create the flow shown in figure 4.4. In addition to creating this flow, these predefined global methods handle environment management tasks.

4.3.2 Steps to Writing an *e* Program

The programming steps required to complete a software program in *e* are:

- Design the program using object oriented programming principles
- Define user defined objects using structs
- Create user data struct hierarchy by extending `sys`

⁵. The `generate()` method is the method that assigns random values to scalar struct members. Even though this method can in principal be extended, this extension is not recommended.

<pre>do_test() is { sys.setup(); sys.do_generate(); sys.do_run(); sys.do_check(); sys.do_finalize(); };</pre>	<pre>do_run() is { me.run(); for each member M of me { if (M is a list of struct) { for i from 0 to M.size() { M[i].do_run(); }; } else if (M is composite) { M.do_run(); }; }; };</pre>
<pre>do_generate() is { me.init(); me.pre_generate(); while (more members to generate) { M = me.next_member_to_gen(); if (M is a list of scalar) { generate(M.size()); for i from 0 to M.size() { generate(M[i]); }; } else if (M is a list of struct) { generate(M.size()); for i from 0 to M.size() { M[i].do_generate(); }; } else if (M is scalar) { generate(M); } else if (M is struct) { M.do_generate(); }; }; me.post_generate(); };</pre>	<pre>do_check() is { me.check(); for each member M of me { if (M is a list of struct) { for i from 0 to M.size() { M[i].do_check(); }; } else if (M is composite) { M.do_check(); }; }; };</pre>
<p>Comments: <u>-me</u> refers to struct containing a method <u>-only bold methods can be extended</u></p>	<pre>do_finalize() is { me.finalize(); for each member M of me { if (M is a list of struct) { for i from 0 to M.size() { M[i].do_finalize(); }; } else if (M is composite) { M.do_finalize(); }; }; };</pre>

Figure 4.4 Pseudo Code for do_test(), do_generate(), do_run(), do_finalize(), and do_check()

- Insert user’s program in the predefined execution flow by extending predefined methods of **sys** and other user-defined structs

For instance, the following *e* code fragment shows a simple *e* program that has one instance of **data_packet** and sets the **data_payload** content before the generation phase and sets the parity bit during run time phase:

```

      :
      :
1  :  extend sys {
2  :      my_data: data_packet;
3  :
4  :      pre_generate() is also {
5  :          my_data.payload.data = 8'b0;
6  :      };
7  :

```

```

8 :      run() is also {
9 :          my_data.payload.set_parity(1'b0);
10 :      };
11 : };

```

Observe that predefined methods of data packet could also be extended to accomplish the same behavior.

4.4 Structure of an e Program

4.4.1 Lexical Conventions

User-defined names in *e* consists of any length combinations of alphabet characters, underscore, and digits (i.e. `_`, A-Z, a-z, 0-9). Valid names cannot start with a number. Though underscores may appear at the beginning of a name, this is not recommended as these names have a unique meaning in the *e* language. Also, reserved keywords in the *e* language (see appendix B) cannot be used as user defined names. Figure 4.5 shows examples of valid and invalid names in the *e* language.

Valid Names:	Valid but Not Recommended	Invalid Names
<ul style="list-style-type: none"> • <code>data</code> • <code>data1</code> • <code>data1_</code> • <code>packet_13b</code> 	<ul style="list-style-type: none"> • <code>_data1</code> • <code>_packet22</code> 	<ul style="list-style-type: none"> • <code>1packet</code> • <code>packet%1</code> • <code>packet@port</code>

Figure 4.5 Valid and Invalid *e* Names

e is a case sensitive language. A period is used to designate struct hierarchy traversal (i.e. `packet.header`). Also, in *e*, code blocks are enclosed with braces and end with a semicolon: `{...;...;};`;

4.4.2 Code Segments

e programs are a collection of *Code Segments*. The beginning and end of a code segment are marked with *begin-code* `<` and *end-code* `>` markers. The begin-code and end-code markers must be the only text on the line containing these markers (i.e. the line must start with the marker and end with the marker therefore containing only 2 characters). Each code segment is composed of multiple statements.

The most trivial *e* program (no program) is:

```
1 : <'
2 : >
```

4.4.3 Comments

All text outside code segments are considered to be comments. Additionally, comments within code segments can be marked with a double dash (`--`) or double slashes (`//`). Since multiple code segments can be included in the same file, it is possible to end a code segment, add comments, and then start a new code segment.

The most trivial *e* program fully commented is:

```
1 : this line is a comment since it is outside a code segment
2 : this line is also a comment
3 :
4 : <'
5 : --this is a comment inside a code segment
6 : // this is also a comment inside a code segment
7 : >
8 :
9 : this line is also a comment after the end of previous code segment
10 :
11 : <'
12 : -- more comments in a new code segment
13 : // additional comments for the new code segment
14 : >
15 :
16 : end of file comments here
```

4.5 Statements

An *e* program is collection of code segments each consisting of a number of statements. The more commonly used statements are:

- import statements
- type declarations
- struct/unit declarations
- extensions

These statements are described in the following sections.

4.5.1 Import statements

Import statements are used to tell the compiler to load another *e* program file before compiling the current *e* program file. Import statements must appear before all other statements inside the first code segment in a file. A simple *e* program using the import statement is:

```

1  : First code segment:
2  : ^
3  : import header.e;
4  : import packet.e;
5  : -- other user code after import statements in this code segment
6  : v
7  :
8  : Second Code Segment:
9  : ^
10 : -- import statements are NOT allowed in this code segment
11 : -- other user code in this code segment
12 : v

```

4.5.1.1 Import Order Dependency

When importing files one at a time, care should be taken to import each file only after importing all files that contain declarations used in the file being imported. However this dependency may at times be difficult to track, and also circular dependencies may exist between declarations and instantiations in each imported file. Under such conditions, it is possible to import multiple files at the same time so that the *e* compiler can resolve the dependencies between code segments in these files. The files that are imported together are placed within parentheses as shown in this example:

```

1  : <
2  : import (beader.e, packet.e);
3  :
4  : extend sys {
5  :     p: packet;
6  : };
7  : v

```

4.5.2 struct Declaration Statement

The **struct** statement is used to define new composite data types. Structs contain data members to store data and methods to perform operations on its data members. Structs may also contain other verification related constructs (i.e. keep statements, events, coverage definitions, etc.). These fields are discussed in detail in their corresponding sections.

4.5.2.1 Struct Data Members

Struct data members may be scalar members, list members, or composite data types (i.e. previously defined structs).

Examples of scalar data members include:

```
1 : struct data_holder {
2 :     data1:    uint[0..100](bits:7);
3 :     flag:     bool;
4 :     time1:    time;
5 :     data2:    uint(bits:16);
6 :     packet1:  packet;
7 : }
```

As shown in this example, other user-defined structs (i.e. **packet**) can be used as a type for a struct member. Examples of list data members include:

```
1 : struct data_holder {
2 :     data_list: list of uint;
3 :     bool_list: list of bool;
4 :     data_list[16]: list of uint;
5 :     packet_list[2]: list of packet;
6 : }
```

In this example, the size of the list is explicitly set to 16. Keyed lists can also be defined to allow for searching lists using values of list members.

4.5.2.2 Methods

Methods are struct members that are used to perform operations on struct data members, or any other value that the method can access. Methods are either procedures or functions and may or may not consume time. Examples of methods are:

```
1 : struct data_holder {
2 :     data: uint (bits:16);
3 :
4 :     set_data(val: uint (bits:16)) is {
5 :         data = val;
6 :     };
7 :
8 :     get_data_version 1(): uint (bits:16) is {
9 :         return data;
10 :     };
11 :     get_data_version2(): uint(bits:16) is {
12 :         result = data;
13 :     };
14 : }
```

Methods are defined by specifying the name, the parameter list, the return type (if any) and the actions in the method. The above example shows two variations of returning a value in a method. In the first version (`get_data_version1()`), the value for data is explicitly returned by issuing a **return** statement. In the second version (`get_data_version2()`), the return value for the method is returned by assigning it to the reserved keyword **result**. When the method returns after completion, the value of result is returned.

4.5.3 Type and Subtype Declaration Statements

The following Boolean and numeric types are predefined in *e*:

- int,
- uint,
- bit,
- nibble,
- byte,
- time,
- bool

New scalar subtypes, enumerated types, and struct subtypes can be defined in an *e* program. The following sections provide an overview of these constructs.

4.5.3.1 Enumerated Type Declarations

Enumerated types are defined by specifying the possible values for the new type as shown in this code fragment. The new type can then be used to define the type for new data objects.

```

1  : type color_t: [RED, BLUE, GREEN];
2  :
3  : extend sys {
4  :     color: color_t;
5  :     run() is also {
6  :         color = RED;
7  :     };
8  : };

```

The enumerated values of the new type can be used in all expressions including the assignment shown in this example.

4.5.3.2 Scalar Subtype Declarations

Scalar subtypes are defined by modifying the bit size and/or the range for a predefined type. If both bit size and the range are specified for a subtype, then the smaller range will be the effective range of a scalar subtype.

```

1 : type int_subrange: int [0..100] (bits:12);
2 : type int_subrange1: int[0..100] (bits:3);
3 : type int16: int (bytes:2);
4 : type int100: int [0..100];

```

The subtype declaration shown in this example limits the range for `int_subrange` to between 0 and 100, and sets the size of the field to 12 bits. This example limits the range for `int_subrange1` to 0 to 7 since the bit size of 3 only allows for values up to 7 to be represented. Either the bit size or the range can be omitted in this construct as shown in this example.

4.5.3.3 Struct Subtype Declarations

When declaring a struct, it is possible to use one of its members as a subtype determinant. The struct definition can then be customized for its different subtypes indicated by different values of this subtype determinant field.

In this example, a pixel is defined to have either a RED or a BLUE color. Depending on its color, the tone is defined to have different types representing different variations of BLUE or RED. The `when` construct is used to define subtypes of pixel so that the struct members are different depending on the value of color field. This example also shows the syntax for instantiating these subtypes where they are instantiated under `sys`. Figure 4.6 shows the instance hierarchy starting at `sys` for this example.

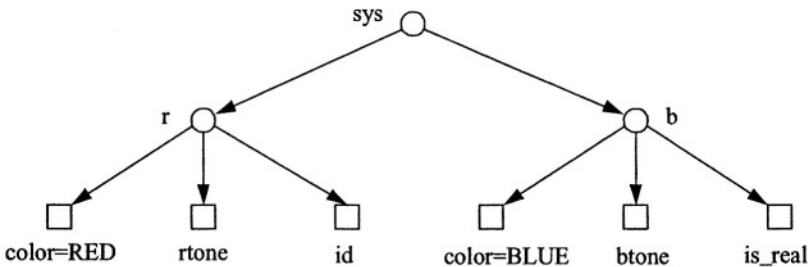


Figure 4.6 Struct instance hierarchy for struct subtypes

```

1 : <
2 : type color: [RED, BLUE];
3 : type red_tone: [RED1, RED2, RED3];
4 : type blue_tone: [BLUE1, BLUE2, BLUE3];
5 : struct pixel {
6 :     color1: color;
7 :     when RED pixel {
8 :         rtone: red_tone;
9 :         id: uint;
10 :     };
11 :     when BLUE pixel {
12 :         btone: blue_tone;
13 :         is_real: bool;
14 :     };
15 : };
16 :
17 : extend sys {
18 :     r: RED pixel;
19 :     b: BLUE pixel;
20 : };
21 : >

```

4.5.4 Extension Statements

The aspect oriented features of e allows different language constructs to be extended after their initial definition. The constructs that can be extended are structs/units, methods/TCMs, enumerated types, coverage groups and items, and events. Extension of structs, methods in structs, and enumerated types are described in this section. Extension of remaining constructs are described in their corresponding chapters.

Enumerated types are extended as shown in this example:

```

:
:
1 : type color: [RED, BLUE, GREEN];
2 : extend color: [YELLOW];
:
:

```

After processing the two lines, the definition for color will include four colors of RED, BLUE, GREEN, and YELLOW. Note that the original declaration of a type and its extensions may be located in different files. An enumerated type can also be extended multiple times.

Structs can also be extended using the extend mechanism. For example:

```

:
:
1 : struct packet {
2 :     header: packet_header;
3 : };

```

```
4 :  
5 : extend packet {  
6 :     data[16]: list of uint(bits:16);  
7 : };  
  
    :  
    :  
    :
```

In addition, method definitions in a struct may also be extended. Methods are extended using keywords of **is empty**, **is undefined**, **is first**, **is only**, and **is also**.

For example, the following code adds **outf** actions to the **set_data()** method so that a text is printed before and after the assignment operation in this method:

```
    :  
    :  
1 : struct data_holder {  
2 :     data: uint (bits:16);  
3 :  
4 :     set_data(val: uint (bits:16)) is {  
5 :         data = val;  
6 :     };  
7 :  
8 :     set_data(val: uint (bits:16)) is first {  
9 :         outf("this text is printed before val is assigned to data");  
10 :     };  
11 :  
12 :     set_data(val: uint (bits:16)) is also {  
13 :         outf("this text is printed after val is assigned to data");  
14 :     };  
15 : };  
  
    :  
    :  
    :
```

To override the definition for a method, the **is only** keyword is used. The definition for the **set_data()** method is changed in the following example to assign the value (**val+1**) to the data field.

```
    :  
    :  
1 : struct data_holder {  
2 :     data: uint (bits:16);  
3 :  
4 :     set_data(val: uint (bits:16)) is {  
5 :         data = val;  
6 :     };  
7 :  
8 :     set_data(val: uint (bits:16)) is only {  
9 :         data = val +1;  
10 :     };  
11 : };  
  
    :  
    :  
    :
```

When using **is empty**, an empty action block is defined for a method. This means that no error will be reported when using this extension mechanism. Using **is undefined** removes any previous block defined for a method. The **is undefined** extension for a method can only be used after defining an action block for that method.

Extensions to a method are applied in the same order that the compiler processes the extension definitions. For example, if the first extension that is processed is an **is only** type, then the original definition of the method is completely replaced and all following extensions are applied to the new method definition. The method signature (name, parameter list, return type) should be exactly the same in the original definition as all extensions of that method.

4.6 Concurrency and Threads

Concurrency is a powerful programming concept that allows for intuitive and effective modeling of systems that are composed of concurrently operating objects. Due to inherent concurrency of hardware models, and the requirements for interacting with such models, supporting concurrency is in fact considered a requirement for languages that are used for design and verification of hardware systems.

The two fundamental concepts in concurrent programming are processes and resources. A process refers to sequential execution of a task and has its own thread of execution. Resources refer to the necessary elements required for executing a process (i.e. memory space, I/O devices, etc.). A concurrent program consists of two or more processes. Processes that share the same address space are called light-weight-processes or threads. Concurrency support in most programming languages is targeted for the same address space as the program itself. Sharing the address space allows multiple threads to access the same variables in the program runtime environment and therefore allows threads to communicate through these shared variables. The processes in an *e* program share the same address space and are therefore considered threads. Therefore this discussion is focused on threads.

In a multi-threaded language such as *e*, mechanisms should be provided for suspending a running thread and to restart a suspended thread in order to give all threads an opportunity to execute. Similarly, the language should provide support for communication between multiple threads. To support concurrent programming, a language should provide utilities for supporting:

- Concurrently running threads
- Thread synchronization
- Starting new threads
- Suspending a running thread
- Ending (removing) a running thread
- Restarting a suspended thread

The *e* programming language has full support for concurrent programming. *Events* and *Temporal Expressions* are used for thread synchronization (i.e. deciding on when to start a new thread or restart a suspended thread). Additional constructs are provided for thread control, including creating new threads and suspending threads. *Time Consuming Methods* (TCMs) are used to further facilitate start of new threads and for enabling more detailed thread synchronization. These language features are discussed in the subsequent sections.

The runtime environment for an *e* program consists of a thread scheduler that has knowledge of all software threads at any given time during the program runtime. At each iteration of its main loop, this scheduler identifies all qualified threads (i.e. threads that are ready to be resumed based on their restart conditions). It then restarts each qualified thread from the point where it was suspended until that thread suspends under its own control (i.e. by calling the **wait** or **sync** action, calling a TCM which includes an implicit **sync** at its sampling event, or terminating). A new iteration starts when all qualified threads at the previous iteration have been restarted. Each iteration of this scheduler corresponds to one tick of the runtime environment.

4.6.1 Events and Temporal Expressions

In the *e* language, events are used to synchronize execution between threads⁶. Events can be emitted explicitly or implicitly. The **event** keyword is used to define an event. The **emit** action is used to explicitly emit an event. Events are emitted automatically by describing an event as a function of other events and Boolean conditions. Temporal expressions are used to describe such functions.

The following *e* program shows an example of defining and emitting named events. It also shows how events are used with the **on** construct.

```

1 : struct transmitter {
2 :     event transmit_completed;
3 :     transmit_count: uint;
4 :     keep transmit_count == 0;
5 :
6 :     on transmit_completed {
7 :         transmit_count += 1;
8 :     };
9 :
10 :    post_transmit_operations() is {
11 :        emit transmit_completed;
12 :    };
13 : };

```

⁶. Events are also used to synchronize to external simulators. This topic is discussed as part of verification support in the *e* language.

The *e* language defines a number of predefined events that are emitted by the *e* program execution environment. The most fundamental predefined event is **sys.any**. This event is emitted for every iteration of the thread scheduler.

4.6.1.1 Temporal Expressions

Temporal Expressions are used to describe temporal behavior using events and temporal operators. Temporal expressions produce events that are triggered when the temporal expression evaluation succeeds. Temporal expressions can be used in the following constructs:

- **wait** and **sync** actions in time consuming methods
- to define named events, and in **expect** or **assume** struct members

A temporal expression is evaluated at every occurrence of its *sampling event*. A sampling event is defined by attaching it to a temporal expression as follows:

```
TE @sampling-event
```

The **sys.any** predefined event is the default sampling event for all temporal expressions. A *sampling period* is the time between current occurrence of a sampling event and the previous occurrence of a sampling event. Sampling period is an important concept in evaluating temporal expressions since the temporal operators are based on how temporal expressions evaluate during the sampling period.

All temporal expressions are constructed by combining *Base Temporal Expressions* using temporal operators. The base temporal expressions are:

```
Event Base TE:           @named-event @sampling-event
Change Base TE:         rise(scalar exp) @sampling-event
                        fall(scalar exp) @sampling-event
                        change(scalar exp) @sampling-event
Boolean Base TE:        true(boolean-expression) @sampling-event
```

For event base temporal expressions, the temporal expression succeeds if **named-event** is emitted at any time during the sampling period specified by the sampling event of the temporal expression. In contrast, a Boolean base temporal expression succeeds only if the **Boolean-expression** evaluates to true when the sampling event is emitted. In other words, a Boolean base temporal expression is evaluated only when the sampling event is activated. Similarly, scalar expressions for a change base temporal expression are evaluated only at occurrence of the sampling event. Figure 4.7 shows examples of how these base temporal expressions are evaluated.

Some commonly used temporal operators are:

```
Logical Operators:      (not TE), (TE and TE), (TE or TE)
Transition Operators:  rise(scalar exp), fall(scalar exp), change(scalar exp)
Fixed Repetition Operator: [exp]*TE
Sequence Operator:     {TE; TE}
```

Examples that use these temporal operators include:

```
TE1:    ((not @A) or @B) @clk
TE2:    ([12]* @A) @clk
```

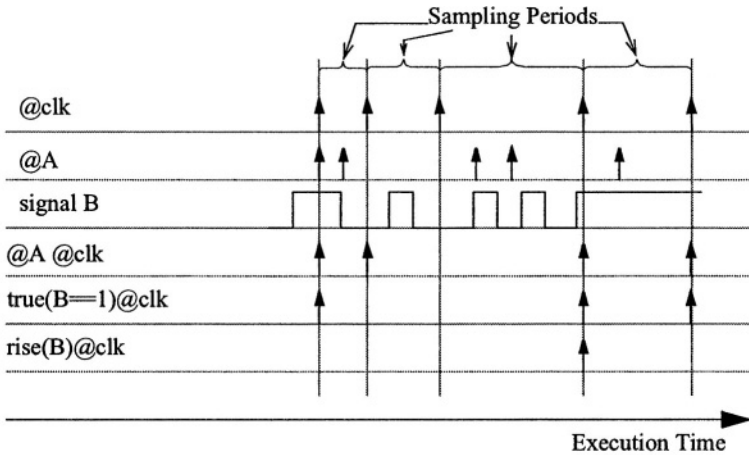



Figure 4.7 Evaluating Base Temporal Expressions

TE3: {not @A; [3]*@B} @clk
 TE4: cycle @clk

TE1 succeeds when event **A** is not detected or event **B** is detected during a sampling period of event **clk**. TE2 succeeds when event **A** occurs 12 times in the previous 12 sampling periods defined by event **clk**. TE3 succeeds when event **A** is not detected during one sampling period and event **B** is detected in the following 3 sampling periods of event **clk**. In TE4, cycle is an alias for the sampling event (i.e. @clk). In essence, TE4 succeeds when event @clk occurs. Temporal operators are discussed in detail in chapter 9.

4.6.2 Time Consuming Methods (TCMs)

Time consuming methods (TCMs) are methods that consume simulation time. TCMs are created by tagging a method with a sampling event. TCMs may contain synchronization actions (which use temporal expressions). The default sampling event of a TCM is used as the default sampling event for any temporal expression that is evaluated as part of executing that TCM.

In this example, method `inject_packet()` is a time consuming methods since it is tagged with the default sampling event `@clk`.

```

    :
    :
1  : struct packet {
2  :     inject_packet() @clk is {
3  :     };
4  : };
    :
    :

```

TCMs may be called only from other TCMs while regular methods may be called from both TCMs and regular methods.

4.6.3 Thread Control

In multi-threaded languages, the program runtime for a program execution consists of many *Runtime Cycles*. A runtime cycle refers to a slice of time where all *Qualified Threads* are given a chance to execute. A qualified thread during a runtime cycle is a thread whose requirements for reactivation are satisfied. During each such cycle, all qualified threads are executed until each thread either exits or is suspended. At this time, the current runtime cycle is completed and a new runtime cycle is started. Within the context of hardware simulation, advancement from one runtime cycle to the next is used to model progression of simulation time⁷, while the execution within a runtime cycle is assumed to take place in zero simulation time.

To support concurrent programming, the *e* language provides:

- Concurrency Actions: **start**, **first of**, **all of**
- Synchronization Actions: **wait**, **sync**

The **start** action is used to explicitly start a new thread. The thread that issued the start action, and the new thread become independent threads that are suspended and restarted independently of each other and based on their suspend and resume conditions. The **first of** and **all of** actions are used to start new threads while suspending the thread that issued these actions. For the **all of** action, the original thread is resumed after all started threads are completed. For **first of** action, upon completion of any of the threads that were started, all started threads are ended and the original thread continues its execution. A visual representation of this behavior is shown in figure 4.8.

wait and **sync** actions both cause the executing thread to suspend until the temporal expression attached to these actions succeeds. For **sync** action, the thread continues to execute if the temporal expression attached to the **sync** action succeeds in the same tick. The thread suspended by a **wait** action requires the sampling event of the TCM to emitted once before it resumes. A visual representation of this behavior is shown in figure 4.9.

The following example demonstrates how new threads are suspended and resumed using the **wait** and **sync** actions:

```

1 : struct packet_injector {
2 :     event clk;
3 :     injector() @clk is {
4 :         wait cycle;
5 :         inject_next_packet();

```

⁷ It is in general possible to have multiple runtime cycles without advancing the simulation time, as is the case with HDL simulator delta cycles.

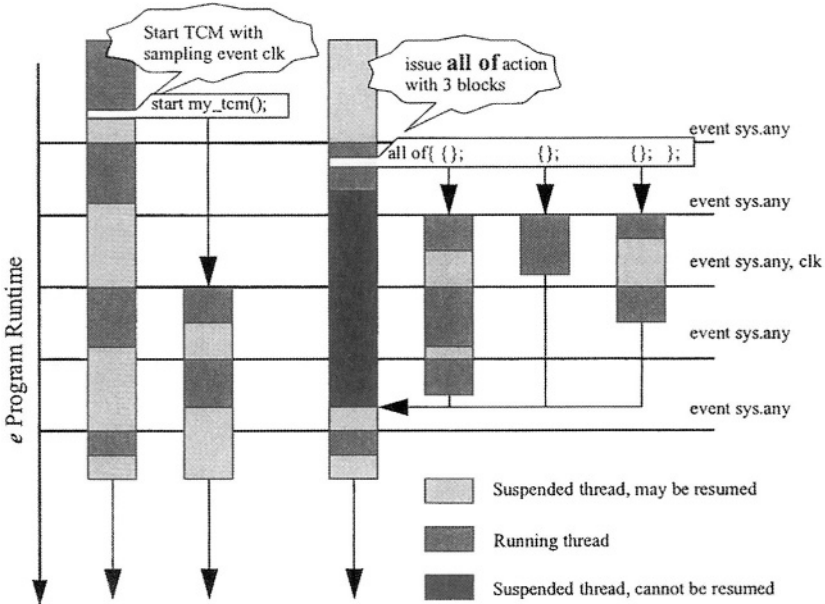


Figure 4.8 Creating New Threads

```

6 :      wait cycle;
7 :      inject_next_packet();
8 :      };
9 :      run() is also {
10 :      start injector();
11 :      };
12 : };
13 : extend sys {
14 :   packet_injector;
15 : };
    :
    :
    :

```

In this example, named event `clk` in `packet_injector` is emitted from other parts of the program not shown in this example. The `clk` event is used as the default sampling event for the `injector()` TCM. The `run()` method of `packet_injector` is extended to start the `injector()` TCM. `injector()` waits for the first occurrence of the default sampling event (i.e. `clk`). It then injects the next packet. It then suspends waiting for the next default sampling event (waiting for a `cycle`). After getting restarted, it injects the next packet. It then returns (i.e. method completes) which terminates the thread started in the `run()` method. At that time, since there are no more sleeping threads, the run phase of program execution completes and the next phase is started.

The use of **first of** and **all of** actions are shown in this example:

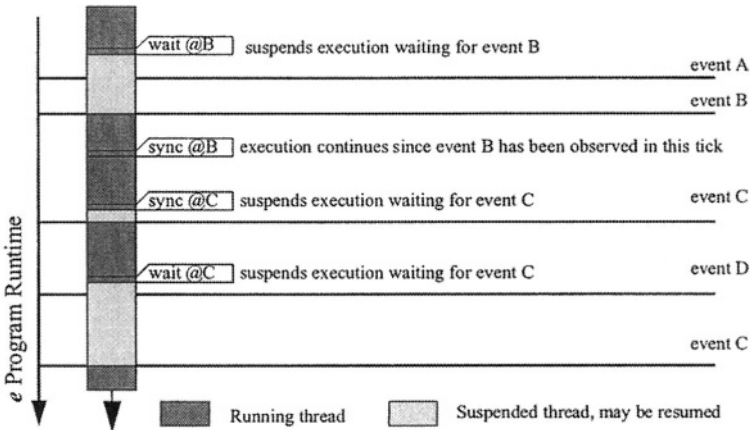


Figure 4.9 Suspending Active Threads

```

1 : struct packet_injector {
2 :     event clk;
3 :     injector() @clk is {
4 :         inject_next_packet();
5 :         first of {
6 :             {wait cycle};
7 :             {wait [2]*cycle};
8 :             {wait [3]*cycle};
9 :         };
10 :     };
11 :     run() is also {
12 :         start injector();
13 :     };
14 : };
15 : extend sys {
16 :     packet_injector;
17 : };

```

In this example, the **first of** statement completes after one occurrence of event `clk` since among the three threads that are started in this action, the first thread completes first and within one cycle. In this example, if the **first of** action is changed to an **all of** action, then this action will complete after 3 occurrences of the `clk` event since the longest running thread takes 3 clock cycles.

Rules to keep in mind when working with threads:

- Only TCMs can be started using the **start** action.
- The **start** action can be called in either methods or TCMs.
- **first of** and **all of** actions can only be used in TCMs.
- The sampling event of a TCM containing a synchronization action (i.e. **wait**, **sync**) is the default sampling event for evaluating the temporal expression attached to these syn-

chronization actions.

- Execution flow moves to the next thread only when a running thread completes or is suspended using a synchronization action.
- **stop_run()** method can be used to explicitly end all running threads.
- Calling a TCM from another TCM results in an implicit **sync** action using the sampling event of the called TCM. Therefore calling TCMs may potentially suspend the current thread.

4.6.4 Semaphores

Semaphores are used to provide mutual exclusion and synchronization across multiple threads. These constructs are described in the following subsections.

4.6.4.1 Mutual Exclusion

It is sometimes required to give a thread exclusive access to a shared resource. The thread that is granted exclusive access is called the locking thread. This exclusive access is necessary in order to prevent other threads from accessing or modifying a resource while the locking thread is temporarily suspended. Consider a TCM that interacts with a DUV port through a multi-cycle handshaking protocol. Clearly, no other thread should be accessing the same DUV port signals while this TCM is suspended because of the protocol requirements.

e provides the **locker** construct when a resource can be used by only one user (i.e. thread) at a time. The following example shows the use of this construct where the resource in consideration is a bus interface TCM:

```

1  :  <'
2  :  struct bus_master {
3  :      lkr: locker;
4  :      event clk;
5  :      write(addr: uint(bits:24), data: uint(bits:32)) @clk is {
6  :          lkr.lock();    -- start mutual exclusion
7  :          -- start multi-cycle bus write operation
8  :          .....
9  :          - end multi-cycle bus write operation
10 :          lkr.release(); -- end of multi-cycle bus write operation
11 :      };
12 : };
13 :  >'

```

In the above, predefined **lock()** and **release()** methods of **locker** are used to mark the beginning and end of code region that should only be used by one thread at a time.

e also provides the **semaphore** construct to provide a general solution to the mutual exclusion (mutex) problem where a resource can be used by N number of users at the same time, where N can be configured in the semaphore. Semaphores in *e* use a fair FIFO type implementation that assures all waiting threads get a fair chance to use that resource.

The above discussion applies only to cases where one resource is being shared across multiple threads. In cases where multiple resources are required for a code segment (i.e. a write operation that requires exclusive access to two or more DUV ports), then use of semaphores as outlined above may lead to a deadlock. For a discussion of mutex algorithms that avoid such deadlocks, the user is referred to any text covering concurrent programming.

4.6.4.2 Thread Synchronization

Thread synchronization is modeled as a producer-consumer type interaction. In this type of synchronization, threads marked as producers can only resume when the resource they have produced has been consumed. Threads marked as consumers can only resume when they consume a product that is made available by a producer. This concept is shown in figure 4.10. Note that multiple producers and multiple consumers can synchronize through the same resource. In this case, a producer will resume when its product is consumed by any of the consumers, and a consumer resumes when a product is made available to it by any of the producers.

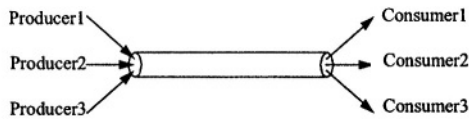


Figure 4.10 Producer-Consumer Model of Thread Synchronization

Rendezvous semaphores are provided in e using the `rdv_semaphore` and are used to implement this type of synchronization. The use of these semaphores is shown in the following example:

```

1  :  <
2  :  extend sys {
3  :      rsem: rdv_semaphore;
4  :
5  :      produce_and_place_in_fifo(id: uint) is empty;
6  :      take_from_fifo_and_consume(id: uint) is empty;
7  :
8  :      producer(id: int) @any is {
9  :          while TRUE {
10 :              produce_and_place_in_fifo(id);
11 :              rsem.up(); -- wait for a consumer to use the resource that was just produced.
12 :          };
13 :      };
14 :
15 :      consumer(id: int) @any is {
16 :          while TRUE {
17 :              rsem.down(); -- wait for a producer to make a resource available
18 :              take_from_fifo_and_consume(id);
19 :          };
20 :      };
21 :
22 :      run() is also {
23 :          start producer(1);

```

```
24 :          start producer(2);
25 :          start consumer(3);
26 :          start consumer(4);
27 :      };
28 :  };
29 :  >
```

In the above, each consumer calls the **up()** predefined method of the semaphore to wait for a consumer to call the **down()** predefined method. Every time a product is consumed by a consumer, both threads for producer and consumer resume. In this case, the thread for producer resumes first, followed immediately by the thread for the consumer.

4.7 Summary

This chapter introduced the concepts and structures in *e* that describe the organization of *e* as a programming language. The discussion presented *e* as a powerful programming language that can be used for effective implementation of general purpose programs. The concepts of aspect oriented programming, declarative programming, and concurrent programming constructs in *e* make *e* a good candidate for a slew of programming environments. The *e* programming language provides an extensive set of abstractions and constructs for verification projects. The verification aspects of the *e* language are discussed in chapter 5.

e as a Verification Language

The *e* language is a powerful hardware verification language architected on the requirements of verification methodologies essential to successful completion of complex verification projects. *e* not only provides the necessary constructs to facilitate the successful and efficient development of verification programs, it also guides the programmer towards a programming style that is better suited to recommended verification methodologies. The execution order of an *e* program, as described in the previous chapter, is one example of the close relationship between *e* and verification projects. This chapter describes the *e* constructs that facilitate the implementation of *e* verification programs.

The abstractions required for implementing a verification program are directly tied into the activities that are performed during a verification project. Verification related constructs and abstractions in the *e* language are in direct correlation with the verification facilities required for implementing these verification activities. Verification activities include:

- Simulation Abstraction
- Generating Stimulus
- Driving Stimulus
- Collecting Device Response and Result Checking
- Measuring Verification Progress and Coverage Collection

Table 5.1 lists these verification activities and the verification facilities and *e* language features that are required for performing these activities. This chapter introduces the *e* language features listed in this table.

Table 5.1: Verification Activities and *e* Language Features

Verification Activity	Verification Facility	<i>e</i> Language Feature
Simulation Abstraction	Notion of Concurrency	Concurrency and Thread Control
Stimulus generation	Constrained Random Generation	Constrained Random Generation
	Stimulus Variation for Specific Verification Requirements	Extension Mechanisms
Driving Stimulus	Interfacing with HDL Simulator	HDL Interface, <i>e</i> -ports
	Associating Verification Objects with Simulation Module Instances	Units
	Synchronizing with HDL Simulator	Events, Temporal Expressions
	Moving from Data Abstraction to Physical Abstraction	Packing
	Applying Stimulus	Methods, TCMs
Result Checking	Collecting data from HDL Simulator	HDL Interface, <i>e</i> -Ports
	Moving from Physical Abstraction to Data Abstraction	Unpacking
	Data Checking	Scoreboarding
	Timing Protocol Checking	Temporal Expression
Coverage	Coverage	Coverage

5.1 Constrained Random Generation

The constrained random generation feature in *e* randomly creates a struct hierarchy for data objects and assigns random values to its scalar fields (leaf nodes)¹. This generation capability is an implicit phase of the execution order in an *e* program when the struct hierarchy rooted at `sys` is randomly generated and populated. Generation can also be used explicitly during the simulation runtime. The pseudo code given in figure 4.4 shows the order of generation for a struct hierarchy.

¹ A struct hierarchy may in fact have different valid structures. Different valid struct hierarchies for the same object type are possible due to: 1) struct members that are lists of structs with potential different size lists, and 2) struct subtypes which may have different members all together depending on the struct type.

In the absence of any generation constraints, generation routines populate scalar fields of a struct hierarchy with fully random values in the range of each scalar type (i.e. an enumerated scalar type is assigned a value from its possible enumerated values).

Fully randomized value assignment is not particularly useful since any realistic scenario for generating values requires valid or interesting ranges to be defined for data objects and that relationships between separate fields are defined during generation. The generation facility in *e* provides a powerful constraint solver engine that allows concise and easy constraint specification for the randomly generated values. In section 5.1.1, the *e* generation mechanism without constraints is described. Constraint specification and usage is introduced in section 5.1.2.

5.1.1 Random Generation

In *e*, the fundamental generation activity is the assignment of random values to members of a struct. For a struct without any subtype definitions, a random value is generated for struct members in their order of appearance in the struct declaration so that for each member:

- if a scalar type: assign random value in its full range
- if a scalar subtype: assign random value in its valid range
- if an enumerated type: assign random value from enumerated literals
- if a list: generate list size, and then generate list items one at a time starting from the first list element
- if a struct: generate this member struct using this same procedure before moving on to next struct member

The steps described above lead to a depth first order of generation for struct members where the struct hierarchy rooted at a struct member, which itself is a struct, is fully generated before moving to the next struct member.

It is possible to prevent generation for a struct member by prefixing that struct member with the “?” character. Such fields will not be assigned during random generation process. The generation order for such an *e* code segment is shown in figure 5.1.

```

1 : '<
2 : type address_type: [LAN, NET];
3 : struct address {
4 :     lan: uint(bits:48);
5 :     type: address_type;
6 : };
7 : struct packet {
8 :     !size: uint; -- will not be generated.
9 :     src: address;
10 :    dest: address;
11 : };
12 : extend sys {
13 :     packet;
14 : };
15 : '>
```

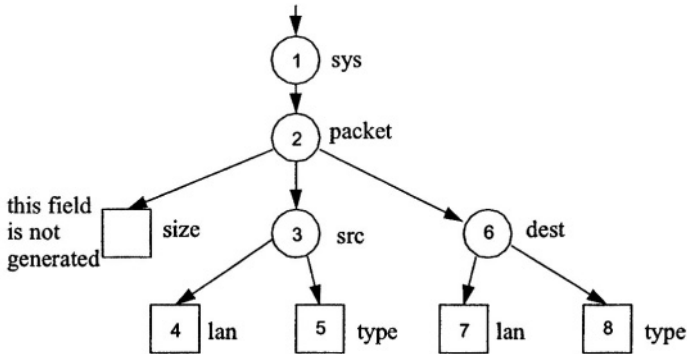


Figure 5.1 Struct Hierarchy Generation Order

Even in the absence of generation constraints, the generation order may change from the default order for a struct that has subtypes. For struct subtypes, one or more members of that struct are used as *subtype determinant struct members*. Subtype determinant members are often used to create struct subtypes that could potentially have different members. Therefore in order to generate a struct subtype, it is necessary to first assign a value to the subtype determinant members of that struct. The generation order is therefore modified to assign a value to subtype determinant members before any of the members specific to that subtype are assigned a value. The affect of subtype definition for the following *e* program is shown in figure 5.2.

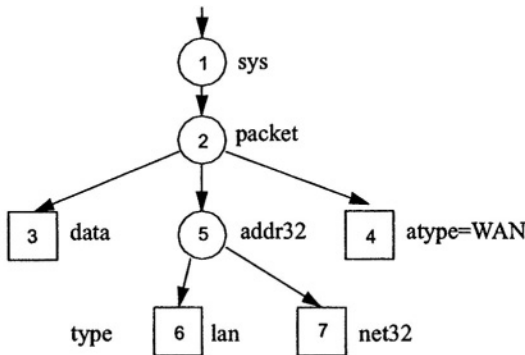


Figure 5.2 Struct Hierarchy Generation Order for Struct Subtypes

```

1 : <'
2 : type address_type: [LAN, WAN];
3 : structaddress_16{
  
```

```

4 :     lan: uint(bits:16);
5 :     net16: uint(bits:16);
6 : };
7 : struct address_32 {
8 :     lan: uint(bits:16);
9 :     net32: uint(bits:32);
10 : };
11 : struct packet {
12 :     data[16]: list of uint;
13 :     when LAN packet {
14 :         addr16: address_16;
15 :     };
16 :     when WAN packet {
17 :         addr32: address_32;
18 :     };
19 :     atype: address_type;
20 : };
21 : extend sys {
22 :     packet;
23 : };
24 : '>

```

5.1.2 Generation Constraints

It is often necessary to constrain the generation process so that a specific verification scenario or a group of verification scenarios can be targeted. *e* provides a powerful constraint solver engine that allows constraints to specify:

- A range of valid values for generated fields
- Relationships that should be maintained between generated fields

The **keep** keyword is used to specify constraints for generated fields. In this example, constraints are specified to limit the valid range for length and height and to define a relationship between these members:

```

1 : struct rectangle {
2 :     length: uint;
3 :     keep length in [1..20];
4 :
5 :     height: uint;
6 :     keep height in [1..20];
7 :
8 :     keep height < length;
9 : };

```

During the generation process, the constraint solver generates values for **length** and **width** so that the specified constraints are met.

Constraint definitions in e are declarative statements, meaning these constraints are declared as struct members and are considered anytime their affected fields are generated anywhere in the e program, or anytime during the simulation runtime.

It is often desirable to have a default constraint for a field, which can later be changed depending on stimulus generation requirements. The **keep soft** keyword is used to define a default constraint that can be overridden in later extensions or in subtypes of a struct using a **keep** statement. For example:

```

      :
      :
1  : type box_type: [SQUARE, RECTANGLE];
2  : struct box {
3  :     type: box_type;
4  :
5  :     length: uint;
6  :     height: uint;
7  :
8  :     keep soft height == length;
9  :
10 :     when RECTANGLE box {
11 :         keep height < length;
12 :     };
13 : };
      :
      :
```

In this example, if the box type is generated to **SQUARE**, then the default soft constraint will guarantee that the generated values for **length** and **height** are equal when randomly generated. However if the type is generated to be **RECTANGLE**, then the constraint defined in the **RECTANGLE** subtype will override the default constraint. During the generation phase, soft constraints are applied only if they do not contradict the combination of all the hard constraints for that field. This means that a soft constraint is applied if it reduces the valid range for a field. Also, soft constraints are applied in the reverse order that they are processed by the e compiler. For example:

```

      :
      :
1  : struct box {
2  :     length: uint;
3  :     height: uint;
4  :     keep soft length == height;
5  :     keep soft length < height-1;
6  :     keep length in [10..20];
7  :     keep height in [1..11];
8  : };
      :
      :
```

In the above example, first the hard constraints on lines 6 and 7 are considered. Then the second soft constraint on line 5 is considered and rejected as it contradicts with the collective

result of all hard constraints. The soft constraint on line 4 is considered next, and is accepted because it can be satisfied while considering the collective effect all previous constraints.

Constraints can affect the generation order that was described in the previous section. Constraints fall into two categories:

- Unidirectional Constraints
- Bidirectional constraints

Bidirectional constraints do not affect the order of generation. *Unidirectional constraints* however require a special ordering of the generated fields and therefore affect the order of generation. Table 5.2 shows examples of unidirectional constraints and the resulting order imposed on fields used in these constraints.

Table 5.2: Unidirectional Constraints

Operation Type	Example	Generation Order
Multiply/Divide	keep area = height *length	height, length, and then area
Method Call	keep area = compute_area(length, height)	length, height, and then area
Bit Extraction	keep length == height[a:b]	a and b, followed by length, followed by height
When Subtype	when SMALL size box {keep length==1};	first size, then length
Explicit Order	keep gen (length) before (height)	length and then height

5.1.3 Pre-Run vs. On-the-Fly Generation

As discussed earlier, a generation phase is performed as part of the execution flow of an *e* program. This implicit generation phase is performed before the simulation run is started and is called the pre-run generation phase. As part of this step, the struct hierarchy rooted at **sys** is generated and populated with random values meeting the declared constraints. Any data object instantiated under **sys** will be generated as part of this pre-run generation phase.

It is possible to generate values during simulation runtime and specify additional constraints during run time generation. In the following example, calling the user-defined method **gen_at_runtime()** during simulation runtime assigns constrained random values to the fields of **packet_injector** struct. Note that constraints are declared for **packet** struct members in the definition of **packet** struct on lines 3 and 5. Additional constraints are also declared for **packet** struct members for the specific instance of this struct in **packet_injector** struct on lines 11 and 12. Additional constraints are also specified during the on-the-fly generation steps for **packet** members, and also for the number of packets to be generated. Due to combination of all specified constraints, during the runtime generation, the generated value for **packet.header** is in the range

4 to 5, value for `packet.data` in the range 3 to 4, value for `num_packets_to_gen` in the range 10 to 50.

```

      :
      :
1   : struct packet {
2   :     header: uint;
3   :     keep header < 10;
4   :     data: byte;
5   :     keep data > 1;
6   : };
7   : struct packet_injector {
8   :     num_packets_to_gen: uint;
9   :     keep num_packets_to_gen in [1..100];
10  :     packet;
11  :     keep packet.header > 3;
12  :     keep packet.data < 5;
13  :
14  :     gen_at_runtime() is {
15  :         gen packet keeping {
16  :             .header in [1..5];
17  :             .data in [3..10];
18  :         };
19  :         gen num_packets_to_gen keeping {
20  :             it in [10..50];
21  :         };
22  :     };
23  : };
      :
      :
```

5.2 HDL Simulator Interface

Seamless interface with the hardware simulation environment is considered another important requirement of a verification language. *e* provides extensive support for HDL simulator access.

Signals in the hardware simulator can be accessed and modified directly from an *e* program. To access an HDL signal, its name is placed in single quotes (*). The following example shows the assignment of value 1'b1 to signal `HDL_signal_name` in the HDL simulator.

```
'HDL_signal_name' = 1'b1;
```

If the name of the signal is provided by a variable, then that variable is further placed in parenthesis before being placed in single quotes. For example, given *e* string variables `signal_name` (a string) and `signal_id` (a uint), the following notation can be used to assign a value to that signal:

```
'HDL_signal_path/(signal_name)_(signal_id)' = 1'b1;
```

The following program shows the interface mechanism for simulator signals:

```

      :
      :
1  :  struct hdl_interface {
2  :      signal_name: string;
3  :      keep signal_name == "~/top/data[8:0]";
4  :      signal_value: uint(bits:9);
5  :
6  :      write_and_read() is {
7  :          signal_value = '(signal_name)';
8  :          '(signal_name)' = signal_value;
9  :      };
10 :      write_and_read_direct() is {
11 :          signal_value = '~/top/data[8:0]';
12 :          '~/top/data[8:0]' = signal_value;
13 :      };
14 : };
      :
      :

```

In this example, method **write_and_read()** uses a string variable as the HDL simulator signal name. In method **write_and_read_direct()**, signal names are specified directly.

When driving HDL signals from an *e* program, it is possible to use **force** and **release** statements to assign and remove the assignment of a hard value to an HDL signal. In *e*, the **verilog variable and vhdl driver** statements are used to describe specifically how an HDL signal is driven from the *e* program (when to drive, how long to drive, when to stop driving).

5.2.1 Multi-Valued Logic

e predefined scalar types support only two-valued logic. As a result, **x** and **z** values are translated into binary values when read from the HDL simulator. When reading HDL signals:

- all **x** values are translated into a binary zero
- all **z** values are translated into a binary one

HDL signals can be driven with **x** or **z** from an *e* program:

```

      :
      :
1  :  struct hdl_interface {
2  :      drive_x_and_z() is {
3  :          '~/top/data[3:0]' = 4'b01xz;
4  :      };
5  : };
      :
      :

```

It is also possible to detect if an HDL signal has an **x** or **z** value. The **@z** operator returns a 1 if an HDL signal has a **z** value. **@x** operator returns a 1 if an HDL signal has an **x** value.


```

1 : struct hdl_interface {
2 :     data: uint(bits:4);
3 :     xdata: uint(bits:4);
4 :     zdata: uint(bits:4);
5 :     drive_and_read_x_and_z() is {
6 :         '~/top/data[3:0]' = 4'b01xz;           -- assigning 4 valued logic to HDL signal
7 :         data = '~/top/data[3:0]';             -- data is set to 4'b0101
8 :         xdata = '~/top/data[3:0]@x';         -- xdata is set to 4'b0010
9 :         zdata = '~/top/data[3:0]@z';         -- zdata is set to 4'b0001
10 :     };
11 : };

```

Comments on lines 7,8, and 9 show the values of **data**, **xdata**, and **zdata** that will be printed after reading signal values from the HDL simulator.

5.3 HDL Simulator Synchronization

The *e* language provides the predefined event **sim** to synchronize operation with the HDL simulator. This predefined event is used as the sampling event for temporal expressions that monitor signal changes in the HDL simulator. The following usage of **sim** event is allowed:

```

event rise_event      is rise('hdl_signal_name') @sim
event fall_event      is fall('hdl_signal_name') @sim
event change_event    is change('hdl_signal_name') @sim
event event_detect    is change('verilog_event_name') @sim

```

After analyzing all such expressions in the *e* program, the *e* compiler automatically creates the necessary interface to the HDL simulator so that a every time a monitored HDL signal is changed, a call back will be made from the simulator to the *e* program runtime environment.

An event that is defined using the **@sim** predefined event can be used as sampling events for TCMs and as part of temporal expressions. In doing so, it is possible to synchronize thread execution in an *e* program to signal transitions in the HDL simulator.

5.3.1 Notion of Time

All thread executions that take place in the same runtime cycle (i.e. tick) are assumed to take place in zero simulation time. When a simulator is attached to the *e* runtime environment, the predefined variable **sys.time** is updated with the HDL simulator time value every time a call-back is made to the *e* runtime environment. In an HDL simulator, it is generally possible for multiple delta cycles to take place at the same simulation time. Under such conditions, multiple

callbacks to the *e* runtime environment may also take place at the same simulation time. The predefined event `sys.new_time` is emitted when simulation time is advanced.

When no simulator is attached to the *e* runtime environment, the `sys.time` variable counts the number of `sys.any` events that have occurred since the beginning of program execution.

5.4 Units

Verification modules that interact with an HDL simulator must be *relocatable*. If a verification module interacts with a number of signals in an HDL module, then the implementation of the verification module should only depend on the HDL module with which it interacts, and should not depend on where in the HDL design hierarchy this HDL module is located. This requirement serves two purposes:

- Simplifies migration from module level to system level simulation
- Improves verification module reusability

During module level verification, an HDL module resides at the top level HDL design hierarchy. During system level verification, however that same HDL module will be placed in its final position in the system level design hierarchy. The same verification module should be usable without any modifications as the project moves from module level to system level verification.

Additionally, an HDL module can be instantiated multiple times in the system level environment. For example, the same ethernet port implementation is instantiated multiple times in an ethernet switch. The verification module for the ethernet port should therefore be usable for all instances of this port without requiring any modifications.

The `unit` construct is used for building relocatable *e* verification modules. Units are the same as structs in all aspects, except the following:

- Units are generated only during the pre-run phase and cannot be generated during runtime.
- Units are instantiated with an additional `is instance` keyword. If the `is instance` keyword is missing for a field, then that field is assumed to be a reference to a unit instance.
- Units are associated with a specific HDL hierarchy path. All HDL signal references inside a unit (unless using an absolute path starting with “~/”) are assumed to be relative to the unit HDL path. This path is specified using the predefined unit method `hdl_path()`.
- The predefined `get_enclosing_unit()` method of a struct or unit instance can be used to find the closest ancestor unit of a given type in its instance hierarchy.
- Units can only be instantiated in units.

These unit features are demonstrated in the following example. In this example, an `hdl_interface` unit is first instantiated under `sys`. The HDL path for this instance is set to the

absolute path of “~/hdl_top”. Unit **hdl_interface** is declared next. Data member **value_to_write** of this unit is a value that will be written to HDL signals. This unit also instantiates an **hdl_writer** unit and sets its HDL path to “channel”. Note that this HDL path is relative to the HDL path of the unit where **hdl_interface** will be instantiated. Next, unit **hdl_writer** is declared. This unit has a string member **signal_name**, which is the target HDL signal for the write operation. This name is constrained to “data,” which again, is relative to the HDL path of the **hdl_writer** unit. **hdl_writer** unit also contains a pointer to the **hdl_interface** unit that contains it. The **get_enclosing_unit()** method is used to constrain the value of this reference to the closest parent of **hdl_writer** which is of type **hdl_interface**. Finally, the **write()** method of **hdl_writer** writes the value of **value_to_write** to the HDL simulator.

```

      :
      :
1   :   unit hdl_interface {
2   :       value_to_write: uint(bits:9);
3   :       keep soft value_to_write in [1..50];
4   :
5   :       hdl_writer is instance;
6   :       keep hdl_writer.hdl_path() == "channel";
7   :   };
8   :   unit hdl_writer {
9   :       signal_name: string;
10  :       keep signal_name == "data";
11  :       hdl_interface_ref: hdl_interface; -- this is a reference
12  :       keep hdl_interface_ref == get_enclosing_unit(hdl_interface);
13  :
14  :       write() is {
15  :           '(signal_name)' = hdl_interface_ref.value_to_write;
16  :       };
17  :   };
18  :   extend sys {
19  :       hdl_if: hdl_interface is instance;
20  :       keep hdl_if.hdl_path() == "~/hdl_top"
21  :   };
      :
      :

```

Analysis of this *e* program shows that after generation and during run time:

- **sys.hdl_interface** has an HDL path of **~/hdl_top**
- **sys.hdl_interface.hdl_writer** has an HDL path of **~/hdl_top/channel**
- **sys.hdl_interface.hdl_writer.hdl_interface_ref** is set to **sys.hdl_interface**
- Method **sys.hdl_interface.hdl_writer.write()** writes the value **sys.hdl_interface.value_to_write** HDL signal **~/hdl_top/channel/data**

5.5 *e*-Ports

Verification code modularity can be drastically improved if the interface for a module is defined using an abstract port model that makes no assumptions about how it is finally connected to other modules. The advantage of this approach is that the internal function of the module can be implemented using the definition for this abstract port, and module connectivity issues can be decided when the module is being instantiated.

***e*-Ports** provide the port abstraction that facilitates the port modeling approach. *e*-Ports can be used to connect:

- *e* modules to *e* modules
- *e* modules to external simulators

e-Ports have the following properties:

- can only be declared inside units
- can be used to pass data or events
- have one of **in**, **out**, or **inout** directions
- are accessed by appending “\$” to the reference name (i.e. **data\$** is the value of port **data**)
- must be bound to either an external simulated object or another *e*-port object. Dangling *e*-ports lead to compilation errors unless they are explicitly indicated by binding them to **empty** port.

Consider a verification module that has an interface consisting of a **read_p** port and **write_p**, and a **clk_p** port that only requires clock transition information. Assume that **read_p** and **write_p** are each seven bits wide. When using this verification module, **read_p**, **write_p**, and **clk_p** may be connected to another *e* module, or to signals in an HDL simulator. Then assume that in one possible configuration, two **verif_module** instances are connected such that each module’s **write_p** drives the other module’s **read_p**. If the **clk** event ports of both modules are driven by an HDL signal named **clk**. Then the implementation of this verification module using *e*-ports is shown in the following *e* program:

```

1  : <'
2  :   unit verif_module {
3  :       write_p: out simple_port of uint(bits:7) is instance;
4  :       read_p: in simple_port of uint(bits:7) is instance;
5  :       clk_p: in event_port is instance;
6  :
7  :       event clk is @clk_p$;
8  :
9  :       write(value: uint(bits:7)) is {
10 :           write_p$ = value;
11 :       };
12 :       read():uint(bits:7) is {
13 :           result = read_p$;
14 :       };
15 :   };
16 :   extend sys {
17 :       vm_1: verif_module is instance;
18 :       keep vm_1.hdl_path() == "~/top";

```

```

19 :
20 :         keep bind(vm_1.clk_p, external);
21 :         keep vm_1.clk_p.hdl_path() == "clk";
22 :         keep vm_1.clk_p.edge() == fall;
23 :
24 :     vm_2: verif_module is instance;
25 :         keep vm_2.hdl_path() == "~/top";
26 :
27 :         keep bind(vm_2.clk_p, external);
28 :         keep vm_2.clk_p.hdl_path() == "clk";
29 :         keep vm_2.clk_p.edge() == rise;
30 :
31 :     keep bind(vm_1.read_p, vm_2.write_p);
32 :     keep bind(vm_1.write_p, vm_2.read_p);
33 : };
34 : >

```

In implementation of **verif_module**:

- **read_p** and **write_p** ports are implemented using the **simple_port** construct, each having type **uint(bits:7)**.
- **clk_p** port is implemented using an **event_port** construct.
- Port values are accessed by using **write_p\$**, **read_p\$**, and **clk_p\$** for both reading the port values and assigning values to ports.
- Event **clk** is defined based on event port **clk_p**.

In instantiating and connecting the two modules:

- event port **vm_1.clk_p** is defined to trigger at the **falling** edge of **~/top/clk** since the effective name of the clk signal is the combination HDL path for **verif_module** instance and **e**-port path name.
- event port **vm_1.clk_p** is defined to trigger at the **rising** edge of **~/top/clk**.
- **vm_1.read_p** is bound to **vm_2.write_p**
- **vm_1.write_p** is bound to **vm_2.read_p**

e-ports provide many options for customization and configuration of the ports. **e** ports can be used to implement buffer ports (queue models). **e**-ports also provide efficient utilities for interfacing to multi-valued signals in the HDL simulator.

5.6 Packing and Unpacking

Two important verification activities are injecting stimulus into the device under verification and collecting device response. It is often the case that abstract data types travel through device ports in a serial bit stream format. An ethernet packet consisting of fields of different lengths travels on an ethernet link in a bit serial format. In a verification program, however, verification data is usually handled at a higher level of abstraction to improve programming productivity. A verification language should therefore provide a utility to translate between an abstract data

type (i.e. an ethernet packet) and a bit serial form. In the *e* language, packing and unpacking mechanisms provide support for this type of translation between logical to physical data abstractions. The following subsections provide an overview of these operations. Packing and unpacking are further described in section 7.3.

5.6.1 Packing

In *e*, structs are used to model abstract data types. A common packing operation is serializing the contents of a struct data object. The following considerations apply when performing a packing operation on an abstract data type modeled by a struct:

- Struct members may need to be serialized in different orders (i.e. first to last, last to first)
- Each scalar struct member may be serialized in a different order (i.e. little-endian, big-endian, and potentially many other variations)
- A struct data object may include composite struct members, necessitating a recursive packing mechanism
- Not all members of a struct may need to be included in the serialized stream since an abstract data type may contain struct members that are used for status and control

The packing mechanism in the *e* language uses the concept of packing options, and physical fields to support these requirements.

The **pack_options** predefined struct is used in the packing utility to specify what order, what direction, and what groupings scalar fields of structs and lists are serialized. This option is also used to specify post processing operations on the final serialized bit stream through swapping bits across different groupings (i.e. swap every two bytes, etc.). The *e* runtime environment provides specific instances of this struct for some commonly used packing requirements. Commonly used predefined instances include **packing.low**, **packing.high**, and **packing.network**². For example, when using **packing.low**, members of a struct are serialized in the order they appear in the *e* program. In contrast, when using **packing.high**, members of a struct are serialized in the reverse order of their appearance in the *e* program. For both these options, the least significant bit of each scalar member appears first in the serialized stream. Using **packing.network** is similar to **packing.high** except that when using **packing.network**, the serialized bit stream is post processed where the final bit stream is byte-order reversed when its size is a multiple of 8. The description of these predefined instances in this section is meant to illustrate the flexibility that the **pack_options** predefined struct afford a programmer in customizing the packing operation for the specific requirements of his verification task.

Struct *Physical Fields* are used to specify members of a struct that should be included in the packing process, struct *Virtual Fields* are all members of a struct that are not physical fields. Physical fields of a struct are marked with a “%” mark before the member name.

² These instances are located under **global** where **packing** is a predefined member of **global**, and **low**, **high**, and **networking** are some predefined members of **packing**.

The syntax for `pack()` method³ is:

```
result = pack(option: pack_options, item1: expression,.....,itemn: expression);
```

Note that multiple items may be passed to the **pack** action. Each item may have a compound or scalar type, and packing options related to the order of including items in the serialized bit stream applies to contents of each compound item and the order of items as listed in the **pack** action. Figure 5.3 shows the pseudo codes for the `pack()` method. As shown, this pseudo code, `pack()` recursively calls the predefined `do_unpack()` method of each composite item to be packed while packing only the physical fields of composite items. The `do_unpack()` method of struct may be extended to modify or enhance the packing behavior. This figure also shows the options in the `pack_options` predefined

<pre>pack(opts, item_list: list of item): list of bit is { result.clear(); while (more items in item_list to pack) { I = next_item_to_pack(opts, item_list); if (I is a list) { do_pack_list(opts, I, result); } else if (I is a scalar) { pack_scalar(opts, I, result); } else if (I is composite) { I.do_pack(opts, result); } }; post_pack_reordering(opts, result); };</pre>	<pre>do_pack(opts, lob: list of bit) is { while (more members to pack) { M = next_member_to_pack(opts, me); if (M is not physical) { continue; }; if (M is a list) { do_pack_list(opts, M, lob); } else if (M is composite) { M.do_pack(opts, lob); } else {-- must be scalar pack_scalar(opts, M, lob); }; }; };</pre>
<pre>Packing Options contains: reverse_fields: used in methods: next_item_to_pack() next_member_to_pack() next_list_item_to_pack() reverse_list_items: used in method: pack_scalar() scalar_reorder: used in method: pack_scalar() final_reorder: used in method: post_pack_reordering()</pre>	<pre>do_pack_list(opts, loi: list of items, lob: list of bit) is { while (more list items to pack) { I = next_list_item_to_pack(opts, loi); if (I is composite) { I.do_pack(opts, lob); } else {-- must be scalar pack_scalar(opts, I, lob); }; }; };</pre>

Figure 5.3 Pseudo Code for `pack()` and `do_pack()`

struct and how they are used during the packing operation.

The following code fragment shows an example of a packing operation.

³ `pack()` and `unpack()` are pseudo-methods. Pseudo-methods look similar to regular methods but are not bound to any structs.

```

      :
      :
1   : struct abstract_subdata {
2   :     valid: bool;
3   :     %val1:uint(bits:2);
4   :     keep val1 == 2'b10;
5   :     %val2: uint(bits:3);
6   :     keep val2 == 3'b100;
7   : };
8   : struct abstract_data {
9   :     !packed_data_low: list of bit; --note: not generated
10  :     !packed_data_high: list of bit; --note: not generated
11  :     data1: abstract_subdata;
12  :     %data2: uint(bits:4);
13  :     keep data2 == 4'b1000;
14  :
15  :     post_generate() is also {
16  :         packed_data_high = pack(packing.high, data1, data2);
17  :         print packed_data_high using bin;
18  :         -- prints bit list: 101001000 (first bit on right)
19  :         packed_data_low= pack(packing.low, data1, data2);
20  :         print packed_data_low using bin;
21  :         -- prints bit list: 100010010 (first bit on right)
22  :     };
23  : };
24  : extend sys {
25  :     abstract_data;
26  : };
      :
      :

```

Observations about this example include:

- Only physical fields (**data1.val1**, **data1.val2**, **data2**) are included in the packing.
- The ordering of scalar fields for serialization spans through compound fields, and across multiple items defined for a pack action. For example, the sequence of scalar items that are packed are (**data1.val1**, **data1.val2**, **data2**). The ordering consideration for **packing.low** and **packing.high** options are applied to this sequence of items and not just to items indicated in the **pack** action.

5.6.2 Unpacking

Unpacking is the reverse operation of packing. During unpacking, a list of bits is split across multiple scalar fields, most commonly the fields of a struct. Unpacking is in general a more complex operation than packing. The reason for this additional complexity is twofold:

- During packing, the size of lists that are participating in packing process are known. During unpacking, however, this size may not be known. Consider unpacking a list of bits of size 10 into two lists of bits whose sizes are not known before hand. How should these 10 bits be divided between the two lists that are the target of unpacking operation?
- Structs may have subtypes that contain different sizes and numbers of physical fields. The struct member that identifies its subtype may in fact be part of the data that is being

unpacked into a struct. Unpacking into such struct data structures may require special handling.

If the target fields of the unpacking operation have no struct subtypes that are indicated by physical fields, and all target lists have predefined fixed sizes, then the unpacking operation is the exact opposite of the packing process. Concepts of **pack_options**, and struct physical fields are used exactly as they are used in the packing operation. The predefined struct **pack_options** is used to specify the order and format that unpacking is done, and serial bits are only unpacked into physical fields. The syntax for the **unpack()** method is:

```
unpack(option: pack_options, serial_data: expression, item1: expression,.....,itemn: expression);
```

Serial data may be represented in many forms (i.e. bytes, list of bit, list of byte, uint, etc.) but the most common form is a list of bits. The serial data is unpacked into the items specified in the unpack method using the same method described for the pack operation. If the size of serial data is less than the minimum amount of data to fill all specified items, then an error condition is reported, but a larger data size is acceptable.

Unpacking for variable size lists and structs with subtypes requires that the default unpacking mechanism modified or extended with additional steps to extract list sizes and identify struct subtypes as information becomes available during the unpacking progresses.

Figure 5.4 shows the pseudo code for the **unpack()** and **do_unpack()** methods. As shown, the **unpack()** method the **do_unpack()** predefined method of structs that they manipulate. Note that during the unpacking process, the order of unpacking follows the order of field definition regardless of the packing options. However, if packing options indicate that the bit stream is packed in reverse order of field definition, then the bit stream is processed from high index bits towards the bit at index 0. To modify the default behavior of unpack operation, this predefined struct method must be extended or redefined.

5.7 Coverage

Measuring verification progress is an essential part of a verification environment that relies on randomly generated values. In this type of randomized environment, it is necessary to keep track of how a specific verification step contributes to the completion, and measuring the total progress of completing the verification plan. Functional coverage collection provides the means to measure the verification progress. Coverage is collected by first defining a metric, which defines the verification progress and then collecting information from the simulation environment to measure this metric. Metrics for measuring coverage are based on:

- Device port traffic
- Temporal and spatial correlations between port traffic
- Device state
- Device state transitions

```

unpack(opts, lob:list of bit, item_list: list of item) is {
  index = 0;
  pre_unpack_reordering(opts, lob);
  --item definition order is used regardless of opts:
  for each item in item_list in order {
    if (I is a list) {
      index=do_unpack_list(opts, I, lob, index);
    } else if (I is a scalar) {
      index = unpack_scalar(opts, I, lob, index);
    } else if (I is composite) {
      index = I.do_unpack(opts, lob, index);
    }
  };
};

do_unpack(opts, lob: list of bit, index:uint): uint is {
  --member definition order is used regardless of opts:
  for each member M of me in order of definition {
    if (M is not physical) {
      continue;
    };
    if (M is a list) {
      index = do_unpack_list(opts, M, lob, index);
    } else if (M is composite) {
      index = M.do_unpack(opts, lob, index);
    } else {-- must be scalar
      index = unpack_scalar(opts, M, lob, index);
    };
  };
  return index;
};

do_unpack_list(opts, loi: list of items, lob: list of bit, index: uint):uint is {
  while (more list items to unpack OR
    if unsized list until no more data to unpack) {
    I = next item in list loi; -- order not dependent on opts
    if (I is composite) {
      index = I.do_unpack(opts, lob, index);
    } else {-- must be scalar
      index = unpack_scalar(opts, I, lob, index);
    };
  };
  return index;
};

```

Figure 5.4 Pseudo Code for `unpack()` and `do_unpack()`

Consider a finite state machine that is specified by its states, state transitions, and input combinations that lead to state transitions. Steps to verifying the functionality of this finite state machine consists of verifying that all possible states are visited, all possible state transitions have occurred, and that state transitions occur for the correct combination of inputs. Collecting coverage on these metrics will provide full confidence that the finite state machine is fully verified when all these metrics are fully covered. Based on this simple example, the following language utilities are required for measuring coverage for a finite state machine:

- Collecting information on signal values
- Collecting information on signal transitions (temporal information)
- Collecting information on combinations of signal values at the same time (spatial information)
- Specifying the times when spatial information should be collected
- Specifying the times across which temporal information should be collected

The requirements for collecting coverage for any verification project can be abstracted to the requirements stated for a finite state machine. Note that although collecting coverage on a state and the input combinations is sufficient for measuring verification progress for a simple finite state machine, the same approach is not practical even for a moderately complex device since state space enumeration is not a practical approach. This means that to collect coverage for a device, all utilities are required so that the coverage collection metrics can be defined in terms of properties that are specifically extracted from the verification plan.

In the *e* language, the concept of *coverage groups* provides coverage collection utilities. A coverage group is a struct member sensitized to an event in that struct. All collection and transition measurement are performed according to the occurrence of this event. A coverage group supports coverage collection for:

- Basic Coverage Items
- Transition Coverage Items
- Cross Coverage Items

Basic Coverage Items are specified to indicate a scalar whose value should be tracked upon the occurrence of the sampling event of the coverage group. This scalar value may be an *e* data object or potentially an HDL signal name. *Transition Coverage Items* collect information on two consecutive values of a previously defined basic item across the sampling event. *Cross Coverage Items* collect information on the cross product of two or more previously defined basic items at the same sampling event.

The coverage group for collecting coverage for a struct representing a finite state machine is shown in this example:

```

1 : type FSM_state: [RESET=2'b00, START=2'b01, LOOP=2'b10, END=2'b11];
2 : struct FSM {
3 :     event clk_rise is rise(~/fsm/clk')@sim;
4 :
5 :     cover clk_rise is {
6 :         item state: FSM_state = '/fsm/state';
7 :         item inp_A: bit = '/fsm/A';
8 :         item inp_B: bit = '/fsm/B';
9 :         transition state;
10 :        cross inp_A, inp_B;
11 :    };
12 : };

```

It is possible to specify additional options for a coverage group based on how information should be collected for its items. Options range from specifying a Boolean expression that qualifies a specific occurrence of the sampling event for coverage collection, to specifying the type of information that should be collected for each item. It is also possible to specify additional options for each item in a coverage group. The full range of these options are discussed in a detailed discussion of coverage constructs.

After the simulation run is completed, the coverage collected during the runtime is analyzed to measure the contribution of that verification run to the completion of all tasks in the verification plan. It is possible to tap into the coverage collection database during runtime to guide the generation towards verification scenarios that have not been generated yet. It is important to observe that gaining more knowledge about the state of verification progress by collecting coverage creates new opportunities for how the verification flow can be improved.

5.8 Summary

This chapter introduced concepts and structures in *e* that describe the organization of *e* as a verification language. The discussion in this chapter highlighted the verification facilities required for effective programming of verification activities and presented the specific features of the *e* language that fulfill those needs.

Chapters 4 and 5 introduce the *e* language as an effective programming and verification language. Details of these constructs are further explained in chapter 5.

This page intentionally left blank

PART 3

Topology and Stimulus Generation

This page intentionally left blank

Random generation is a fundamental operation in implementing a coverage driven verification methodology. The *e* hardware verification language provides constructs that support the full range of random generation requirements.

Detailed understanding of the generation mechanism is an essential part of programming in *e*. The understanding proves specially useful as program size increases and the relationships between generated data items grow in complexity. A detailed understanding of generation is especially useful in both tracing a constraint conflict to its source, and also in making sure that random generation behavior follows the intended implementation.

This chapter provides a detailed look at *e* generation constructs and the details of the generation constraint solver. Issues related to constraints and how they affect both generation and the generation order are also discussed.

6.1 Generator Execution Flow

The fundamental generator activity for a data type is to create its struct hierarchy and to populate the scalar members of this hierarchy with random values. Generation constraints control the hierarchy structure (in case of struct subtypes), the generation order, and also the range for the generated values. This chapter focuses on details of this operation. Issues regarding use of generation during program runtime are discussed in section 6.5.

The following example shows a simple struct definition and constraints for its fields.

```

1 : <
2 : struct data_packet_s {
3 :     data: list of byte;
4 :     keep data.size()== 0x100;
5 :     address: uint;
6 :     keep address in [0x100000..0x200000];
7 :     check: bool;
8 :     keep soft check == FALSE;
9 : };
10 : >

```

In this example, simple constraints are specified for each struct member.

The generator for compound data types (i.e. structs, lists) is implemented in terms of basic operation of generating constrained random values for scalar data items. The **do_generate()** method (figure 4.4) shows the generation execution flow for a compound data type. Note that generation order within a compound data type may be affected by generation constraints, therefore, this flow uses method **next_member_to_gen()** to select the next member that should be generated. This flow implies a depth-first-order for the generation of the struct hierarchy. As shown, the implementation of the generator is based on the following four methods.

- **next_member_to_gen()**
- **pre_generate()**
- **generate()**
- **post_generate()**

The **generate()** method is used to assign a constrained random value to a scalar data item. Once the constraints for a scalar data item are known, then creating a random value in that range is easily accomplished. The power of the generator is therefore in its ability to analyze multiple constraints to choose the next scalar value to generate and the constraints to use during that generation. Section 6.3 presents the *e* constraint solver and section 6.4 shows how constraints can affect generation order. The remainder of this section shows how **pre_generate()** and **post_generate()** predefined struct methods are used to control the generation behavior.

6.1.1 pre_generate()

pre_generate() is a predefined method of all structs. It is initially empty but is extended to indicate the operations that should take place before members of a struct are generated. In this example, the value of member **x** of **xyz_s** struct is assigned in **pre_generate()** method. Note that **x** is designated as a do-not-generate field by prefixing it with “!”, and therefore the value assignment to in **pre_generate()** is not over-written by the generation phase. Running this program has the effect that when instance **xyz_i** in **sys** is generated, after generation **x** will have a value of 10, where **y** and **z** will be generated randomly to satisfy the specified constraint.

```

      :
      :
1  :  struct xyz_s {
2  :      !x: int;
3  :      y: int;
4  :      Z: int;
5  :      keep z = x*y;
6  :      pre_generate() is also {
7  :          x = 10;
8  :      };
9  :  };
10 :  extend sys {
11 :      xyz_i: xyz_s;
12 :  };
      :
      :

```

6.1.2 post_generate()

The **post_generate()** method of a struct is executed after the generator has completed assigning values to all members of that struct. This method is used as a place holder for all post-processing operations that should be performed after random value generation. Post processing activities depend on the specific requirements of a project and may, for example, include computing non-generated struct members from the randomly generated values and printing information on the screen after generation for a struct is completed. User code is added to this method by extending **post_generate()**.

The following example shows the use of **post_generate()** method to compute non-generated values, and to print the generated values on the screen. In this example, a new field **z1** is added to struct **xyz_s**, and in the **post_generate()** method, **z1** is assigned. The value of **z1** will also be printed every time a data object of type **xyz_s** is generated.

```

      :
      :
1  :  extend xyz_s {
2  :      !z1: uint;
3  :      post_generate() is also {
4  :          z1 = z/20;
5  :          print z1;
6  :      };
7  :  };
      :
      :

```

6.2 Constraint Types

Constraints affect both the generation order and generated values. Different types of constraints are defined in this section. These definitions are used in the next sections to describe the behavior of the generation utility.

Simple Constraints are constraints that have only one clause alternative. Some examples of simple constraints include:

```
keep x < 10;
keep not (y in [10..100]);
keep y < x;
keep (y == a * b) and (z == a - b);
keep not (z == filter(x)) and (y in [3..5]);
```

Compound Constraints are constraints with more than one clause alternative. Compound constraints include:

- **or** Constraints: **(A or B)**
- **imply** Constraints: **(A ==> B)**
- Boolean Equivalence Constraints: **(A == B)**
- Boolean Non-Equivalence Constraints: **(A != B)**

Note that **imply** constraints, boolean equivalence constraints, and boolean non-equivalence constraints are in fact short-hand notations for an equivalent **or** constraint¹:

- Constraint **(A ==> B)** is equivalent to **(not A or B)**.
- Constraint **(A == B)** is equivalent to **((not A and not B) or (A and B))**
- Constraint **(A != B)** is equivalent to **((not A and B) or (A and not B))**

Also note that for an **imply** constraint, there is no requirement to generate A before generating B. Either B or A can be generated first to satisfy the required boolean clause.

Some examples of compound constraints are:

```
keep (x < 10) or (x > 100);
keep (x < 10) or (y > 10);
keep (x < a * b) or (y == filter(x));
keep (z < y) ==> (z > 20);
keep (x==1) == (y==2);
```

Note that any constraint can be re-written as a multiple-clause **or** constraint where each clause is a simple constraint.

Constraints are further divided into unidirectional and bidirectional. These constraints are described in section 5.1.2.

¹. Boolean equivalence and Boolean non-equivalence are in fact the familiar exclusive-nor and exclusive-or boolean operators.

6.3 Generation Steps and the Constraint Solver

The *e* generator uses a constraint solver to generate random values subject to generation constraints. This constraint solver applies an iterative process to derive the collective effect of all constraints on the generated data items. The results of the constraint solver is passed to the step that assigns random values to scalar fields. The application of the constraint solver and the Set-Scalar steps are repeated until all data items are generated. The constraint solver is applied only to items that do not have generation order dependency. So first, an ordered list of item groups is created to allow the constraint solver to be applied to items within each group.

The iterative steps performed by the generator are:

- 1: Create an ordered list of item groups, where items in a group have to be generated before items in the next group can be generated, and no generation order exists for items in the same group.
- 2: For Each Group:
- 3: Apply Constraint Solver: iterative application of:
 - Constraint Reduction
 - Constraint Evaluation
- 4: Set-Scalar: assign next item using the reduced set of constraints created by the constraint solver, back to constraint solver if more items remain in group
- 5: If more groups remain, back to step 2.

These steps are described in more detail in the following sections.

6.3.1 Item Generation Order

Unidirectional constraints imply generation ordering dependencies between items that are to be generated. Consider the following constraints:

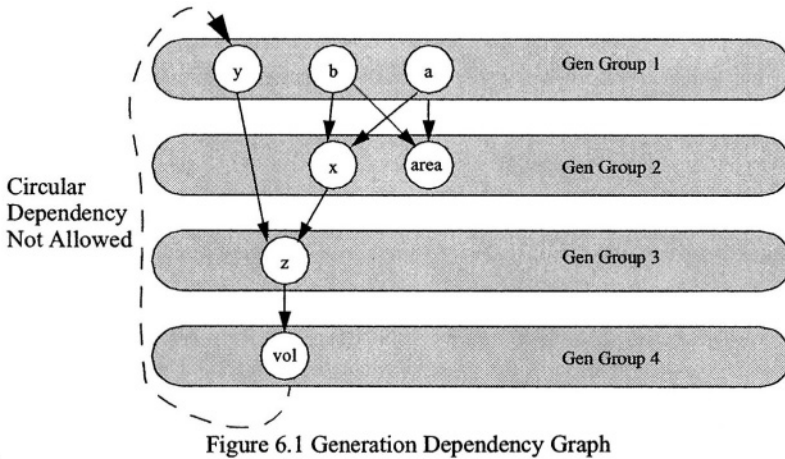
```
keep x > a * b;
keep usable_area < compute_area(length, width);
```

These constraints require that **a** and **b** be generated before **x** is generated, and that **length** and **width** are generated before **usable_area** is generated.

Unidirectional constraints can potentially lead to multiple groups of items that must be generated in a given order. Consider the following constraints:

```
keep x > a * b;
keep z > x * y;
keep volume > compute_volume(z)
keep area == a*b;
```

In this example, **a** and **b** must be generated before **x** and before **area**, **x** and **y** must be generated before **z**, and **z** must be generated before **volume**. These ordering requirements is illustrated in figure 6.1. One possible grouping of generation items to satisfy these requirements is also shown in the figure. This grouping is not unique. For example, **area** can be placed in group 2, 3 or 4, and **y** can be placed in group 1 or 2.



Beware that circular dependencies during this analysis can lead to a situation where the requirements for grouping items cannot be satisfied. Adding a constraint requiring that **vol** be generated before **y** leads to a circular dependency and causes an error condition in the generator.

Note that during generation, one item (scalar or compound object) is completely generated before generation moves to the next item. Therefore, constraints that apply to members of different structs can potentially lead to circular dependencies when same constraints placed on members of the same struct would not lead to a circular dependency. Consider the following example:

```

1 : struct xyz_s {
2 :     data1: uint;
3 :     data2: uint;
4 : };
5 : extend sys {
6 :     xyz1_i: xyz_s;
7 :     xyz2_i: xyz_s;
8 :     keep xyz1_i.data1 == compute_vol(xyz2_i.data2);
9 :     keep xyz2_i.data1 == compute_vol(xyz1_i.data2);
10 : };

```

The constraint on line 8 requires that **xyz2_i** be generated before **xyz1_i** even though the constraint is on members of these structs. The constraint on line 9 requires the reverse, therefore creating a circular dependency that will lead to an error condition during generation.

Once groups of items are created, then each group is passed to the next generation step. Within each group, items are assigned random values based on their order of appearance in the code.

6.3.2 Reduction

The reduction step combines simple constraints to produce more narrowly defined constraint ranges for items in the same generation group.

For example, given the following simple constraints:

```
keep x < 10;
keep not y < 6;
keep x > 5;
keep y < x;
keep z > y;
```

The reduction step produces the following reduced constraints:

```
keep x in [7..9];
keep y in [6..];
keep z in [7..];
```

The reduction steps performed at this stage are well defined and no alternatives exist in how these constraints may be reduced. The results of this step are deterministically defined based on the given set of constraints.

6.3.3 Constraint Evaluation

The constraint evaluation step uses the results obtained from the constraint reduction step to reduce compound constraints into simple constraints. Consider the following example:

```
keep y < 10;
keep x < y;
keep (x > 9) or (y < 5);
```

The first two constraints are simple constraints that are reduced to:

```
keep y in [0..9];
keep x in [0..8];
```

During the constraint evaluation phase, this new result is combined with the compound constraint to reduce it to the simple constraint:

```
keep y < 5
```

The original compound constraint is then marked as considered and does not come into consideration again. At this point, iteration continues since a compound constraint was reduced to a simple constraint. In the new iteration, constraint reduction is applied to the following three constraints:

```
keep y < 10;
keep x < y;
keep y < 5;
```

Leading to simplified constraints:

```
keep y in [0..4];
keep x in [0..3];
```

Since no more compound constraint are left, operation moves to the Set-Scalar step.

In general, it is possible to have simple and compound constraints that can no longer be simplified using the reduction and evaluation steps. In such cases, generation moves to the set-scalar step to assign a random value to one of the items. This value is then used in the next iteration of reduction and evaluation to further simplify the constraints during the next iteration.

6.3.4 Set-Scalar

The Set-Scalar step assigns random values subject to the simplified constraints obtained from the constraint reduction and evaluation steps. During this step:

- The next item to be assigned a value is based on the order of fields definition in the program.
- For lists, list item $n-1$ is assigned before list item n is assigned.

For example, if field y was defined first, then y is assigned a random value in the range 0 to 4, say 2. In the next iteration of the reduction and evaluation steps, the range for x is reduced to a random number between 0 and 1.

6.4 Controlling the Generation Order

It is often useful to explicitly change the default generation order. Controlling the generation order is sometimes necessary to control the distribution of the generated values, and to also prevent generation constraint contradictions.

Generation order can affect value distribution for the generated items. Consider the example shown in figure 6.2. In this example, all constraints are bidirectional and the generation order is completely based on fields definition order. If x is generated first, then it is assigned a value of either 0 or 1 with equal probability. That means probability of y being in range [0..9] is the same as probability of y being in range [10..99]. Therefore, if x is generated first, then generation of y is highly biased toward numbers in the [0..9] range. However, if y is generated first, then all values in the range [0..99] are generated with equal probability, but then generation of x is biased toward value 1. Depending on the desired value distribution, either x or y may need to be generated first.

Generation order may also lead to unexpected constraint contradictions that cannot be detected during the static analysis phase and depend on the actual random values that are generated during runtime. Consider the following *e* code fragment:

```
┌  
└  
  :  
  :  
1 : struct transaction {  
2 :     opcode1: uint;  
3 :     opcode2: uint;  
4 :     kind: [DATA, CNTL];
```

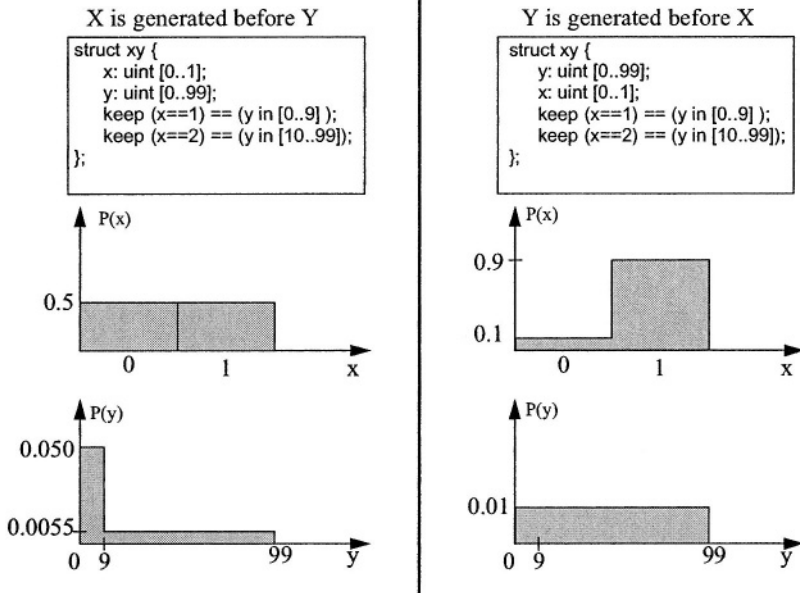


Figure 6.2 Generation Order Affects Value Distribution

```

5 :      keep kind== DATA => opcode1== 1 and opcode2 == 2;
6 :      keep kind== CNTL => opcode1== 2 and opcode2 ==3;
7 : };
      :

```

In this example, the value of **kind** is used to control the allowed values and the relationship between values for **opcode1** and **opcode2**. If **kind** is generated first, then no generation contradictions will ever occur. However, if **kind** is generated last, then generating this data object will almost always lead to a generation contradiction. The reason is that the value of **opcode1** and **opcode2** will be generated to values other than the allowed values and the allowed combination, and when the generator attempts to generate the value for **kind**, it realizes that the specified constraints can never be satisfied.

Generation order can be controlled by using the following constructs:

- Replacing **imply** constraints with **when** blocks.
- Explicitly defining the generation order using the **gen before** construct.
- Using **value()**.
- Using method calls.

6.4.1 when Blocks

Replacing **imply** statements with **when** blocks leads to a more readable program and also eliminates the occurrence of runtime generation contradictions. The previous example can be corrected as shown below:

```

      :
      :
1  : struct transaction {
2  :     opcode1: uint;
3  :     opcode2: uint;
4  :     kind: [DATA, CNTL];
5  :     when DATA transaction {
6  :         keep opcode1 == 1 and opcode2 ==2;
7  :     };
8  :     when CNTL transaction {
9  :         keep opcode1 == 2 and opcode2 ==3;
10 :     };
11 : };
      :
      :

```

Use of the **when** block eliminates any possibility of a generation contradiction by forcing the generator to assign a value to **kind** first.

6.4.2 Explicit Order Definition

Runtime generation contradictions can be prevented by adding explicit ordering instructions to the generator. This approach is shown in the following code segment.

```

      :
      :
1  : struct transaction {
2  :     opcode1: uint;
3  :     kind: [DATA, CNTL];
4  :     keep gen (kind) before (opcode1);
5  : };
      :
      :

```

6.4.3 value()

Using **value()** in a simple constraint assures that value of its parameter is generated before any other item involved in that constraint. Use of **value()** is demonstrated in the following example constraints:

keep $x < y + z$;	bi-directional constraints. Items ordering is decided by field ordering
keep $\text{value}(x) < y + z$;	item x is generated before either y or z
keep $x < \text{value}(y) + z$;	item y will be generated before either x or z

keep $x < \text{value}(y+z)$; items x is generated after generating both items y and z

However, a compound constraint, using the **value()** construct has no effect on the generation order. This is because in a compound constraint alternative clauses exist that allows the generator to bypass the ordering imposed by **value()**. Therefore, the example for **transaction** struct could not be corrected by using the **value()** construct.

6.5 Generation and Program Execution Flow

Generation is an integral part of the execution flow of an *e* program. The phases of the program execution and generation activities during each phase are shown in figure 6.3. These generation activities are:

- Static Analysis during the setup and initialization phase.
- Pre-Generation of the struct hierarchy rooted at *sys*.
- On-the-fly generation activity during simulation runtime.

Pre-generation of *sys* struct hierarchy was discussed in section 4.3. Issues related to the static analysis and on-the-fly generation are discussed in this section.

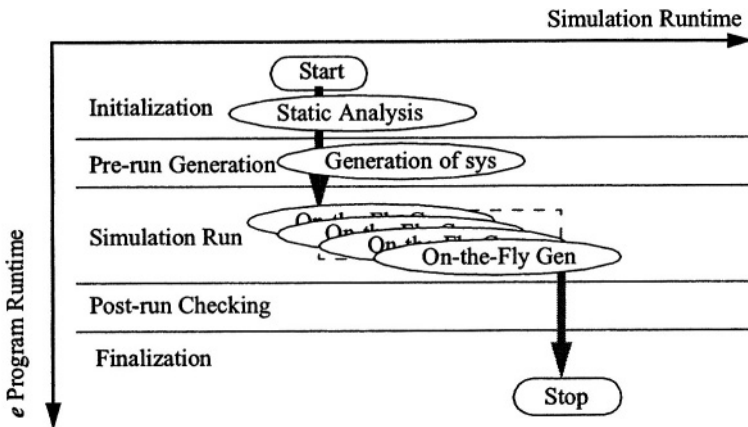


Figure 6.3 Generation and *e* Program Runtime Phases

6.5.1 Static Analysis

During the initialization phase, a static analysis phase is performed on all constraints specified in the *e* program. This static analysis consists of the initial reductions and evaluations that are done in the setup phase before simulation starts. No scalar assignment is performed during the static analysis phase.

Static analysis provides the following benefits:

- Prevents generation errors by identifying generation constraints that lead to generation contradictions.
- Performs constraint reduction and evaluation for type definition and not for each type instance. Not repeating these steps for each instance can lead to savings in program runtime.

Static analysis does not take into account the following constraint types:

- Constraints that depend on runtime values
 `keep x == 'hdl_signal';`
- Constraints with method call
 `keep x == f(y);`
- Soft constraints

6.5.2 On-the-Fly Generation

During simulation runtime, it is usually necessary to allocate new data objects and to generate their value. While the declared constraints for a data object (defined as part of struct definition) should be considered during on-the-fly generation, it should be possible to specify additional constraints while generating data values during runtime. These additional constraints are used to reflect the requirements of the specific instance of the object that is being generated.

The ability to allocate new data objects raises the issue of software memory management to ensure that all allocated memory no longer needed is freed. *Garbage Collection* is a powerful feature of the *e* language that provides for automatic management of memory that is no longer needed. The garbage collection works by keeping track of number of data references for each allocated memory chunk. Once the number of reference is reduced to zero, then that memory chunk is returned to the pool of available memory space. The power of garbage relieves the programmer from having to actively manage the memory space. However, the programmer does need to make sure that if a data object is no longer needed, then all references to that object in the program are destroyed.

e provides constructs for allocating new data objects and for on-the-fly generation of data values. It is also possible to specify additional constraints during on-the-fly generation. The usage of these constructs are described in the following sections.

6.5.2.1 Data Allocation: new

Keyword **new** is used to create and initialize a new struct. No generation is performed on the allocated data object. To populate the struct hierarchy rooted at the newly allocated data, an explicit generation step should be performed. The syntax for this action is:

```
new [struct-type [[(name)] with {action;...}]]
```

name in above syntax refers to the new name given to the allocated data object. If the struct type is not indicated, then the data object type is derived from the context of the type of variable that holds the result of this statement.

The steps performed while creating a new struct are:

- Allocate memory space for the struct type
- Invoke the **init()** method of the struct to initialize all struct members with default values
- Invoke the **run()** method of the struct (unless **new** expression is in a method that is invoked before **sys.run()**, for example in **sys.setup()**).
- Execute the action block.

The default values assigned by invoking the **init()** method are:

- Scalar fields: set to 0
- Enumerated fields: set to first enumerated literal (index 0) in its definition.
- Sized Lists: size is set to the specified size. Each list member is then set to a default value according to the rules for assigning scalar default values.
- Un-sized Lists: set to NULL.
- Structs: set to NULL.

Note that since the default value for all scalar members is NULL, then the data allocation creates a shallow struct where no struct hierarchy exists below the created data object.

Consider the following example:

```

1  :  <'
2  :  struct packet {
3  :      kind: [DATA, CONTROL, ACK];
4  :      len: int [4..10];
5  :      data: list of byte;
6  :          keep data.size() == len;
7  :      address: uint;
8  :          keep address in [0x10000..0x20000];
9  :  };
10 :
11 :  extend sys {
12 :      post_generate() is also {
13 :          var apkt: ACK packet;
14 :          var pkt: packet = new;
15 :          print pkt;
16 :          apkt = new ACK packet;
17 :          print apkt;
18 :          pkt = new packet (p) with {
19 :              p.len = 5;
20 :          };
21 :          print pkt;
22 :      };
23 :  };
24 :  >

```

In this example, variable **pkt** is of the type **packet**. The action on line 14 allocates a new data object and assigns all its fields to default values (no generation is performed). The action

on line 15 allocates a new data object of type `packet` where the `type` field is set to `ACK`. Action starting on line 16, allocates a new data object of type `packet` and as a post processing step, sets the `len` field of the allocated object to value 5. Note that in this assignment, the previous data object pointed to by variable `pkt` is lost and will be freed by the garbage collection utility.

Running this `e` program will produce the following results:

```
Specman c> test
Doing setup ...
Generating the test using seed 1...
  pkt = packet-@0: packet
----- @c
0   kind:          DATA
1   len:           0
2   data:          (empty)
3   address:       0
  apkt = ACK packet -@1: ACK packet
----- @c
0   kind:          ACK
1   len:           0
2   data:          (empty)
3   address:       0
  pkt = packet-@2: packet
----- @c
0   kind:          DATA
1   len:           5
2   data:          (empty)
3   address:       0
```

Even though the value for `len` must be between 4 and 10 as defined by the subtype declaration on line 4, the default value is still set to 0 since `len` is a scalar. Also note that `data` is set to `NULL` since it is an un-sized list. Also `kind` is set to `DATA` since `DATA` is the first enumerated literal in its definition.

6.5.2.2 Data Generation: `gen`

The `gen` action populates the struct hierarchy for a data object with constrained random values. The syntax for this action is:

```
gen gen-item [keeping {[it].constraint-bool-exp; ...}]
```

`gen-item` above refers to the data item to be generated. This data item can be any instance of a scalar or compound data type. If it is a struct, the instance for which it is allocated and all the field values inside are generated according to the constraints.

If the `gen-item` is a field struct member, then random values are assigned to the `gen-item` given the constraints applied via “keeping”. Other constraints existing for that item in the enclosing struct and its children are also considered, but constraints defined at a higher level than the enclosing struct are not considered when generating the values.

Different instances of the same struct will have different `gen` action.

```

      :
      :
1  :  struct packet {
2  :      len: int[0..10];
3  :      kind: [DATA, CONTROL, ACK];
4  :      data: list of byte;
5  :      address : uint;
6  :  };
7  :  extend sys {
8  :      pkt_x: packet;
9  :      pkt_y: packet;
10 :      run() is also {
11 :          gen pkt_x keeping {
12 :              .data.size()==0x100;
13 :          };
14 :          gen pkt_y keeping {
15 :              .data.size()== 0x10;
16 :          };
17 :      };
18 :  };
      :
      :

```

Soft constraints can be specified with a **gen** action. The soft constraints applied with the **gen** action are added to any other declarative soft constraints that apply to the generated fields. Example of soft constraint usage is shown in the following example:

```

      :
      :
1  :  struct packet {
2  :      len: int [0..10];
3  :      kind: [DATA, CONTROL, ACK];
4  :      data: list of byte;
5  :      address: uint;
6  :  };
7  :  unit lmn {
8  :      ! pkt: packet;
9  :      m_gen() is {
10 :          for i from 1 to 100 {
11 :              gen pkt keeping {
12 :                  soft it.len == select {
13 :                      10: [0..1];
14 :                      50: [3..7];
15 :                      20: [8..9];
16 :                      20: [10];
17 :                  };
18 :              };
19 :          };
20 :      };
21 :  };
      :
      :

```

In this example, 10% of the time, **len** is in the range of [0..1], 50% of the time in range [3..7], 20% of the time in range [8,9], and 20% of the time **len** will equal 10.

The `gen` action is usually used inside a TCM to create consecutive data packets that will be injected into a DUV port. In such cases, generation of the data packet has to be synchronized with the DUV operation. Synchronization actions are used to achieve this goal while the `gen` action is used to generate the packet that is to be injected. Use of **TCMs** and **gen** is shown in the following example:

```
1 : <
2 : struct packet {
3 :     len:int[0..10];
4 :     kind: [DATA, CONTROL, ACK];
5 :     data: list of byte;
6 :     address: uint;
7 : };
8 : unit pkt_driver {
9 :     event drv_clk is rise('clk') @sim;
10 :
11 :     gen_ctrl_pkt() @drv_clk is {
12 :         var pkt : packet;
13 :         for i from 0 to 10 {
14 :             gen pkt keeping {
15 :                 it.len in [0..2];
16 :                 it.kind == CONTROL;
17 :             };
18 :             drive_control_packet(pkt); -- TCM driving the packet.
19 :             wait cycle;
20 :         };
21 :     };
22 : extend sys {
23 :     drv_i: pkt_driver is instance;
24 :     keep drv_i.hdl_path() == 'top.system';
25 : };
26 : >
```

6.6 Summary

This chapter discussed issues on how the generator operation fits in program execution flow. It also presented the operation of the constraint solver and how random values are assigned to generated items. Constraint types were presented and the effect of these constraints on generation order was presented. This chapter also motivated the need to control generation order by showing that accepting the default order may lead to unacceptable random value distributions and also unexpected generation contradictions. The means to explicitly control the generation order was also presented.

The use of the generator to create clocks, resets, the verification environment, and also the verification stimulus will be described in the following chapter.

Data Modeling and Stimulus Generation

In a modern verification environment, data is generated and stored using abstract representations. This abstract representation leads to a more productive and modular programming style. Productivity is gained by eliminating the need to deal with the details of operations for physical data manipulation. Modular representation of data items allows operations on all instances of a data type to be encapsulated along with its abstract representation. This modularity in turn leads to further increase in productivity by improving code readability and ease of maintenance.

An abstract data item includes fields that are not necessarily present in the item's physical manifestation. Such fields include flags, status, and subtype determinants of a data type which are used to guide constrained random generation for that data type, and to store information about that abstract data item collected from physical data.

Data items existing in a verification environment are either generated in the environment or derived from physical signals (i.e DUV ports or internal signals). As such, it is necessary to provide mechanisms to translate between the physical view of a data item and its logical abstraction. Encapsulating such operations as part of data type, allows the verification environment and verification suite developers to focus on verification details rather than the tedium of performing data translations.

This chapter describes data modeling in *e* and discusses data type structure, organization, and operations required for creating abstract data types that can lead to easier and more productive use of the verification environment.

7.1 Data Model Fields

Abstract data objects are not necessarily associated with a specific location in the design or verification hierarchy. These objects may flow through the verification environment and can be allocated, generated, and/or freed during the program and simulation runtime. As a result, data objects are modeled using the **struct** construct. Observe the following considerations when developing a data model:

- A data model may contain information that does not exist in the physical equivalent of that data model. As such, special considerations must be made to differentiate between these two types of information. This differentiation is required so that operations on this data type can be done accordingly.
- An abstract data model may have different structures, or specific field values, depending on its subtype specification. Data modeling should therefore allow for subtype specification and customization of such subtypes.
- A data model may require special fields to guide generation and to store information about its status if extracted from physical data. A data model should therefore provide such fields, as required by the data being modeled, and both consider and update these values while performing operations on this data type.

The *e* language provides mechanisms and techniques for supporting the considerations above. These issues are discussed in the following sections.

7.1.1 Physical Fields

In the *e* language, concepts of *physical fields* and *virtual fields* are used to differentiate between fields that only exist in the physical equivalent of a data type. Struct member definitions corresponding to physical fields are preceded by a *%*. Consider the following example showing a partial description for a UART frame:

```

1  :  <
2  :  struct uart_frame {
3  :      frame_size: uint;
4  :          keep frame size in [5..8];
5  :      %bits[frame_size]: list of bit;
6  :  };
7  :  >
```

Depending on the configuration, a UART frame may contain anywhere from 5 to 8 bits. In above example, **frame_size** is a virtual field that corresponds to the number of bits in the frame. The size of this frame is hard constrained to keep it in the legal range. **bits** is a physical fields whose size is derived from **frame_size**.

Note that physical and virtual information are differentiated to guide the packing and unpacking steps. Also, this makes it easier to identify physical fields while reading an *e* program.

7.1.2 Determinant Fields

Determinant Fields are virtual fields of a data type that identify its subtypes. Determinant fields are used in defining **when** subtypes of a struct, or in extending that subtype in the same or a separate file. Determinant fields can be used to enhance the definition of the UART frame to indicate whether this packet contains a parity bit, and how the parity bit is calculated:

```

1  | <
2  | struct uart_frame {
3  |     parity_type: [NONE, EVEN_PARITY, ODD_PARITY];
4  |     frame_size : uint;
5  |     keep frame_size in [5..8];
6  |     %bits[frame_size]: list of bit;
7  |     when EVEN_PARITY uart_frame {
8  |         %even_parity: bit;
9  |         keep even_parity == compute_even_parity(bits);
10 |         compute_even_parity(lob: list of bit): bit is {
11 |             return lob.sum(it)[0:0];
12 |         };
13 |     };
14 |     when ODD_PARITY uart_frame {
15 |         %odd_parity: bit;
16 |         keep odd_parity == compute_odd_parity(bits);
17 |         compute_odd_parity(lob: list of bit): bit is {
18 |             return ~lob.sum(it)[0:0];
19 |         };
20 |     };
21 | };
22 | >

```

In this example, field **parity_type** is used to indicate the type of parity that this frame contains. The **when** construct is used to extend the base definition of this struct to add physical fields corresponding to even and odd parities depending on **parity_type**. Though the same fields name could be selected for both parity types, different names are used to further differentiate between the two types. By using **parity_type** in a **when** construct, that field is generated before any field defined within any subtype definition. **bits** is also passed to these methods to force the generator to generate the contents of **bits** before attempting to compute the parity.

Additional constraints are specified to set parity to its correct value. These constraints include the user defined methods **compute_odd_parity()** and **compute_even_parity()**. Note that these methods are very simple and the operation performed in the body of each method could replace the method call in its corresponding constraint. However a method call is used to demonstrate use of user defined methods in constraint definitions.

7.1.3 Utility Fields

As previously mentioned, an abstract data model may include information that is not present in its physical form. For example, in the above example, **frame_size** is a utility field as it is not present as a UART frame travels on a link, but in this case, it was used to model the frame.

Two common uses of a utility field are index fields indicating the position of a data item in a list of items, and flags for guiding the generation flow.

The following *e* program shows a simple example of using an index field to keep track of the position of data item in a generated list.

```

1 : <
2 : struct data_frame {
3 :     indx: uint;
4 :     %data: byte;
5 : };
6 : extend sys {
7 :     data_frame_list: list of data_frame;
8 :     keep data_frame_list.size() == 10;
9 :     for each in data_frame_list {
10 :         keep.indx == index;
11 :     };
12 : };
13 : >

```

In this example, the **for each** construct is used to constrain the **indx** field of each list member to the index of that member within the list. In this case, the index value is used to report information for a data item. Note that the same technique can be used to assign ordered values to physical data fields which can be useful for checking the transmission of consecutive data items in the verification environment.

A common use of a utility field is to guide the generator into creating specific variations of a data object. For example, a UART frame may include a field indicating if correct parity should be generated for that frame. The previous example is enhanced below to include a flag indicating whether the parity error should be generated to its correct value.

```

1 : <
2 : struct uart_frame {
3 :     has_parity_error: bool;
4 :     keep soft has_parity_error == FALSE;
5 :     parity_type: [NONE, EVEN_PARITY, ODD_PARITY];
6 :     frame_size: uint;
7 :     keep frame_size in [5..8];
8 :     %bits[frame_size]: list of bit;
9 :     when EVEN_PARITY uart_frame {
10 :         %even_parity: bit;
11 :         keep even_parity == compute_even_parity(bits, has_parity_error);
12 :         compute_even_parity(lob: list of bit, has_err: bool): bit is {
13 :             return (has_err ? (lob.sum(it)[0:0]) : (~lob.sum(it)[0:0]) );
14 :         };
15 :     };
16 :     when ODD_PARITY uart_frame {
17 :         %odd_parity: bit;
18 :         keep odd_parity == compute_odd_parity(bits, has_parity_error);
19 :         compute_odd_parity(lob: list of bit, has_err: bool): bit is {
20 :             return (has_err ? (~lob.sum(it)[0:0]) : (lob.sum(it)[0:0]) );
21 :         };
22 :     };

```

```

23 : };
24 : >

```

Passing `has_parity_error` to the methods computing either the odd or even parity modified the previous example where `has_parity_error` is used to indicate if parity should be generated correctly. The advantage of this approach is that by passing `has_parity_error` to a method call in a constraint, `has_parity_error` is generated before the parity field is generated. In this case, the generation order is controlled so that correct generation order is maintained.

It is possible to implement the above enhancement by only using generation constraints, without modifying the methods that compute the parity value. However, if not used carefully, this approach may lead to potential pitfalls during the generation phase.

7.1.3.1 Avoiding Data Generation Inconsistencies

Consider a data item whose definition is shown in this example:

```

1 : <
2 : struct data_frame {
3 :     %data: byte;
4 :     keep data == compute data() ;
5 : };
6 : >

```

Now consider adding a generation flag and the necessary constraints to generate erroneous data values. The goal in this new implementation is to always generate consistent values where the value for `gen_err` indicates if `data` has a valid value.

```

1 : <
2 : struct data_frame {
3 :     gen_err: bool;
4 :     keep soft gen_err == FALSE;
5 :     %data: byte;
6 :     keep gen_err => data != compute data() ;
7 : };
8 : >

```

Although this implementation may seem correct at first, it can lead to inconsistent generation results! Since the **imply** constraint in this example does not impose an order and the method call in this constraint does not have a parameter, the only generation order is dictated by the order of appearance of fields `gen_err` and `data`. In the above implementation, where `gen_err` is generated before `data`, the generated data item content is always consistent. However, if `data` is generated first (i.e. because of other generation considerations or if `gen_err` is defined after `data`), inconsistencies may result. Assume `data` is generated to a value other than the value returned by the `compute_data()` method, then the constraint on line 6 does not impose any restrictions on the value of `gen_err`. In the absence of any other constraint on `gen_err`, the soft constraint on line 4 forces the generator to assign its value to false. Thus, leading to an inconsis-

tent state for this data item where **gen_err** is false but **data** is not equal to the correctly calculated value returned by **compute_data()**.

Such an issue may be resolved using a number of different approaches. One is to explicitly force the generator to generate **gen_err** before **data** using a **gen before** constraint. Another approach would be to pass the value of **gen_err** to the method that computes a value for **data**. The UART frame example followed this approach. Again, the advantage of this approach is that the generation order is explicitly defined.'

Another interesting approach to solving this problem is to replace the **imply** constraint on line 6 with a Boolean equivalence constraint. With this new change, if the value of **data** is assigned to a value other than the one returned by **compute_data()**, then the constraint on line 6 still imposes a hard constraint on **gen_err** (in this case forces it to be set to true). An additional benefit of this approach is that this generation constraint can be used to compute the value of **gen_err** flag when assigning data into this data object.

```

1  : <
2  : struct data_frame {
3  :     gen_err: bool;
4  :     keep soft gen_err == FALSE;
5  :     %data: byte;
6  :     keep gen_err == ( data != compute_data() );
7  : };
8  : extend sys {
9  :     dframe: data_frame;
10 :     run() is also {
11 :         dframe.data = 10;
12 :         gen dframe.gen_err;
13 :     };
14 : >

```

The value of **data** for the instance **dframe** of data type **data_frame** is set to 10. An explicit generation action on **gen_err** of **dframe** (line 12), then uses the constraint on line 6 to generate the correct value for **gen_err** field of **dframe**. This approach can be used when unpacking data into a data object to compute virtual fields of that data object.

7.2 Data Model Subtypes

The specification of any non-trivial data model includes field value customization and conditional fields. Data field customization refers to attaching special meaning to specific values of data fields (i.e. tags, tokens, etc.). Conditional field refers to a data field that is present only under specific conditions. The ethernet packet shown in figure 7.1 shows examples of both these concepts. **SIZED** and **QTAGGED** are two commonly used ethernet packet formats. The **TAG** field in an ethernet packet is used to indicate the type of ethernet packet. Additionally, the

data payload portion of the ethernet packet for QTAGGED packets, contains a substructure that consists of its own flags, tags, and data payload.

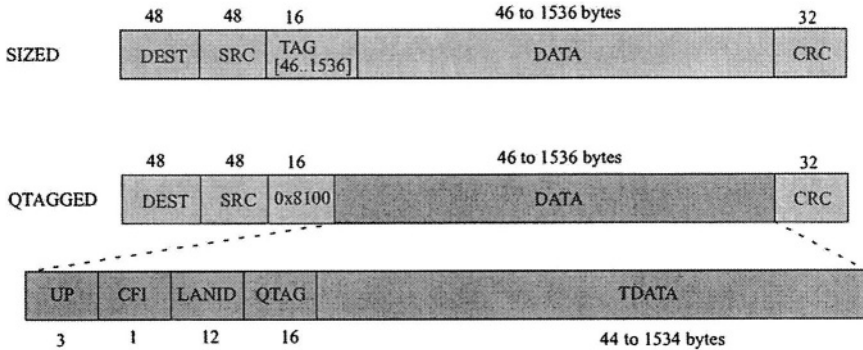


Figure 7.1 ethernet Packet Formats

Using data subtypes is a common approach used to implement both these requirements. These topics are discussed in the following sections.

7.2.1 Field Value Customization

As shown for an ethernet packet, data definitions may include fields that are used to control the valid range or the exact value of other data fields. In an ethernet packet, the tag field indicates the packet type.

```

1  | <
2  | struct ethernet_packet {
3  |     type: [SIZED, QTAGGED];
4  |     %dest: uint(bits:48);
5  |     %src: uint(bits:48);
6  |     %tag: uint(bits:16);
7  |     when SIZED ethernet_packet {
8  |         keep tag in [46.. 1536];
9  |     };
10 |     when QTAGGED ethernet_packet {
11 |         keep tag ==0x8100;
12 |     };
13 | };
14 | >

```

The above implementation assigns the value of **tag** to the correct range depending on the packet type. Modeling of the data payload field containing conditional fields is described in the next section.

7.2.2 Conditional Fields

As shown in figure 7.1, the structure of data payload in an ethernet packet depends on its type. In a SIZED ethernet packet, the tag field contains the size of the data payload in bytes. In a QTAGGED ethernet packet, the tag field is set to a constant value and the **qtag** field contains the size of the data payload. The implementation of the ethernet packet to include conditional fields of QTAGGED type is shown below.

```

1  :  <
2  :  extend SIZED ethernet_packet {
3  :      %data[tag]: list of byte;
4  :  };
5  :  extend QTAGGED ethernet_packet {
6  :      %up: uint(bits:3);
7  :      %cfl: bit;
8  :      %lanid: uint(bits:12);
9  :      %qtag: uint(bits:16);
10 :      keep qtag in [44..1534];
11 :      %data[qtag]: list of byte;
12 :  };
13 :  >

```

In this implementation, the SIZED subtype is extended to include a data payload with its length specified by the **tag** field. The QTAGGED subtype is extended to include additional fields and a data payload whose size is defined by field **qtag**.

7.2.3 Conditional Fields and Generation Constraints

It is often necessary to generate data types where the fields that need to be constrained only exist in a subtype of that data type. Multiple approaches can be used to specify generation constraints for conditional fields of a data type as part of the generation step. It is also possible to enhance the data model to make constraint specification(s) for conditional fields easier.

The easiest approach for specifying constraints for a subtype conditional field of a data object is to specify the data subtype in the declaration of that object. In this example, **epkt** is defined as a QTAGGED packet and therefore **qtag** and **lanid** conditional fields can be used directly during constraint specification.

```

      :
      :
1  :  extend sys {
2  :      run() is also {
3  :          var epkt: QTAGGED ethernet_packet;
4  :          gen epkt keeping {
5  :              .qtag == 1300;
6  :              .lanid == 0x123;
7  :          };

```

```

8 :      };
9 :      };
      :
      :
      :

```

Sometimes a packet may need to be generated either as **QTAGGED** or **SIZED**. In such cases, the packet subtype cannot be specified as part of its declaration. Instead, *Tick Notation* must be used to specify the necessary constraints:

```

      :
      :
1 :  extend sys {
2 :      run() is also {
3 :          var epkt: ethernet_packet; -- packet subtype not defined in declaration
4 :          gen epkt keeping {
5 :              .type == QTAGGED;
6 :              .QTAGGED'qtag == 1300;
7 :              .QTAGGED'lanid == 0x123;
8 :          };
9 :      };
10 : };
      :
      :

```

In the above example, the conditional fields **qtag** and **lanid** are specified using the tick notation as part of their subtype. Tick notation constraints only take effect when the generated data has the subtype indicated in that notation. For instance, if the constraints defined on line 5 in the above example are removed, then the generated **epkt** may have subtype of either **QTAGGED** or **SIZED**. Constraints on lines 6 and 7 are only taken into consideration if the generated type is **QTAGGED** and are ignored if type is generated as **SIZED**.

It is often convenient to add additional utility fields to a data model to make constraints simpler to specify. In the case of an ethernet packet, the goal is often to generate a packet with special data size constraints. The implementation of a data packet is extended below to make size constraint specification more straight forward.

```

1 :  <
2 :  extend ethernet_packet {
3 :      psize: uint;
4 :      when SIZED ethernet_packet {
5 :          keep tag == psize;
6 :      };
7 :      when QTAGGED ethernet_packet {
8 :          keep qtag == psize;
9 :      };
10 : };
11 : extend sys {
12 :     run() is also {
13 :         var epkt: ethernet_packet;
14 :         gen epkt keeping {
15 :             .psize == 200;
16 :         };
17 :     };

```



```
18 : };  
19 : >
```



Given the above enhancement, the packet size can be constrained as shown by directly constraining **psize** (a field in the base definition and not any of the subtypes). Of course care should be taken in constraining size, since data payload size limit is different for each subtype and specifying the wrong size for a subtype will lead to generation contradictions.

7.3 Data Abstraction Translation

Translations between an abstract data type (i.e a logical view) and its physical view are a commonly used operation when interacting with a DUV. Translating from a logical view to a physical view consist of creating a bit stream that represents the contents of a data object. Translating from a physical view to a logical view consists of translating a stream of bits into an abstract view. The *e* language provides packing and unpacking utilities to support both these activities. The following section describes the details of these operations.

7.3.1 Packing: Logical View to Physical View

The *e* language provides the **pack()** operation to group the physical fields of a data object into a bit stream. As described in section 5.6, the packing operation creates a list of bits according to the pack options specified when calling the method.

As mentioned in section 5.6, the default behavior of pack option can be modified by extending the definition for the **do_pack()** method. Since the packing operation can be controlled through the **pack_options** structure, there are few cases where the default packing behavior may need to be modified. Possible scenarios where the this modification may be required include:

- Enhancing the packing operation to update program fields storing packing statistics and adding debugging information as packing proceeds.
- Since pack options passed to the **pack()** method apply to all items being packed, then changing packing options for one specific struct in the struct hierarchy or one specific item requires modification to the **pack()** method.
- Even though unlikely, a struct member may need to be included in the packing but not in the unpacking (or vice versa). In such a case, this field is defined as a virtual field and explicitly included in the packing operation by overriding the default packing operation.

The following code fragment shows an example of the packing order of **dataframe** set to **packing.low** regardless of the packing option specified for the **pack()** method that packs objects of type **dataframe**. The struct member **data1** is also included in the packing result for **dataframe** objects, even though it is not a physical field.

```

      :
      :
1   : struct dataframe {
2   :     data1 :uint(bits:10);
3   :     %data2: uint(bits:12);
4   :     do_pack(options:pack_options, l: *list of bit) is only {
5   :       var lob : list of bit =
6   :         lob = pack(packing.low, data1, data2);
7   :         l.add(lob);
8   :     };
9   : };
      :
      :

```

7.3.2 Unpacking: Physical View to Logical View

Unpacking is the reverse operation of packing. Unpacking is a more difficult operation to program, however, because of the reasons described in section 5.6. Because of these considerations, special attention must be paid to unpacking into lists and unpacking into structs whose structure depends on the data that is being unpacked. These issues are discussed.

7.3.2.1 Lists

Unpacking into data models that contain variable size lists requires special considerations to achieve proper unpacking behavior. The **unpack()** method applies the following rules when unpacking into lists:

- Sized Lists: Unpack and create new list items until size is reached.
- Un-sized Lists: Unpack until no more data is available.
- Lists with existing Size: Unpack until all existing list items are replaced with new unpacking data.

Based on these unpacking rules, it is suggested to only use sized lists in data models because if a struct member is un-sized and has a size of zero, then any unpacking into the struct will not progress beyond the un-sized list, and all remaining data is unpacked into that un-sized list.

A good approach for testing the unpacking of data into a data object is to first pack that data then unpack the pack results. The unpack result should be the same as the original data content. In the following discussion this approach is considered in analyzing the behavior of the unpacking operation.

Consider **dataframe** shown in the following example.

```

1   : <
2   : struct dataframe {
3   :   %size: byte;
4   :   %data[size]: list of bit;

```

```

5 : };
6 : >

```

The following rules are useful in analyzing packing and unpacking behavior:

- During packing, the packing order may be **data** and then **size**, or **size** followed by **data** depending on packing options (i.e. **packing.high**, **packing.low**).
- During unpacking, **size** is always unpacked first, followed by **data**. Packing options only affect how data is extracted from the bit stream (i.e. for **packing.low**, size is extracted from lowest bits of the bit stream, for **packing.high** size is extracted from higher significant bits of the bit stream).
- While unpacking data into a sized list, the value of size should already be set to its correct value.

This analysis indicates that as long as the size field for a sized list is a physical field and is defined before that list, the default unpacking method does not need to be modified.

However, consider a case where list size is not a physical field. Two possibilities exist:

- List size is derived from the environment.
- List size is derived from the data that is being unpacked.

If the list size is derived from the verification environment and its configuration, then this field should be set before calling **unpack()** for this data object.

The more interesting scenario is where the list size must be derived from the data that is being unpacked into a data object. One solution is to process the bit stream before calling **unpack()** to set the size field. As mentioned, the goal of data model development is to encapsulate as much of the data manipulation operations in the data model itself. Therefore it is best that extraction of size be performed implicitly as part of calling the **unpack()** method. This is accomplished by extending the predefined **do_unpack()** method.

```

1 : <
2 : struct dataframe {
3 :     size1: byte;
4 :     %data1[size1]:list of bit;
5 :     size2: byte;
6 :     %data2[size2]: list of bit;
7 :     do_unpack(options:pack_options, l: list of bit, from: int):int is first {
8 :         size1 = extract_size1(l, from);
9 :         size2 = extract_size2(l, from);
10 :     };
11 : };
12 : extend sys {
13 :     dframe_list: list of dataframe;
14 :     dframe: dataframe;
15 :     unpack_dframes(lob: list of bit) is {
16 :         unpack(packing.low, lob, dframe);
17 :         dframe_list.clear(); -- make list of size zero.
18 :         unpack(packing.low, lob, dframe_list);
19 :     };

```

```

20 : };
21 : >

```

In the above example, the `do_unpack()` method of `dataframe` is extended to compute the values for `size1` and `size2` fields when `do_unpack()` is first called. The default implementation of `do_unpack()` then continues to unpack the correct amount of data into `data1` and `data2` fields.

Encapsulating field extraction as part of the data model definition (i.e. extending `do_unpack()`) not only hides extraction complexity from the verification environment user, but also allows a single bit stream to be unpacked into multiple data objects or even a list of data objects. In the above example, two unpack operations are shown. The call to `unpack()` on line 16 unpacks enough bits from `lob` to populate fields of `dframe`. Assuming `dframe_list` size is zero, the call to `unpack()` on line 17 creates as many items in `dframe_list` as allowed by the available data in `lob`.

7.3.2.2 Subtypes

A data model structure may look completely different depending on its subtype. This means that subtype determinant fields affecting each field must be known as the unpack operation moves from one field to next. Since subtype determinant fields are usually enumerated types, it is likely that they are not set to correct values as part of the unpacking process. Consequently, additional code must be developed to extract the necessary subtype determinant fields before or during the unpacking process.

This problem solving approach is essentially the same as for managing list sizes described in the previous section. The ethernet packet example is enhanced to support unpacking in the following code fragment.

```

1  : <
2  :   extend ethernet_packet {
3  :       do_unpack(options:pack_options, l: list of bit, from: int):int is first {
4  :           var tagvalue : uint(bits:16) = extract_tag_from_list(l, from);
5  :           if(tagvalue<1537){
6  :               type = SIZED;
7  :           } else if (tagvalue == 0x8100) {
8  :               type = QTAGGED;
9  :           } else {
10 :               error("Invalid tag field while unpacking in ethernet_packet");
11 :           };
12 :       };
13 :   };
14 : >

```

7.4 Data Generation Constraints

A data model includes hard constraints as part of defining field valid ranges or field relationships. It is useful to include additional constraints to improve a data model's ease of use. Approaches to achieving this goal are discussed in the following sections. The following implementation of a rectangle is used to motivate and illustrate these concepts.

```

1  :  <'
2  :  struct rectangle {
3  :      width: uint;
4  :          keep width in [1.. 1000];
5  :      height: uint;
6  :          keep height in [1.. 1000];
7  :  };
8  :  >'

```

7.4.1 Abstract Ranges

Special ranges of data values usually correspond to abstract properties of that data object. For example, a rectangle may be defined as **WIDE** or **NARROW**, and **TALL** or **SHORT** depending on the settings of its width and height parameters. The definition for such abstract properties is usually a part of the context that the data object.

Using abstract ranges is useful for simplifying constraint specification when working with a data model. Defining such abstract properties as enumerated types and then using them as subtype determinants is a robust way to allow constraint definition.

Using abstract ranges also allows for easier specification of non-continuous ranges whenever these ranges are constrained in the subtype that is defined by that abstract property. The rectangle data model is extended in the following code fragment to define abstract properties and allow for using these abstract properties when using this data type.

```

1  :  <'
2  :  extend rectangle {
3  :      height type: [SHORT, AVERAGE, TALL, SHORT_OR_TALL];
4  :      width type: [NARROW, MEDIUM, WIDE, WIDE_OR_NARROW];
5  :      when SHORT rectangle { keep height < 100 ; };
6  :      when AVERAGE rectangle { keep height in [100..900] ; };
7  :      when TALL rectangle { keep height > 900 ; };
8  :      when SHORT_OR_TALL rectangle {
9  :          keep height < 100 or height > 900;
10 :      };
11 :      when NARROW rectangle { keep width < 100 ; };
12 :      when MEDIUM rectangle { keep width in [100..900] ; };
13 :      when WIDE rectangle < keep width > 900 ; };
14 :      when NARROW_OR_WIDE rectangle {
15 :          keep width < 100 or width > 900;
16 :      };
17 :  };

```

```

18 : extend sys {
19 :     d1: SHORT rectangle;
20 :     d2: NARROW rectangle;
21 :     d3: NARROW_OR_WIDE rectangle;
22 : };
23 : >

```

In this example, **NARROW_OR_WIDE**, and **TALL_OR_SHORT** are abstract non-continuous ranges. By defining these abstract properties, it is no longer necessary to remember the exact definition of these abstract properties during verification code development

7.4.2 Coordinated Ranges

Coordinated ranges refer to interesting constraints that define specific combinations of different data fields. The following example shows use of coordinated ranges to define **ANY**, **SMALL**, **MEDIUM**, and **LARGE** rectangle types, where no coordinated constraints are applied for **ANY**.

```

1 : <'
2 : extend rectangle {
3 :     rectangle_type: [ANY, SMALL, MEDIUM, LARGE];
4 :     when SMALL rectangle {
5 :         keep width_type== NARROW and height_type== SHORT;
6 :     };
7 :     when LARGE rectangle {
8 :         keep width_type==WIDE and height_type==TALL;
9 :     };
10 :    when MEDIUM rectangle {
11 :        keep not (width_type==NARROW and height_type!=SHORT) and
12 :        not (width_type==WIDE and height_type==TALL);
13 :    };
14 : };
15 : extend sys {
16 :     d1: SMALL rectangle;
17 :     d2: rectangle;
18 :     keep soft d2.rectangle_type == {
19 :         10: SMALL;
20 :         20: MEDIUM;
21 :         30: LARGE;
22 :     };
23 : };
24 : >

```

7.4.3 Default Ranges

A data model should include default constraints that provide typical behavior for that data type. Soft constraints are used to specify default constraints. However, though using soft constraints to provide default behavior is a powerful technique for defining typical behavior, use of soft constraints may lead to problems when the default constraints must be overridden. The *e* lan-

guage provides a mechanism to remove sort constraints if it is necessary to do so. The following extension of rectangle implementation illustrates this issue in more detail.

```
1 : <
2 : extend rectangle {
3 :     keep soft rectangle_type == SMALL;
4 : };
5 : extend sys {
6 :     d1: rectangle;
7 :         keep d1.rectangle_type.reset_soft();
8 :         keep d1.rectangle_type in [SMALL, MEDIUM];
9 : };
10 : >
```

In the above example, the definition of `rectangle` is extended to include a soft constraint to set the default rectangle type to **SMALL**. The hard constraint on line 8 indicates that while generating the value for `d1`, rectangle type should be generated to either **SMALL** or **MEDIUM** with equal likelihood. However without the `reset_soft()` constraint on line 7, rectangle type will always be generated as **SMALL** since the soft constraint on line 3 does not contradict the hard constraint on line 8. By adding the `reset_soft()` constraint on line 7, all soft constraints applied to `rectangle_type` (including the one on line 3) are ignored

7.5 Summary

Data is either generated in the verification environment or collected from the physical data extracted from the DUV. This chapter described the structure of a data model and how physical, subtype determinant, and utility fields are used to model and manage a data object. Packing was described in this chapter as a means of translating a data object into a bit stream so that it could drive physical DUV ports. Unpacking was described as the means to populate data structure fields from serial data collected from the DUV. Special techniques for enhancing the definition of a data model were introduced in order to simplify constraint definition when a data model is being used in a verification environment.

Chapter 8 discusses how sequences of data items are generated to form special verification scenarios. Building of a verification environment is discussed in chapter 11.

Verification scenarios are created through a series of interactions between the verification environment and the DUV. Considering that in a modern verification environment interactions with the DUV take place through a BFM, a verification scenario is more accurately defined as a series of interactions with a BFM. Interactions with a BFM take place through its user interface defined as a set of transactions, packets, or instructions. Thus, a verification scenario is implemented as a sequence of items where each item represents an atomic step in interacting with a BFM.

As discussed in chapter 2, such sequences should be randomly generated in order to gain the full benefits of constrained random verification methodology. A good random sequence generation utility should have the ability to:

- Parameterize and set generation constraints
- Synchronize between multiple sequences at different DUV ports
- Define hierarchical sequences (sequence of sequences)
- Define reactive sequences (sequences responding to DUV output)
- Define layered sequences (sequences driving sequences)

The *e* language provides full support for constrained random sequence generation and the necessary utilities to support the desired sequencing features. This chapter will first present an overview of sequence implementation and then discusses the details of using sequences to create verification scenarios for the verification requirements listed above.

8.1 Verification Scenarios as Sequences

A verification scenario is described using the following three components:

- Verification Item
- Verification Sequence
- Verification Item Driver

A *Verification Item* describes an atomic verification activity. An atomic verification activity is carried out as a whole. Theoretically, a verification item may describe an operation as complex as the full configuration cycle of a DUV, or as simple as injecting a single packet into a DUV port. But since complex atomic verification activities are often implemented as a sequence of simpler operations, in most cases a verification item will describe simple operations such as read/write, a configuration step, or a data packet injection. A verification item might include randomly generated parameters. For instance, the payload size for a verification item corresponding to a valid ethernet packet can be generated randomly. A sequence then, is the random generation of its verification items, and a *Generated Instance of an Item* refers to such a generation step.

A *Flat Verification Sequence* contains only verification items, and describes the number of and properties of its verification items. These properties include:

- Required item orderings
- Data Dependencies between verification items
- Synchronization mechanism with the simulator
- How to drive each verification item into the verification environment

The number of items in a verification scenario may be generated randomly if it is relevant to the goal of that verification sequence.

As an example, a verification scenario for an ethernet port may indicate that between 5 to 10 valid ethernet packets with data size between 50 and 200 bytes must be injected into a DUV port. In this case, the verification item is defined as a valid ethernet packet of between 50 to 200 bytes of data. The verification sequence indicates that it contains 5 to 10 ethernet packets, and the number of items is also randomly generated.

A more complex verification scenario might involve writing to a memory location and then reading that same memory location to check that the memory write/read operation works as expected. In this case, the verification item is either a memory write or a memory read operation. The address and data values may both be generated randomly but the address for both read and write operations must be the same. The verification sequence indicates that write is followed by the read operation and also indicates that the address for both operations must be the same.

A *Hierarchical Verification Sequence* contains verification subsequences as well as verification items. A verification sequence might consist of 5 memory write operations followed by a write/read sequence. In this instance, the flat verification sequence above is a subsequence of

this new verification sequence. A *Virtual Verification Sequence* refers to a hierarchical verification sequence that contains only other sequences and no verification items.

A *Verification Item Driver* is needed to apply the generated item to the verification environment. For an ethernet packet, the item driver is the step that passes a data packet to the ethernet BFM. For a more complex item (i.e. performing a multi-cycle configuration step) the item driver may be implemented as a method that implements that required task. A verification item driver operates in two modes:

- Push Mode
- Pull Mode

In the push mode, the verification item is driven into the verification environment once it is generated. In this case, the synchronization mechanism generating the sequence determines when an item is driven into the environment. In the pull mode, the verification environment asks the sequence to provide it with the next item. In the latter case, synchronization between driving sequence items into the environment and the simulation is controlled by the agent pulling the next verification item.

Verification sequences are further divided into homogeneous and heterogeneous sequences. All verification items in a *Homogeneous Verification Sequence* have the same item kind. Verification items of the same kind are items that can be injected into the verification environment using the same item driver (and hence the same BFM). A *Heterogeneous Verification Sequence* contains verification items that have different kinds. A hierarchical sequence whose items all have the same type is still considered homogeneous regardless of the kind of items contained in its subsequences. For example, verification items in a homogeneous verification sequence may correspond to ethernet packets that have different payload size ranges and are injected into the same ethernet port. Verification items in a heterogeneous verification sequence may correspond to ethernet packets that are sent to two different ethernet ports (requiring two different BFMs). A flat heterogeneous sequence can be modeled using a hierarchical sequence consisting only of homogeneous sequences. The predefined sequence generation utilities in ϵ do not support flat heterogeneous sequences, so all heterogeneous sequences must be modeled as hierarchical sequences containing only heterogeneous sequences.

The order that a sequence is created is not affected by how a sequence item is driven into the environment. Therefore, for the purposes of this discussion, sequence generation and the steps for driving sequence items are discussed separately.

Figure 8.1 shows the graphical view of a sequence definition and the steps for generating an instance of the sequence. A sequence definition includes:

- Action Block
- Member Verification Items and/or Verification Subsequences

The structure of a sequence generator is defined by its member verification items and verification subsequences. However, these members are only the templates that specify the types of items and subsequences that can be generated. The action block is used to generate item and subsequence instances and to specify the order and dependencies between items and sub-

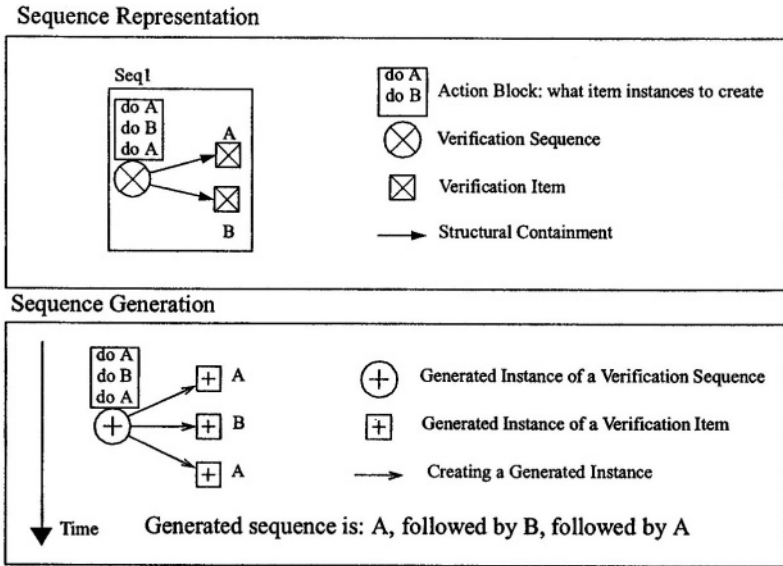


Figure 8.1 Graphical View of Sequence Representation and Generation

quences generated. As constrained random generation is a required feature of sequence generation, generation constraints can be specified in the action block for verification items and subsequences that are being generated. In figure 8.1, the generation of an item or subsequence is signified by the **do** action. In this figure, the action body indicates that item **A** must be generated first, followed by item **B**, and then again item **A**.

Figure 8.2 shows an example of generating a hierarchical sequence using the view described in figure 8.1. In this example, **Seq3** is a virtual sequence generated recursively according to the action blocks specified for each of its subsequences. The sequence generated is shown on the vertical time axis in this figure. No assumption is made about how each item is driven into the verification environment or how it synchronizes with the simulator because these decisions are separate from how a sequence is actually generated and will be discussed later in the chapter.

8.2 Sequence Generation Architecture

The *e* language contains built-in facilities for generating and driving sequences into the verification environment. The architectural view of sequence generation and sequence driving constructs of the *e* language is shown in Figure 8.3. This view consists of 4 main components¹:

- Sequence Item: Specifies the items that form the sequence (i.e. ethernet packet).

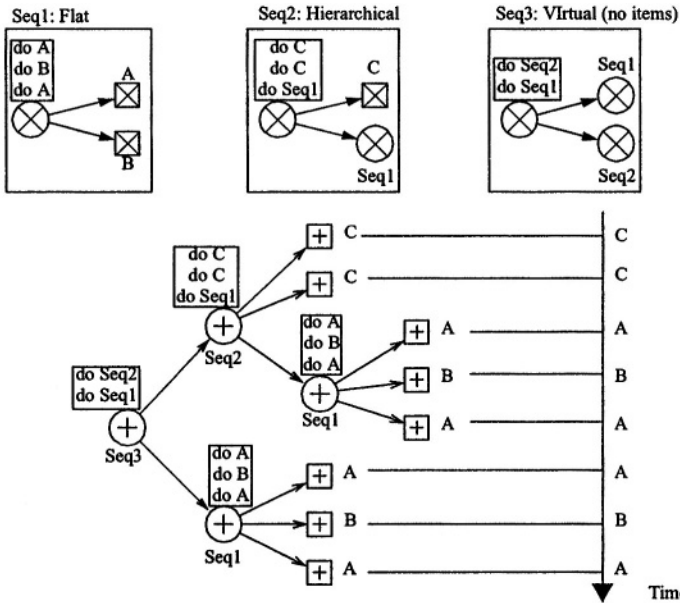


Figure 8.2 Hierarchical and Virtual Sequences

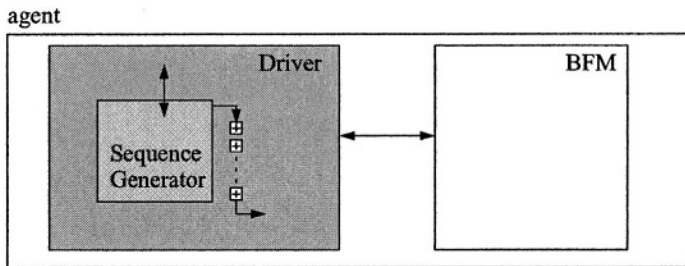


Figure 8.3 Sequence Generator Architecture

- Sequence Generator: Generates a sequence of items according to the specifications in its action block.
- Sequence Driver: Synchronizes with the sequence generator to receive the generated items.
- BFM: The generated item is injected into the environment by the BFM.

¹ A heterogeneous sequence requires access to multiple BFMs, one for each item kind generated by that sequence. Therefore this architectural view may in fact contain multiple drivers and multiple BFMs. These topics are discussed in detail in section 8.5.

Some properties of this architecture:

- Sequence generator interacts only with the sequence driver to inject the generated items into the environment one at a time.
- A new item is generated by the sequence generator only after the previous generated item is passed to driver.
- Sequence generator is physically located inside the sequence driver. This structure has implications on how sequence generators are made to interact with multiple drivers when implementing heterogeneous sequences.
- Sequence driver interacts with the BFM to inject the items it receives from the sequence generator into the verification environment. The BFM interacts only with the driver.
- The BFM shown in this architecture is an abstract component that contains the knowledge of how a verification item can be injected into the environment. For ethernet packets, this BFM is an ethernet device equivalent, while for an item corresponding to a configuration step for this BFM corresponds to the method calls necessary to perform the specified environment configurations.
- The sequence generator and sequence driver each have their own execution threads.
- The driver and sequence threads interact only for passing a generated item to the driver. Synchronization between these two threads is achieved through built-in mechanisms in the predefined driver and sequence constructs.

The sequence generator produces a series of verification items. The sequence driver is oblivious of how this sequence is internally created in the generator² and interacts with the sequence generator only when a sequence item is handed off from the sequence generator to the sequence driver. The operation process of the sequence generator can be described in two steps:

- Step1: Sequence Generator operation
- Step2: Sequence Driver operation

In section 8.3, the implementation of a sequence generator will be explained using the default item driving mechanism. Section 8.4 focuses on different modes of driving items and transferring a generated item to the BFM. The discussion in these two sections is limited to homogeneous sequences in order to keep the focus on a single driver and single BFM implementation. Heterogeneous sequences require multiple sequence drivers and BFMs, and are discussed in section 8.5.

8.3 Homogeneous Sequences

The *e* language provides predefined constructs for the driver, sequence, and items. These constructs are:

² The sequence may have been generated using a hierarchical or a flat sequence.

- **any_sequence_driver**: a **unit** describing the sequence driver
- **any_sequence_item**: a **struct** describing a sequence item
- **any_sequence**: a struct describing a sequence generator and its sequence

These predefined constructs contain the predefined implementation that performs all generic operations for creating a sequence generator and driving the generated items into the environment. All user defined drivers, sequences, and sequence items are derived from these predefined constructs using **like** inheritance. By using **like** inheritance all predefined methods and features of these predefined constructs become available to user defined constructs. User defined sequences are implemented by extending these predefined constructs to specify additional information necessary to customize the implementation. The user defined steps are:

- Define Item Structure: Structure of the items being generated
- Name Sequence Kind: A name for the sequence being created
- Name Sequence Driver: A name for driver that interacts with the BFM
- Define Sequence Driver Interaction: Where sequence driver is placed in the environment, and how items generated by the sequence generator move to the BFM.
- Define Sequence Generator Structure: Items and subsequences contained in the sequence generator
- Define Sequence Action Block: How items and subsequences are generated

For hierarchical sequences, the above operations must be performed for each sequence kind. Once a sequence kind is defined, then it can be used as a subsequence in other sequences.

8.3.1 Verification Environment Enclosing a Sequence Generator

Consider the following *e* program used to implement a partial verification environment for injecting an ethernet packet:

```

1 : <'
2 : struct eth_packet {
3 :     type: [SIZED, QTAGGED];
4 :     size: uint;
5 :     src: uint(bits:48);
6 :     dest: uint(bits:48);
7 :     -- implementation now shown
8 : };
9 : unit eth_bfm {
10 :     agent: eth_agent; -- pointer
11 :
12 :     inject_packet(packet: eth_packet): bool is {
13 :         -- implementation not shown
14 :     };
15 : };
16 :
17 : unit eth_agent {
18 :     event clock;
19 :
20 :     bfm: eth_bfm is instance;
21 :     keep bfm.agent == me;
22 : };

```

```
23 :  
24 :   extend sys {  
25 :       eth_agent: eth_agent is instance;  
26 :   };  
27 : }>
```

This environment consists of an ethernet agent that contains a BFM used for injecting packets into the DUV ethernet port. A partial list of **eth_packet** fields is also shown. These fields will be used to create different variations of ethernet packets during the sequence generation process. The master clock is located in the **eth_agent** and in the complete environment is driven by its clock source.

8.3.2 Verification Item Definition

The first step in creating a sequence is to define a verification item by extending the predefined `any_sequence_item` construct:

```
1 : struct eth_packet_item like any_sequence_item {  
2 :     packet: eth_packet;  
3 :     keep packet.size < 500;  
4 : };
```

The item is defined as containing a member **packet** which holds the generated ethernet packet during sequence generation. Note that constraints can be applied along with the definition of this item. These constraints will hold throughout the sequence generation process. In this example, all generated packets in the sequence will have a size less than 500.

8.3.3 Driver and Sequence Creation

The next step in creating a sequence is to create the driver and the sequence generator, and is demonstrated in the following *e* code fragment:

```
1 : sequence eth_sequence using  
2 :     item = eth_packet_item,  
3 :     created_driver = eth_driver,  
4 :     created_kind = eth_sequence_kind;  
5 :
```

The **sequence** construct is an *e* statement specified at the highest level of *e* code hierarchy in parallel with **struct**, **unit**, **extend**, **import** and **other e** statements. The item type, driver name, and the sequence kind name are specified along with this statement. The item type is the name of the item that was created for the ethernet packet. The effect of this statement is:

- A sequence driver named **eth_driver** is created.
- A sequence generator named **eth_sequence** is created. This sequence generator is a member of **eth_driver** structure.
- An enumerated type named **eth_sequence_kind** is created
- **MAIN**, **RANDOM**, and **SIMPLE** sequence kinds are added to enumerated type **eth_sequence_kind** and corresponding sequence generator structure and action block are specified for each kind. These predefined kinds are created to facilitate a default behavior and are discussed later in this chapter.

8.3.4 Verification Environment Attachment

Next, the created sequence driver is attached to the existing verification environment and how data is moved from the sequence generator to the BFM is specified. This step is shown in the following code fragment:

```

      :
      :
1  :  extend eth_agent {
2  :      driver: eth_driver is instance;
3  :      keep driver.agent == me;
4  :  };
5  :
6  :  extend eth_driver {
7  :      agent: eth_agent;
8  :      event clock is only @agent.clock;
9  :
10 :      keep bfm_interaction_mode == PUSH_MODE;
11 :      send_to_bfm(item: eth_packet_item) @clock is only {
12 :          item.status = agent.bfm.inject_packet(item.packet);
13 :      };
14 :  };
      :
      :

```

The sequence driver is included in the environment by extending **eth_agent** unit. Additionally, **eth_driver** is extended to create a field to hold a pointer to the agent it is contained in. The clock for the driver is also redefined to be that of the agent. In this example, driver is configured as a **PUSH_MODE** driver which means that an item is passed to the BFM anytime that item is ready, and therefore injection time is controlled by the sequence generator. To accomplish this setting, the **bfm_interaction_mode** of the driver is constrained to **PUSH_MODE** and the predefined driver method **send_to_bfm()** is redefined to pass the generated item to the BFM. Other BFM interaction modes are explained in section 8.4.

8.3.5 User Defined Sequences

What remains is the definition of new sequence kinds and customization of the environment to merge the new sequence into the default sequence generation flow. Creating a new sequence kind requires specifying the structure of the sequence (i.e. what items, and what subsequences it holds) and the action block that will define exactly how each instance is generated.

To complete this step, the enumerated type for sequence kind is extended to add a new kind that corresponds to the new sequence being generated. The sequence structure is then defined by extending the subtype corresponding to this newly defined sequence kind. The following sections present details of flat and hierarchical sequence implementation.

8.3.5.1 Flat Sequences

A flat sequence refers to a sequence that has no subsequence. A flat sequence is implemented by defining its structure (member verification items), and the action block specifying the constraints and sequence that item instances are generated. Consider the following *e* code segment:

```

      :
      :
1  : extend eth_sequence_kind: [SEND_MIXED_PKTS];
2  : extend SEND_MIXED_PKTS eth_sequence {
3  :     min_size: uint;
4  :     keep soft min_size == 300;
5  :     !sized: eth_packet_item;
6  :     keep sized.packet.type == SIZED;
7  :     !qtagged: eth_packet_item;
8  :     keep qtagged.packet.type == QTAGGED;
9  :
10 :     body() @ driver.clock is only {
11 :         do sized keeping {
12 :             .packet.size > min_size;
13 :         };
14 :         do qtagged keeping {
15 :             .packet.size > min_size + 50;
16 :         };
17 :         do sized keeping {
18 :             .packet.size > min_size + 100;
19 :         };
20 :     };
21 : };
      :
      :

```

In the above code, the new sequence kind **SEND_MIXED_PKTS** is defined. This is a flat sequence that contains only items. The **SEND_MIXED_PKTS** subtype of **eth_sequence** is then extended to include two items, both of type **eth_packet_item**, that are constrained to generate either **SIZED** or **QTAGGED** packets. Both these items are marked as non-generated since they are generated by specific generation actions in the sequence generator. The action block for this sequence is specified by extending the **body()** predefined method. The **do** action is a predefined action of the *e* language that can only be used in methods defined in sequences. A generation

constraint may be specified for a **do** action. Sequence member `min_size` is used to show how sequences can be parameterized. The soft constraint for `min_size` sets its value to 300 in the absence of any other constraints. The use of this field is shown later in this section when building a hierarchical sequence. In this example, the action block first generates an item instance of type `eth_packet_item` with type **SIZED** and item packet size in range[301 ..499]. The type is set to **SIZED** because of the declarative constraint in defining `sized` (line 4 in above code). The item packet size is larger than 300 because of constraints specified with the **do** action, and item packet size will be less than 500 because of the declarative constraint when specifying the structure for `eth_packet_item`. The sequences of items generated are:

```
eth_packet_item with packet type = SIZED and size in [301..499];
eth_packet_item with packet type = QTAGGED and size in [351..499];
eth_packet_item with packet type = SIZED and size in [401..499];
```

Every time a new item is generated, it is passed to the BFM in **PUSH_MODE** by calling the `send_to_bfm()` predefined method.

8.3.5.2 Hierarchical Sequences

A hierarchical sequence can now be created using the **SEND_MIXED_PKTS** sequence and other items. This is shown in the following example.

```

┌
      :
      :
1  :  extend eth_sequence_kind: [SEND_SHORT_AND_MIXED_PKTS];
2  :  extend SEND_SHORT_AND_MIXED_PKTS eth_sequence {
3  :      !short: eth_packet_item;
4  :          keep short.packet.size in [60..80];
5  :      !mixed.seq: SEND_MIXED_PKTS eth.sequence;
6  :  }
7  :      body() @ driver.dock is only {
8  :          do short;
9  :          do mixed.seq keeping {
10 :              .min size ==350;
11 :          };
12 :          do short keeping {
13 :              .packet.size == 70;
14 :          };
15 :      };
16 : };
      :
      :
└
```

In this example, `SEND_SHORT_AND_MIXED_PKTS` is a hierarchical sequence that contains one item `short` and one subsequence `send_mixed_pkts_seq`. The action block specifies the sequence using these sequence members to generate item instances. The `min_size` parameter of **SEND_MIXED_PKTS** is used in the action block as a parameter for that subsequence to change its default behavior. In this example, the sequences of items generated are:

```
eth_packet_item with packet type in [SIZED, QTAGGED] and size in [60..80];
eth_packet_item with packet type = SIZED and size in [351 ..499];
eth_packet_item with packet type = QTAGGED and size in [401..499];
```

```
eth_packet_item with packet type = SIZED and size in [451..499];
eth_packet_item with packet type in [SIZED, QTAGGED] and size = 70;
```

8.3.6 Default Sequence Generation Starting Point

In *e*, the predefined sequence kinds **MAIN**, **SIMPLE**, and **RANDOM** are defined as part of processing the **sequence** construct. The implementation for these predefined sequence implementations is shown in figure 8.4. The default implementation of the sequence driver includes an instantiation of the **MAIN** sequence subtype. Therefore, the **MAIN** sequence type is the start point of a sequence generator. As shown in figure 8.4, the **MAIN** sequence randomly produces sequences of kind **SIMPLE**, and other user defined sequence kinds. In the absence of any user defined sequence kinds, the default behavior of the sequence generator is to generate count instances of **SIMPLE** subsequence and exit. The **SIMPLE** sequence simply generates one instance of the item and exits. In this system, user defined sequence kinds can be added to the flow by either over-riding the default flow or by merging the user sequence into the default flow.

<pre>extend sequence_kind: [MAIN]; extend MAIN sequence_name { count: int [1..1k]; lsequence: sequence_name; keep count <= driver.max_random_count; keep sequence.kind not in [RANDOM, MAIN]; body() @driver.clock is only { for i from 1 to count do { do sequence; }; }; };</pre>	<pre>extend sequence_kind: [SIMPLE]; extend SIMPLE sequence_name { !item: item; body() @driver.clock is only { do item; }; };</pre>
<pre>extend sequence_kind: [RANDOM]; extend RANDOM sequence_name { count: int [1..1k]; keep count <= driver.max_random_count; lsequence: sequence_name; keep sequence.kind not in [RANDOM, MAIN]; keep depth_from_driver >= driver.max_random_depth => sequence.kind == SIMPLE; body() @driver.clock is only { for i from 1 to count do { do sequence; }; }; };</pre>	

Figure 8.4 Predefined Sequence Kinds: MAIN, SIMPLE, RANDOM

8.3.6.1 Merging New Sequence Kind with the Default Start Point

A new user defined sequence kind is by default merged into the sequence generation flow since the MAIN sequence type generates random subsequence kinds that are of kind **SIMPLE** or any user defined sequence kinds. More precise control can be achieved by using the following code fragment:

```

      :
      :
1   :   extend MAIN eth_sequence {
2   :       keep sequence.kind == [SEND_SHORT_AND_MIXED_PKTS];
3   :       keep count == 1;
4   :   };
      :
      :

```

Above, the MAIN sequence is constrained so that it only generates one subsequence which is constrained to be the new sequence kind.

8.3.6.2 Over-riding the Default Start Point to a New Sequence Kind

One approach towards merging user defined sequence kind into the flow is to completely over-ride the default behavior of the MAIN sequence. This approach is shown in the following user defined replacement of the default flow:

```

      :
      :
1   :   extend MAIN eth_sequence {
2   :       !my_seq: SEND_SHORT_AND_MIXED_PKTS eth_sequence;
3   :
4   :       body() @driver.clock is only {
5   :           do my_seq;
6   :       };
7   :   };
      :
      :

```

The generation order for this sequence is shown in figure 8.5.

8.3.7 Sequence Generator Flow Customization

The sequence driver for a sequence starts that sequence in its **run()** method (see section 8.4). Once a sequence has been started, it goes through a predefined order of execution. The pseudo code for sequence generation flow is shown in figure 8.6. Note that this pseudo code is only accurate to the extent that it reflects the sequence generator operational flow.

As mentioned, the sequence driver starts the default sequence (i.e. **MAIN**) by calling its **start_sequence()** method in its **run()** method. This method starts the **internal_body()** method

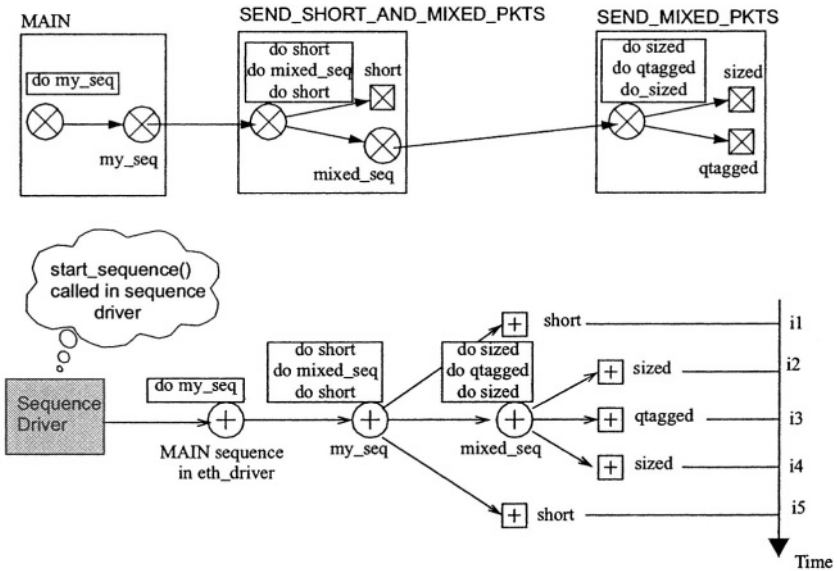


Figure 8.5 Generated ethernet Packet Sequence

for that sequence. The execution flow for **internal_body()** is shown in figure 8.6 and consists of calling **pre_body()**, **body()**, and **post_body()** methods. Note that all methods are by default empty and therefore if no extension is performed for these methods in a new sequence kind, then the sequence simply exits. The action block for a sequence is implemented using the **do** action. This action can be called for a subsequence or for an item. The flow for each of these cases is shown in methods **do_sequence()** and **do_item()**. The following observations can be made about these methods:

- Methods emit events throughout their operation. These events can be used to synchronize or learn about the internal operation of a sequence generator.
- Only **do_item()** method interacts with the driver. Synchronization with the sequence driver and its BFM are done while in **do_item()** method. The interaction between **do_item()** method and the driver is explained in section 8.4.
- The methods indicated in this figure with **BOLD** typeset can be extended by to merge the user's program with the default flow. Since some of these methods are TCMs, additional implementation specific synchronization can be obtained by using wait statements in these TCMs.
- For subsequences, the **internal_body()** method is not started, but its **do_sequence()** method is called by its parent sequence. As a result, **pre_body()** and **post_body()** methods of a sequence are not executed unless that sequence is explicitly started through its **start_sequence()** method.

<pre> struct any_sequence { start_sequence() is { start internal_body(me); }; internal_body(seq) @sys.any { seq.pre_body(); emit started; seq.body(); emit ended; seq.post_body(); }; }; </pre>	<pre> extend any_sequence { do_sequence(seq: any_sequence) @sys.any is { pre_do(FALSE); gen seq keeping { .parent_sequence == me; .driver == me.driver; --Additional constraints specified with - do action are placed here }; mid_do(seq); emit seq.started; seq.body(); emit seq.ended; post_do(seq); }; do_item(item: any_sequence_item) @sys.any is { driver.add_item_to_queue(item); emit driver.new_item_added; pre_do(TRUE); wait @ready_to_gen; -- emitted by driver gen item keeping { .parent_sequence == me; .driver == me.driver; --Additional constraints specified with - do action are placed here }; mid_do(item); emit @driver_done; wait @driver.item_done; post_do(item); }; }; </pre>								
<p>Method in BOLD can be extended by user.</p> <p>TCMs:</p> <table border="0"> <tr> <td>pre_body()</td> <td>@sys.any</td> </tr> <tr> <td>body()</td> <td>@driver.clock</td> </tr> <tr> <td>post_body()</td> <td>@sys.any</td> </tr> <tr> <td>pre_do(is_item: bool)</td> <td>@sys.any</td> </tr> </table> <p>Methods:</p> <pre> mid_do(s: any_sequence_item) post_do(s: any_sequence_item) </pre> <p>Note: any_sequence is derived from any_sequence_item through like inheritance. Therefore it is possible to call mid_do() and post_do() with either a sequence or an item</p>	pre_body()	@sys.any	body()	@driver.clock	post_body()	@sys.any	pre_do(is_item: bool)	@sys.any	
pre_body()	@sys.any								
body()	@driver.clock								
post_body()	@sys.any								
pre_do(is_item: bool)	@sys.any								

Figure 8.6 Sequence Generator Pseudo Code

In the generation order shown in figure 8.5, **start_sequence()** method is called by the sequence driver. **do_item()** method is called for all do actions on items and **do_sequence()** methods is called on all do actions on subsequences. Using the diagram in figure 8.5, it is easy to determine the order of method calls for the generates items and subsequences. Given this predefined ordering, the user code can then be merged into this flow, according to the specific requirements of the generations scenario.

8.4 Sequence Synchronization

Synchronization between a verification environment and a sequence as it generates items- takes place through the sequence driver. The pseudo code representing the internal implementation of the sequence driver is shown in figure 8.7³. As shown in this implementation, the sequence

driver contains a predefined **MAIN** sequence. This sequence is generated in the **run()** method of the driver and then its **start_sequence()** method is called. The sequence driver can be in either **PULL_MODE** or **PUSH_MODE**. In **PUSH_MODE**, once a new item is generated by the sequence generator, that item is passed to the BFM by the driver. In the **PULL_MODE**, the BFM asks the driver for the next item once it is ready to process that item. If the driver is configured to be in the **PUSH_MODE**, then method **send_loop()** is started in its **run()** method. The use of **PULL_MODE** vs. **PUSH_MODE** is discussed in the following subsections.

<pre> unit any_sequence_driver { lsequence: MAIN any_sequence_item; run() is also { gen sequence; sequence.start_sequence(); if (bfm_interaction_mode==PUSH_MODE) { start send_loop(); }; }; send_loop() @clock is { while TRUE { item = get_next_item(); send_to_bfm(item); emit item_done; }; }; }; </pre>	<pre> extend any_sequence_driver { get_next_item():any_sequence_item @clock is { item = get_next_queued_item(); if (item == NULL) { wait @new_item_added; }; emit item.parent_sequence.ready_to_gen; wait @item.parent_sequence.driver_done; return item; }; try_next_item(): any_sequence_item @clock is { item = get_next_queued_item(); if (item == NULL) { return NULL; }; en.t item.parent_sequence.ready_to_gen; wait @item.parent_sequence.driver_done; return item; }; }; </pre>
--	--

Figure 8.7 Sequence Driver Pseudo Code

8.4.1 Sequence and Sequence Driver Interaction

The sequence driver and the sequence generator each have their own execution thread. The synchronization between the sequence generator and sequence driver is accomplished through handshaking between the **do_item()** method of the sequence generator and **get_next_item()** method and **item_done** event of the sequence driver. This handshaking mechanism is shown in figure 8.8. Both **PULL_MODE** and **PUSH_MODE** are implemented using the **get_next_item()** method. The driver also contains the **try_next_item()** which behaves the same as

³. Multiple threads of **do_item()** method can run in parallel for the same sequence generator, which means that all these threads must synchronize with the same sequence driver. Even though it is not shown explicitly in this pseudo code, each thread running **do_item()** maintains the required handshaking protocol to pass its generated item to the sequence driver.

`get_next_item()` if an item is available in the driver queue. Otherwise, `try_next_item()` returns immediately with a NULL return value.

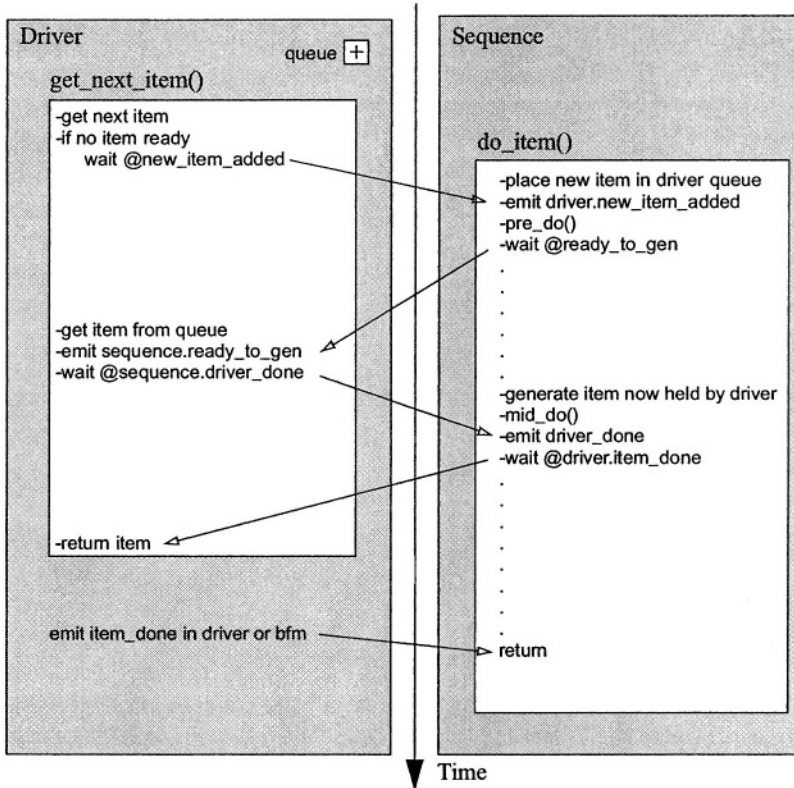


Figure 8.8 `do_item()` and `get_next_item()` synchronization

8.4.1.1 Push Mode

In the push mode, the driver injects the next generated item into the environment as soon as that item is available. If the driver is configured as push mode, then predefined method `send_loop()` of the driver is started in the `run()` method of the driver. The use of `get_next_item()` and `item_done` event are clearly shown in the implementation of `send_loop()` shown in figure 8.7. If the driver is configured in the push mode, then all that remains is to extend the definition for the `send_to_bfm()` to perform the necessary steps. Configuration of the `eth_driver` in push mode was shown in section 8.3.4.

8.4.1.2 Pull Mode

In the pull mode, the driver does not start its `send_loop()` method. Instead it is up to the BFM to issue the `get_next_item()` method and to also emit the `item_done` event. The implementation of `eth_driver` in pull mode is shown in the following:

```

      :
      :
1  :  extend eth_bfm {
2  :      inject_pkt(epkt: eth_packet) is {
3  :          --implementation now shown
4  :      };
5  :
6  :      execute_items() @agent.clock is {
7  :          var eitem: eth_packet_item;
8  :          while TRUE {
9  :              eitem = agent.driver.get_next_item();
10 :             inject_pkt(eitem.packet);
11 :             emit agent.driver.item_done;
12 :          }
13 : };
14 :     run() is also {
15 :         start execute_items();
16 :     };
17 : };
      :
      :

```

8.4.2 Multiple Sequence Synchronization

It is possible to have multiple sequences pass items to the same sequence driver. An example of a sequence with multiple threads generating items for the same driver is shown in the following sequence kind:

```

      :
      :
1  :  extend eth_sequence_kind: [SEND_PARALLEL];
2  :  extend SEND_PARALLEL eth_sequence {
3  :      !mixed_seq1: SEND_MIXED_PKTTS eth_sequence;
4  :      !mixed_seq2: SEND_MIXED_PKTTS eth_sequence;
5  :
6  :      body() @ driver.clock is only {
7  :          all of {
8  :              {do mixed_seq1 keeping {.min_size == 100};
9  :              {do mixed_seq2 keeping {.min_size == 200};
10 :          };
11 :      };
12 : };
      :
      :

```

The synchronization mechanism between a sequence generator and the driver takes into account that multiple threads from the same sequence implementation may be passing items to the driver and operates in a first-come-first-serve manner.

8.5 Heterogeneous Sequences

Items in a heterogeneous sequence are injected into the environment using different BFM. These items may all have the same or different types. A sequence that injects ethernet packets into two different ports of a DUV is considered a heterogeneous sequence even though all its items have the same type. A sequence that injects ATM and ethernet packets into two DUV ports is also a heterogeneous sequence since it requires two different BFM to inject its item.

Consider the ethernet packet sequence generated in figure 8.5. In that implementation, all the generated items were injected into the environment using the same BFM. The case shown in figure 8.9 where the packet is generated must be sent to different BFM. In this instance, the implementation of `eth_packet` must be changed to a heterogeneous sequence to support the two target BFM.

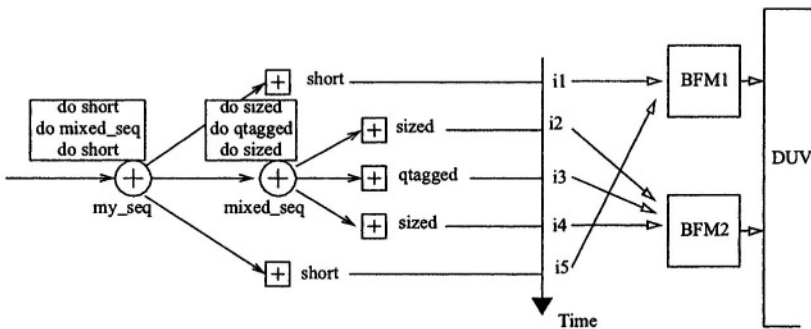


Figure 8.9 ethernet Packet Sequence Targeted to Multiple BFM

Two approaches can be used to implement heterogeneous sequences: 1) using virtual BFM, 2) using virtual Drivers. These approaches are discussed in the next subsections.

8.5.1 Implementation Using Virtual BFM

A *virtual BFM* refers to an abstract BFM that can handle item types that are headed to multiple BFM. In this implementation, the definition for an item is extended to identify the destination BFM for an item. The abstract BFM will then send a generated item to its corresponding BFM.

The abstract BFM approach can only be used in the `PUSH_MODE` driver mode. With this approach the same sequence driver is used to drive items for different BFMs. In push mode, all BFMs are waiting to receive the next item and the abstract BFM can decide which BFM should receive the new item. However, in pull mode, BFMs will request for items when they are ready to handle that data item. Since all BFMs request items from the same driver, then the driver will provide each BFM with the next generated item on a first-come-first-serve basis. The effective order of BFM pull operations may not be the one required by the sequence, and this leads to corrupted sequence item distribution between BFMs.

In the following example, the implementation of the ethernet packet sequence generator is modified using an abstract BFM to send the generated items to the BFMs indicated in figure 8.9:

```

      :
      :
1  :  extend eth_packet {
2  :      dest_bfm: [BFM1, BFM2];
3  :  };
4  :  extend eth_bfm {
5  :      bfm1: eth_port_bfm;
6  :      bfm2: eth_port_bfm;
7  :      inject_packet(packet: eth_packet) is only {
8  :          if(packet.dest_bfm == BFM1) {
9  :              bfm1.inject_packet(packet);
10 :          } else {
11 :              bfm2.inject_packet(packet);
12 :          };
13 :  };
14 :  extend SEND_MIXED_PKTS eth_sequence {
15 :      keep sized.packet.dest_bfm == BFM2;
16 :      keep qtagged.packet.dest_bfm == BFM2;
17 :  };
18 :  extend SEND_SHORT_AND_MIXED_PKTS eth_sequence {
19 :      keep short.packet.dest_bfm == BFM1;
20 :  };
      :
      :

```

In this implementation, the `eth_packet` is extended to include a destination field. The implementation of `eth_bfm` is extended to include two BFMs and method `inject_packet()` is extended to redirect the generated packet to the BFM indicated by its `dest_bfm` field. The definitions for `SEND_MIXED_PKTS` and `SEND_SHORT_AND_MIXED_PKTS` `eth_sequence` structs have also been extended to indicate the destination BFM for its items. It should be emphasized that this implementation will only work for `PUSH_MODE` interaction mode.

8.5.2 Implementation Using Virtual Drivers

A virtual driver is a sequence driver that contains no sequence items and is used to drive other sequence drivers. Virtual drivers can be used to implement a robust heterogeneous sequence that works for both `PULL_MODE` and `PUSH_MODE` configuration of the environment.

A virtual driver is implemented using the **sequence** statement without defining any item.

```

      :
      :
1  :  sequence virtual_eth_sequence using
2  :      created_driver = virtual_eth_driver;
      :
      :

```

The approach for building a heterogeneous sequence using virtual drivers is as follows:

1. Create an item type for each item generated by the sequence.
2. Define a new sequence using the **sequence** statement for each item type that is to be generated by the scenario. Extend each driver to setup the desired BFM interaction mode.
3. Create a virtual driver using the **sequence** statement. Extend the driver structure to contain pointers to all drivers in the environment.
4. Instantiate a driver in the environment for each BFM that is used to drive items from the sequence. Customize each driver instance to point to the BFM used for driving its items.
5. Extend the virtual driver to include pointers to all BFM drivers instantiated in step 4.
6. At this point, the virtual driver and BFM drivers each include a sequence. The driver field of each sequence points to its own driver. Disable the sequences for BFM drivers (over-ride their body() method) and modify the MAIN subtype of the virtual sequence to generate a sequences using the sequence definitions for each item type.
7. Extend each sequence to include a pointer to the virtual driver. This extension is not necessary for the virtual sequence since its driver field already points to the virtual driver. When building a new sequence, update this pointer by adding the necessary constraints. This is an example of parameters passing across sequences. The virtual driver contains pointers to all BFM drivers and by keeping this pointer updated during sequence generation, all subsequences have access to BFM drivers.
8. When using a **do** action on a subsequence, constrain the driver field of that sequence to point to the appropriate driver, which should be available from the pointer to the virtual driver.

The steps for building the sequence generator for the sequence shown in figure 8.9 is shown in the following implementation. Note that in this sequence, item types are the same but sent to different BFMs.

Assume the verification environment has the following structure:

```

      :
      :
1  :  struct eth_packet {
2  :      type: [SIZED, QTAGGED];
3  :      size: uint;
4  :      src: uint(bits:48);
5  :      dest: uint(bits:48);
6  :      -- implementation now shown
7  :  };
8  :  unit eth_bfm {
9  :      agent: eth_agent; -- pointer
10 :      inject_packet(packet: eth_packet): bool is {
11 :          -- implementation not shown
12 :      };
13 :  };

```

```

14 :
15 :   unit eth_agent {
16 :     event clock is @sys.any;
17 :
18 :     bfm1 : eth_bfm is instance;
19 :     keep bfm1.agent == me;
20 :     bfm2: eth_bfm is instance;
21 :     keep bfm2.agent == me;
22 :   };
23 :
24 :   extend sys {
25 :     eth_agent: eth_agent is instance;
26 :   };

```

The environment consists of an agent that contains two instantiation of the same BFM. Each BFM has a pointer to its agent container. Only one sequence item type is necessary since only ethernet packets are being generated. This is implemented as:

```


```

```

1 : struct eth_packet_item like any_sequence_item {
2 :   packet: eth_packet;
3 :   status: bool;
4 : };

```

The next step is to define a new sequence for each item type being generated and to extend the driver for that sequence for attachment to its BFM. In this case, only one BFM sequence for ethernet packet item is required.

```


```

```

1 : sequence eth_sequence using
2 :   item = eth_packet_item,
3 :   created_driver = eth_driver,
4 :   created_kind = eth_sequence_kind;
5 :
6 : extend eth_driver {
7 :   bfm: eth_bfm;
8 :   event clock is only @bfm.agent.clock;
9 :
10 :   keep bfm_interaction_mode == PUSH_MODE;
11 :   send_to_bfm(item: eth_packet_item) @clock is first {
12 :     item.status = bfm.inject_packet(item.packet);
13 :   };
14 : };

```

Next, a virtual sequence and its corresponding virtual driver are created. The virtual driver is extended to contain a pointer for each driver it must drive.

```

      :
      :
1  : sequence virtual_eth_sequence using
2  :     created_driver = virtual_eth_driver;
3  :
4  : extend virtual_eth_driver {
5  :     agent: eth_agent;
6  :     event clock is only @agent.clock;
7  :
8  :     driver1: eth_driver; -- pointer
9  :     driver2: eth_driver; -- pointer
10 : };
      :
      :

```

Following that, all sequence drivers are instantiated in **eth_agent** and appropriate constraints are set to set the pointers in virtual driver point at BFM drivers.

```

      :
      :
1  : extend eth_agent {
2  :     edriver1: eth_driver is instance;
3  :     keep edriver1.bfm == bfm1;
4  :     edriver2: eth_driver is instance;
5  :     keep edriver2.bfm == bfm2;
6  :     vdriver: virtual_eth_driver is instance;
7  :     keep vdriver.driver1 == edriver1;
8  :     keep vdriver.driver2 == edriver2;
9  :     keep vdriver.agent == me;
10 : };
      :
      :

```

Then the **eth_sequence** is extended to add a pointer to the pointer to virtual pointer. This pointer is used to pass the pointer to the virtual driver in the sequence generation hierarchy.

```

      :
      :
1  : extend eth_sequence {
2  :     vdriver: virtual_eth_driver; -- pointer
3  :     keep soft vdriver == NULL;
4  : };
      :
      :

```

Next the default definition for **eth_sequence** is modified to disable its sequence thread. The definition for **virtual_eth_sequence** is also redefined so that the sequence generates a sequence of type **eth_sequence**. The constraint on line 6 sets the virtual driver pointer inside the subsequence. By default, a subsequence inherits the driver for its parent sequence. The constraint on line 11 over-rides this default behavior and forces the driver for the **eth_seq** subsequence to use **driver1** which is connected to **bfm1**.

```

      :
1 : extend MAIN eth_sequence {
2 :     keep count == 0;
3 : };
4 : extend MAIN virtual, eth_sequence {
5 :     !my_seq: SEND_SHORT_AND_MIXED_PKTS eth_sequence;
6 :     keep my_seq.vdriver == driver;
7 :
8 :     body() @driver.clock is only {
9 :         do my_seq keeping {
10 :             .driver == driver.driver1 ;
11 :         };
12 :     };
13 : };
      :

```

The sequence kinds for **eth_sequence** sequence are now defined while the virtual driver pointer is updated for all subsequences and using any driver that is required for a subsequence.

```

      :
1 : extend eth_sequence kind: [SEND_SHORT_AND_MIXED_PKTS];
2 : extend SEND_SHORT_AND_MIXED_PKTS eth_sequence {
3 :     !short_eitem: eth_packet_item;
4 :     keep short_eitem.packet.size in [60..80];
5 :     !send_mixed_pkts_seq: SEND_MIXED_PKTS eth_sequence;
6 :     keep send_mixed_pkts_seq.vdriver == vdriver;
7 :
8 :     body() @ driver.clock is only {
9 :         do short_eitem;
10 :         do send_mixed_pkts_seq keeping {
11 :             .driver == vdriver.driver2;
12 :             .min_size == 350;
13 :         };
14 :         do short_eitem keeping {,packet.size == 70;};
15 :     };
16 : };
      :

```

In the code above, the definition for **SEND_MIXED_PKTS** is as defined in section 8.3. The definition for **SEND_SHORT_AND_MIXED_PKTS** however is modified by adding a constraint on line 6 to update the virtual driver pointer, and a constraint on line 11 to specify the driver that is to be used while performing the **do** action for a subsequence. The **do** action on line 10 can be issued multiple times in the **body()** method each time using a different driver, causing the generated items to be sent to different BFM. Creating a virtual driver pointer in this case allows subsequences at any depth of the sequence hierarchy to use any driver available in the environment.

Figure 8.10 shows a pictorial view of driver assignment while the sequence is being generated. In this diagram, items *i2*, *i3*, and *i4* are driven by driver2 and items *i1* and *i5* are driven by driver 1.

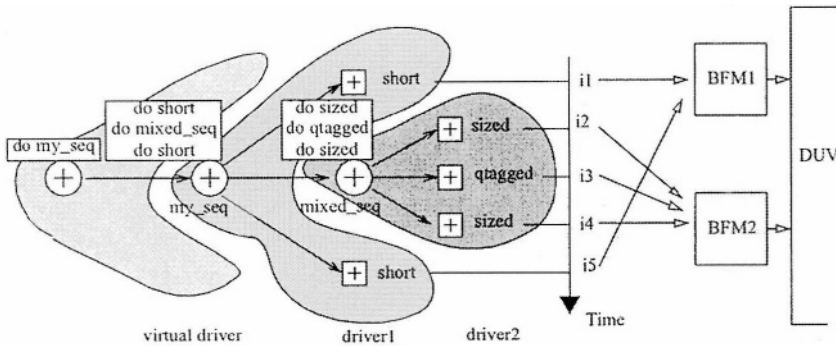


Figure 8.10 ethernet Packet Sequence Targeted to Multiple BFM

Note that this implementation will work as expected in both pull and push operation mode. When in pull mode, each BFM will request the next item when it is ready to drive it. Assume the sequence is generating item *i3* and both **bfm1** and **bfm2** try to pull the next item. The pull action by **bfm2** will block until the sequence generator moves back to **driver1** which is when item *i5* is being generated. Therefore, even if both BFMs request the next item at the same time, only the BFM that is targeted by the sequence generator will receive the generated item. Also, each driver may be configured to operate either in push or pull mode. This is an important property that allows the creation of complex handshaking mechanisms between multiple BFMs.

Even though the above example was constructed for a sequence with same item types, the same technique can be used to generate sequences with different types of items and those using different BFMs.

8.6 Summary

This chapter introduced the concept of sequences and how they are used to generate verification scenarios. The architecture and implementation of sequences in the *e* language were explained and approaches for building homogeneous and heterogeneous sequences were discussed. Homogeneous sequences are used when only BFM is required for driving the generated sequence. Heterogeneous sequences are used with multiple BFMs. The concept of virtual sequences and virtual drivers were also introduced and used to implement heterogeneous sequences that will operate with any combination of pull and push modes from its multiple drivers.

This page intentionally left blank

PART 4

Response Collection, Data
Checking, and Property Moni-
toring

This page intentionally left blank

Temporal expressions are a powerful method for defining and identifying complex behaviors that span multiple program runtime ticks. Once a specific temporal behavior has been described using a temporal expression, its definition is used to achieve thread synchronization and implement property assertions, which in turn are used to implement monitors and protocol checkers.

Temporal expressions provide a declarative approach for defining temporal behavior and afford the programmer the benefits of declarative programming discussed in chapter 4. Moreover, assertion checking is implemented using declarative constructs to check that a desired temporal behavior maintained throughout the program execution.

This chapter describes temporal expressions in detail and introduces temporal operators that are used to combine events and base temporal expressions to describe complex program behavior. This chapter also describes common temporal behaviors and their equivalent implementation in *e*.

9.1 Temporal Expression Basics

Base temporal expressions were introduced in section 4.6.1.1. One important property of base temporal expressions is that they are evaluated within one sampling period of their sampling event (either in the sampling period or at the sampling event). As such, these base expressions define only properties that can be identified in a single sampling period. The conditions for a base temporal expression to succeed are described in section 4.6.1.1.

Temporal expressions describe behaviors that span multiple sampling periods, and are constructed by using temporal operators to combine base and non-base temporal expressions. The conditions for a composite temporal expression to succeed are defined recursively based on the success of base temporal expressions and the temporal operator that is being used.

The *Sequence Temporal Operator* is used to create temporal expressions:

```
{temporal_subexp1;temporal_subexp2;..} @sampling_event
```

In this definition, each temporal sub-expression may be a base temporal expression or a temporal expression defined by the sequence operator. Each temporal sub-expression separated by the semicolon operator is evaluated over successive occurrences of the sampling event. The evaluation for each temporal sub-expression begins in the sampling period following that in which the preceding temporal sub-expressions succeeded. The temporal expression defined by the sequence operator succeeds when its last temporal sub-expression succeeds. The evaluation for this sequence fails if evaluation of any of these temporal sub-expressions fails.

An example of using temporal expressions is shown below.

```

1  :  <
2  :  unit config_env {
3  :      event clk is rise('top.clk')@sim;
4  :      event start_sim is rise('top.start_sim')@clk;
5  :
6  :      configure() @clk is {
7  :          wait {fall('top.reset'); [2]; @start_sim};
8  :          -- configure here
9  :      };
10 :
11 :      run() is also {
12 :          start configure();
13 :      };
14 :  };
15 :  >

```

In the above example, **'rise('top.clk')@sim'**, **'rise('top.start_sim')@clk'**, **'fall('top.reset')**, and **'@start_sim'** are base temporal expressions. The temporal expression used with the **wait** action on line 7 is defined using base temporal expressions and the repeat temporal operator (see section 9.3.3.2). The sampling event for this temporal expression is the default sampling event for the **configure()** TCM which is **clk**.

Figure 9.1 shows the visual notation used to indicate stages of evaluation for a temporal expression. This figure also shows the stages of evaluation for the complex temporal expression on line 7 of above example. As shown in the figure, the valuation of the temporal expression sequence starts after the first occurrence of **fall('top.reset')**. There is a 2 cycle wait after evaluation starts. However, since there is no occurrence of a **start_sim** event, evaluation fails. With the second occurrence of **fall('top.reset')** the evaluation of the temporal expression sequence is again started. After a 2 **clk** cycles, event **sim_start** occurs and the evaluation of the sequence succeeds. Temporal expression evaluations are discussed in detail in section 9.2.1 using an abstract model of temporal expression evaluation presented in section 9.2.1

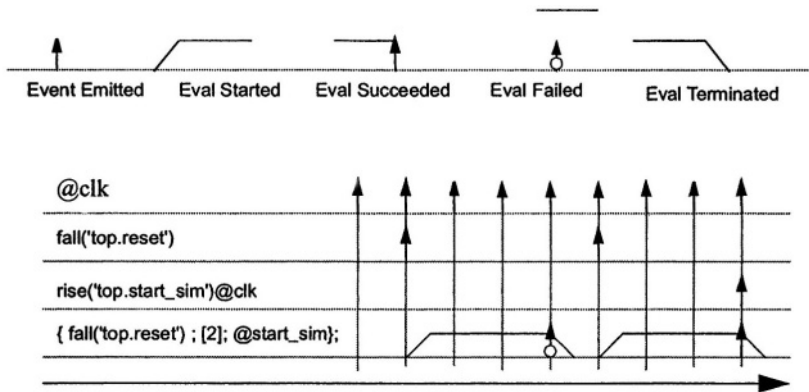


Figure 9.1 Evaluating Complex Temporal Expressions

9.2 Temporal Expression Evaluation

A number of factors affect the evaluation of temporal expressions. These factors include program context, temporal operators used to create the expression, temporal expression attachment to a specific thread, and number of sub-threads needed to evaluate multiple ways the same temporal expression may succeed. A clear understanding of these factors is important for understanding the expected behavior of a temporal expression and for building temporal expressions that match the intended behavior. Temporal expression evaluations are discussed in detail in section 9.2.1 using an abstract model of temporal expression evaluation. Section 9.2.3 discusses factors related to concurrent evaluation of temporal expressions and how temporal expression evaluations are started.

9.2.1 Evaluation Abstract Model

Evaluation of a temporal expression always begins with a single thread. As evaluation progresses, multiple threads that run in parallel may be created to evaluate the multiple ways a temporal expression may succeed. The creation of new evaluation sub-threads are controlled by the temporal operators used to construct a temporal expression.

Figure 9.2 shows the abstract model used to describe the operation of a temporal operator. This model consists of the following components:

- Number of Evaluation Sub-threads
- Success Equation

- Termination Condition
- Failed Status

The number of sub-threads for a temporal operator is defined by the function of that operator. For example, an **or** operator requires two sub-threads while a repeat operator requires only one sub-thread. Sub-threads may span multiple cycles of the sampling event and do not necessarily take the same number of cycles to complete. The terminate condition is defined either as ALL or ANY. For the ALL conditions, sub-threads must complete before the evaluation for an operator terminates. For an ANY condition, all sub-threads are terminated once any of the sub-threads terminate. The success equation defines the times at which evaluation of an operation succeeds. The evaluation of a temporal operator may result in multiple successes. Evaluation of a temporal operator fails when all evaluation sub-threads terminate without the temporal expression ever succeeding. Failed status for a temporal expression is raised when a failed status is identified in the last cycle of evaluation. This abstract model is used for defining the behavior of each temporal operator in section 9.3.

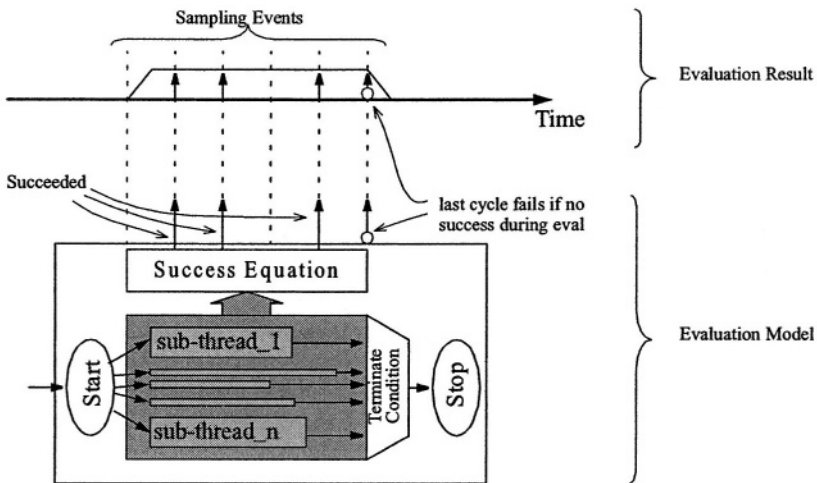


Figure 9.2 Abstract Model for Temporal Operator Evaluation

The evaluation model of base temporal expressions requires only one cycle of the sampling event, and only one thread (using the main evaluation thread). The success equation for base temporal expressions was described in section 4.6.1.1. Evaluation flow of all temporal expressions is described recursively in terms of the evaluation model of its temporal operators and the evaluation model of base temporal expression.

9.2.2 Sequence Temporal Operator

The syntax for the sequence operator is:

$$T: \{T_1; T_2; \dots\}$$

The evaluation flow for a temporal expression created by the temporal operator $\{T_1; T_2; \dots\}$ is described as follows:

- Number of Evaluation Sub-threads: T_1 , the first temporal sub-expression in the sequence, is evaluated in the thread that started the evaluation. For any successful temporal sub-expression T_n at position n , a new evaluation sub-thread is started for temporal sub-expression T_{n+1} . This new evaluation thread is started at the sampling event following the one in which T_n succeeded. An evaluation sub-thread is terminated if the temporal sub-expression it is evaluating never succeeds (i.e. fails), or when it evaluates the last sub-expression in the sequence.
- Success Equation: The temporal expression created by a sequence operator succeeds when any of the sub-threads evaluating the last temporal sub-expression in the sequence succeeds.
- Termination Condition: The evaluation of a sequence temporal expression terminates with all its evaluation sub-threads have terminated.

Figure 9.3 shows an example of how the evaluation of a sequence temporal expression progresses in terms of the evaluation model of the sequence operator and evaluation flow of its temporal sub-expressions. Observations on this evaluation are:

- Evaluation starts at time 0 by evaluating temporal expression T_1 in the main thread.
- T_1 Succeeds once at cycle 1 and once at cycle 3.
- Evaluation A of T_2 starts at cycle 1 and succeeds once at cycle 3.
- Evaluation B of T_2 starts at cycle 3 and succeeds 2 times at cycle 4 and 5.
- Evaluation A of T_3 starts at time 3 and fails at time 5.
- Evaluation B of T_3 starts at time 4 and succeeds at time 6. This sub-thread of evaluation is terminated since the **fail** operator is used for sub-expression T_3 . Evaluations along this thread will not continue.
- Evaluation C of T_3 starts at time 5 and fails at time 7.
- Evaluation A of T_4 starts at time 5 and succeeds at times 6 and 8. Both these successes are reported for the temporal expression T and the sub-thread is terminated since T_4 is the last sub-thread in the sequence.
- Evaluation B of T_4 starts at time 7 and succeeds at time 9. This success is reported for the temporal expression T and sub-thread is terminated since T_4 is the last sub-expression in the sequence.
- Evaluation flow for this temporal expression terminates at time 9 when all evaluation sub-threads have terminated.

In this example, the evaluation started with one thread, and grew to three sub-thread, two of which produced three successes for the evaluation of this temporal expression.

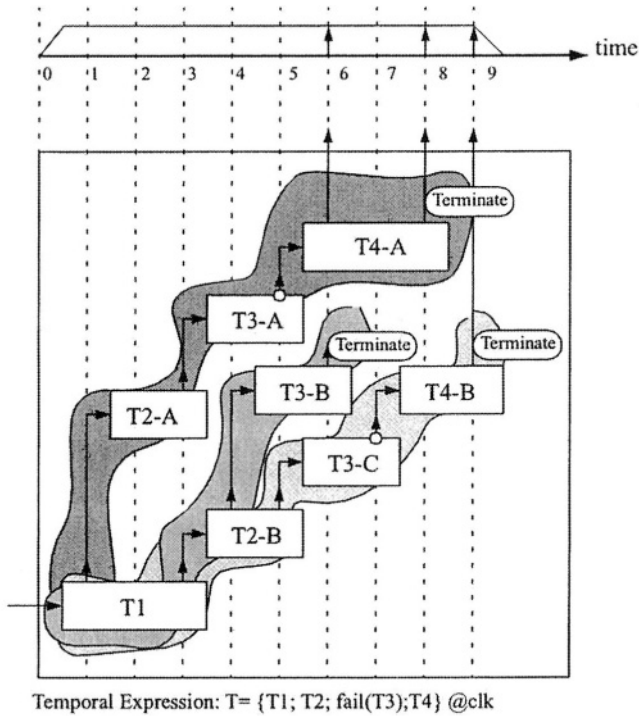


Figure 9.3 Evaluating Temporal Expressions Using the Abstract Model

Given the abstract model of temporal operators, the behavior for any temporal expression is computed using the procedure illustrated in figure 9.3. Start of an evaluation is controlled by the temporal expression context and is discussed in section 9.2.3. The temporal operators in the *e* language are described using the evaluation model in section 9.3.

9.2.3 Evaluation Threads and Program Context

Evaluation threads for temporal expressions are derived from the context of that temporal expression. Temporal expressions may be used in

- Declarative Statements
- Sequential Actions

Declarative statements include **event**, **expect**, and **assume** struct members. Sequential actions include the **wait** and **sync** (section 4.6.3).

Evaluation of a temporal expression used in a sequential action is started when that action is reached during program execution. The default sampling event of the TCM containing that

sequential action is used as the default sampling event for any temporal expression contained in that TCM. Even though a temporal expression may succeed multiple times for the same evaluation, the evaluation of a temporal expression used in a sequential action is terminated when it succeeds for the first time.

Evaluation of a temporal expression used in a declarative statement is started in the `run()` method of that struct, and a new evaluation of that temporal expression is started at every occurrence of its sampling event thereafter. Each evaluation of this temporal expression is continued until all sub-threads for evaluating that temporal expression complete.

Events in a struct definition can be used to define the temporal expressions for other events. Under such conditions, the temporal expression for each event definition runs in its own evaluation thread. Consider the following example:

```

1  :  <
2  :  struct event_holder {
3  :      event A is {fall('reset'); rise ('clk')} @sim;
4  :      event B is rise('simulation_stop') @sim;
5  :      event C is rise('simulation_start') @sim;
6  :      event D is {@C;@A;@B} @sim;
7  :      event E is {@C; { fall('reset'); rise ('clk'); } @B}@sim;
8  :  };
9  :  >

```

In this example, events A, B, C, and D are all evaluated in their own evaluation threads. As such, the detail of each evaluation is not visible to the other temporal expressions. Evaluation for event D can only check if event A succeeded during its sampling event. This is an important observation in evaluating temporal expressions since it allows for concurrent evaluation of expressions. For example, by the time event A is emitted, **fall(reset)** and **rise(clk)** have been detected in consecutive cycles. Event D checks that by its second evaluation cycle, **fall(reset)** followed by **rise(clk)** has been observed. Event E however is evaluating this sequence as part of its own evaluation thread and is therefore consuming the necessary cycles to evaluate this sub-expression. Depending on the expected temporal behavior, either method may need to be used.

9.2.4 Detach Operator

Detach operator is used to explicitly create a new thread that runs in parallel with the evaluation thread of its containing temporal expression. The evaluation of a sub-thread started by the **detach** operator starts at the same time as the evaluation thread for its parent temporal expression.

The *Detach Temporal Operator* creates an implicit event which is then used to replace the detached temporal expression. In the following example, events D and E are equivalent.

```

1  :  <
2  :  struct even_holder {

```

```

3 :      event A is {fall('reset'); rise ('clk')} @sim;
4 :      event B is rise('simulation_stop') @sim;
5 :      event C is rise('simulation_start') @sim;
6 :      event D is {@C;@A;@B} @sim;
7 :      event E is {@C; detach((fall('reset'); rise ('clk'))); @B}@sim;
8 :      };
9 :      >

```

9.2.5 exec Construct

The **exec** construct allows a non-time-consuming procedural code-block to be attached to a temporal expression such that once that temporal expression succeeds that code-block is executed. This construct allow for procedural code to be inserted in the evaluation flow of a temporal sequence. Since temporal sequences are composed of a series of temporal sequences, the **exec** construct allows for non-time-consuming procedural code to be inserted in the evaluation flow of a temporal sequence.

```

1 :      <'
2 :      extend exec_ex {
3 :          data : list of uint;
4 :          event clk is rise ('top.clock')@sim;
5 :          event load_data is (
6 :              rise('top.valid') exec {
7 :                  var tmp_data : uint = 0x1000;
8 :                  for i from 1 to 100 {
9 :                      data.add(tmp_data);
10 :                     tmp_data = tmp_data + 0x10;
11 :                 };
12 :                 [..];
13 :                 fall('top.valid') exec {
14 :                     outf("valid deasserted");
15 :                 };
16 :             } @clk;
17 :         };
18 :     >

```

In the above example, the data list is populated with 100 integers when a rise in signal **top.valid** is detected. Also a message is printed when a fall in signal **top.valid** is detected.

9.3 Temporal Operators

Temporal operators are divided into three categories:

- Base Temporal Operators
- Atomic Temporal Operators
- Composite Temporal Operators

Base Temporal Operators are used to create base temporal expressions from variables in the e program or in the external simulator. Atomic Temporal Operators define a new type of operation that is described in terms of the evaluation model presented in section 9.2.1. Composite Temporal Operators are described in terms of atomic temporal operators¹.

9.3.1 Base Temporal Operators

Base temporal operators are used to create base temporal expressions. Use of these operators was discussed in section 4.6.1. These operators are:

true, change, fall, rise, @ unary event operator

9.3.2 Atomic Temporal Operators

Atomic temporal operators are:

sequence, detach, fail, and, or, first-match-variable-repeat

The sequence operator was described in section 9.2.2. Detach operator was described in section 9.2.4. The remaining base temporal operators are described in the following subsections.

9.3.2.1 fail

Temporal expression **fail T** succeeds if failed termination condition (section 9.2.1) is raised at the end of the evaluation flow for temporal expression T. This operator does not change the evaluation flow for temporal expression T, and T evaluates according to its own evaluation flow. This operator does, however, modify the sense of success for this temporal expression. Temporal expression **fail T** consumes as many sampling event cycles as required for the evaluation of temporal expression T to terminate.

9.3.2.2 and

The syntax for the **and** operator is:

T: (T₁ or T₂ and or T_n)

The evaluation flow for temporal expression T defined using the **and** operator is described as follows:

- Number of Evaluation sub-threads: Each temporal sub-expression of the **and** operator is evaluated in a separate evaluation sub-thread. Each sub-thread is started when the eval-

¹. Even though composite operators are described in terms of atomic operators, the actual implementation of these operators is not necessarily based on using the atomic operators.

uation of the temporal expression T is started. For the above notation, evaluation flow for T will consist of n evaluation sub-threads.

- Success Equation: Temporal expression T succeeds when all its sub-threads start evaluating in the same sampling period and succeed in the same sampling period. The evaluation flow for an **and** operator can succeed only one time.
- Termination Condition: The evaluation of temporal expression T terminates when any of its evaluation sub-threads succeeds, or when all sub-threads terminate. The reason is clear: if any sub-threads continue to evaluate when at least one sub-thread terminates, then there is no possibility for all sub-threads to succeed at the same time and the evaluation of T should fail.

One important consequence of the evaluation of the **and** operator is that the and of two separate temporal expressions which take different number of cycles to evaluate will never succeed. The temporal expression for event D in the example below will never succeed even though definition for event C is the same as that of $\{ @A; @B \}$ in the temporal expression for event D. The reason is that the sub-thread evaluating $@C$ for the **and** operation on line 4 always evaluates in one cycle while a sub-thread evaluating $\{ @A; @B \}$ requires two cycles to complete.

```

1 : <
2 : struct event_holder {
3 :     event C is { @A; @B } @sys.any;
4 :     event D is { { @A; @B } and @C } @sys.any; -- event D is never emitted.
5 : };
6 : >
```

9.3.2.3 or

The syntax for the **or** operator is:

$$T: (T_1 \text{ or } T_2 \text{ and } \dots \text{ or } T_n)$$

The evaluation flow for temporal expression T defined using the **or** operator is described as follows:

- Number of Evaluation sub-threads: Each temporal sub-expression of the **or** operator is evaluated in a separate evaluation sub-thread. Each sub-thread is started when the evaluation of the temporal expression T is started. For the above notation, the evaluation flow for T will consist of n evaluation sub-threads.
- Success Equation: Temporal expression T succeeds when any of its evaluation sub-threads succeed. The evaluation flow for anor operation may potentially succeed as many times as the number of sub-expressions in the or expression.
- Termination Condition: The evaluation of temporal expression T terminates when all of its evaluation sub-threads terminate.

Figure 9.4 shows an evaluation of a temporal expression that is the OR of two temporal sub-expressions. The first sub-expression $\{ @req; @ack \} @pclk$, succeeds at the first **ack** occurrence (second **pclk** occurrence). The second sub-expression $\{ @re; [1]; @ack \} @pclk$, succeeds at

the second **ack** occurrence (third **pcclk**). When **req** occurs again at the fourth **pcclk** occurrence, a new evaluation of the sequence starts. This evaluation succeeds upon the occurrence of **ack** at the fifth **pcclk** cycle. Evaluation of the second sub-expression continues at the sixth **pcclk**, where it terminates. The evaluation started at cycle 10 fails because both its sub-threads fail without generating any successes.

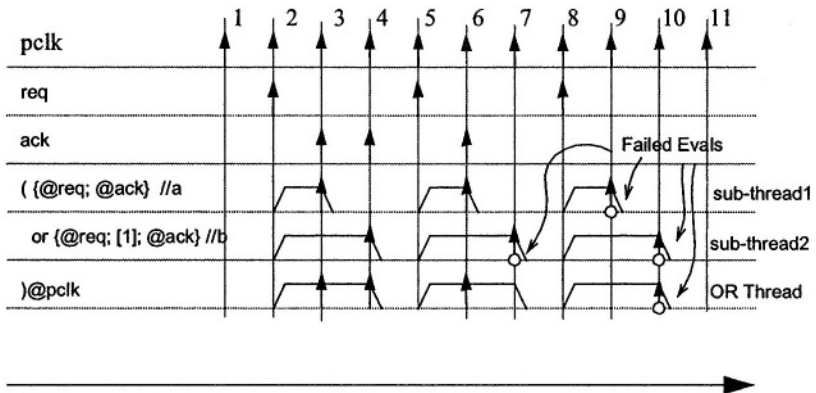


Figure 9.4 Evaluating OR Temporal Expressions

9.3.2.4 First Match Variable Repeat [from..to]

The syntax for the **first match variable repeat** operator is:

$$T: \{ \dots; [\text{min}.. \text{max}] * T_1; T_2; \dots \}$$

This operator can only be used within a sequence operator as shown above. Parameters **min** and **max** are unsigned integers and if missing, are assumed to have a value of zero and **MAXUINT** respectively.

The evaluation flow for temporal expression T is described as follows:

- Number of Evaluation sub-threads: Temporal expression T as shown above is computed $(\text{max} - \text{min} + 1)$ parallel sub-threads (section 9.3.3.2) each evaluating fixed repetition expression $\{[\text{count}] * T_1; T_2\}$ where each sub-thread **count** ranges from **min** to **max**. Note that if **min** is zero, then the part of the expression requiring a match for expression T1 immediately succeeds.
- Success Equation: Temporal expression T succeeds when any of its evaluation sub-threads succeed. The evaluation flow for the first match variable repeat operator may succeed only once.
- Termination Condition: The evaluation of temporal expression T terminates when the

first of its evaluation sub-threads succeed, or when all sub-threads terminate.

9.3.3 Composite Temporal Operators

Composite temporal operators are:

not, fixed-repetition, true-match-variable-repeat, yield, eventually

The definition of composite temporal operators in terms of atomic temporal operators is discussed in the following sections.

9.3.3.1 not

The syntax for the **not** operator is:

$$T: \text{not } (T_1)$$

The **not** operator is equivalent to the following temporal expression:

$$T: \text{fail } (\text{detach}(T_1))$$

This means that temporal expressions T and T_1 evaluate in separate threads that are started independently. The **not** operator always terminates in one cycle of the sampling event, regardless of how many cycles T_1 needs to complete its evaluation. This is evident since the **fail** operator in above equation operates on an implicit event which is generated by the detached thread and **fail** operation on an event always completes in one cycle.

9.3.3.2 Fixed Repetition

The syntax for the **fixed repetition** operator is:

$$T: [\text{count}] * T_1$$

The **fix repetition** operator is equivalent to the following temporal expression:

$$T: \{T_1; T_1; T_1; \dots; T_1\} \text{ (with count elements in the sequence)}$$

If the value of count is 0, then temporal expression T succeeds at all times in the same cycle of the sampling event.

9.3.3.3 True Match Variable Repeat

The syntax for the **true-match-variable-repeat** operator is:

$$T: \sim[\text{min}.. \text{max}] * T_1$$

The **true-match-variable-repeat** operator is equivalent to the following temporal expression:

$$T: [\text{min}] * T_1 \text{ or } [\text{min}+1] * T_1 \text{ or } \dots \text{ or } [\text{max}-1] * T_1 \text{ or } [\text{max}] * T_1$$

Parameters **min** and **max** are unsigned integers and if missing, are assumed to have a value of zero and MAXUINT respectively. The evaluation flow for true-match-variable-repeat operator may succeed as many as (max-min+1) times.

9.3.3.4 Yield

The syntax for the **yield** operator is:

$$T: T_1 \Rightarrow T_2$$

The **yield** operator is equivalent to the following temporal expression:

$$T: \text{fail } T_1 \text{ or } \{T_1; T_2\};$$

Even though the equivalent representation of the **yield** operator has two **or** sub-expressions, evaluation of a **yield** operator may only succeed once because of the use both failed and succeeded versions of T1 in the two **or** sub-expressions.

9.3.3.5 eventually

The syntax for the **eventually** operator is:

$$T: \text{eventually } T_1$$

The **eventually** operator is equivalent to the following temporal expression:

$$T: \{[..]*\text{not } @\text{quit}; T_1\}$$

The **quit** event is a predefined event of all structs and is emitted when the **quit()** method of a struct is called as the run phase of the program is completing. The **eventually** operator succeeds if temporal expression **T₁** occurs sometime before the **quit** event is emitted. This operator is usually used in conjunction with the **yield** operator in an **expect** struct member to monitor that an event B occurs before simulation ends after occurrence of some event A, (i.e. @A => eventually @B).

9.4 Temporal Operator Arithmetic

Often, temporal expressions that use a combination of different operators may seem difficult to understand. In such cases, a temporal expression can be reduced to a temporal expression consisting only of base temporal expressions and atomic temporal operators. Applying the evaluation model for the atomic temporal operators then provides a systematic approach for understanding the expected behavior of a complex temporal expression.

Consider the following event definition:


```

1 : <'
2 : struct event_holder {
3 :     event x is {
4 :         ~[2..3]*{@A;@B};
5 :         (@C => {@D;@F}) or (@C and not(@D;@F;@F));
6 :         @G
7 :     } @clk;
8 : };
9 : >'

```

The composite temporal operators in this event can be reduced to atomic operators to make the cycle by cycle evaluation of this event easier to understand:

```

1 : <'
2 : struct event_holder {
3 :     event P is {@D;@F;@F} @clk;
4 :     event X is {
5 :         {@A;@B;@A;@B} or {@A;@B;@A;@B;@A;@B};
6 :         fail(@C) or {@C;@D;@F} or (@C and fail(@P));
7 :         @G
8 :     } @clk;
9 : };
10 : >'

```

The temporal evaluation model for **sequence**, **and**, **or**, and **fail** operators can be used to evaluate the expected behavior or event x.

Many interesting reductions can be derived for temporal operators. For example, assuming the same sampling event, the following two properties hold for **and** and **or** temporal operators:

$$\{...\{T1 \text{ or } T2\};\{T3 \text{ or } T4\};...\} = \{...\{T1;T3\};...\} \text{ or } \{...\{T1;T4\};...\} \text{ or } \{...\{T2;T3\};...\} \text{ or } \{...\{T2;T4\};...\}$$

$$\text{fail}(@C) \text{ or } (@C \text{ and fail}(@P)) = \text{fail}(@C) \text{ or fail}(@P) \text{ since } @C \text{ and } @P \text{ evaluate in one cycle}$$

Using the above two equivalence relationships, definition for event x can be further enumerated to the following description:

```

1 : <'
2 : struct event_holder {
3 :     event P is {@D;@F;@F} @clk;
4 :     event X is {
5 :         {@A;@B;@A;@B;fail(@C);@G} or
6 :         {@A;@B;@A;@B;@C;@D;@F;@G} or
7 :         {@A;@B;@A;@B;fail(@P);@G} or
8 :         {@A;@B;@A;@B;@A;@B;fail(@C);@G} or
9 :         {@A;@B;@A;@B;@A;@B;@C;@D;@F;@G} or
10 :        {@A;@B;@A;@B;@A;@B;fail(@P);@G}
11 :    } @clk;
12 : };
13 : >'

```

9.5 Temporals Dictionary

The operators introduced in section 9.3 can be used to define complex temporal behaviors that are used to emit events, or to assert properties using the **check** struct member. The following two subsections provide translations between common temporal behaviors and their equivalent implementation.

9.5.1 English Phrases and event Definitions

One event

```
event clk is rise('top.clock')@sim;
```

Two or more events occur at same time

```
event e1 is (change('top.signal1') and rise('top.signal2')@clk;
event e2 is (fall('top.signal1') and change('top.signal2') and change('top.signal3'))@clk;
```

Event e1 occurs before event e2

```
event e3 is {@e1 ; @e2}@clk;
```

This description is equivalent to event e2 occurs after e1 occurs.

Event e2 occurs N cycle after e1 occurs

```
event e4 is {@e1; [N-1]; @e2}@clk;
```

If N=3 then e2 occurs 2 cycles after e1 occurs. This description is equivalent to event e1 occurs N cycle before e2 occurs.

Event e2 occurs within N cycles after e1 occurs

```
event e5 is {@e1; [...]; @e2}@clk;
```

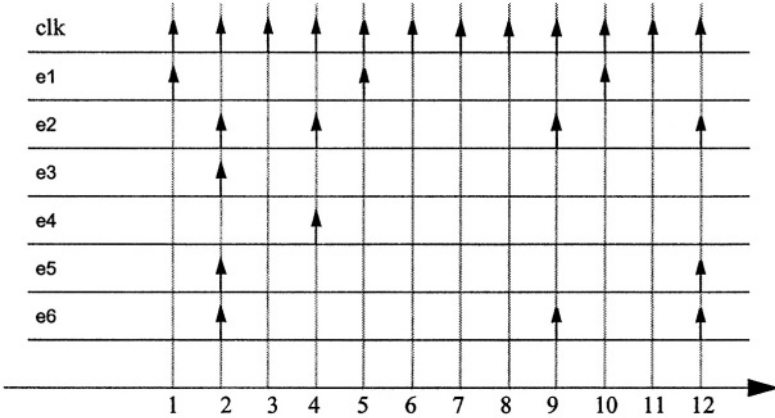
In above example, event e5 is emitted if event e2 occurs after 1, 2, or 3 cycles of sampling event after the occurrence of e1.

Event e2 occurs some time interval after e1 occurs

```
event e6 is {@e1; [...]; @e2}@clk;
```

The **exec** construct can be used to measure the time interval between e2 and e1.

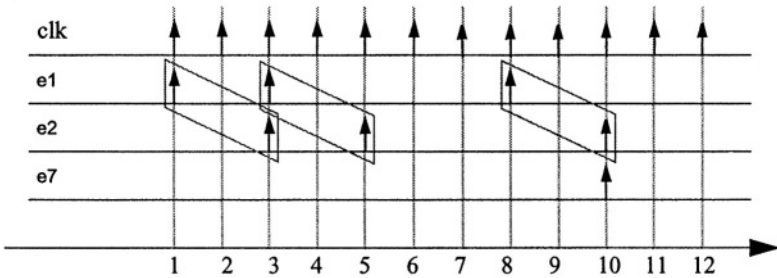
```
event e6 is {
  @e1 exec {e1_time = sys.time};
  [...];
  @e2 exec { out("e2_time is = ", sys.time - e1_time , "cycles after e1_time") };
} @clk;
```



Sequence (event e2 occurring 2 cycles after e1) repeated N times

event e7 is `{@assertion.start_of_test; [N]* {[.]; detach(@e1;[1];@e2;)}}@clk;`

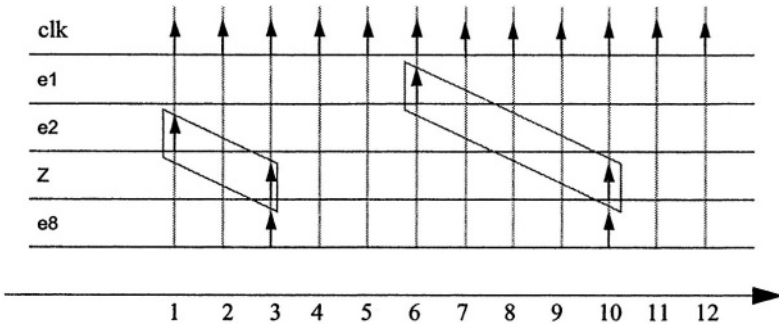
The detach operator creates an independent evaluation thread for temporal sub-expression `{@e1;[1];@e2;}`. If $N = 3$ then event e7 is emitted after the above temporal sequence is repeated 3 times.



Event Z occurs within N1 to N2 cycles after either e1 or e2 occurs

event e8 is `{{@e1 or @e2}; [N1-1..N2-1]; @Z}@clk;`

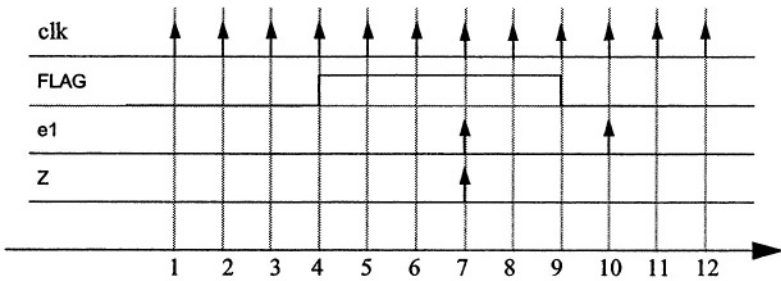
An example of a condition for triggering event e8 for $N1=2$ and $N2=4$ is shown below.



Event e1 occurs N cycles after a flag 'FLAG' becomes true

event e9 is $\{true(FLAG); [N-1]; @e1\}@clk;$

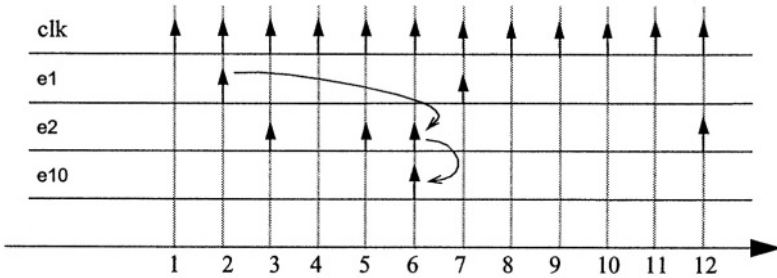
An example of a condition for triggering event e9 for N=3 is shown below.



Event e1 does not occur within N cycles before event e2

event e10 is $\{[N] * not @e1; e2\}@clk;$

An example of a condition for triggering event e10 for N=4 is shown below.

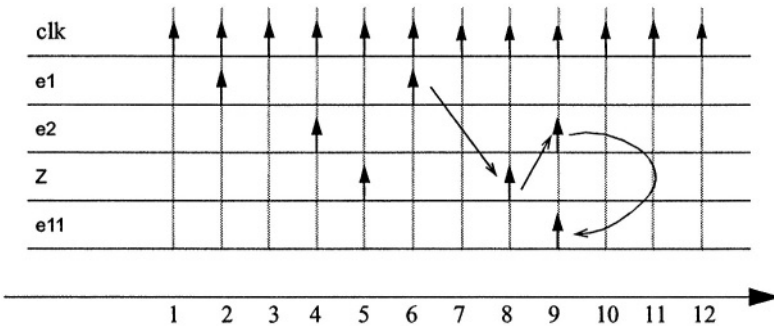


Event Z occurs after e1 and before event e2

```

event e11 is{
    @e1;                --e1 occurs
    [...] * not @e1;    --e1 does not occur for any number of cycles
    (@Z and not @e1);  -- Z occurs and e1 does not occur
    [...] * not @e1;    --e1 does not occur for any number of cycles
    @e2;                --e2 occurs.
};
    
```

An example of a condition for triggering event e11 is shown below.

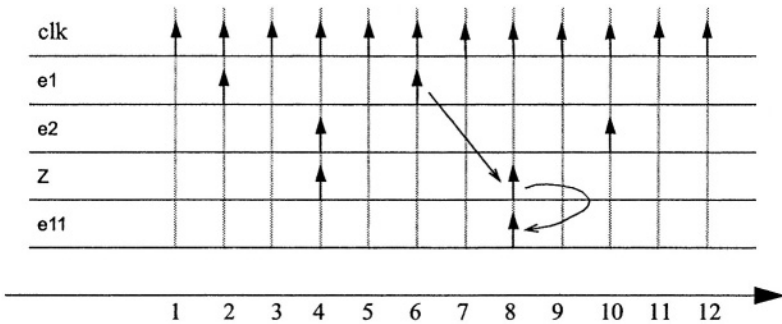


Event Z occurs after e1 occurs with no occurrence of e2 in between

```

event e12 is {
    @e1;                --e1 occurs
    [...] * not @e2;    --e2 does not occur for any number of cycles
    @Z;                --Z occurs.
} @clk;
    
```

An example of a condition for triggering event e2 is shown below.



9.5.2 English Phrases and Property Checking:

Section 9.5.1 showed translations between English phrases and temporal expressions used for defining events. Temporal expressions are used in the **check** struct member to perform property checking. This section explains the conversion of English phrases to temporal expressions used for property checking.

Event e2 should occur N cycle after event e1

If a check is to be done to determine that an event/expression occurred before or after another event/expression, it can be done as follows.

The following example shows the implementation of the check that an event e1 is expected to occur N cycles before another event e2. This check is equivalent to checking that event e2 is expected to occur N cycle after e1.

```

1 : <
2 : struct check_ex {
3 :     event e1 is rise('top.data_sent')@clk;
4 :     event e2 is rise('top.ack')@clk;
5 :
6 :     expect @e1 => {[N-1]; @e2} @clk
7 :     else dut_error("the event e2 did not occur one cycle after e1 happened");
8 : };
9 : >

```

Event e2 should occur within N1 to N2 cycles after e1 occurs

The implementation of this check for N1=3 and N2=5 is shown below.

```
expect @e1 => {[2..4]; @e2}@clk
    else dut_error ("the ack did not happen within 3-4 cycles after data was sent");
```

Event e2 should eventually happen after e1

```
expect @e1 => {eventually @e2}@clk
    else dut_error ("the ack did not happen ever, eventhough data was sent");
```

Event e2 must occur N cycles after e1, with no repetition of e2 before e1

```
expect @e1 => {[N-1]* not @e1 ; @e2}@clk
    else dut_error ("the ack happened before the next data could be sent");
```

Event e2 should not occur N cycles after e1 has occurred

```
expect @e1 => {[N] * not @ev_2} @clk
    else dut_error ("the ack occurred before 2 cycles after which data was sent") ;
```

Event e1 should have occurred N cycles before e2

```
expect @e2 => detach ({@e1 ; [N+1]})@clk
    else dut_error ("the event e1 did not happen 2 cycles before e2 happened.");
```

Event e1 must have happened sometime within N cycles before e2

```
expect @e2 => detach ({@e1; ~[N+1..]})@clk
    else dut_error ("the ack happened before the data could be sent");
```

There must be no more than N occurrences of Z after e1 and before e2

```
expect (@e1 => {~[.N-1]*{[.];@Z}; [.] * not @Z; @e2})@clk
    else dut_error ("the data_write_done happened after the ack was sent");
```

Event e1 must not occur more than N times throughout the test

```
expect @session.start_of_test=> [N+1] * {[.]; @e1}@clk
    else dut_error ("the test occurred less or more than 16 times.");
```

9.6 Performance Issues

Evaluation of temporal expressions is performed by evaluation threads that are started at every new sampling event of a temporal expression. As such, indiscriminate use of temporal expressions may degrade simulation performance. This section discusses performance issues that can happen with temporals and tips to overcome such issues and to enhance performance.

9.6.1 Over-sampling

Temporal expressions are created using base temporal expressions. Base temporal expressions are defined based on the value or change in value of signals in the DUV or *e* data values. Over-sampling refers to sampling a data value or a DUV signal more number of times than required while evaluating a base temporal expression. If a signal used in a temporal expression does not change very frequently then sampling it at each clock cycle is considered over-sampling.

Consider the following example:

```

1  :  <'
2  :  extend agent {
3  :      event clk is rise ('top.clock')@sim;
4  :      event rst rise('top.reset')@clk;
5  :
6  :      drive() @clk is {
7  :          wait rise('top.start_test');
8  :      };
9  :  };
10 :  >'

```

Struct **agent** has 2 events declared. If signal **top.reset** changes only once or twice during the simulation, then the event **rst** should not be sampled on **@clk**, as **@clk** event occurs very frequently. In this case event **rst** should use event **@sim** instead of **@clk** for its sampling event. This change reduces the number of times temporal expression for event **rst** is evaluated. By using **@sim** as the sampling event, evaluation occurs only when the signal **top.reset** changes in simulator. If **@clk** is used, then the temporal expression for **rst** is evaluated every time event **clk** occurs.

Also in the **drive()** TCM, the temporal expression used in the **wait** action has sampling event **@clk** by default as it is inside a TCM. Signal **top.start_test** occurs only in the beginning, thus sampling it on every **@clk** event also leads to over-sampling. Thus, explicitly specifying **@sim** as the sampling event for this temporal expression eliminates over-sampling. Note that by changing the default sampling event to **@sim**, the wait action computes when rise of signal **top.start_test** is detected and not necessarily when event **@clk** is emitted. Therefore it is necessary to use the sync action to synchronize the execution flow with the **@clk** event.

```

1  :  <'
2  :  extend agent {
3  :      event clk is rise ('top.clock')@sim;
4  :      event rst rise('top.reset')@sim;
5  :
6  :      drive() @clk if{
7  :          wait rise('top.start_test') @sim;
8  :          sync @clk;
9  :          -- body of driver here.
10 :      };

```



```

11 : };
12 :

```

9.6.2 Missing Sampling Event

Predefined event **sys.any** is the default sampling event for all temporal expressions that do not have a sampling event specified explicitly or derived from context. This event is the most frequently occurring event in the simulation runtime (occurs whenever any event occurs) and can result in over-sampling of the temporal expression.

```

1 : extend agent {
2 :     event clk is rise('top.clock')@sim;
3 :     event e1 is rise('top.signal1');
4 :     event e2 is fall('top.signal2')@sim;
5 :     event e3 is change ('top.signal3')@sim;
6 :     event evx is { @e2 ; @e3 };
7 : };

```

In the above example, All the clocks in DUV are synchronized to **top.clock**. Event **e1** is defined to occur upon rise of **top.signal1**, but there is no sampling event. Hence **@sys.any** is used as the sampling event. Event **evx** is the event which occurs when **e3** happens in the cycle following **e2**. Since no sampling event is specified, **sys.any** is used as the default sampling event. Thus, the sampling of the DUV signal happens more frequently than required even though sampling at event **@clk** is sufficient to achieve the desired behavior

The solution for this issue is to specify a proper sampling event for all temporal expressions. The above example is corrected as follows:

```

1 : extend agent {
2 :     event clk is rise('top.clock')@sim;
3 :     event e1 is rise('top.signal1')@clk;
4 :     event e2 is fall('top.signal2')@clk;
5 :     event e3 is change ('top.signal3')@clk;
6 :     event evx is { @e2 ; @e3 } @clk;
7 : };

```

9.6.3 Unanchored Sequences

Another reason performance hit can occur when in the coding, any sequence is defined which starts with a frequent event.

An anchor temporal expression is defined as a temporal expression that when satisfied, starts the evaluation of a temporal sequence. An anchored temporal sequence is a sequence that has an anchor temporal expression. Unanchored sequences are prone to over-sampling. This is shown in this example.

```
event ev_y is { [2000]* true('top.rst' == 1'b0); [..]; @e1 } @clk;
```

In this example, a new evaluation for event `ev_y` is started at every clock cycle searching for the first 2000 times `top.rst` is inactive at the `clk` sampling event. The resources required for starting all these evaluations may in fact exceed the limits of the `e` runtime environment.

The real intent in defining this example however is to detect event `e1` sometime after 2000 cycles have passed since reset became inactive. By adding an anchor expression to the temporal expression for event `ev_y`, the number of started evaluation can be reduced to one. This enhancement is shown below:

```
event ev_y is (fall('top.rst') ; [1999]* true('top.rst' == 1'b0) ; [..] ; @e1 ) @clk;
```

9.6.4 Nested Sampling

Nested sampling refers to using different sampling events for evaluating the same temporal expression. Nested sampling can be specified explicitly by over-riding the default sampling event or overriding the default sampling events in a TCM. Use of nested sampling reduces the potential for evaluation optimizations performed by the program runtime environment. Therefore, it is best to minimize use of nested sampling as much as possible. Consider the following example:

```

1 : <
2 :   extend agent {
3 :     event clk_1 is rise('top.clk1')@sim;
4 :     drive1 ()@clk is {
5 :       wait (@evz ; @evu)@clk_1;
6 :       -- body of drive1
7 :     };
8 :   };
9 : >
```

In the above example, nested sampling is used for evaluation of condition on line 5. The default sampling event for the method is `@clk` and this default is overridden in the `wait` action to event `@clk_1`. Enhancement of this program shown below removes the nested sampling and improves performance.

```

1 : <
2 :   extend agent {
3 :     event clk_1 is rise('top.clk1')@sim;
4 :     event ev_zu is (@evz;@evu)@clk_1;
5 :     drive1() @clk is {
6 :       wait @clk_zu;
7 :       --body of drive1
8 :     };
9 :   };
10 : >
```

In this enhancement, event `@ev_zu` is emitted using its own independent evaluation thread. The wait action on line 6 samples event`@clk_zu` using sampling event`@clk`.

9.7 Summary

This chapter introduced temporal expressions and temporal operators. Temporal operators are divided into base temporal operators, atomic temporal operators, and composite temporal operators. Base operators are used to translate values and transitions in signal values into temporal expressions. Atomic temporal operators provide unique temporal evaluations. An abstract evaluation model for a temporal operator was presented and evaluation of atomic temporal operators were described using this abstract model. Composite temporal operators were described in terms of atomic operators.

This chapter also presented a temporal dictionary showing implementation of common event definitions and check phrases. Issues affecting performance of temporal expressions evaluations were discussed.

The messaging feature in *e* provides a standard and uniform mechanism for printing informative text to screen or to log files. Typical uses for messaging include:

- Program Tracing Information
- Debugging Aid
- Simulation Run Summaries

Program tracing prints messages at the occurrence of a specific event. This is useful for printing messages that a user requires upon a specific event. Debugging prints detailed messages during the simulation run to help understand unexplained behavior. Simulation run summaries print information at the beginning or end of some specific activity.

The messaging utility in *e* allows the user to insert and display formatted as well as colored messages to the screen. The messages can be enabled or disabled as needed by the user.

Messaging is different from **out()** and **outf()** actions since it allows the user to disable or enable messages and control messaging behavior by routing messages through configurable message handler constructs (loggers). Messaging in *e* is also different from the **dut_error()** usage since messages do not necessarily indicate error conditions and do not increment error counts as is done with the **dut_error()** action.

Details on messaging are described in this chapter. Section 10.1 describes the messaging strategy and its advantages. Section 10.2 describes the message action syntax. Message loggers are described in section 10.3.

10.1 Messaging Strategy

The messaging strategy provided in *e* is based on two constructs: message actions, and message loggers. Message actions are used to initiate a messaging activity and to provide the text printed on the destination for that message (i.e. file, terminal). Message loggers control the format and destination of messaging activity initiated by a message action.

Message loggers include the following properties:

- One or more tags
- Destination
- Messaging format
- Verbosity
- Unit hierarchy depth

Tags are used to categorize messages into user-defined groups. Message destinations can be a file, or the display terminal. The messaging format specifies the amount of data printed for a message (long, short, etc.). Verbosity is defined by an enumerated list specifying ordered list of verbosity levels (i.e. HIGH, MEDIUM, LOW). The verbosity setting for a message logger is used to filter messages based on the verbosity setting(s) (e.g. a logger verbosity setting of MEDIUM indicates that all messages handled by this logger should be printed only if their verbosity is set to MEDIUM or lower). Logger depth gives the number of levels in the **sys** unit hierarchy that separates a logger instance from **sys**. As such, a logger instantiated in **sys** has a depth of 0.

A message logger is identified by its tag, destination, and depth. This identification mechanism is used to decide which message loggers should handle a given message action. Once a message logger is identified as one of the handlers for a message action, its format, and verbosity settings are used to control the message output through that logger.

A message action consists of a tag specification, a verbosity setting, and a text message that contains the message. Every message action is aware of the unit in which it is initiated. Method **get_unit()** is a predefined method of all structs used to identify the unit containing a message action.

The message handling mechanism consists of two steps: Identifying all loggers that should handle a message, and then deciding how each of those loggers processes a message action.

Identifying handling loggers for a message action consists of the following steps:

- Identify the hierarchical path from **sys** to the unit containing the message action.
- Create a list of all loggers instantiated in units on this hierarchical path.
- For message loggers on this list that have the same tag and destination, keep only the message logger whose depth is closest to the unit containing the message action.

Handling the message action consists of the following steps:

- For each logger identified as a handler, print the message action if the verbosity level

for the action is the same or lower than the verbosity setting for that logger.

- If verbosity level satisfies the verbosity requirement for a logger, then format the message according to the logger format requirements and send to the logger destination.

The messaging utility in *e* allows filters to be defined as message and logger properties. The ability to define tags, destinations, formats, and filters provides a utility for creating a powerful reporting mechanism for a verification environment.

Details of message loggers and message actions are described in the following sections.

10.2 Message Actions

A message action is initiated using the **message** action. The syntax for this construct is:

```
message([tag], verbosity, exp, ...) [action-block]
messagef([tag], verbosity, format, exp, ...) [action-block]
```

messagef allows the user to format the output string by specifying a format string as part of the syntax. Action block is used to provide additional text producing code, and the text produced by this action block is processed along with text provided with the message parameters.

Examples of calling the message action in procedural code are shown below.

```

1 : <'
2 :   unit uart_tx {
3 :     //uart_evc: uart;
4 :     data[8]: list of bit;
5 :     sout_tmp[11]: list of bit;
6 :     event clk is rise("top.clk")@sim;
7 :     event reset_start is rise("top.reset")@sim;
8 :     event reset_end is fall("top.reset")@sim;
9 :     event tx_ready is rise("top.txready")@sim;
10 :     data_drv()@clk is {
11 :       wait @reset_start;
12 :       message(LOW, "uart_module", me, "is reset") {
13 :         -- beginning of message action block
14 :         var uart_id: uint;
15 :         --compute uart_id so it can be printed below.
16 :         print uart_id;
17 :       };
18 :       wait[1000]*cycle;
19 :       wait @reset_end;
20 :       message(LOW, "uart_module", me, "reset is removed");
21 :       while TRUE {
22 :         var start : bit = 1'b0;
23 :         var stop : bit = 1'b1;
24 :         //gen data;
25 :         var num_of_frame: uint = 100;
26 :         var parity : bit;
27 :
28 :         wait @tx_ready;
29 :         message(HIGH, "uart module", me, "transmit data is", sout_tmp);

```

```
30 :           for j from 0 to num_of_frame {
31 :               gen data;
32 :               for i from 0 to 7 {
33 :                   parity= bitwise_xor(data[i]);
34 :               };
35 :               sout_tmp = {start; data; parity; stop};
36 :               message(HIGH, "uart module", me, "transmit data is", sout_tmp);
37 :               for i from 0 to 10 {
38 :                   'top.sout'= sout_tmp[i];
39 :               };
40 :               sout_tmp.clear();
41 :           }; // for j
42 :
43 :       }; // for while
44 :   }; // data_drv
45 : }; // uart tx
46 : >
```

10.2.1 Message Tags

Message tags are used to link a messaging action with a specific message logger. It is a pre-defined enumerated type named **message_tag** and is defined as:

```
type message_tag : [NORMAL];
```

The default tag for a message is **NORMAL**.

Type **message_tag** can be extended to create new tag values needed for messaging control:

```
extend message_tag: [UART_XMT];
```

If the tag is not provided with a message action, then the default tag is used. **NORMAL** is the default message **TAG** for all messages that do not provide a message tag.

```
message(HIGH, "uart module ", me, " transmit data is", sout_tmp);
```

is equivalent to:

```
message(NORMAL,HIGH, "uart module ", me, " transmit data is", sout_tmp);
```

The following is an example of a specific message tag in the message action:

```
message(UART_XMT_DATA, MEDIUM, "uart module ",uart_ev, "transmit data is", sout_tmp);
```

This message action is handled by loggers tagged with a **UART_XMT_DATA** tag value. Tags can be used to enable a message action during specific debugging modes only. Thus:

```
message(DEBUG, MEDIUM, "uart module ",me, "transmit data is", sout_tmp);
```

In this approach, a message logger can be created to handle only messages that are used for debugging purposes. Such a logger would be enabled only during debugging mode.

10.2.2 Verbosity

This parameter provides different levels of verbosity for filtering messages. Verbosity is identified by the enumerated type `message_verbosity`. Possible values for this parameter are:

- NONE: means messages cannot be disabled.
- LOW: used for messages that are displayed only once in a test run.
- MEDIUM: used for messages displayed once per transaction.
- HIGH: used for messages displayed with some details.
- FULL: used for messages displayed with a lot more details.

10.2.3 Format Type

The message output format is chosen from **Short**, **Long**, and **None**. This setting is configured using commands or constraints as shown later in this chapter.

Short format is the default output format. The syntax for this format is:

```
[time] short-name-path: message
```

The syntax for **Long** format is:

```
[time]short-name-path (verbosity) source in struct-instance:
message
```

For format **None**, the text is printed as it is provided in the messaging action.

Given string “Reset Occurred” as the text for a messaging action, the following messages will be printed for formats None, Short, and Long respectively:

```
Reset Occurred
[12030] AHB_0 M1: Reset Occurred
[12300] AHB_0 M1 (HIGH) at line 12 in @vr_ahb_send in vr_ahb_bfm-@77: Reset Occurred
```

10.2.4 Action Block

The action block is used for any procedural block that is to be executed when printing a message (i.e. computing values to be printed as part of message action text). An example of using an action block is shown below:

```

1 : <
2 :   extend xyz {
3 :     the_packet : packet;
4 :     m()@clk {
5 :       message(HIGH, "Master ", me, " received packet:"){
6 :         print the_packet;
7 :       };
8 :     };
9 :   };
10 : >
```


Message actions cannot be used in any program execution flow initiated inside the action block for another message action. This means a message action cannot be used in an action block, and moreover, no method containing a message action can be called either directly or recursively inside an action block.

Execution of an action block is dependent on the setting for the message action and its handling message loggers, so it is not guaranteed that the procedural code in an action block is always executed. Therefore, no part of the program essential for correct operation of the verification environment should be placed in an action block.

10.3 Message Loggers

Message loggers manage outputs from message actions. Messaging actions create the output message and send it to message loggers for further formatting. These loggers further filter the message text using the verbosity setting, reformat the text according to the format settings, and then direct these messages to their destination set for the logger. The predefined unit **message_logger** contains the implementation for a logger.

A message logger is instantiated inside a unit, becoming part of unit hierarchy.

```
1  : <
2  :   extend protocol_checker {
3  :       log_out : message_logger is instance;
4  :   };
5  : >
```

sys.logger is a default message logger that is by default instantiated under **sys**.

The functions handled by message loggers are:

10.3.1 sys.logger

sys.logger is the default logger in *e*. It has Low verbosity, NORMAL tag, and sends messages to the screen. These settings can be changed using constraints and Specman commands.

In the example shown below, three message actions are shown. Since no tag is specified in any of these messages, the default NORMAL tag is used. And since no message loggers are instantiated, **sys.logger** is the only logger present in the environment. The default tag accepted by **sys.logger** is NORMAL with default LOW verbosity. All three message actions below are handled by **sys.logger**; however only the first message action is printed and the last two are filtered because of their verbosity setting.

```
1  : <
2  :   extend sys {
```

```

3 :         run() is also {
4 :             message(LOW, "Reset is removed");
5 :             message(MEDIUM, "the num of transactions", num_transaction);
6 :             message(HIGH, "the transactions are done");
7 :         };
8 :     };
9 : >

```

10.3.2 Message Handling Using Loggers

Consider the example shown below:

```

1 : <
2 : extend sys {
3 :     aud_env : env is instance;
4 : };
5 : extend env {
6 :     i2s_agent : agent is instance;
7 :     logger : screen_log is instance;
8 :     filelogger : file_log is instance;
9 : };
10 : extend agent {
11 :     data_gen : data_generator is instance;
12 :     logger : agent_screen_log is instance;
13 :     filelogger : agent_file_log is instance;
14 : };
15 : extend data_generator {
16 :     run() is also {
17 :         message(I2_S, LOW, "Reset Detected");
18 :     };
19 : };
20 : >

```

The unit containing the message action on line 17 is **sys.aud_env.i2s_agent.data_gen**. Calling **get_unit()** a predefined method will return **sys.aud_env.i2s_agent.data_gen**. Message loggers on the hierarchy path from this unit to **sys** are:

```

sys.logger (NORMAL, screen, 0)
sys.aud_env.logger (NORMAL, screen, 2)
sys.aud_env.filelogger (NORMAL, screen, 1)
sys.aud_env.i2s_agent.logger (I2_S, screen, 1)
sys.aud_env.i2s_agent.filelogger (I2_S, screen, 2)

```

The tag, destination, and depth for each logger is shown in parenthesis. All the loggers that ignore the message tag are removed from this list. The message loggers that will handle the message action above are:

```

sys.aud_env.i2s_agent.logger (I2_S, screen, 1)
sys.aud_env.i2s_agent.filelogger (I2_S, screen, 2)

```

The predefined method **accept_message()** of each of above message logger is used to decide whether or not the message action is accepted by that logger. If this method returns

FALSE, then the message is filtered. This method will return TRUE by default but can be extended to change the default behavior.

If a message is accepted by a logger, then the predefined method **format_message()** is called to format the message text and send it to the logger destination. The default behavior for this method is based on the format setting of the logger (None, Long, Short) and may be extended by the user to change the desired format.

10.3.3 Configuring Loggers

The default configuration for a user defined message logger is:

- Verbosity NONE (ignores all input)
- Destination is to the screen
- No write to any file
- Empty tag list {}

The default setting for **sys.logger** is:

- Verbosity LOW
- Destination is to the screen
- No write to any file
- Tag NORMAL

There are three approaches to configure loggers:

- Commands (any time during the run)
- Methods of the logger
- Constraints (the most common way)

10.3.3.1 Using Commands

The commands are specifically for a tool that interprets *e* code. The commands shown in this section refer to Specman Elite.

All the commands refer to **sys.logger** unless the logger name is specified using **-logger** option, to identify a particular logger or all the loggers. The **-logger=all** option refers to all the loggers.

Some of the commands to configure message loggers are shown below.

```
set message [-logger=explall] ..... Specifies which message logger is being
                                     addressed by the command. If the -logger option is
                                     not given then the command refers to sys.logger
set message [-logger=explall] units=exp[onloff]
..... Specifies which units are being watched by the log-
                                     ger
set message [-logger=explall] screen[onloff]
..... Turns the writing to screen by the logger on and off.
```

- show message.....Shows a short summary of all the loggers which includes a list of tags, high verbosity and number of message actions.
- show message –units[=exp]..... Show all units, each with its associated loggers.

10.3.3.2 Using Methods

The next way to configure message loggers is to use predefined methods of message loggers. These of the predefined methods are shown below.

- set_units(root: any_unit, to: message_on_off)
.....This method is used to set the unit tree under root to be either on or off for the message logger.
- set_screen(to:message_on_off) .Turns the writing to screen by the logger on and off.
- show_message(all :bool, full:bool)
.....Shows a summary of all the loggers with full verbosity.
- show_units(root: any_unit).....Shows all the units with their associated loggers.

10.3.3.3 Using Constraints

Constraints are used during pre-run generation to set the fields of the logger. These fields are used during **post_generate()** to configure the logger.

In the example below, the use of constraints for configuring a message logger is shown. **message_logger** is a predefined unit, hence it is extended.

```

1  | <'
2  | : extend message_logger {
3  | :   // The message tags for selecting the actions for this logger
4  | :   m_tags: list of message_tag;
5  | :   keep soft m_tags == {};
6  | :
7  | :   // The verbosity for selecting the actions for the logger
8  | :   m_verbosity: message_verbosity;
9  | :   keep soft m_verbosity == NONE;
10 | :
11 | :   // The modules wildcard for selecting the actions for the logger
12 | :   m_modules: string;
13 | :   keep soft m_modules == "";
14 | :
15 | :   // The pattern to match against the string in the message action
16 | :   m_string_pattern: string;
17 | :   keep soft m_string_pattern == "...";
18 | :
19 | :   // File name the logger should write to (or none if "")
20 | :   // Default extension for the file name is ".elog".
21 | :   m_to_file: string;
22 | :   keep soft m_to_file == "";
23 | :
24 | :   // True if we want the message_logger to write to screen
25 | :   m_to_screen: bool;
26 | :   keep soft m_to_screen == TRUE;

```

```
27 : };  
28 : >
```

Note that only one file can be associated with a logger via constraints. Using the commands and method calls, additional files can be associated with a logger.

The fields shown above can be constrained for each instance of a logger by providing constraints for an instance in its containing unit. This is shown in the following example:

```
1 : <  
2 :   unit env {  
3 :     filelogger : file_log is instance;  
4 :     keep filelogger.m_verbosity == LOW;  
5 :     keep filelogger.m_to_screen == TRUE;  
6 :   };  
7 : >
```

10.4 Summary

This chapter introduced the concept of messaging used in *e*. Messaging is based on message loggers and message actions. Message loggers are instantiated as part of the environment struct hierarchy and message loggers handle message actions depending on their tags and destination settings. Loggers also provide filtering and formatting utilities that can be customized for each message logger.

The messaging mechanism provided in *e* facilitates the development of a uniform and customizable reporting strategy for a verification environment composed of components developed independently. Therefore, taking advantage of the messaging features described in this chapter, allows for easier development, customization, and integration of a verification component.

A Monitor module tracks the activity in DUV signals to perform protocol checking, and also to collect data and report status. A monitor is a passive component of the verification environment and it does not drive any DUV signals.

A monitor collects output from a DUV, prints output messages, performs protocol checking, and facilitates coverage collection. Extracted events and status information collected by a monitor is used by other components in the verification environment (i.e. Scoreboard). It is important that a monitor implementation be independent of the information collected by active blocks in a verification environment (i.e. BFM). This independence allows a monitor to be used when active components are removed during system level verification.

Issues related to design and implementation of monitors are discussed in this chapter. Section 11.1 discusses considerations for designing and architecting a monitor. Section 11.2 presents approaches for implementing the protocol checking functionality of a monitor. Section 11.3 presents approaches for collecting and status reporting use of a monitor.

11.1 Monitor Architecture

A monitor is an important verification module in implementing a coverage driven verification methodology. A monitor facilitates automatic protocol checking, property checking, and collects the data necessary for scoreboarding and coverage measurement. A monitor should be:

- A Passive Module
 - Independent of BFMs and Drivers
-

- Relocatable

It should be possible to add or remove monitors to a verification environment without affecting the stimulus generation and driving activity in the environment. This allows the same monitor to be used in both module and system level verification environments. A monitor has to be a passive module¹ so that it can be added or removed from the environment without affecting the stimulus and sequence generation activity. The operation of a monitor must also be independent of BFM and drivers so that when these verification modules are removed during system level verification, the monitor can be used without any modification. Figure 11.1 shows how a monitor is reused in the system level verification environment after the stimulus generator and the verification BFM are replaced by a DUV submodule.

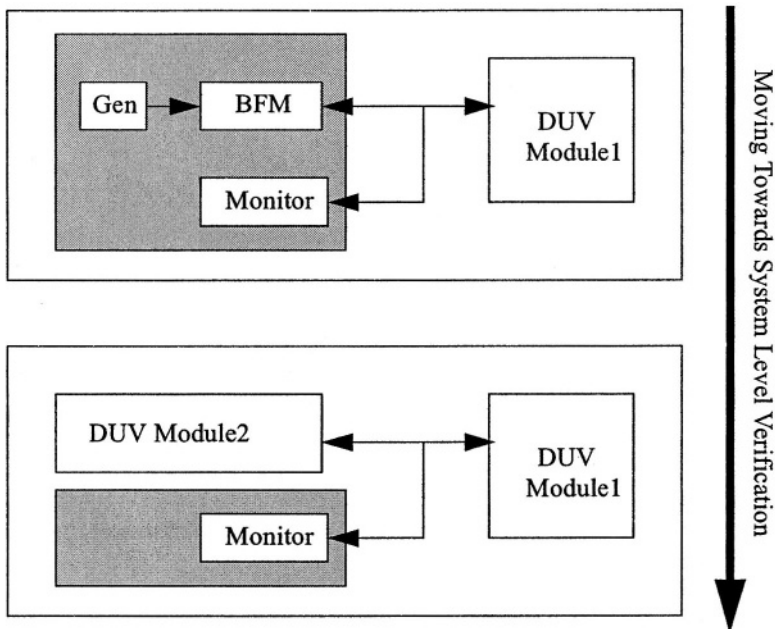


Figure 11.1 Migrating Monitors to System Level Verification

A monitor may need to be attached to different DUV ports of the same type. Therefore the monitor implementation has to be relocatable (see section 5.4) so that multiple instances of the same monitor can be attached to different points in the HDL hierarchy. Monitors must therefore be implemented using the *e* **unit** construct. The following *e* code fragment shows the implementation of a monitor for a UART module:

¹ A passive module never drives any DUV signals.

```

1 : <
2 : unit si_uart_monitor {
3 :     rcv_monitor_port : in simple_port of bit is instance;
4 :     rcv_frame : si_uart_frame;
5 :
6 :     xmt_monitor_port : in simple_port of bit is instance;
7 :     xmt_frame : si_uart_frame;
8 :
9 :     event frame_transfer_started;
10 :    event frame_transfer_ended;
11 :    event frame_receive_started;
12 :    event frame_receive_ended;
13 :
14 :    break_count : uint;
15 :    keep soft break_count == 0;
16 :    timeout_count : uint;
17 :    keep soft timeout_count == 0;
18 : };
19 :
20 : unit si_uart_agent {
21 :     monitor : si_uart_monitor is instance;
22 :     keep monitor.agent == me;
23 : };
24 : struct si_uart_frame {
25 :     has_parity_error : bool;
26 :     has_framing_error : bool;
27 :     size : uint ; //size of data bits in the transmitted frame.
28 :     Keep size in [5..8];
29 : };
30 : >

```

The monitor is modeled using a **unit** and is instantiated inside a UART agent. The monitor has two input ports of type **bit** that are used to monitor the receive and transmit ports of a UART module. The structure of a UART packet is also shown in above example.

The fundamental function of a monitor is to track and look at DUV signals to check for correct protocol and other device properties that require inspection during simulation runtime. A monitor's implementation is enhanced with additional features to facilitate other activities supported by a monitor (e.g. collection and reporting). Protocol checking is discussed in section 11.2. Collection and reporting functions of a monitor are discussed in section 11.3.

11.2 Protocol Checking

Protocol checking is implemented using a combination of declarative and procedural statements. Declarations of temporal struct members (**on**, **expect**, **assume**) are used to check for design properties. At the same time, TCMs containing **wait** and **sync** actions are used to check for complex protocols across many cycles.

Protocol checking is an important function provided by a monitor. At times however, running protocol checking is not required during the simulation flow. The conditions for whether to run a checker or not depend on DUV conditions during the simulation or on verification requirements of a simulation run. Protocol checkers lead to runtime overhead, and therefore it may be desirable to deactivate a checker depending on the DUV state. At the same time, if verification is focused on specific modules of a system, then it may not be necessary to include a protocol checker in the simulation run.

To support the requirement to selectively enable a protocol checker, create a protocol checker as an independent unit instantiated under a monitor unit only when a determinant field in the monitor indicates the presence of a protocol checker. This approach is shown in the following *e* code fragment:

```

1 : <
2 : unit si_uart_checker {
3 :     monitor: si_uart_monitor; -- pointer to monitor
4 :     -- add checker expect members
5 :     protocol_check()@rclk is also {
6 :         --implement checker method
7 :     };
8 :     run() is also {
9 :         start protocol_check();
10 :    };
11 : };
12 : extend si_uart_monitor {
13 :     has_checker : bool;
14 :     .....
15 :     // checker is instantiated only when has_checker is TRUE, else no checker.
16 :     when TRUE'has_checker si_uart_monitor {
17 :         checker : si_uart_checker is instance;
18 :         keep checker.monitor == me;
19 :     };
20 : };
21 : >

```

In the above, determinant struct member **has_checker** is used to instantiate a checker under the monitor unit.

11.2.1 Protocol Checks

A protocol checker is implemented by building temporal expressions based on events generated inside monitors. These monitor events are extracted from the DUV and allow for the protocol checker to be implemented without specific knowledge as to how an event is emitted in the monitor. These temporal expressions corresponding to DUV properties are then used with temporal struct members or in TCMs to check for any protocol errors. These topics are discussed in the subsequent subsections.

11.2.1.1 Monitor Events

Monitor events refer to the events extracted by a monitor. These events are used by the checker to check for any protocol errors. The most common event is the clock on which the signals are sampled and signal timings are checked. Other events refer to significant changes in the DUV control signals, which define transactions and protocols. For instance, the occurrence of an interrupt in a system or start of a transaction refer to specific events.

The following example shows extracted events in a monitor that facilitate the checker implementation.

```

1 : <'
2 :   extend si_uart_monitor {
3 :     event clk is @agent.rclk;
4 :     event rcvport_rise is rise(agent.bfm.rcv_port$) @clk;
5 :     event rcvport_fall is fall(agent.bfm.rcv_port$) @clk;
6 :     event frame_transfer_started is true(agent.bfm.start) @rcvport_rise;
7 :     event frame_transfer_ended is true(agent.bfm.end) @rcvport_fall;
8 :   };
9 : >'

```

11.2.1.2 Temporal Struct Members

Temporal struct members are used in declarative statements to perform protocol checking based on events and data values extracted by the monitor.

Use of on and expect temporal struct members are shown in the following examples:

```

1 : <'
2 :   extend si_uart_checker {
3 :     event frame_transfer_started is @monitor.frame_transfer_started;
4 :     event frame_transfer_ended is @monitor.frame_transfer_ended;
5 :
6 :     on frame_receive_ended {
7 :       num_of_transfer += 1;
8 :       if(monitor.rcv_frame.has_parity_error) {
9 :         message(LOW, "frame has parity error");
10 :       };
11 :     };
12 : };
13 : >'

```

```

1 : <'
2 :   extendsi_uart_checker{
3 :     event rcvport_rise is @monitor.rcvport_rise;
4 :     event rcvport_fall is @monitor.rcvport_rise;
5 :
6 :     expect @rcvport_rise => [31] * not @rcvport_fall
7 :       else dut_error("error in transmission:Data should change every 16 clock cycles");
8 :
9 :     expect @rcvport_fall => [31] * not @rcvport_rise
10 :      else dut_error("error in transmission:Data should change every 16 clock cycles");

```

```
11 : };  
12 : >
```

11.2.1.3 Protocol Checking Reports

Reports may need to be printed based on the success or failure of protocol checking activity. Reports can be printed using the **print** action or **out()** method call. Reports can also be printed by using the **check** and **dut_error** constructs.

```
1 : <  
2 : extend si_uart_checker {  
3 :     check() @clk is {  
4 :         gen has_parity_error;  
5 :         if (parity_type != NO_PARITY) {  
6 :             paradd = 1  
7 :         } else {  
8 :             paradd = 0;  
9 :         };  
10 :         check that !size() == (size+paradd) else  
11 :             dut_error("Size of collected data bits do not match the expected value);  
12 :         return from + size + paradd;  
13 :     };  
14 : };  
15 : >
```

The use of the message construct and message loggers allows the user to control the verbosity and format of the printed messages.

11.2.2 Protocol Checker Activation

Checkers added to an *e* verification environment increase simulation overhead. It is therefore important to provide a mechanism for selectively disabling protocol checkers. Checkers can be disabled or enabled using either constraint during pre-run generation or by using method calls.

11.2.2.1 Static Checker Activation

Static checker activation designates whether a specific protocol checker should be activated during any particular simulation run. Static activation is required when the need for a checker is determined by factors that can be decided before simulation begins. For example, the verification environment for a system supporting multiple protocols consisting of PCI, ethernet, UART, and serial ATA includes protocol checkers for all its sub-blocks. For a verification engineer working on the UART module, the protocol checker for all other blocks is not required. Thus having a choice whether to disable the other checkers will reduce the simulation time.

Static checker activation is implemented by using generation constraints and a determinant struct member for the monitor to decide if a protocol checker should be included in the environment (see page 202).

During regression runs, only critical checkers should be enabled in order to reduce the regression time. For debugging purposes, only the relevant checkers for a particular feature should be enabled. This selective enabling and disabling of checkers leads to a reduction in simulation time overhead.

11.2.2.2 Dynamic Checker Activation

Dynamic checker activation refers to the enabling or disabling of a checker depending on DUV signal values or transitions. For dynamic checker activation, constant polling of DUV signals should be avoided. Checkers should be enabled or disabled based on events extracted from the DUV signals relevant to starting or stopping the checking. Consider a system where protocol checker **chkr1** is dedicated to checking the functionality of a calculus engine when FIFO overflow and underflow conditions occur and **chkr2** is used to check the functionality in normal FIFO conditions. In this case, only **chkr1** should be enabled when the FIFO full or empty conditions occur and disabled otherwise.

11.3 Collection and Reporting

In addition to protocol checking, a monitor also performs functions that facilitate the operation of other modules in the verification environment. These functions include:

- Data and Transaction Collection
- Coverage Collection
- Event Extraction
- Reporting using the Messaging Feature

Coverage collection is described in chapters 13 and 14. The remaining operations and their implementations are discussed in the following subsections.

11.3.1 Data and Transaction Collection

Data collection is an important function provided by a monitor, as the collected data is used in other functions provided by a monitor (i.e. scoreboarding, checking). This data can be packet data, serial/parallel data, transaction data, instructions, or commands, etc. The collected data for an ethernet packet is the complete ethernet packet and frames. Similarly, the collected data for USB traffic maybe one of bulk, control, isochronous, or control type packets For a UART interface, the data is a complete UART frame.

Data collection should be synchronized to a clock that reflects the time where valid data is present on the DUV signals. For example, when collecting a memory write transaction, it is necessary to identify bus idle cycles so that only valid data is collected from the bus. To summarize, the collector function of a monitor must understand the handshaking mechanism used to transfer data over DUV signals so that it can extract abstract data packets and frame from the physical traffic over the DUV signals.

Following example shows an *e* code fragment where a UART frame is collected.

```

1 : <
2 : extend si_uart_monitor {
3 :     incoming_bits : list of bit;
4 :     keep incoming_bits.size() == 0;
5 :
6 :     rcv_bit_value(): bit is {
7 :         result = rcv_port$;           // receiving from the port.
8 :         message(MEDIUM, "receiving data bits =", result);
9 :     };
10 :    rcv_next_data_bit() is {
11 :        incoming_bits.add(rcv_bit_value()); //The incoming bits getting collected.
12 :    };
13 :    rcv_stop_bit(): @sys.any is {
14 :        -- now unpack the bits collected into rcv_frame;
15 :        gen rcv_frame keeping {
16 :            .has_parity_error == FALSE;
17 :            .has_framing_error == framing_error
18 :            .size == config.get_word_length();
19 :            .parity_type == config.get_parity_selection();
20 :            .stop_bit_type == config.get_stop_bit_length();
21 :        };
22 :        unpack(packing.low, incoming_bits, rcv_frame);
23 :        emit agent.monitor.frame_receive_ended;
24 :
25 :        incoming_bits.clear();
26 :        framing_error = FALSE;
27 :    };
28 : };
29 : >

```

These TCMs defined above are then used in a finite state machine to start and stop data collection:

```

1 : <
2 : extend si_uart_monitor {
3 :     rcv_state : si_uart_receive_state;
4 :     keep rcv_state == IDLE;
5 :
6 :     rcv_fsm()@rclk is {
7 :         state machine rcv_state {
8 :             IDLE => IDLE {
9 :                 wait until true(not rcv_next_bit_is_zero())@rclk;
10 :            };
11 :            IDLE => START {
12 :                wait until true(rcv_next_bit_is_zero())@rclk;
13 :                wait until [7]*cycle;

```

```

14 :           };
15 :           START => SHIFT {
16 :               .....
17 :           SHIFT => STOP {
18 :               .....
19 :           };
20 :           // actions taken in a state
21 :           IDLE {
22 :               out("RCV FSM in IDLE State");
23 :           };
24 :           .....
25 :           START {
26 :               out("RCV FSM in START State");
27 :               emit frame_receive_started;
28 :           };
29 :           .....
30 :           SHIFT {
31 :               out ( " rcv FSM in SHIFT state");
32 :               rcv_next_data bit();
33 :           };
34 :           .....
35 :           STOP_1BIT {
36 :               out ( " rcv FSM in SHIFT state");
37 :               chk_stop_bit();
38 :               rcv_stop_bit();
39 :           };
40 :           .....
41 :           };
42 :       }; // rcv_state
43 :   }; // si_uart_monitor
44 : >

```

After collecting data, the following step is to use the collected data in scoreboarding, the protocol checker, or to store the collected data to a log file. These issues are discussed next.

11.3.1.1 Sending collected data to Scoreboard:

Data collected from the DUV ports may need to be sent to a Scoreboard if the signals tracked by the monitor are part of the signal path checked by a Scoreboard. Using monitors is the preferred approach for attaching scoreboards to the verification environment (see section 12.3). Approaches for using the monitor to attach a Scoreboard to a verification environment were discussed in section 12.3 and include using a hook method or events to call a method that inject the collected data into the Scoreboard.

Example below uses a direct method call to inject the collected data into the Scoreboard.

```

1 : <'
2 : extend si_uart_monitor {
3 :     rcv_stop_bit(): @sys.any is {
4 :         -- now unpack the bits collected into rcv_frame;
5 :         gen rcv_frame keeping {
6 :             .....
7 :         };
8 :         unpack(packing.low, incoming_bits, rcv_frame);

```

```

9 :             emit agent.monitor.frame_receive_ended;
10 :
11 :             // send the data collected to Scoreboard
12 :             compute agent.scoreboard.insert_next_frame(
13 :                 deep_copy(rcv_frame), TRUE, FALSE);
14 :
15 :             incoming_bits.clear();
16 :             framing_error = FALSE;
17 :         };
18 :     };
19 : };
20 : unit si_uart_agent {
21 :     Scoreboard : si_uart_scoreboard is instance;
22 : };
23 : '>'

```

11.3.1.2 Sending to a Checker

A protocol checker requires access to data collected from the DUB by the monitor. This data may be required for syntax checking or for deciding protocol behavior, which at times may depend on DUV signal values. In the following example, the checker verifies that any time event **packet_started** is emitted in the monitor, then in the next cycle, value of **packet_data** collected by monitor is set to 1'b0. In this approach, **packet_data** is collected and stored in the monitor, but the checker uses the collected data directly from the monitor.

```

1 : <'
2 : extend si_uart_monitor {
3 :     has_checker: bool;
4 :     keep soft has_checker == TRUE;
5 :     when has_checker si_uart_monitor {
6 :         checker: si_uart_checker is instance;
7 :         keep checker.monitor == me;
8 :     };
9 : };
10 : unit si_uart_checker {
11 :     monitor: si_uart_monitor; -- up pointer
12 :
13 :     expect @monitor.packet_started => true(monitor.packet_data == 1'b0)@monitor.clk;
14 : };
15 : '>'

```

11.3.1.3 Sending to a File

Collecting and storing data in a file is a common necessity as the collected data may need to be processed by a post-processing script. If audio data is generated through an input file, then by storing the output data in a file it is possible to compare the files using a file, compare utility. Message actions and message loggers are used to store the collected data in a file (see chapter 10).

11.3.2 Event Extraction

A monitor should extract timings related to DUV data activity since the operation of the protocol checker depends on this signal timing and activity relationship. This timing information is collected by extracting events from the DUV signals. These events are defined based on the specific checking requirements for a protocol. Examples of these events include an event indicating the time valid data arrives on the interface, the event indicating when a valid transaction ends, and the event indicating when a specific instruction like enabling a register happens. These events are then passed to other blocks (i.e. protocol checker) to check the timings and validity of control signals. For a basic data item, a monitor usually provides events such as **item_started** and **item_ended**. The monitor collects the **item_data** as soon as the **item_started** event occurs and ends collecting when the **item_ended** event occurs and then passes the collected data to its needed destinations. The example below shows the approach for emitting an event indicating the end of receiving a packet.

```

1  : <
2  : extend si_uart_monitor {
3  :     event frame_receive_ended;
4  :     rcv_stop_bit(): @sys.any is {
5  :         -- now unpack the bits collected into rcv_frame;
6  :         gen rcv_frame keeping {
7  :             .....
8  :         };
9  :         unpack(packing.low, incoming_bits, rcv_frame);
10 :         emit frame_receive_ended;
11 :         .....
12 :     };
13 : };
14 : >
15 :

```

The events extracted by the monitor are also very useful in determining the coverage information. For instance, a coverage group sampling event should be defined in the monitor containing that coverage group. The following shows an example of a coverage group in a monitor that uses event **frame_receive_started** as its sampling event.

```

1  : <
2  : extend si_uart_monitor {
3  :     cover frame_receive_started {
4  :         item rcv_frame;
5  :         item rcv_state;
6  :         item has_parity_error;
7  :     };
8  : };
9  : >

```


11.3.3 Reporting

A monitor should not only collect data and extract events but also print messages and log verification information for debugging purposes. Use of message and message loggers are described in the following subsections.

11.3.3.1 Messages

A message refers to the information printed on the screen or into a file when a particular event happens. Printing of messages is associated with the occurrence of data error or protocol error specifying when and where the error occurred. It is, however, important to print messages whenever an important event occurs as error at a particular point in time is a cumulative effect of many other events. Messages can be printed when the data is collected (i.e. start of every data frame), when a control signal changes, or when information is being passed to other blocks/components of verification environment. The following example shows the use of **message** construct for reporting purposes.

```
1 : <
2 :   extend si_uart_monitor {
3 :     rcv_bit_value() : bit is {
4 :       result = rcv_port$;           // receiving from the receiver port.
5 :       message(MEDIUM , "receiving DUV output data bits =" , result);
6 :     };
7 :     rcv_xmt_bit_value() : bit is {
8 :       result = rcv_xmt_data_port;   // receiving from the xmt port.
9 :       message(MEDIUM , "receiving DUV input data bits =" , result);
10 :    };
11 :  };
12 : >
```

11.3.3.2 Message Loggers

Message loggers are used to manage the output from message actions by filtering and formatting the messages and sending them user defined destinations such as a file or screen. By using a logger different trace levels can be defined so that by using these levels the verbosity of messages can be controlled.

The implementation of a messaging scheme for messages filtered based on three tags INFO, WARN, and ERROR is shown in the following:

```
1 : <
2 :   extend message_tag : [INFO,WARN,ERROR];
3 :   extend message_logger {
4 :     verbosity : message_verbosity;
5 :     tags : list of message_tag;
6 :   };
7 :   extend si_uart_checker{
8 :     logger1 : message_logger is instance;
9 :     keep soft logger1 .verbosity == LOW;
```

```

10 :         keep soft logger1.tags == {INFO};
11 :     logger2: message_logger is instance;
12 :         keep soft logger2.verbosity == HIGH;
13 :         keep soft logger2.tags == {ERROR};
14 :
15 :     check() @rclk is also {
16 :         wait @frame_receive_ended;
17 :         If (parity_type == EVEN_PARITY and parity_value == ODD_PARITY) {
18 :             message(ERROR, HIGH, "the parity has error", has_parity_error);
19 :         } else
20 :         If (parity_type == EVEN_PARITY and parity_value == EVEN_PARITY) {
21 :             message(INFO, HIGH, "the parity has no error", has_parity_error);
22 :         };
23 :     };
24 : };
25 : '>

```

In this implementation, message tag for **logger1** is set to INFO and the message tag for **logger2** is set to ERROR. Therefore, all message actions with tag INFO are handled by **logger1**, and messages with tag ERROR are handled by **logger2**. Therefore, message action on line 18 is handled by **logger2** and printed since its verbosity level matches the setting for the verbosity for **logger2**. Message action on line 21 is handled by **logger1**, but since its verbosity setting is higher than the verbosity setting for **logger1**, the message is filtered. Messages are described in detail in chapter 10.

11.4 Summary

This chapter introduced the concept of monitors, collectors, and checkers. Monitors are the verification modules that contain collectors, checkers, and reporting utilities. Checkers are used for protocol checking and are modeled as a checker instance that is contained in a monitor. The monitor extracts the data and emits the events that are required by the checkers to implement the necessary property checking statements. Monitors contain collectors used to collect coverage and data and transactions used in scoreboards and checkers.

Monitors must be implemented as passive modules that do not depend on the verification BFM or any active modules. This guideline allows for a monitor to be used without a BFM being present in the environment, therefore allowing for the easy migration of a monitor to the system level verification environment.

This page intentionally left blank

Earlier chapters in this book described the driver and monitor blocks of a verification environment. A driver injects stimulus data into the DUV input ports. A monitor performs protocol checking and data collection. Data checking is used to verify that data movement through the DUV meets the expected DUV behavior. A Scoreboard is the verification environment block that handles the data checking. Scoreboarding refers to the methodology used to perform data checking using a Scoreboard.

In scoreboarding, a data item (i.e. packet, instruction, etc.) that is driven in to the DUV is posted in a Scoreboard. The response data received from the DUV is also posted in to the scoreboard and the two data items are compared according to the expected behavior of the DUV.

The role of Scoreboard in the verification environment is shown in figure 12.1. The packets injected into the DUV by the driver are posted into the Scoreboard where the expected DUV output is computed using a predictor. This expected output is placed in the Scoreboard expected list. After the collector receives the DUV output, it is posted in to the Scoreboard and compared against the expected list. In the example shown in this figure, **pkt1** and **pkt2** are received correctly and matched in the Scoreboard. However **pkt3** is not generated by the DUV and **pkt4** is the next packet that is collected. Comparing **pkt4** in the Scoreboard leads to a mismatch as **pkt3** was the next expected packet.

The predictor in a Scoreboard deals only with data checking behavior. Timing checks related to the data output are handled in other blocks such as protocol checker.

Every input injected into the DUV has a matching output if indicated by the Scoreboard predictor. To verify this behavior, two types of properties are checked in a Scoreboard:

- Checks to make sure that no output data corresponding to input data is missed
-

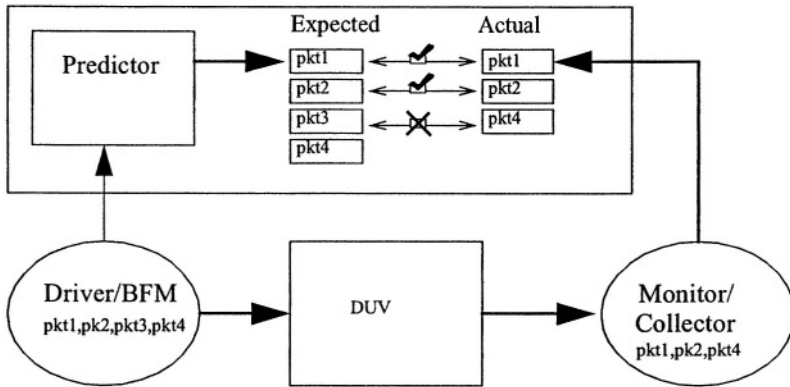


Figure 12.1 Scoreboarding Flow

- Checks to make sure that no extra data should appear at the DUV output ports.

12.1 Scoreboard Implementation

A Scoreboard is implemented using a struct as it is not associated with any HDL component. A Scoreboard contains the following fields:

- List of expected transactions
- List of actual transactions
- Predictor Method
- Injector Method
- Match Method

The predictor method is used to create an expected transaction based on the transaction injected into the DUV. An expected transaction is added to the end of the list of expected transactions. An actual transaction is collected from the DUV output. Every actual transaction that has been compared against the Scoreboard contents can be discarded. Therefore, the list of actual transactions is created only when necessary to maintain a record of actual transactions that have been checked against the Scoreboard (i.e. performing data comparison only after all transactions have been collected). The Scoreboard user calls the injector method to insert an input transaction into the Scoreboard. A match method is called by the user to match a DUV output transaction against the Scoreboard.

The following example shows a generic Scoreboard implementation for a UART module:

```

1  |
2  | : <'
   | : struct si_uart_scoreboard {

```

```

3 :     expected_uart_frames : list of si_uart_frame;
4 :     actual_uart_frames : list of si_uart_frame;
5 :
6 :     predictor(in_transaction: si_uart_frame): si_uart_frame is {
7 :         --build predictor based on expected DUV behavior;
8 :     };
9 :     insert(in_transaction: si_uart_frame) is {
10 :         expected_uart_frames.add(predictor(in_transaction));
11 :     };
12 :     match(actual_transaction : si_uart_frame): bool is {
13 :         -- insert into actual list if necessary
14 :         actual_uart_frames.add(actual_transaction);
15 :         --implement the comparison and matching actions
16 :     };
17 : };
18 : >

```

When a new transaction is injected into the DUV, that transaction is passed to the scoreboard using method **insert()**. This method uses method **predictor()** to create the expected output transaction for the input transaction. The result produced by the predictor is placed on the expected list of transactions.

During DUV output response collection, the collected transaction is matched against the Scoreboard contents by calling the **match()** method. This method is implemented based on the requirements of how transactions should be compared. In the most generic case, the matching compares the incoming actual transaction against the first transaction on the list of expected transactions and checks that all fields are exactly equal. These steps may be modified depending on the specific requirements of a DUV.

Unique IDs can also be used to match the data items. These unique IDs are usually carried in the data payload portion of transaction and are specially generated during stimulus generation. The ID of the received transaction should match the ID of the injected data transaction. The content of the data items for both transactions is compared only after transactions IDs are matched. An error is issued if there is a mismatch.

Data comparisons between actual and expected data items should be performed during the simulation runtime. This approach allows expected data and collected actual transactions to be discarded after the matching; therefore preventing memory limitations for long simulation runs. Sometimes it might be necessary to store the data processed by a Scoreboard. In such cases, it is best to print to a file and then discard its data.

12.2 Scoreboard Configuration Types

Scoreboards receive data from two places: source data injected into the DUV that is used to create the expected transaction, and actual data collected from the DUV output. The actual data item is collected at the DUV output by the monitor or collector. The injected data can be placed

into a Scoreboard by either a monitor or a driver. These configurations are discussed in the following sections.

12.2.1 Driver/BFM Based Scoreboard

In this type of Scoreboard, the source data is injected by the BFM that drives the data into the DUV port. Figure 12.1 shows the configuration for this type of Scoreboard.

Advantages of using a Driver/BFM based Scoreboard are:

- Can build the Scoreboard quickly as it does not require extra code for collecting the source data from the environment. Whenever a BFM injects data to DUV, the same data is directed to the Scoreboard.
- Better performance because no extra TCMs are required for capturing signals and reconstructing data.

12.2.2 Monitor Based Scoreboard

In this type of Scoreboard, the source data item is injected into the Scoreboard by the monitor. The monitor tracks the interface port through which input data is injected. As part of this process, the monitor collects injected data items on this DUV port and inserts them into the Scoreboard whose source data is collected from the monitored input port. In this configuration, both source data for a Scoreboard and the actual data collected from output ports are collected by monitors. Figure 12.2 shows the configuration for a monitor based Scoreboard.

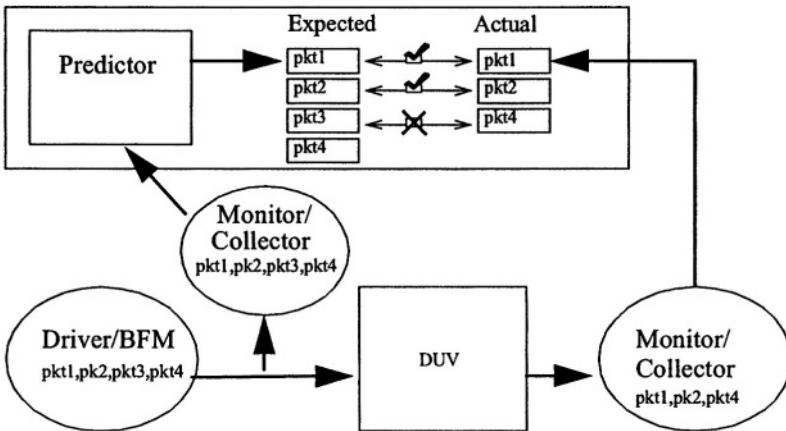


Figure 12.2 Monitor Based Scoreboard

Monitor based scoreboards are very useful when migrating from a module level verification to system level verification. When moving from module level to system level verification, the driver verification component for a module is replaced by the DUV component driving that module in the system level environment. Since monitor based scoreboards do not rely on driver components and extract the source data from the environment, they can be used without modification during system level verification. This concept was illustrated in figure 3.9.

Advantage of using a monitor based Scoreboard are:

- The migration from module level verification environment to system level verification is easy. No additional effort is required and an existing Scoreboard can still be used in the system level environment.

12.3 Attaching Scoreboards to the Environment

As discussed earlier the interface to a Scoreboard is usually through a BFM/driver or a monitor. The interface between the BFM and Scoreboard, or monitor and Scoreboard should be maintained in such a way that it is easy to modify or extend when the verification environment moves from module level to system level. The ports, events, or any fields that are going to be used for providing input to the Scoreboard should be put in a separate file so as to provide enough flexibility while moving from one verification environment to another. The following subsections describe techniques for attaching a Scoreboard to the verification environment.

12.3.1 Direct Method Call

As soon as the driver/BFM sends the data item to the DUV, a method can be called in the driver or the monitor to insert the same data into the Scoreboard. Using this approach for a driver is shown in the following example:

```

1 : <'
2 :   extend evc_driver {
3 :     !data_item: packet;
4 :
5 :     main_tcm()@clk is {
6 :       for i from 0 to n-1 {
7 :         gen data_item;
8 :         bfm.send_data_to_duv(data_item);
9 :         sb.insert(data_item);
10 :       };
11 :     };
12 : };
13 : '>
```


In this case, method `send_data_to_duv()` is used for injecting `data_item` into the DUV port. `data_item` is inserted into the Scoreboard at the same time by calling the `insert()` method of the Scoreboard.

Using direct methods calls is easy to implement. However, it requires that driver and Scoreboard implementations be closely tied to one another. To improve reusability, it is better to remove this dependency between the driver or monitor and the Scoreboard. Techniques for achieving this goal are shown in the next subsections.

12.3.2 Using Hook Methods

An empty hook method can be declared in the driver or the monitor with `data_item` as its parameter. When attaching the Scoreboard to the environment, this hook method can be extended to pass `data_item` to the Scoreboard. This approach is shown in the following example:

```

1 : <
2 : extend evc_monitor {
3 :     process_captured_data(data_item: packet_type) is empty;
4 :
5 :     a_monitor_tcm() @clk is {
6 :         var captured_data_item : packet_type;
7 :
8 :         // capture the data item from the DUV interface
9 :         captured_data_item = capture();
10 :        //.. .. and it is ready to be inserted into Scoreboard
11 :        process_captured_data(captured_data_item);
12 :        //.. .. Continue operation .. ..
13 :    };
14 : };
15 : >
```

In the above, the hook method `process_captured_data()` is declared as an empty method. After the monitor TCM `a_monitor_tcm()` captures the `data_item` using method `capture()`, it is passed to the hook method. At this point, the hook method is empty and code will function even without a Scoreboard in place.

The next step is to extend the hook method to include the necessary code for passing the collected data to the Scoreboard. This step is shown in the example below where the collected data is passed to the Scoreboard.

```

1 : <
2 : extend evc_monitor {
3 :     process_captured_data(data_item: packet) is also {
4 :         sys.sb.insert( data_item);
5 :     };
6 : };
7 : >
```

If there are one or more monitors using the same Scoreboard and the Scoreboard requires information from the monitor collecting the data, then the monitor can be passed to the scoreboard using the following modification:

```

1 : <'
2 :   extend evc_monitor {
3 :     process_captured_data(data_item: packet) is also {
4 :       sys.sb.insert( data_item , me);
5 :     };
6 :   };
7 : >'

```

This approach is notable for its reusability, and it also isolates monitor and driver implementation and scoreboarding. In this approach, the hook mechanism can be used not only for scoreboarding, but also for any other steps that require information from the collected data.

12.3.3 Using Events

Another approach for attaching a Scoreboard to a verification environment is implementing the drivers and monitors so that an event is emitted when the BFM drives a data item into the environment; or when the monitor collects a data item. This approach is shown in the following example:

```

1 : <'
2 :   extend evc_monitor {
3 :     event data_ready_to_be_processed;
4 :     captured_data_item : packet ;
5 :     a_monitor_tcm () @ clk is {
6 :       //data item ready to be posted after monitor captures it
7 :       captured_data_item = capture();
8 :       emit data_ready_to_be_processed;
9 :       //.....continue.....code...
10 :     };
11 :   };
12 : >'

```

The event **data_ready_to_be_processed** is used in the Scoreboard to pull the data from the relevant field **captured_data_item** as shown below:

```

1 : <'
2 :   extend Scoreboard {
3 :     env_pointer : evc_env; // pointer to the evc environment
4 :     event data_sent is @env_pointer.monitor.data_ready_to_be_processed;
5 :     on data_sent {
6 :       insert (env_pointer.monitor.captured_data_item);
7 :     };
8 :   };
9 : >'

```

12.4 Scoreboarding Strategies

Scoreboarding can be performed across DUV ports, or in multiple steps by checking correct data transfer through each DUV sub-block. Strategies are selected based on the specific Scoreboarding application. These strategies are described in the following subsections.

12.4.1 End-to-end Scoreboarding

A DUV consists of many sub blocks. But for end-to-end scoreboarding, a DUV acts as a single block. The Scoreboard looks only at the data items going in and the data items coming out of the DUV ports.

An end-to-end scoreboarding strategy has the following advantage:

- The end-to-end scoreboarding approach is useful when DUV modules are not very complex and debugging at module level is not required. The amount of code to be written is less, and therefore fast to develop.

This approach has the following disadvantages:

- The end-to-end scoreboarding approach is not good for complex designs with big modules because debugging becomes difficult. If there is an error, then all the modules will be targeted for debugging.
- Module implementations cannot be checked independently.

12.4.2 Multistep Scoreboarding

In a multistep scoreboarding strategy, each sub-block in the DUV has its own Scoreboard. In this approach data checking is done across sub-blocks.

Figure 12.3 shows the setup for multistep scoreboarding. Here, the verification environment is sub-divided into parts for each DUV sub-block. In this case, the DUV consists of two sub-blocks with one Scoreboard for each module. The sub-block verification environments are combined to create the chip level verification environment once the verification environments for each sub-block is created. The DUV in this instance consists of **module1** and **module2**. Disconnecting the driver from **module2** and driving its input from the output of **module1** creates the chip level environment. All other verification modules are transferred without modification from the module level verification environment.

Multistep scoreboarding is helpful for easy migration from a module level verification to chip level verification, but the size of the code increases because each module now has its own monitor and Scoreboard.

A multi-step scoreboarding strategy has the following advantage:

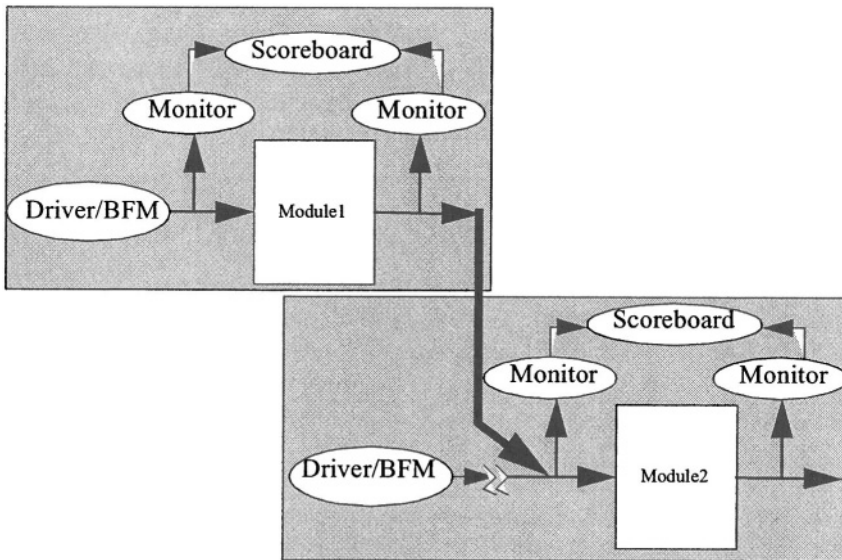


Figure 12.3 Multistep Scoreboarding

- A multi step Scoreboard is a good implementation for complex designs with big modules. The development and testing can be split into small sub-blocks and multistep scoreboard expedites debugging at each sub block level.
- All the modules or sub blocks of a bigger system can be verified independently by different engineers simultaneously.
- Debugging for errors is easy and quick since the environment can indicate which block caused the error. Instead of debugging the whole system, a specific module can be targeted. This is useful for catching bugs that might otherwise be filtered out when the data path goes from one block to another.
- The migration from module level to system level environment is very straightforward.

This approach has the following disadvantages:

- The amount of code to be written increases.
- This approach may not be necessary for small designs where a simple end-to-end scoreboard would save environment development time.

12.5 Summary

This chapter introduced the concept of scoreboarding and presented different approaches for configuring scoreboards. A generic implementation of a scoreboard was presented and approaches for connecting a scoreboard to the verification environment were discussed.

Multistep and end-to-end scoreboarding strategies were introduced along with their advantages and disadvantages in the context of migrating from a module to system level verification.

PART 5

Coverage Modeling and Measurement

This page intentionally left blank

A coverage tool monitors signal values in the DUV and verification environment before passing the collected data to the coverage engine. The coverage engine then collects and analyses functional coverage information based on the data collected by the coverage monitor before generating reports based on the collected data.

The data collected by a coverage engine is used to produce the following report types:

- Combination of features, DUV data, states and transitions of fields and events
- Whether verification goals are met
- Value distribution for data items that are randomly generated

13.1 Coverage Collection Steps

Coverage collection is the first step in determining functional coverage of a DUV. During this phase, all the required data for measuring coverage is collected. Coverage collection is implemented using the following steps:

1. Scalar data objects in the DUV and in the verification code that are necessary for collecting functional coverage information are identified. These objects define the coverage items
 2. The type of coverage collection should be defined for each coverage item (i.e. transition, etc.)
 3. A sampling event is decided for each coverage item. Coverage information for that data object is collected upon occurrence of this sampling event.
 4. Coverage items with the same sampling event are grouped into coverage groups.
 5. Coverage buckets are defined for each coverage item. A coverage bucket identifies scalar values that need to be differentiated during coverage collection.
-

6. The necessary *e* program is developed to implement the coverage items, groups, buckets, and customization of each.
7. The coverage engine automatically collects coverage data during the program runtime.
8. Functional Coverage reports are produced using the data collected in the coverage engine.

The following sections describe the implementation of these coverage concepts in the *e* language.

13.2 Coverage Terminologies

The following coverage terminologies are used in this chapter.

A *Coverage Item* defines the data object that should be sampled for coverage collection. A coverage item can be a field, variable, or HDL signal from the DUV. For example, in the verification of an ethernet switch or a router the values of the length and address of each data packet should be collected for covering the full range of possible address and lengths.

A *Coverage Group* is a construct used to group items with the same sampling event. For example, the two items for collecting packet length and address information are defined inside a coverage group. All items in a coverage group are sampled upon the occurrence of its sampling event.

A *Coverage Bucket* is a symbolic representation for the container, which stores a single or a range of values for an item. For each item that collects coverage data on a CPU instruction type, one bucket is assigned for each instruction type. For an item that collects data on data packet length ranging in value from 64 to 1024 byte, a bucket may represent values in the range [64..511] and in a second bucket, values in the range [512..1024].

A *Cross Item* is an item that collects the data for the combination of all possible values of two or more items. A *Transition Item* is an item that collects data for value transitions of a data object at the occurrence of the sampling event.

A *Bucket Hit* or a Hit refers to the case when the sampled value for a data object falls in the range specified for a bucket. *Coverage Goal* specifies the target number of hits for a bucket of an item. If coverage goal for a bucket is set to at least 10 hits, then that goal has been covered 100% only if the bucket is hit 10 times. A *Coverage Hole* refers to a bucket whose coverage goal has not been reached. *Coverage Grading* refers to measuring the coverage percentage of each bucket using values in the range [0..1.0]. A bucket with no hit shall have a 0 grade and the one that has reached the required hits has a grade of 1.0.

13.3 Scalar Coverage Constructs

Scalar coverage collection refers to collecting information about the value of a data object sampled at specific time instances indicated by a sampling event,; and filtered according to certain boolean qualifiers. Questions answered by scalar coverage analysis include:

- How many times an ADD instruction was executed by the CPU?
- How many ethernet packets of size less than 50 were observed on the link?
- How many memory read bus transactions were observed on the bus while system initialization was in progress?

This section describes the basic concepts of scalar coverage modeling and the language constructs for implementing these models. Information about the relationship between data objects across cycles and across different design signals is provided by composite coverage analysis. Composite coverage is discussed in section 13.4.

13.3.1 Coverage Groups

A *Coverage Group* is a struct member that contains all coverage items that use the same event for sampling item values. The **cover** construct is used to model coverage groups. A previously defined event in the same struct is used to identify the coverage group. Coverage groups are defined using the following syntax:

```
cover event_type is empty;
cover event-type [using coverage-group-option,...]is {
    coverage-item-definition;...
};
```

In this syntax, **event_type** is a previously defined event for the parent struct of the coverage group. **event_type** is used as the name of group. Whenever this event occurs the coverage data for the group is collected. **coverage-item-definition** is used to define the coverage items inside the coverage group. Coverage item definitions are discussed in the next section. Now, consider the following example:

```
1 : <
2 : type cmd : [LOAD, MEM_LOAD, ADD, SUB, MULT, DIV, NOP];
3 : struct cpu_inst{
4 :     opcode : cmd;
5 :     data_size : byte;
6 :     event inst_cov;
7 :     cover inst_cov is {
8 :         item opcode1;
9 :         item data_size;
10 :     };
11 : };
12 : >
```

In the above, coverage group **inst_cov** is defined inside struct **cpu_inst**. Note that **inst_cov** is declared as an event in the same struct. Whenever event **inst_cov** occurs, the value of **opcode** and **data_size** items in the cover group are collected.

The following options can be used when defining a coverage group:

- no_collect** = boolWhen True, no coverage data is collected for the coverage group.
- text** = string.....string describes the coverage group.
- when** = bool-exp.....bool-exp is used to filter event occurrences during which coverage items are sampled. Coverage items are sampled only when bool-exp evaluates to TRUE.
- global** = boolwhen True, this coverage group becomes a global coverage group. The sampling event for a global coverage group is expected to occur only once, otherwise a DUT error is generated.
- radix** = declhexbin.....Specifies the format for displaying the collected values in the buckets.
- weight** = uint.....This option specifies the grading weight of the current group relative to other groups. It is a nonnegative integer with a default of 1.

Above options are used in the following example:

```

1  :  <
2  :  type usb_pkt_type : [bulk, control, iso, intr];
3  :  struct pkt { data: byte;};
4  :  struct usb_pkt{
5  :      datapkt : pkt;
6  :      num_pkt : uint;
7  :      size : uint;
8  :      pkt_type : usb_pkt_type;
9  :      event data_sent;
10 :      cover data_sent using no_collect is {
11 :          item data: uint(bits:8) = datapkt.data using radix = HEX;
12 :      };
13 : };
14 : struct usb_trans {
15 :     init_done : bool;
16 :     num_err : uint;
17 :     event state_change;
18 :     event reset_event;
19 :     event trans_done;
20 :     cover reset_event using global is {
21 :         item reset : bit = "top.usb.reset";
22 :     };
23 :     cover state_change using
24 :         text = "USB transaction state-machine",
25 :         when = (init_done == TRUE) is {
26 :             item st: byte = 'top.usb.main_trans state';
27 :         };
28 :     cover trans_done using weight = 3 is {
29 :         item len: uint (bits: 8) = sys.i_usb_pkt.size;
30 :         item num_err ;

```

```

31 :     };
32 : };
33 : extend sys {
34 :     i_usb_pkt::usb_pkt;
35 :     i_usb_trans:usb_trans;
36 : };
37 : >

```

The following observations can be made for the above example:

- **no_collect** option: The coverage group `pkt_data` does not save coverage data.
- **text** option: The text “USB transaction state machine” appears at the beginning of the data for the group in the ASCII coverage report.
- **when** option: Coverage is collected for **st** when the **state_change** event occurs and **init_done** is TRUE.
- **global** option: The **reset_event** is expected to occur only once. If it occurs more than once a DUT error is issued.
- **weight** option: The **trans_done** coverage group is assigned a weight of 2. If there are 10 other coverage groups that all have default weights of 1, the **trans_done** group contributes $(2/12)*grading(trans_done)$ to the **all** grade.

13.3.2 Basic Coverage Items

A basic coverage item is used for collecting coverage information for the value of a data object at the occurrence of a sampling event. A basic coverage item may collect coverage information on a struct member, or DUV signals. A basic coverage item is defined in the `cover` construct may correspond to an existing struct member or collect coverage data on the result of an expression based on `e` and DUV signal values. Coverage items are defined in the `e cover` construct.

```
item item-name [ : type=exp ] [ using coverage-item-option, ... ]
```

In this syntax, **exp** is the expression that produces the data value used for coverage collection, **type** is the type of data produced by **exp**. Simple examples of item definition were shown in the previous section.

The coverage item construct has various options available that can be used for defining a coverage item with a variety of features. These options show the strength of the tool and give flexibility to the user for determining the coverage according to the functionality of the DUV and coverage collection requirements.

- per_instance** = boolWhen set to TRUE, coverage information for separate instances of struct or units are collected and graded separately.
- no_collect** = bool.....When set to TRUE, no coverage data is collected for this coverage item.
- no_trace** = boolWhen set to TRUE, this item is not traced by tje simulator.

text = string,.....This option provides a descriptive text.

when = bool-exp,.....The item is sampled only when bool-exp is True.

ranges = {range(parameters);...} Provides the range of values defining the buckets for the current item. See below for range syntax.

ignore = item -bool-exp,.....Defines the values that can be ignored completely, and thus not used while generating coverage report.

illegal = item-bool-exp,.....Specifies illegal values for the current item.

radix = DEC | HEX | BIN Specifies the radix for items of type int & uint. The default is decimal.

weight = uint,..... Specifies the weight of the current item in a coverage group with respect to other items. Default is 1 .

name = alt-nameAssigns an alternative name for specific items like cross or transition item. This option cannot be extended by using also.

The **range** construct is used to define bucket ranges for a verification item. The syntax for this construct is:

```
range(range: range, name: string,
      every-count: int, at_least-num: int
```

The range construct has 4 parameters. The first parameter, **range**, provides the range for the buckets, such as a bucket with range[2..10] or a bucket with range [1] only. The second parameter, name, assigns a name to the range. **every-count** gives the number of buckets within a range. If the range is defined to [0..256] and **every_count** is 16 then 16 buckets each with a range of 16 will be created. This option cannot be used if the **name** parameter is used. **at_least** specifies the minimum number of samples for each bucket. Anything less than that number is considered a hole. So, if a bucket is required to have 3 minimum samples but has only 2 hits during the run, then that bucket is counted as a hole in the coverage report.

Coverage item usage is shown in the following example:

```

1 : <
2 : type state_type: [INIT, RESET, RUN];
3 : extend sys {
4 :     init_done : bool;
5 :     state: state_type;
6 : };
7 : struct packet {
8 :     pkt_type : [ETH , UART, USB];
9 :     pkt_len: uint(bits: 12);
10 :     addr: byte;
11 :
12 :     event send;
13 :     cover send is {
14 :         item pkt_type using per_instance;
15 :         item len : uint(bits: 12) = pkt_len using ranges = {
16 :             range( (16..255], "small");
17 :             range( [256..3k-1], "medium");
18 :             range( [3k..4k-1), "big");
19 :         };
20 :         item illegal: bool = (pkt_len < 16 or pkt_len > 4000) using
```

```

21 :                 when = (sys.init_done==TRUE);
22 :                 item state: state_type = sys.state using text = "packet transmit state";
23 :                 item start_addr : byte = addr using
24 :                 ranges = (range([0..255], "", 32, 16));
25 :             };
26 :         };
27 :     '>'

```

In the above example, using the **per_instance** option for the **pkt_type** item, collects and grades coverage information for all other items in the same coverage group separately for each possible value of **pkt_type**. The range for **start_addr** item is set to 0 to 255 where this range is further divided into 32 sub-ranges each having a range of 8 values. The **at_least** option is set to 16 and indicates that a bucket with less than 16 hits is considered a hole.

13.3.3 Sampling Events

A coverage item is sampled at the occurrence of the sampling event for its parent coverage group. Selecting the correct sampling event is therefore important in generating accurate coverage reports. For example, coverage should be collected on bus transaction types only when bus is in use and at the stage where the transaction type is being signaled on the bus. Similarly, coverage information for a CPU instruction execution unit should only be collected when a new instruction is being executed. Consider the following:

```

1 : <'
2 : type op_type : [LOAD, MEM_LOAD, ADD, SUB, MULT, DIV, NOP];
3 : unit cpu {
4 :     opcode : op_type;
5 :     value : uint;
6 :     interrupt : bool;
7 :     event cpu_exec is rise('top.cpu.exec') @sim;
8 :     //Indicates start of inst execution
9 :
10 :     cover cpu_exec {
11 :         item opcode;
12 :         item value;
13 :         item intrpt;
14 :     };
15 : };
16 : '>'

```

As shown, the **cpu_exec** event occurs when the signal **top.cpu.exec** rises. The coverage group for this event samples values for **opcode**, **value**, and **interrupt** at the occurrence of event **cpu_exec**.

Defining a sampling event dedicated to collecting coverage is recommended. With this approach, it is possible to control coverage collection more explicitly. The following example demonstrates how coverage on a packet is collected when the sampling event for its coverage group is explicitly emitted as new packets are generated.

```

1 : <
2 :   unit driver {
3 :       pkt : packet;
4 :       event clk is rise('top.clk');
5 :       drive() @ clk is {
6 :           for i from 0 to 10 {
7 :               gen pkt;
8 :               emit pkt.cov;
9 :               --drive pkt here.
10 :              wait cycle;
11 :           };
12 :       };
13 :       run() is also {
14 :           start drive();
15 :       };
16 :   };
17 :   struct packet {
18 :       event cov;
19 :       addr: uint;
20 :       data: byte;
21 :       cover cov is {
22 :           item addr;
23 :           item data;
24 :       };
25 :   };
26 : >

```

13.3.4 Coverage Buckets

The values sampled for a coverage item are organized using coverage buckets. These values are then used to analyze the results of coverage collection. It is important to specify the following bucket properties for a coverage item accurately:

- Ranges of values for buckets
- Minimum number of required hits for a bucket
- Illegal buckets and buckets that should be ignored

By default, a coverage bucket refers to each possible value of a coverage item. However, the definition for a bucket can be altered to include a range of possible values. The range of its possible values and the grouping of these values determine the number of buckets for a coverage item.

In the example below, **opcode1** is an item for which coverage needs to be collected. This item has a type defined by the **cmd** enumerated type with 7 possible values. Therefore opcode 1 by default has 7 buckets, one for each possible value of its enumerated type.

```

1 : <
2 :   type cmd : [LOAD, MEM_LOAD, ADD, SUB, MULT, DIV, NOP];
3 :   struct cpu_inst {
4 :       opcode1 : cmd;
5 :       data_size : byte;
6 :       keep data_size in [1..100];

```

```

7 :      event inst_cov;
8 :      cover inst_cov is {
9 :          item opcode1;
10 :         item data_size;
11 :     };
12 : };
13 : >

```

During a simulation run, a bucket hit occurs every time the sampled value for **opcode1** matches the bucket range. Therefore, in a series of 10 samples, if the sampled values for **opcode1** are **LOAD, NOP, NOP, ADD, LOAD, SUB, SUB, DIV, SUB, LOAD** then the corresponding bucket hits are shown in figure 13.1

type cmd: [LOAD, MEM_LOAD, ADD, SUB, MULT, DIV, NOP];

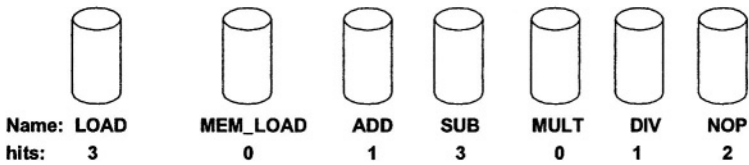


Figure 13.1 Coverage Buckets for Enumerated Types

The default range for a bucket can be redefined using the **range** construct. In the above example, coverage item **data_size** by default has 100 buckets. The ranges for the buckets can be redefined so that only 7 buckets are created with each bucket receiving a hit for a range of values. The hits for each bucket for the sequence of sampled values 10,15,45,87,46,23,12,98,9 is shown in figure 13.2.

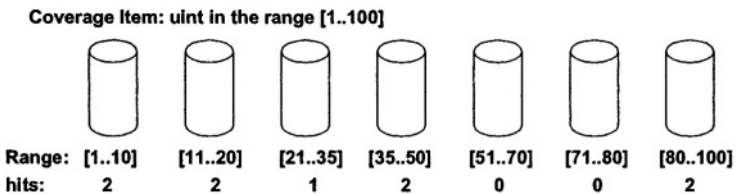


Figure 13.2 Coverage Bucket Ranges for Scalar Value

13.3.4.1 Bucket Ranges

Every possible value of a coverage item is assigned one bucket by default. But if the number of possible values is very large, then to improve performance, a bucket should be assigned a range of values to reduce the total number of buckets. The **ranges e** construct is used to define bucket

ranges. The ranges to be assigned to coverage items should be decided when defining the coverage model. Coverage result analysis should be performed based on the ranges covered by the bucket.

Consider the length field for an ethernet packet. Packet length can have values from 0 to 2048. Assigning a bucket to each value leads to significant overhead during simulation time. To eliminate this unnecessary overhead, the valid range for the packet length can be divided into sub-ranges where one bucket is used for each sub-range. It is not necessary for all buckets to have equal range, as each bucket can represent a different range of values. The code below shows the assignment of ranges for buckets.

```

1 : <'
2 : struct frame {
3 :     legal: bool;
4 :     length: uint;
5 :     keep length in [0..2048];
6 :     event send_frame;
7 :     cover send_frame is {
8 :         item length using ranges = {
9 :             range([64..65], "short_legal");
10 :            range ([66-512], "normal_legal1");
11 :            range([513-1024], "normal_legal2");
12 :            range([1025-1516], "normal_legal3");
13 :            range([1517], "big_legal1");
14 :            range([1518], "big_legal2");
15 :        };
16 :    };
17 : };
18 : >'

```

In this example, the ranges for the item **length** are defined, creating 7 buckets consisting of 6 ranges that are explicitly defined, and one bucket for all remaining values. The last bucket is used for values [0..63,1519..204] which were not included in any of the explicitly defined sub-ranges. A name is also assigned to each bucket.

For some applications, assigning equal ranges to buckets of an item may be acceptable. This can be done as shown below:

```

1 : <'
2 : struct frame {
3 :     addr : uint;
4 :     keep addr in [0x101..0x200];
5 :     event send_frame;
6 :     cover send_frame is {
7 :         item addr using ranges = {
8 :             range([0x101.. 0x200], NULL, 32);
9 :         };
10 :    };
11 : };
12 : >'

```

In this instance, the range for **address** is divided equally into 32 sub-ranges creating 8 buckets where the first bucket has the range [0x101..0x120], the second [0x121..0x140], and so on.

13.3.4.2 Default Buckets

Default buckets are used for a coverage item if the **ranges** construct is not used to explicitly define bucket ranges. The default buckets are defined as follows:

- Bool: one bucket for TRUE and one for FALSE.
- Enumerated Types: One bucket is allocated for each enumerated value.
- **int** or **uint** less than 32 bits: If the number of possible values is less than 16, then one bucket is allocated for each value.
- Types for which the range is not implicit: One bucket is created for each value.

The items for which bucket ranges are explicitly defined, an **others** bucket is automatically created for all item values that do not belong to one of the explicitly defined buckets.

```

1  : <'
2  : struct inst {
3  :     op2: byte;
4  :         keep op2 in [1..24];
5  :     event inst_driven;
6  :     cover inst_driven is {
7  :         item op2 using ranges = {range([1..16], "", 4)};
8  :     };
9  : };
10 : '>

```

In the above example, the range for field **op2** is [1..24] while defining it as a coverage item, but only range [1..16] is defined by 4 buckets. Sampled values in the range [17..24] are placed in bucket **others** which is the default bucket.

13.3.4.3 Illegal Buckets


Some values for a coverage item may be considered illegal during the program runtime because of DUV or verification environment requirements. Such values or ranges of values must be declared as illegal during coverage collection. Coverage item option **illegal** is used to specify invalid ranges for a coverage item.

```

1  : <'
2  : struct frame {
3  :     length: uint;
4  :     event send_frame;
5  :     cover send_frame is {
6  :         item length using
7  :             ranges = {
8  :                 range([64..127], "short_legal");
9  :                 range([127..1518], "long_legal");

```

```
10 :           },
11 :           illegal = (length <64 or length==1519);
12 :       };
13 : };
14 : >
```




In the above example, the `illegal` option is used to define all packets smaller than 64 and larger than 1513 bytes as illegal. If during coverage collection a value in the illegal range is sampled, a runtime error is generated.

13.3.4.4 Ignored Buckets

Some item values can be ignored completely when analyzing coverage results. The `ignore` option is used to remove a value or ranges of values from any consideration during coverage collection.

```
1 : <
2 : struct cpu_instruction {
3 :     itype: [NOP, ADD, SUB, MULT, DIV];
4 :     event send_inst;
5 :     cover send_inst is {
6 :         item itype using ignore = (itype==NOP);
7 :     };
8 : };
9 : >
```



Coverage above is collected on the instruction type. Even though a NOP instruction is expected to occur, these NOP instructions are completely ignored during coverage collection.

13.3.4.5 Bucket Grading

Coverage Grading refers to measuring the coverage for buckets of coverage items. Coverage grade for a bucket is calculated to a value between 0 and 1.0 using the equation (number-of-hits/minimum-required-hits). Each bucket has a parameter for the required minimum number of hits which is specified using the **at_least** option. If a bucket has more hits than the minimum required, then its grade is set to 1.0. Figure 13.3 shows an example of coverage grading for values samples for a CPU instruction type. Grading is discussed in section 14.3.

13.4 Composite Coverage Items

Composite coverage collection refers to collecting coverage information about correlation of values between a number of data objects. This correlation may be defined as relationship between values of different data objects sampled at the same time using the same sampling event, or changes in the value of single data object across multiple sampling events.

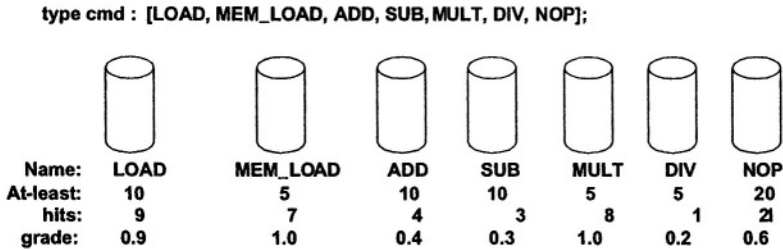


Figure 13.3 Coverage Buckets for Enumerated Types

Questions that can be answered by composite coverage analysis include:

- How many bus read transactions for address range [0x100..0x200] were observed during system initialization?
- How many ethernet packets of size in range [0x100..0x200] with destination address 0xA6B3:D87E:82AB?
- How many times a CPU ADD instruction execution was followed by a CPU DIV instruction execution while the CPU was executing an interrupt service routine?

Composite coverage items include cross coverage items and transition coverage items, and are described in the next subsections.

13.4.1 Cross Coverage Items

Cross coverage provides information about values for multiple data objects sampled at the same time. The number of possible values for cross coverage is the cross product of possible values for each item that is included in the cross coverage. Consider two items A with possible values $[A_1, A_2]$, and B with possible values $[B_1, B_2]$. The cross coverage of A and B has the possible values $[A_1B_1, A_1B_2, A_2B_1, A_2B_2]$.

A *Cross Coverage Item* is defined in terms of previously defined coverage items in a coverage group. The **cross** construct is used to define a cross items. The syntax for this construct is:

```
cross item-name1, item-name2, ... [ using coverage-item-option, ... ]
```

The options for the cross coverage are as follows:

- name** = label.....Label specifies a name for the cross coverage item. The default name is 'cross__item-a__item-b'.
- text** = string.....String provides text description for the cross coverage item.
- when** = bool-exp.....Item is sampled only when bool-exp evaluates to TRUE.
- at_least**=numnum specifies the minimum number of samples each bucket should have so that it is not consid-

- ered a hole.
- ignore** = item-bool-exp.....Defines cross values that can be ignored and excluded from coverage measurement.
 - illegal** = item-bool-exp.....Defines cross values that are illegal. A DUT error is generated when an illegal cross value is sampled.
 - weight** = uint.....Specifies the weight of the current cross item relative to other items.

The following example shows a definition for a cross item based on the definitions for opcode and reg coverage items.

```

1  :  <'
2  :  type cpu_opcode: [ADD, SUB, OR, AND, JMP];
3  :  type cpu_reg : [LOAD_REG, EN_REG , INT_REG ];
4  :  struct inst {
5  :      opcode : cpu_opcode;
6  :      opcode1 : cpu_opcode;
7  :      opcode2 : cpu_opcode;
8  :      reg : cpu_reg;
9  :      event inst_drv ;
10 :      cover inst_drv is {
11 :          item opcode;
12 :          item opcode1;
13 :          item opcode2;
14 :          item reg;
15 :          cross opcode, reg using
16 :              when = (opcode != ADD and reg!= LOAD_REG),
17 :              name = cross_exclude_ADD_LOAD;
18 :          cross opcode, reg using
19 :              illegal = (opcode == JMP and reg == INT_REG),
20 :              name = cross_with_illegal_JMP_INT;
21 :          cross opcode1, opcode2;
22 :          cross cross_exclude_ADD_LOAD, cross__opcode1__opcode2;
23 :      };
24 : };
25 :  >

```

Two cross items are defined based on the items **opcode** and **reg** in the above code. For the first cross on line 11, the cross coverage collection is filtered by using the **when** option on line 12, and for second cross definition on line 14 an illegal case is defined. The default name for both cross items on lines 11 and 14 is **cross__opcode_reg**. To prevent naming conflicts between the two cross items, a new name is assigned for each cross item using the **name** option.

It is possible to define a cross item based on other cross items. In above example, cross items defined on lines 18 and 21 are used to define a new cross item on line 22.

13.4.2 Transition Coverage Items

A transition coverage item is used to collect coverage information for transitions across the sampling events for data values. The syntax for a transition coverage item is:

```
transition item-name [using coverage-item-option]
```

The various options for transition coverage item are as follows:

name = label.....Label specifies a name for the transition coverage item. The default name is 'transition__item-name'.

text = string.....String provides text description for the transition coverage item.

when = bool-exp.....Item is sampled only when bool-exp evaluates to TRUE.

at_least=numnum specifies the minimum number of samples each bucket should have so that it is not considered a hole.

ignore=item-bool-expDefines transition values that can be ignored and excluded from coverage measurement.

illegal= item-bool-exp.....Defines transition values that are illegal. A DUT error is generated when an illegal transition value is sampled.

weight = uint.....Specifies the weight of the current transition item relative to other items.

Definition of the coverage transition item, by default, creates a variable called **prev_item-name** that holds the previous sampled value of **item-name**. This variable can be used in constructing boolean expressions that depend on the previous sampled value of a transition item.

The following program shows an example of transition coverage item definition.

```

1  :  <'
2  :  struct processor {
3  :      st: machine_state;
4  :      event state_change is change('top.processor.cur_state') @sim;
5  :      cover state_change is {
6  :          item st;
7  :          transition st using illegal = (prev_st == START and st == LOAD1);
8  :      };
9  :  };
10 :  >'

```

In the above example variable **prev_st** is used to define transition **START -> LOAD1** as illegal. All other transitions are collected as part of coverage collection.

13.5 Coverage Extension

Coverage groups and coverage item can be extended to add new items, or to change or add new option settings. Coverage extension is discussed in the following sections.

13.5.1 Coverage Group Extension

An existing coverage group can be extended using the **is also e** keyword. A coverage group can be defined as an empty in the beginning and extended later by adding the declarations of new items. This is shown in the example below.

```

1  :  <
2  :  struct inst {
3  :      event trans_done;
4  :      cover trans_done is empty;
5  :  };
6  :  >

1  :  <
2  :  extend inst {
3  :      len: uint;
4  :      data: byte;
5  :      cover trans_done is also {
6  :          item len;
7  :          item data;
8  :      };
9  :  };
10 :  >

```

13.5.2 Coverage Item Extension

Coverage items are extended by **using also e** keyword. Use of **using also** to extend or change a **when**, **illegal**, or **ignore** option, a special variable named **prev** is automatically created. This variable holds the result of all previous **when**, **illegal**, or **ignore** calculations before the current extension. Therefore **prev** can be used to extend the boolean expressions for **when**, **illegal**, or **ignore** options before the current extension of the coverage item.

The default or user assigned name of a coverage item is used for extending a coverage item. The default name for each coverage item type was described in its corresponding section.

Consider the following coverage group definition:

```

1  :  <
2  :  struct packet {
3  :      length: uint (bits: 12);
4  :      type : [ETH , UART, USB];
5  :      event send;
6  :      cover send is {
7  :          item length;
8  :          item type;
9  :          cross length, type;
10 :          transition type;
11 :      };

```

```

12 : };
13 : >

```

The definition of **send** coverage group is extended in the following code segment. The simple item **length** is extended to use a **HEX** radix for report outputs. The cross item is extended using its default name to indicate that **ETH** is an illegal type during cross coverage collection. Transition item is extended by using its default name where a transition from **USB->ETH** is defined to be illegal.

```

1 : <'
2 : extend packet {
3 :     cover send is also {
4 :         item length using also radix = HEX;
5 :         item cross__length__type using also illegal = (type == ETH);
6 :         item transition__type using also illegal = (prev_type == USB and type == ETH);
7 :     };
8 : };
9 : >

```

The definition of **send** coverage group is further extended in the following code fragment. Simple item length is extended to use a **BIN** radix for report outputs. The cross item is extended to add a new option for illegal cross values. In this case, type **UART** is defined to be illegal in addition to all previously calculated illegal conditions (i.e. **prev** means **type == ETH**). The transition item is extended by including transaction **USB->UART** to the illegal list.

```

1 : <'
2 : extend packet {
3 :     cover send is also {
4 :         item length using also radix = BIN;
5 :         item cross__length__type using also illegal = (prev and type == UART);
6 :         item transition__type using also
7 :             illegal = (prev or (prev_type == USB and type == UART));
8 :     };
9 : };
10 : >

```

13.6 Minimizing Coverage Collection Overhead

In order to collect the necessary coverage information, the coverage engine continuously samples data values throughout the program runtime and stores the sampled data in a database. The collected data consists of a counter for each bucket of each coverage item where the counter is incremented every time a bucket is hit during the program runtime. It is clear that if left unchecked, the storage space for storing coverage results and coverage program runtime overhead can lead to inefficiencies in maintaining and using the verification environment. It is therefore important to follow guidelines to limit these overheads.

The size of collected data is directly proportional to the number of buckets defined in the coverage model. Therefore, the amount of stored data is directly proportional to:

- the number of buckets for scalar coverage items
- the product of the number of buckets for each member of a cross coverage item
- the square of the number of buckets for the base item of a transition coverage item
- the number of instances of a struct if coverage information for that struct is collected for each instance

Given the above observations, data storage can be reduced by:

- Making sure that all coverage item definitions are really needed for coverage measurement.
- Reducing the number of buckets for scalar types using the **ranges** option. Reducing the number of buckets for scalar coverage items will directly lead to reducing the number of buckets for cross and transition coverage items.
- Reducing the number of buckets for scalar coverage items by using the **ignore** option. Using the **ignore** option directs the coverage collector to bypass coverage collection for buckets specified by the **ignore** option parameters. Reducing the number of buckets for scalar items will indirectly reduce the number of buckets for composite items using this scalar item. Consider three scalar coverage items A, B, and C with number of buckets n_A , n_B , and n_C ; a cross coverage item D which crosses A, B, and C; and a transition coverage item E which is defined in terms of item A. Removing one bucket from the definition for item A, removes $(n_B * n_C)$ buckets for coverage item D and $(2 * n_A - 1)$ buckets from item E.
- Reducing the number of buckets for composite coverage items by using the **ignore** option.
- Using the **no_collect** option to remove coverage collection for scalar items that are only used to construct composite coverage items.
- Collecting per-instance coverage only for units which cannot be generated during program runtime (static number of instances), or only for structs that are generated only a few times during the program runtime.

Program runtime overhead is directly proportional to:

- Frequency of sampling event for coverage groups.
- Whether or not a coverage item is sampled at its sampling event based on its filter settings.

Program runtime overhead is minimized by:

- Minimizing the amount of data that needs to be collected.
- Judiciously selecting sampling events.
- Using the **when** option for coverage groups to minimize the number of times a coverage group items are sampled
- Using the **when** option for coverage items to minimize the number of times a coverage item is sampled.

13.7 Summary

This chapter introduced the concepts of coverage groups, coverage items, and coverage buckets and described their use in implementing a coverage model. Each of these constructs allows the user to specify options for controlling the behavior of that construct. These options were presented along with their use in creating a coverage model.

Composite coverage items were introduced as an extension of scalar coverage items to collect coverage about multiple data values while using cross coverage items, or collecting coverage about transitions in the value of a data object using the transition coverage item. Coverage extension was presented to facilitate the application of aspect oriented programming style to coverage modeling. Constructs for extending coverage groups as well as coverage items were presented. Approaches for reducing the impact of coverage collection to program runtime performance were also introduced with guidelines for reducing the amount of collected data.

This chapter focused on describing the basics of coverage collection and their implementation in the *e* language. Coverage collection methodology is described in the next chapter.

This page intentionally left blank

Chapter 13 described coverage concepts and the utilities provided in *e* that support coverage collection facilitating the implementation of coverage collection and analysis of the collected data.

This chapter will describe coverage modeling. A *Coverage Model* includes all aspects related to coverage data collection and handling. Developing a coverage model consists of the following phases:

- Coverage Planning and Design
- Coverage Implementation
- Coverage Grading
- Coverage Analysis

During coverage planning, a coverage plan is derived from the verification plan and its element details are identified. During the coverage implementation phase, the coverage plan is implemented so that during simulation runs, coverage data is collected. Coverage grading describes the approach used to specify quantitative measures for simulation goals, and computing the coverage grade which give a measure of the portion of simulation goals that have been reached. During coverage analysis phase, the results of coverage grading is used to identify enhancements to the simulation environment and simulation rerun constraints that will lead to improving DUV coverage as specified in the coverage plan.

Section 14.1 describes coverage planning and design. Coverage implementation is discussed in section 14.2. Coverage grading and analysis are discussed in sections 14.3 and 14.4 respectively.

14.1 Coverage Planning and Design

A coverage plan is derived from a verification plan. Detailed steps for developing a verification plan were described in section 3.1. Given a verification plan, a *Coverage Plan* describes the information necessary to identify whether or not a specific verification item or scenario has occurred. A coverage plan is only focused on detecting the occurrence of verification scenarios, with the implicit assumption that the environment checkers will confirm correct device operation for each observed scenario.

In the *Coverage Design* phase, information necessary to implement the coverage plan is defined. This information includes:

- What elements of environment or design represent plan attributes
- When to sample based on events in the environment or design
- Where in the environment architecture should each coverage be placed

The code necessary to capture the elements of a coverage plan is developed in the *Coverage Development* phase.

The structure of an ethernet packet was described in section 7.2. A partial verification plan for an ethernet port is shown in table 14.1. This verification plan describes the type of scenarios that should be observed on the DUV receive port (i.e. BFM transmit port), and consists of 4 columns. Column 1 lists the destination address for the packet (multicast or unicast). Column 2 lists the packet types that should be injected into the device port for each destination type. Column 3 lists the ranges of interest for each destination and packet type. Note that the required ranges for QTAGGED and SIZED packets are different due how ethernet packet types are defined. Column 4 lists the LANID field that is only valid for QTAGGED type packets. In this verification plan, only LANID values of 1 and 2 should be generated. Column 5 gives a brief description of each verification item.

The coverage plan and its design are shown on table 14.2. The first column describes the item that should be included in the coverage implementation. Column 2 lists the buckets that define ranges of interest for each item. Each bucket is described with a range and a condition. The condition refers to the packet type, which in this case, impacts the range of interest for packet size. Information regarding the required number of hits for each bucket may need to be specified with each bucket. Column 3 lists the location where coverage item should be sampled; in this case the BFM that transmits a packet to the DUV. Column 4 lists the time when the coverage should be collected. Here, coverage is collected when a packet is successfully transmitted to the DUV, which implies that incomplete packet transmissions (i.e. due to collisions) are not counted as part of coverage measurement. The actual coverage implementation will consist of cross and transition items that is created by combining the items listed in table 14.2.

Table 14.1: Partial Verification Plan for an ethernet Port

Destination	Type	Size	LANID	Description
Unicast	SIZED	<46	--	Small Illegal
		46	--	Small Corner-case
		[47..1535]	--	legal
		1536	--	Large Corner-case
		>1536	--	Large Illegal
	QTAGGED	<44	1,2	Small Illegal
		44	1,2	Small Corner-case
		[45..1533]	1,2	legal
		1534	1,2	Large Corner-case
		>1534	1,2	Large Illegal
Multicast	SIZED	1536	--	Large Corner-case

14.2 Coverage Implementation

Coverage model implementation depends on the source of, and the relationship between the data objects sampled during coverage collection. These considerations and their corresponding coverage implementation are described in the subsequent sections. Section 14.2.1 describes the implementation of different coverage model organizations that describe the relationship between coverage data items. Section 14.2.2 discusses implementation of coverage collection for different data sources (i.e. DUV signals, generated data, state machine behavior).

14.2.1 Coverage Model Organization

Coverage model organization is divided into hierarchical models and multi-dimensional models. A *Hierarchical Coverage Model* is used to collect coverage in cases where a value X is only valid for particular values of a value Y (i.e. **LANID** is only defined when ethernet packet type is **QTAGGED** in table 14.2). A *Multi-dimensional Coverage Model* is used to collect coverage in cases where values for data object X have different meanings for each value of Y. (i.e. small corner case condition for **SIZED** packet is considered legal size packet for **QTAGGED** packet in table 14.2). These models are described in the following subsections.

Table 14.2: Partial Coverage Plan Design for an ethernet Port

Item	Buckets		Location	Sampling Time
	Condition	Range		
destination	none	[Multicast, Unicast]	XMT BFM	Upon completion of packet xmt in XMT BFM
type	none	[SIZED, QTAGGED]	XMT BFM	Upon completion of packet xmt in XMT BFM
size	SIZED	[1..45]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[46]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[47..1535]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[1536]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[1537..]	XMT BFM	Upon completion of packet xmt in XMT BFM
	QTAGGED	[1..43]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[44]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[47..1533]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[1534]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[1535..]	XMT BFM	Upon completion of packet xmt in XMT BFM
LANID	QTAGGED	[1]	XMT BFM	Upon completion of packet xmt in XMT BFM
		[2]	XMT BFM	Upon completion of packet xmt in XMT BFM

14.2.1.1 Hierarchical Coverage Models

A hierarchical coverage model is used to collect coverage when definition of a coverage item or a coverage group is only meaningful under conditions defined by other parameters in the environment. Hierarchical coverage models are implemented using struct subtypes.

In the example below, struct **instruction** has a determinant field **size**, which can be **WIDE** or **REGULAR**. For each subtype of this struct, the definition of cov_sample coverage group is extended to include coverage items that are relevant only for their corresponding subtype.

```

1  : <'
2  :   struct inst {
3  :       size: [WIDE, REGULAR];
4  :       event cov_sample;
5  :       cover cov_sample is {};

```

```

6 : };
7 : '>'
|
|-----|
|
1 : <'
2 : extend inst {
3 :     when WIDE inst {
4 :         cover cov_sample is also {
5 :             item len: uint (bits: 3) = sys.len;
6 :             item data: byte = data;
7 :             item mask;
8 :         };
9 :     };
10 :     when REGULAR inst {
11 :         cover cov_sample is also {
12 :             item reg_addr;
13 :             item wr_data;
14 :             item rd_data;
15 :         };
16 :     };
17 : };
18 : '>'
|-----|

```

New coverage groups can be introduced for struct subtypes. Such coverage groups contain coverage items relevant only to their corresponding struct subtype. All sampling events for conditional coverage groups should be declared in base definition of a struct and not in its subtype.

In the example below, the struct **operation** includes the determinant field **opcode**. Coverage group **ready3** is defined on line 20 only for subtype **ADD** of struct operation. Note that event **ready3**, which is used as sampling event for this coverage group is defined in the base definition of the struct.

```

|-----|
1 : <'
2 : struct operation {
3 :     opcode: [ADD, SUB];
4 :     op1: uint;
5 :     op2: uint;
6 :     op3: uint;
7 :
8 :     event ready is rise('top.ready');
9 :     event ready3 is rise('top.op3ready'); // Must define here
10 :
11 :     cover ready is {
12 :         item op1;
13 :         item op2;
14 :         cross op1, op2;
15 :     };
16 : };
17 : extend operation {
18 :     when ADD operation {
19 :         //event ready3 is rise('top.op3ready'); // Can't define here
20 :         cover ready3 is {
21 :             item op1;
22 :             item op2;
23 :             item op3;

```



```

24 :           cross op1, op2, op3;
25 :           };
26 :   };
27 : };
28 : '>'

```

14.2.1.2 Multi-dimensional Coverage Models

The true meaning of data samples collected during coverage collection are often clear only in combination with other data values during the simulation. Thus, for example, the ethernet packet in section 7.2, the packet sizes that indicate corner case conditions are different depending on the packet type being transmitted. Also, when collecting coverage data on CPU instructions, the value of instruction parameters should only be considered in correlation to the instruction that is being executed as the corner case conditions for add and divide operations are different.

Cross items are used to implement multi-dimensional coverage models. In the example below, the items **opcode**, **op1**, and **op2** are defined in the coverage group **inst_driven**. The item **opcode** is constrained to have only two values **ADD**, **SUB** and **op1** to have values **REG0** and **REG1**. The coverage item **op2** can have values in the range [1..24] with buckets defined for the ranges [1..4], [5..8], [9..12], [13..16], and a bucket for the remainder range [17..24].

```

1 : <'
2 : type cpu_opcode: [ADD, SUB, OR, AND, JMP, LABEL];
3 : type cpu_reg: [REG0, REG1, REG2, REG3];
4 : struct instruction {
5 :     opcode: cpu_opcode;
6 :     keep opcode in [ADD, SUB];
7 :     op1: cpu_reg;
8 :     keep op1 in [reg0, reg1];
9 :     op2: byte;
10 :    keep op2 in [1..24];
11 :    event inst_driven;
12 :    cover inst_driven is {
13 :        item opcode;
14 :        item op1;
15 :        item op2 using ranges = {range([1..16], "", 4)};
16 :        cross opcode, op1, op2 using name = opcode_op1_op2;
17 :    };
18 : };
19 : '>'

```

Results of coverage collection for 10 samples of cross item are shown below.

Sample	opcode	op1	op2
1	ADD	reg0	2
2	SUB	reg1	3
3	ADD	reg1	16
4	SUB	reg0	1
5	SUB	reg1	12
6	ADD	reg0	9
7	ADD	reg0	17

8	SUB	reg0	15
9	SUB	reg1	10
10	ADD	reg1	11

This data can be organized as shown in figure 14.1 to get a clear understanding of the registers and data values that were used with each instruction type.

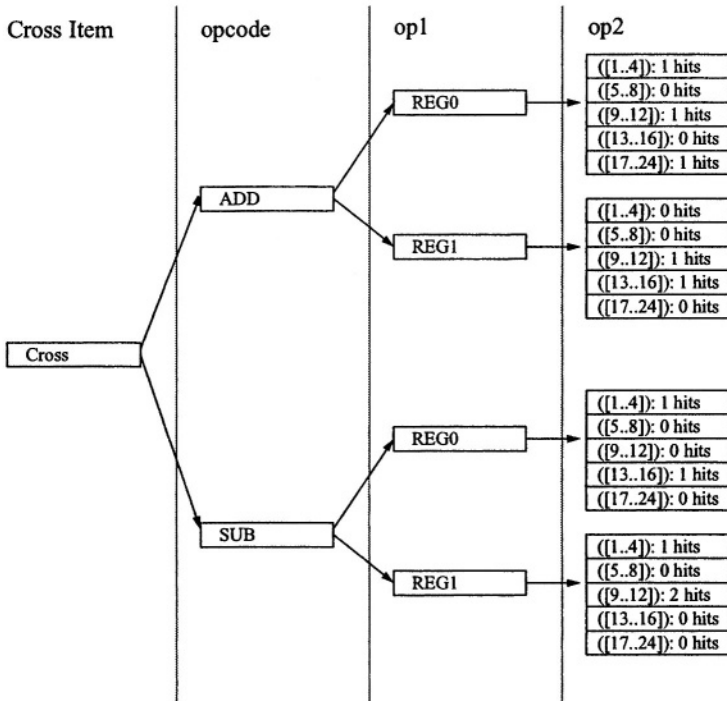


Figure 14.1 Multi-dimensional Coverage Model

14.2.2 Coverage Data Source

Coverage data is collected through sampling of data values that correspond to coverage items. The type of collected data can range from data values in the *e* program to DUV signals. In addition, the data that is collected may correspond to composite constructs in the verification environment or the DUV. Based on these considerations, coverage item implementation is categorized into three types: DUV signal coverage, generated data coverage, and finite state machine coverage. These implementations are discussed in the following subsections.

14.2.2.1 DUV Signal Coverage

All verification activities are directed at driving DUV ports and collecting and checking DUV ports and internal signal values. As such, collecting coverage on DUV signal values is an important part of coverage collection. Issues regarding coverage collection for DUV signals are discussed in this section.

The first step in collecting DUV signal coverage is to identify all important DUV signals that need to be sampled during coverage measurement. Special attention must be paid to the following signal types:

- Main Interface Signals
- Control Signals
- State Variable Signals
- Signals Indicating Exceptions (i.e. overflow, underflow)
- Interrupt Signals

Sometimes it is necessary to collect coverage on a condition computed from a number of DUV signals. In such a case, it is best to create a variable in the *e* program and assign it based on DUV signal values and collect coverage on this variable in the *e* program.

The second step is to define a sampling event to sample the DUV signals. This sampling should be based on another DUV signal like a clock, or a signal that indicates a specific condition in the design (i.e. register load, state transition, etc.). Although an existing event definition in the verification program can be used for collecting coverage, it is better to define a new event solely for coverage collection. Using a dedicated event keeps the coverage environment separate from the rest of the verification environment.

Some examples of DUV signal sampling are shown below:

- Sampling one signal where the sampling event is the change in the signal itself

```
event cov_valid is change ('top.controller.valid') @sim;
cover cov_valid is {
    item valid: bit = 'top.controller.valid';
};
```
- Sampling more than one signal at the same time. All coverage items in this case are sampled based upon a change in signal **top.controller.valid**.

```
item ready: bit = 'top.controller.ready';
item read: bit = 'top.controller.read_sig';
item write : bit = 'top.write_sig';
```

In the last step, coverage groups containing coverage items for DUV signals are placed in coverage groups located in the unit that interacts with the DUV level of hierarchy that contains the signals being sampled. By placing the coverage groups in the appropriate unit, coverage can be collected for all HDL signals corresponding to different instances of that unit. In this case, coverage should be collected for each instance of that unit. Define any cross or transition items for the combinations of values according to the coverage based test plan.

```

1 : <
2 : struct st_controller {
3 :     event cov_valid is change ('top.controller.valid') @sim;
4 :     cover cov_valid is {
5 :         item valid: bit = 'top.controller.valid' using per_instance;
6 :         item ready : bit = 'top.controller.ready';
7 :         item read : bit = 'top.controller.read_sig';
8 :         item write : bit = 'top.write_sig';
9 :     };
10 : };
11 : >

```

14.2.2.2 State Machine Coverage

State machines are commonly used to implement DUV controllers. Measuring coverage on all visited states and transitions of a state machine is an important steps in coverage collection.

The first step in collecting coverage on a state machine is to establish the relationship between the states of the state machine and the equivalent state variable in the *e* code. This is accomplished by using the HDL define-file to declare an enumerated type in the *e* program corresponding to valid states of that state machine. Consider the state machine shown in figure 14.2 and the following HDL define-file that assigns constants to strings in the HDL code:

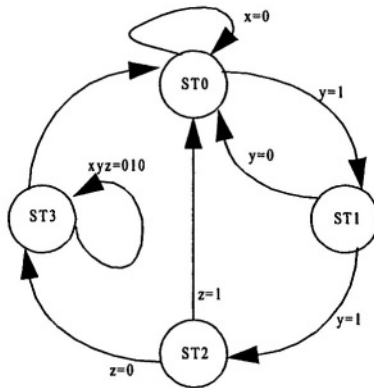


Figure 14.2 State machine coverage collection

```

// file rtl_define.v
`define state0 2'b00
`define state1 2'b01
`define state2 2'b10
`define state3 2'b11

```

Import the above HDL file into the *e* program and use the defined values to declare a new enumerated type:

```

1 : <'
2 : verilog import rtl_define.v
3 : type my_state : [ST0='state0, ST1 = `state1, ST2='state2, ST3='state3];
4 : >

```

Next, define the struct/unit in which this enumerated type is used:

```

1 : <'
2 : struct state_xyz {
3 :     st : my_state;
4 : };
5 : >

```

Now, declare the sampling event based on the signal that causes the state machine to change its state. This signal is usually the state machine clock.

```

1 : <'
2 : extend state_xyz{
3 :     event st_cov is rise('top.xyz.clock')@sim;
4 : };
5 : >

```

Next, define the cover group based on this sampling event and declare a coverage item for the state:

```

1 : <'
2 : extend state_xyz {
3 :     cover st_cov {
4 :         item st;
5 :     };
6 : };
7 : >

```

State transitions are an important part of state machine coverage collection. The following example shows how to use the **transition** operator to collect coverage on state transitions while checking that illegal transitions do not occur for this coverage item.

```

1 : <'
2 : type machine_state: [START, LOAD1, LOAD2, EXEC, STOP];
3 : extend sys {
4 :     xyz : state_xyz ;
5 :     is_illegal_transition(ps :my_state, cs: my_state) :bool is {
6 :         result = ps==ST0 and cs==ST2 or
7 :             ps==ST0 and cs==ST3 or
8 :             ps==ST1 and cs==ST1 or
9 :             ps==ST1 and cs==ST3 or
10 :            ps==ST2 and cs==ST1 or
11 :            ps==ST2 and cs==ST2 or
12 :            ps==ST3 and cs==ST1 or
13 :            ps==ST3 and cs==ST2;

```

```

14 :     };
15 : };
16 : struct state_xyz {
17 :     event st_cov;
18 :     st: my_state;
19 :     cover st_cov is {
20 :         item st;
21 :         transition st using illegal = sys.is_illegal_transition(prev_st, st);
22 :     };
23 : };
24 : >

```

In this example, struct **state_xyz** is instantiated under **sys**. A method is also defined in **sys** that detects whether a transition is illegal. The **transition** item on line 28 uses this method to flag illegal transitions during coverage collection. Note that the boolean expression defined for the **illegal** option is computed in the context of **global** and therefore an absolute path for the method **is_illegal_transition()** is required. For this reason, method **is_illegal_transition()** is declared under **sys**.

14.2.2.3 Coverage of Generated Data

Collecting coverage on the randomly generated DUV stimulus is an important part of coverage collection. Not only does it provide data on the covered verification scenarios, but also the effectiveness of the implementation of the random stimulus generation steps. Another important part of collecting this coverage data from the generated data is to determine the time for sampling data values. Data generated and fed continuously to a DUV, such as packets, transactions, or instructions should be sampled when injected into the DUV.

```

1 : <
2 : extend sys {
3 :     i_swch : switch is instance;
4 :     keep i_swch.hdl_path() == 'top.switch';
5 : };
6 : unit switch {
7 :     event clk is rise('clock');
8 :     event send_pkt;
9 :     i_pkt : uint;
10 :     driver() @clk is {
11 :         while TRUE {
12 :             if (valid == 1) {
13 :                 gen i_pkt;
14 :                 send(i_pkt);
15 :                 emit send_pkt;
16 :                 wait [1]*cycle;
17 :             };
18 :         };
19 :     };
20 :     cover send_pkt is {
21 :         item i_pkt;
22 :     };
23 :     run() is also {
24 :         start driver();
25 :     };

```

```

26 : };
27 : >

```

Data values that change infrequently (i.e. configuration options) can be sampled at the start or end of the test. For configuration data define, emit, and use the events that signify these changes. The predefined method **finalize()** is used to emit event **end_of_test** below

```

1 : <
2 : struct uart {
3 :     fifo_mode : bool;
4 :     num_stop_bit: uint(bits:2);
5 :     event end_of_test;
6 :     finalize() is also {
7 :         emit end_of_test;
8 :     };
9 :     cover end_of_test is {
10 :         item fifo_mode;
11 :         item num_stop_bit;
12 :     };
13 : };
14 : >

```

14.3 Coverage Grading

Coverage grading is the process of measuring the portion of coverage goals that have been met in the currently completed simulation runs. coverage grading assigns a grade to each construct in the coverage model (i.e. coverage buckets, bucket sets, items, and groups), and then computes a global coverage grade based on individual coverage grades; the grade for each coverage construct is computed recursively. First a coverage grade is computed for each bucket. Coverage grade for a bucket is computed using its goal setting and its number of hits. The goal for a bucket specifies the number of desired samples for that bucket. Next, the grade for each coverage item is computed using the grades for its bucket(s), and the grade for the coverage group is computed from the grades for its coverage item(s).

Each coverage bucket, item, and group has an assigned weight that indicates the impact of the grade for that construct to the overall grade calculation.

Coverage grades can be computed using two different formulas: a linear formula, and a root-mean-square formula. Syntax for selecting the coverage grade formula is:

```
set_config(cover, grading_formula, linearroot_mean_square)
```

A linear coverage grade assigns the same significance to consecutive hits in buckets whose goals have not been met. The equations for computing linear bucket grades, item grades, group grades, global grade are shown below:

$$\text{linearGrade(bucket)} = \min\left(1.0, \frac{\text{hits(bucket)}}{\text{goal(bucket)}}\right)$$

$$\text{linearGrade(item)} = \frac{\sum \text{linearGrade(bucket)}}{\text{numberOfBuckets}}$$

$$\text{linearGrade(group)} = \frac{\sum (\text{linearGrade(item)} \times \text{weight(item)})}{\sum \text{weight(item)}}$$

$$\text{linearGrade(global)} = \frac{\sum (\text{linearGrade(group)} \times \text{weight(group)})}{\sum \text{weight(group)}}$$

With a root-mean-square formula, earlier hits for buckets whose goals have not yet been met contribute more to the overall grade (i.e. first hit in a bucket affects overall coverage grade most). The equations for computing root-mean-square bucket grades, item grades, group grades, and the global grade are shown below:

$$\text{RMSGrade(bucket)} = \min\left(1.0, \sqrt{\frac{\text{hits(bucket)}}{\text{goal(bucket)}}}\right)$$

$$\text{RMSGrade(item)} = \sqrt{\frac{\sum ((\text{RMSGrade(bucket)})^2)}{\text{numberOfBuckets}}}$$

$$\text{RMSGrade(group)} = \sqrt{\frac{\sum ((\text{RMSGrade(item)})^2 \times \text{weight(item)})}{\sum \text{weight(item)}}$$

$$\text{RMSGrade(global)} = \sqrt{\frac{\sum ((\text{RMSGrade(group)})^2 \times \text{weight(group)})}{\sum \text{weight(group)}}$$

For coverage groups that collect coverage per-instance, each instance is calculated as a separate group.

To summarize, in order to setup coverage grade calculation:

- Assign weights for coverage buckets, items, and groups (default weight is 1).
- Assign goal for coverage buckets (default goal is 1).
- Select coverage calculation formula (default is linear).

Figure 14.3 shows an example of a coverage model hierarchy, the hits for each bucket, and how overall coverage grade is calculated using a linear grade equation and default weight and goal values.

Item and bucket weights, and bucket goals are discussed in more detail in the following subsections.

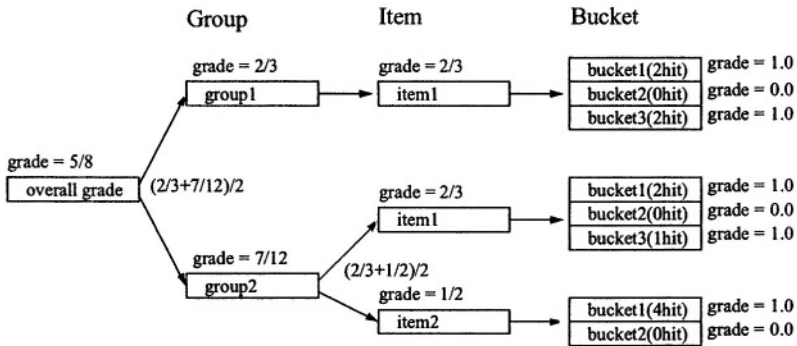


Figure 14.3 Coverage grade calculation for default weight and goals

14.3.1 Changing Default Weights

All coverage group and items have a pre-assigned default weight. The weight of an item determines the importance of that item relative to other items in the coverage group. The weight of a coverage group determines the importance of that group with respect to the other groups in the coverage model. The default weight for any coverage group or item in the coverage model is 1. This weight can be forced to be 0 or to a value greater than 1. Making the weight 0 for an item or group removes the item from coverage grading. Increasing the weight for a group or item increases its contribution to the overall coverage grade. Depending on the verification requirements, some items may also need to be given higher weights than other items.

The weight is assigned using the **weight** option for coverage group or item.

```

1 : <
2 : struct packet {
3 :     addr : uint;
4 :         keep addr in [0x101..0x200];
5 :     length : uint;
6 :     data[10] : list of byte;
7 :     cover cov_pkt {
8 :         item addr;
9 :         item length using weight = 2; // weight of 2 assigned for length.
10 :         item data ;
11 :     };
12 : };
13 : >

```

Figure 14.4 shows an example of grade calculation using modified weight settings for coverage items and groups.

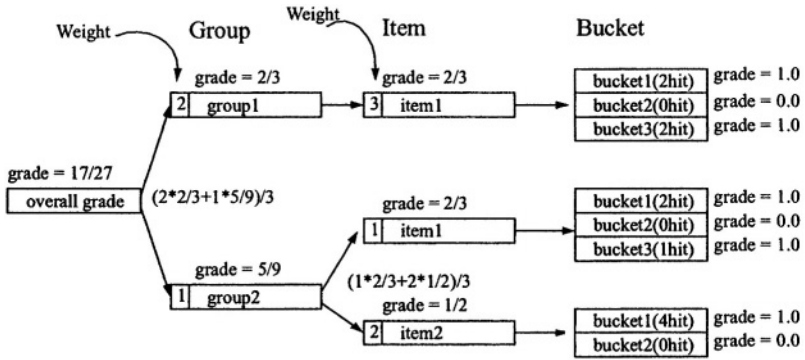


Figure 14.4 Coverage grade calculation for default goals and modified weight

14.3.2 Changing Default Goals

Every coverage bucket requires a goal number of hits in order to be considered completely covered. The default value for this goal is 1. This goal can be changed using the **at_least** option when defining coverage items, as shown below:

```

1 : <'
2 : struct inst {
3 :     op2: byte;
4 :     keep op2 in [1..24];
5 :     event inst_driven;
6 :     cover inst_driven is {
7 :         item op2 using ranges = {range([1..16], "", 4)}, at_least = 2;
8 :     };
9 : };
10 : >

```

Figure 14.5 presents an example of grade calculation for the example in figure 14.4 where bucket goals have been changed from their default value.

14.3.3 Ungradeable Items

Some coverage items may have a very large number of buckets. For example, coverage items of type **int** or **uint** that have no limit over their range of values have a very large number of buckets. Items that have a very large number of buckets are considered Ungradeable and do not participate in coverage grading. An Ungradeable item can be collected but will not contribute towards the coverage by assuming its goal to be 0. The maximum limit for the items with no specific limit is set using option like **max_int_bucket**, which specifies a maximum value for

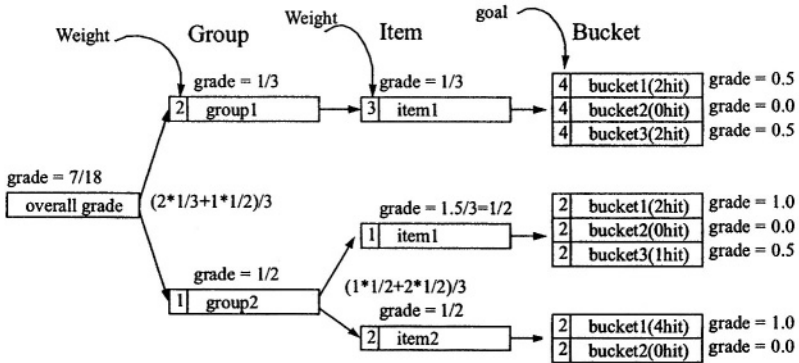


Figure 14.5 Coverage grade calculation for modified goals and weight

the number of buckets, which can be assigned for each item. If this limit is exceeded, the item becomes ungradeable.

The following guidelines are used to differentiate between gradeable and ungradeable coverage items:

- For simple items, boolean and enumerated types are always gradeable
- String types are always ungradeable
- Coverage item of type **int** and **uint** are gradeable only if the range of possible values is less than the maximum configurable value for the coverage tool (default is 16).
- For cross item, items containing ungradeable items are also ungradeable.
- For transition items, items that have transitions of gradeable items are gradeable and transition items from ungradeable items cannot be graded.

Use of these guidelines is shown in the following example:

```

1 : <
2 : extend packet {
3 :   cover cov_packet is {
4 :     item x: byte = x;           // x is ungradeable (256 values)
5 :     item y: byte = y using     // y becomes gradeable (8 buckets: [0..31], etc.)
6 :       ranges = { range([0..255], "", 32)};
7 :     item z: uint(bits: 3) = z; // z is gradeable (8 values)
8 :     item xyz : int = xyz;      // xyz is ungradeable 2**32 values possible
9 :     cross x,y;                 // ungradeable as x is ungradeable item
10 :    cross y,z;                  // gradeable as both items are gradeable.
11 :    transition x;               // ungradeable as x is ungradeable item
12 :    transition z;               // gradeable as z is gradeable.
13 :   };
14 : };
15 : >

```

14.3.4 Illegal and Ignored Items

Coverage items that are declared illegal or ignored using coverage item options, do not contribute to any coverage grading because the goal set for those items is 0.

14.4 Coverage Analysis

Coverage grades identify coverage holes that still remain in spite of the simulation runs that have already been completed. Such coverage grades can be used to:

- Identify inefficiencies in the coverage plan
- Decide whether or not to run further simulation cycles using the current constraints for random generation to improve coverage
- Identify changes in the generation constraints or enhancements to the verification environment that would cover missing scenarios

Careful studying of individual values for item and group coverage grades identify coverage holes for the current set of simulation runs. Given a coverage hole, the flow chart in figure 14.6 shows the methodology used to improve the coverage.

After a simulation run, coverage grades are computed and coverage holes identified. The first step in coverage analysis is to decide whether or not this coverage hole represents an important verification scenario. If the represented case is not important or relevant, the coverage model should be fixed. In some cases, the **ignore** option can be used to exclude the coverage hole from consideration during coverage grade calculation.

If the coverage hole corresponds to a relevant case, then that hole may be covered if the simulation is run for additional cycles. It is important that the missing case occurs with good probability using the existing random generation constraints; otherwise, many simulation runs will be wasted before the coverage hole is covered. If running more cycles will cover the missing case, then more simulation cycles should be run.

If running more simulation cycles does not cover the missing case with good probability, then the environment has to be analyzed to examine if changing the generation constraints is sufficient to cover the missing case. If not, then the environment has to be enhanced so that with new generation constraints, the missing case is generated. Either way, new generation constraints are added and specified before running new simulation cycles.

The above flow is repeated until no more coverage holes remain. Though this flow is described in terms of one coverage hole at a time, this analysis should be done simultaneously for all coverage holes obtained after each simulation run to cover multiple holes after each new simulation run. Thus, the flow shown in the shaded area in figure 14.6 should be done for all or most of the coverage holes detected after each simulation run. A new simulation run should start only after the necessary changes for all analyzed coverage holes have been made.

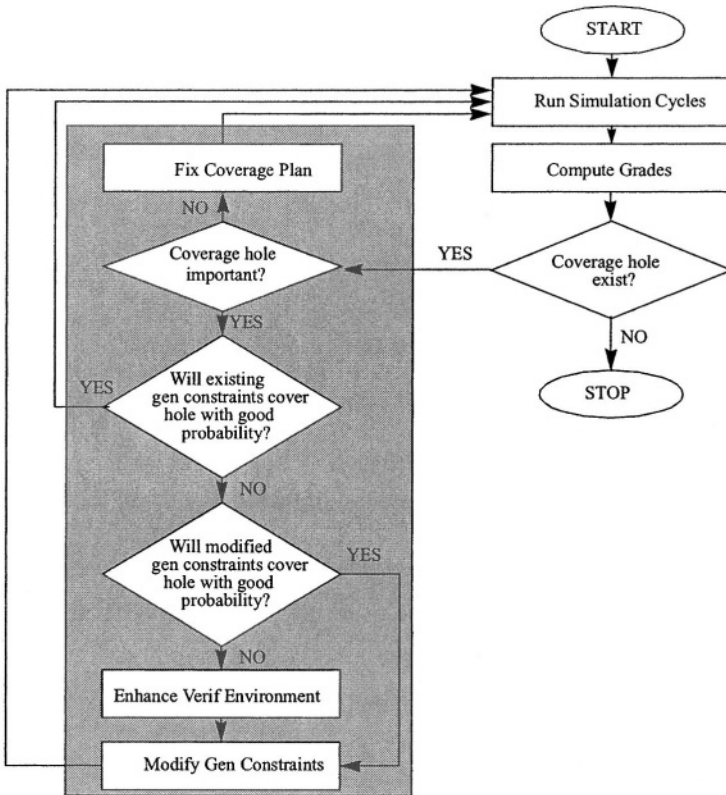


Figure 14.6 Coverage Analysis Flow

14.5 Summary

This chapter presented the flow for building a coverage plan, and how to implement the necessary code to collect coverage according to the coverage plan. Coverage collection styles were discussed as part of the subject of on coverage implementation, and examples were presented for different implementation styles.

Coverage grade calculation, used to measure verification progress, was described using equations in terms of bucket, item, and group goals and weights. The use of coverage grades to refine simulation runs for the purpose of increasing verification coverage was also described.

This page intentionally left blank

PART 6

e Code Reuse

This page intentionally left blank

Code reuse is an important concept in bridging the design and verification productivity gap. A significant portion of design productivity is gained by using existing blocks or off-the-shelf design IPs to construct larger, more complex systems. Applying the same reuse concept to verification of these complex systems makes it possible to significantly reduce verification effort as design size grows.

A design IP is a pre-designed block with well-defined functionality and port behaviors. In practice, any block that can be packaged for later use in the current or a different project can be considered a design IP. Ideally, the verification environment for a design IP would be used in creating the verification environment for the system using this IP. In practice, verification environments for design IPs are actually highly specialized without much consideration for verification reuse. Lack of planning for future reuse leads to difficulties in merging multiple verification environments and complicates adding or reusing verification code; even for slightly different verification requirements. But taking verification reuse into consideration in the early stages of architecting and building a module level verification environment facilitates verification reuse without demanding extra effort for building that environment.

This chapter presents the e Reuse Methodology (eRM), with considerations for building reusable verification environments. Section 15.1 discusses the requirements for building *e* Verification Components. Section 15.2 presents guidelines for preventing conflicts during the deployment of eVCs from multiple sources. Section 15.3 presents architectural features that would make an eVC reusable for a variety of verification requirements. The reuse guidelines discussed in this chapter are described in the context of eVC development. These guidelines should be followed for any verification environment development in order to facilitate its reuse in future projects.

15.1 eVCs: e Verification Components

An *e Verification Component (eVC)* is a configurable and reusable verification environment that focuses on a specific architecture or protocol (i.e. PCI-Express, ethernet, USB, AMBA, etc.).

An eVC provides a complete verification environment for applying coverage driven verification methodology to the protocol targeted by the eVC. In that regard, an eVC architecture is very similar to the verification environment architecture shown in figure 3.2. An eVC contains a complete set of components for stimulus generation, device response collection, scoreboarding, protocol checking, and coverage collection.

Ideally, an eVC is a plug-and-play verification environment that can be used either as a stand-alone environment or to build larger verification environments. The internal implementation of an eVC is usually not visible to the eVC user, but knowledge of its interface and configuration parameters is sufficient to use an eVC.

At one extreme, productized eVCs focus on a standardized engineering specification that is used in multiple projects and in multiple companies. A productized eVC comes with extensive documentation, well designed and flexible user interface, and an implementation that is usually encrypted in order to hide the intellectual property of the eVC developer. A shareware eVC or a project specific eVC may correspond to a subcomponent of a system level design, where the verification environment for that subcomponent is used during module level design and verification, and later in building the system level verification environment.

An eVC can be used as an independent verification environment, or as a component in a larger verification environment. The component level verification environment composed of smaller eVCs may actually be used to build a system level verification environment. Therefore, any verification environment can be considered an eVC and should be designed with such reuse considerations in mind. The distinction between a verification environment and an eVC is therefore somewhat indistinct. In general, an eVC is targeted for use in more than one verification setting.

Each eVC in a verification environment composed of eVCs from multiple sources should satisfy the following requirements:

- No interference between eVCs
 - No name space collision
 - No complex SPECMAN_PATH or directory dependencies
 - Handles dependencies on common modules
 - No dependencies on different versions of Specman Elite and utilities
 - No timing dependencies
 - No dependencies on global settings
- Common look and feel, similar activation, similar documentation
 - Common way to install eVCs
 - Common way to patch eVCs

- Common tracing and debugging
- Handles DUT errors
- Gets eVC identification
- Waveform viewer data
- Custom visualization
- Common way of specifying simulator-specific material
- Common way to do back-door initialization
- Common programming interface to standard blocks
- Common eVC taxonomy
- Common style of documentation
- Support for combining eVCs (control, checking, layering, etc.)
 - Common way to configure eVCs
 - Common way to write tests
 - Common way to create sequences
 - Common way to do checking
 - Combined determination of end of test
 - Common way to do layering of protocols
 - Common way to do combined coverage
- Support for modular debugging
 - Understands combined constraints
 - Reconstructs the behavior of a single eVC in the verification environment
- Commonality in implementation
 - Common data structures
 - Common eVC testing methodology
 - Common way to use ports and packages

Above requirements are stated for the perspective of the eVC user. These usage requirements translate into guidelines for eVC developers, which are discussed in the following chapters.

15.2 Packages and Package Libraries

Reusable verification components are organized using verification packages and verification libraries. A Verification Package is a grouping that consists of:

- Verification Code
- Documentation
- Examples
- Versioning Information
- PACKAGE_README.txt File

A verification package may contain an eVC or a shareware utility. Reusable verification code is distributed by transferring package directories.

Verification packages have two main types:

- Environment Packages
- Utility Packages

Environment Packages are used to instantiate a new verification sub-environment in the process of creating a verification environment. The environment instantiated in this type of package is defined using the **any_env** predefined unit (section 15.2.6). eVCs and shareware packages are built as environment packages. *Utility Packages* do not provide an environment that can be instantiated, but instead provide useful utilities that can be used throughout the verification environment. Chapter 16 describes a utility package that provides useful additions to the core language constructs.

A Package Library contains multiple verification packages. These packages may be grouped based on vendor, project, version, or another user defined requirement. A package library is placed on the default path that is searched with the **e import** statement. All references to packages inside a package are treated relative to the library path (see section 15.2.3).

15.2.1 Naming Conventions

Package names should be unique across all verification package producers (eVC vendors, shareware donations, utility providers). Other than for the obvious reasons being able to uniquely identify package providers, packages with the same name cannot be placed in the same library directory.

Package name uniqueness is guaranteed by centrally allocating package name prefixes to verification package providers. This central management is currently handled through Verisity Inc. Table 15.1 shows a list of currently assigned name prefixes.

Names inside a verification package should be selected so that a name conflict never arises when loading multiple packages provided by different vendors, or different developers. Name conflicts can happen for two reasons:

- Importing files with the same name, even if in different directories
- **e** program name space conflicts

The **import** statement in an **e** program can only be used for files that have different names even if these files reside in two different directories. If two files with the same name are imported into an **e** program, then the second **import** statement is ignored. It is therefore necessary to guarantee that file names in different packages are distinctly different. Pre-pending all file names with a unique package name guarantees that file names originating in different packages will never be the same.

Object definitions in different packages must always use unique names. If the same unit name is used in multiple packages for different purposes, then loading these packages at the

Table 15.1: Predefined Package Name Prefixes

Prefix	Reserved For
erm	Various eRM utilities supplied by Verisity Inc.
evc	Various general utilities supplied by Verisity Inc., such as evc_util.
ex	Small, example packages.
rf	Specman Elite reflective facility.
shr	Shareware that does not use a company name prefix. For example, if a user wants to donate a register creation package as shareware, s/he can call it “shr_register”.
si	SiMantis Inc. provided eVCs.
sn	Internal Specman Elite entities.
vr	Verisity Inc. provided eVCs

same time will lead to conflicts when parsing the loaded program(s). Identifiers in *e* statements: including **type** statements, **struct/unit** statements, **define** statements, and **extend** statements, always require unique names. Enumeration literals, struct members, and struct methods, however, do not require unique names, as these objects will be qualified by the unique name of the object containing them.

15.2.2 Directory Structure

The directory structure for a package library containing two verification packages is shown in figure 15.1. The library directory contains the verification package directory roots, as well as the file LIBRARY_README.txt (section 15.2.4). The program runtime environment uses this file to read information about the contents of that library. Each package directory contains files PACKAGE_README.txt and demo.sh and subdirectories *e*, examples, and docs. File PACKAGE_README.txt provides information to the environment about the contents of the package (section 15.2.5). File demo.sh is used to run a complete demo of the package. The docs directory contains all the documentation for the package. The examples directory contains examples that showcase the special and default features of the package. One of these examples is the example run by the demo.sh file. The examples directory also contains an EXAMPLES_README.txt file that describes the contents of the examples directory. Additionally, a configuration template file should be included in the examples directory to demonstrate how the user can configure the eVC to fit various requirements. The *e* directory contains all the *e* programs for the package. This directory may be flat, or the root of a directory sub hierarchy. Either of these two structures can be used if correct accessing method (section 15.2.3) is used for the files in this sub-hierarchy.

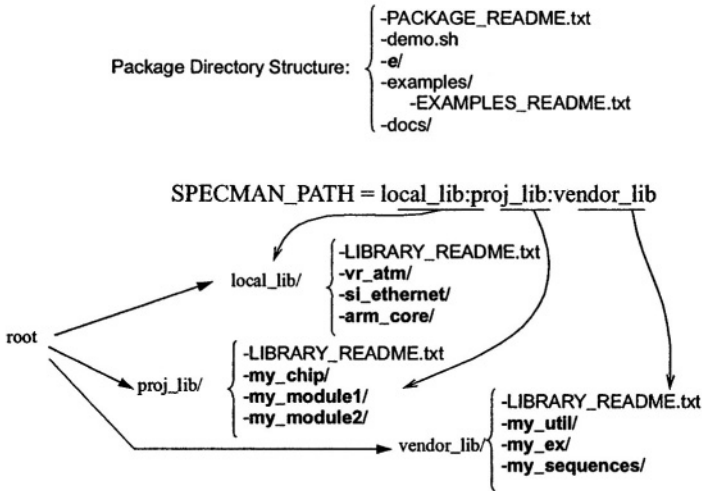


Figure 15.1 Package and Library Directory Structure

A package may contain other directories necessary to organize the package. However these additional subdirectories are used on a case-by-case basis.

15.2.3 Accessing Files

Package relative names should be used in specifying file locations. For example, to import a file named `vr_atm_top.e` in the `e` directory of a package named `vr_atm`, the following import statement is used:

```
import vr_atm/e/vr_atm_top.e;
```

Package `vr_atm` is placed in a package library whose path is already included in the `SPECMAN_PATH` indicating the list of directories to be searched. Using this approach, new packages are added to an existing environment by simply placing the package under a package library already in the `SPECMAN_PATH` list of directories. No changes to `SPECMAN_PATH` will be required. The following guidelines should be followed when specifying imported file:

- Imports within a package should be local (with respect to the `e` directory of that package) as much as possible. This approach will ensure that all files for a package are imported from within the same package even if multiple versions of that package exists on the search path.
- Import statements should never use “`..`” to traverse up a directory hierarchy, as the expected behavior can be different than expected if the directory path includes soft links.
- Package relative paths should always be used for files that reside in a different eVC or

utility package. This approach guarantees that the needed file will always be found regardless of the package library that contains it.

Occasionally, the need may arise to use a different version of a package that is already included in one of the package libraries. *Package Shadowing* is used to achieve this goal. When importing a file, the import statement loads the first copy of that file that it finds in its search paths. By placing the desired version of a package in a library that is defined first in the path list in environment variable `SPECMAN_PATH`, the import statement loads the desired version of that file even though both versions are available on the search path.

To make this file access strategy usable with other tools in the operating systems, the `sn_while.sh` shell script is used to derive the absolute file path name for a package relative file name.

15.2.4 LIBRARY_README.txt File

The format of this file is:

```
1 : * Title: This is the project-wide shareware library.
```

The Title field in this file is used to provide information about the contents of this library.

15.2.5 PACKAGE_README.txt File

This file contains a number of headers that describe the contents of the package. Header names are case and blank insensitive. A header is started with `*` and is followed by a colon. A header line may contain the header as well as a description required for that header, or the header description may continue on the following lines. Headers **Title**, **Name**, and **Version** are mandatory and have to be included in all `PACKAGE_README.txt` files. The following listing shows an example of header types and their description.

```
1 : * Title: Verisity AHB eVC
2 :   --Should be on one line.
3 :   -- Another example: An IEEE 335 foo-transfer protocol eVC
4 : * Name: vr_ahb_evc
5 :   -- Must be the same as the package name
6 : * Version: 3.3
7 :   -- Can also be e.g.: 0.1 (Experimental Version)
8 : * Modified: 14-Jul-2001
9 :   -- Please use dates in exactly that format
10 : * Category: eVC
11 :   -- Can be eVC, shareware, or utility for now
12 : * Support: evc_support@verisity.com
13 :   -- Where to send requests for support, questions, etc..
14 : * Documentation: docs/user_man.pdf
15 :   -- File name containing the documentation
16 :   -- Note: File names should be specified relative to the package dir
17 : * Release notes: docs/rel_notes.txt
```



```
18 : * Description:
19 :   The Verity AHB eVC is...
20 :   ....
21 : * Directory structure:
22 :   This package contains the following directories:
23 :   e/ - All e sources
24 :   ...
25 : * Installation:
26 :   To install it:
27 :   ....1.
28 : * To demo: run demo.sh
29 :   -- A description of how to run a demo from scratch.: 0.
```

15.2.6 any_env unit

The **any_env** unit is a predefined unit used as the base definition for all environments in a package. For example, the top level eVC environment **vr_xbus_env** for the **vr_xbus** is defined using like inheritance from the **any_env** unit, as shown below:

```
1 : unit vr_xbus_env like any_env {
2 : };
   :
```

At the end of **sys** generation, the runtime environment prints a banner that provides information about the title and version number of all instances of the **any_env** unit. Using this information, it is possible to trace the name and version number of all loaded packages after a verification run is completed.

The **any_env** unit has a number of predefined methods that are used in the execution flow and can be used by the user to modify their behavior. These methods are described in table 15.2.

15.3 Features

The typical eVC architecture is shown in figure 15.2. In this view, the eVC top level contains multiple agents that all interact with the same DUV port. The actual configuration of an eVC is dependent on the DUV port properties with which it interacts. An eVC consists of the following subcomponents:

- eVC Environment
- Agent
- Configuration Settings

Table 15.2: Predefined Methods of any_env

Method	Return type	Description	User Extendable
get_name()	string	Computed from definition file of package.	No
get_file(fname)	string	Return the full file name (starting with “/”) of fname, looking for it under the package directory (as it exists now). For an empty string - return the package directory. Issues an error if it does not exist.	No
ge_title()	string	Return the title of the package. By default, returns PACKAGE_README.txt title.	Yes
get_version()	string	Return the version of that package, as a stringBy default, returns PACKAGE_README.txt version.	Yes
show_banner()	none	Print the banner for this instance of the env. By default, shows a single line, with get_title() and get_version(). User can modify it, adding, for example, copyright notice and a short description of the configuration of this instance.	Yes
show_status():	none	Print the current status of this instance of the env. By default, empty.	Yes
add_wave_info()	none	Add waveform info for that package. By default, empty.	Yes

- Sequence Generator and Driver
- e-Port Interface
- BFM
- Monitor

These subcomponents are described in the following sections.

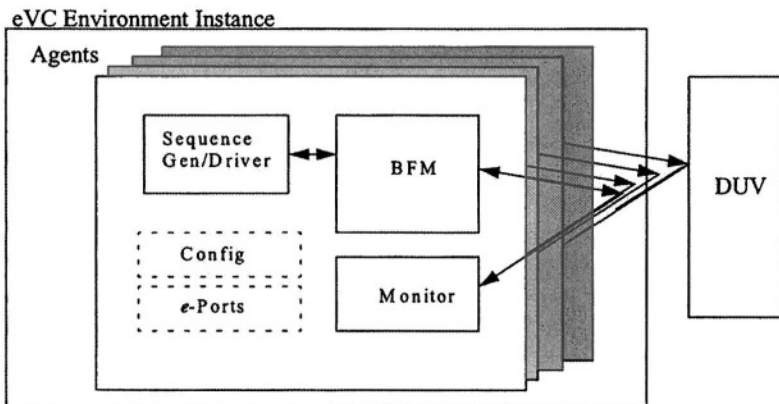


Figure 15.2 eVC Architecture

15.3.1 eVC Environment

An eVC environment instance is created by instantiating the top level environment of an eVC created by using like inheritance on `any_env` predefined unit. One eVC environment instance is created for each DUV port that must be supported by an eVC. If multiple ports with same functionality exist, then multiple instances are created. Each eVC instance communicates with one port of the DUV.

An eVC environment may contain a single or multiple agents. Even when the environment contains only one agent, the environment and agent hierarchy should be maintained.

15.3.2 eVC Agents

An eVC environment may contain multiple agents if it connects with a DUV port that supports multiple agent interaction (i.e. a bus interface). The environment instance for a serial type interface will only contain one agent.

If the environment contains multiple agents for the same port, then it is not necessary to have a monitor for each agent, as all monitors would be monitoring the same port. In this case, either the monitor is moved inside the eVC environment, or only one of the agent monitors is activated.

Agents are classified as active or passive. Active agents are further defined as proactive or reactive agents. *Passive Agents* do not drive any DUV signals. *Active Agents*, on the other hand, drive DUV signals. *Proactive Agents* initiate transactions while *Reactive Agents* only drive transactions only in response to a request.

Agents should use a determinant field to define subtypes corresponding to its active or passive modes. The predefined type `erm_active_passive_t` should be used for this field, as shown below:

```

┌
│
│
│
1  :  --Predefined in evc_util: erm_active_passive_t: [ACTIVE, PASSIVE];
2  :  extend vr_atm_agent_u {
3  :      active_passive: erm_active_passive_t;
4  :  };
│
│
└

```

Passive agents do not require a BFM or a sequence driver. The implementation for a passive agent consists only of a monitor, port interface descriptions, and configuration fields. The active/passive determinant field should be used to include the BFM and the sequence driver only for active agents:

```

      :
      :
1  :   unit vr_atm_agent_u {
2  :       active_passive: erm_active_passive_t;
3  :       monitor: vr_atm_monitor_u is instance;
4  :       when ACTIVE vr_atm_agent_u {
5  :           bfm: vr_atm_bfm_u is instance;
6  :           seq_driver: vr_atm_sequence_driver is instance;
7  :       };
8  :   };
      :
      :

```

15.3.3 Configuration Settings

A group of fields that allow configuration of the agent's attributes and behavior.

15.3.4 Sequence Generator and Driver

The sequence generator is used to generate and drive scenarios using the BFM. The generation of scenarios using sequences is described in detail in chapter 8.

15.3.5 *e*-Port Interface

An eVC interacts with its outside environment through *e*-Ports. The advantage of using *e*-Ports is that no assumptions need to be made about the name or location of port signal connections while the eVC is being developed. During deployment, the port connections are customized according to its deployment requirements. During deployment, the agent ports may be connected to signals in an external simulator, or to ports of an *e*-based module.

eVC ports should be defined in the eVC top-level environment. All agents in the same environment connect to the same signals and can therefore use the common ports defined in the eVC top level environment to interact with the outside device. *e*-Ports are discussed in section 5.5.

15.3.6 BFM

The design and architecture of a verification BFM is described in detail in section 3.2.2.

15.3.7 Monitor

Monitors are used for both passive and active agents. Monitors are independent implementations and do not depend on the BFM in any form. The reason for this is because passive agents only contain one monitor and therefore a monitor should be able to operate independent

of the BFM. Monitors can emit events when they notice interesting things happening in the DUV or on the DUV interface. They can also check for correct behavior or collect coverage. Monitors also only sample signals from the ports and do not drive any values. Monitors are thus passive components.

15.4 Summary

This chapter presented code reuse issues and introduced concepts of *e* Reuse Methodology (eRM) and *e* Verification Components as an embodiment of eRM. Issues related to organization and architecture of eVCs development was presented. This chapter focused mostly on deployment and architectural issues of code reuse. Sequence generation and messaging are also two important considerations in code reuse and these issues are discussed in their corresponding chapters.

This chapter describes the contents of the **si_util** eRM compatible utility package¹. The utilities provided by this package are:

- *Stop-Run Controller*: provides a declarative approach for managing simulation termination conditions across multiple modules. This utility can be integrated into the implementation of an existing verification environment with minimal changes.
- *Memory Package*: provides a memory manager that can manage an *e* based or HDL based memory core. Also provides an *e* based sparse memory core where sparsity can be configured from fully sparse to fully instantiated.
- *Native e Time Manager*: provides a native *e* based time wheel that handles time values in the *e* environment in the absence of an attached simulator. Time management automatically transitions to HDL based time when an HDL simulator is attached.
- *Signal Generator*: provides a declarative approach to generating clocks, resets, and timers. Also includes a centralized mechanism for controlling the stopping and restarting of user defined groups of signals. Uses the time manager utility.
- *Native e Float Arithmetic Package*: Provides a native implementation of IEEE compliant floating point operations that eliminates the need to use non-*e* programs to perform float operations.

The Motivation for these utilities and their usage methodology are described in the subsequent sections.

¹ The **si_util** package may be downloaded from the SiMantis Inc. web site by visiting www.simantis.com.

16.1 Stop-Run Controller

In an *e* program, the run execution phase is completed by calling the predefined **stop_run()** method. Given a verification environment composed of independently developed modules, a centralized mechanism is necessary to decide when the **stop_run()** method should be called.

The *e* language provides an objection mechanism that can be used to synchronize the end-of-test condition between multiple agents by using the predefined **raise_objection()** and **drop_objection()** methods. The Stop-Run Controller mechanism of the **si_util** package provides a modular and declarative approach for managing the end of simulation condition for independently developed modules.

One approach for deciding when to call the **stop_run()** method is to define a stop-run-condition (simulation end condition) for each module of the verification environment hierarchy. In this approach, the stop-run-condition for each module is defined in terms of the stop-run-conditions for its sub-modules and any agents inside that module (i.e. methods, etc.) that might participate in calling **stop_run()**. The formalization of this approach consists of the following abstract objects:

- Stop-run agent
- Stop-run group

A *stop-run agent* corresponds to a module, activity, or event in the verification environment that impacts the decision for calling **stop_run()**. Examples of a stop-run agent include:

- A generator module that requires program execution to continue until the module has generated all its items
- A collector module that requires program execution to continue until it has collected a required number of items.
- A statistic collection activity (i.e. coverage grade, random generation profile, etc.) that requires program execution to continue until it has reached a user defined value

A *stop-run group* defines a grouping of stop-run agents that requires all agents in that group to complete their activity before program execution can be stopped. Multiple stop-run groups may exist corresponding to different requirements. Examples of stop-run groups include:

- Group of all agents that use coverage measurement to decide program termination
- Group of all agents that require generators to complete before program execution is ended

A visual representation of this model is shown in figure 16.1. The verification environment hierarchy shown in this figure contains a number of stop-run agents at different levels of the hierarchy. A possible grouping of these stop-run agents into stop-run groups is also shown. This grouping implies that for the run execution phase to end, either agent F has to terminate, or agents A, B, and D should all terminate, or agents C and F should both terminate.

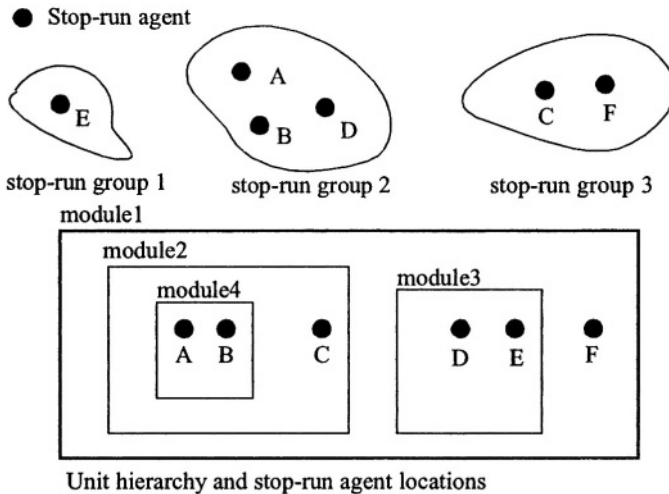


Figure 16.1 stop-run agents and groups

16.1.1 Stop-Run Controller and Stop-Run Interface

The `si_util` package provides a declarative approach for modeling the stop-run model consisting of stop-run agents and stop-run groups. The declarative style of this utility contrasts with the procedural style of using objection mechanisms to implement an end of test strategy. The following predefined structs are provided in the `si_util` package:

- `si_util_stop_run_controller`: controller used for modeling stop-run groups
- `si_util_stop_run_controller_if`: interface used to connect stop-run agents with groups

Details of these structs are shown in tables 16.1, 16.2, and 16.3. A stop-run interface struct is used to attach a stop-run agent to a group. This struct has a method `stop_run()` which is used by each stop-run agent to indicate that it has completed its operation. Each stop-run interface struct includes a pointer to a stop-run controller. Agents are placed in stop-run groups by constraining the controller pointer for each instance of the stop-run interface. Each of these structs also includes a name field that can be constrained to provide meaningful messages as different agents indicate that they no longer require the simulation execution to continue by calling the `stop_run()` method of the interface struct.

16.1.2 Migrating to Using Stop-Run Interfaces

Consider the code fragment shown below:

Table 16.1: si_util_stop_run_controller_if struct member

Struct Member	Type	Valid Range	Description
name	string		Name is used when reporting on the interface status Default: "SRUNCIF"
srunc	si_util_stop_run_controller		Constrained during instantiation to the stop-run controller that this interface belongs to. If not constrained, then internal stop-run controller is used forming a stop-run group consisting of only this interface.

Table 16.2: si_util_stop_run_controller_if methods

Method Name	Parameters	Return Type	Description
stop_run()	none	none	Indicates that the stop-run interface containing this method no longer objects to end of simulation run.

Table 16.3: si_util_stop_run_controller struct member

Struct Member	Type	Valid Range	Description
name	string		Name used when reporting on the interface status Default: "SRUNC"
srunc	si_util_stop_run_controller		Constrained during instantiation to the stop-run controller that this interface belongs to. If not constrained, then internal stop-run controller is used instead; forming a stop-run group consisting of only this interface.

```

1 : <'
2 : unit module1 {
3 :     stop_this_module() is {
4 :         stop_run();
5 :     };
6 : };

```

In the above, the predefined global **stop_run()** method is used to stop the simulation. The migration of this example to an implementation using a stop-run interface is shown below:

```

1 : <
2 : unit module1 {
3 :     srunc_if: si_util_stop_run_controller_if; -- uses its internal stop-run controller
4 :     stop_this_module() is {
5 :         srunc_if.stop_run();
6 :     };
7 : };

```

In the example modification, a stop-run interface is instantiated; but instead of using the global **stop_run()** method, the **stop_run()** method of this interface is used when stopping the simulation on line 5. Since the **srunc** member of **srunc_if** is not constrained to point to any other stop-run controllers, an internal stop-run controller is used inside the interface struct. In the above implementation, the simulation stops with the method call on line 5. This behavior is identical to the code sample using the global **stop_run()** method. Approaches for combining multiple stop-run interfaces into stop-run groups are discussed next.

16.1.3 Multiple Stop-Run Groups in the Same Module

Use of these constructs for stop-run agents at the same level of hierarchy are shown for the example in figure 16.2. In this instance, 4 stop-run agents exist at the same level of hierarchy. These agents are placed in two groups so that the program run phase is completed when both agents A and C, or both agent B and D indicate the end of their need for simulation to continue. As shown in the figure, 4 stop-run interfaces are instantiated for the four stop-run agents. Two stop-run controllers are also instantiated to create the stop-run groups shown in this figure. The *e* program implementing this example is shown below:

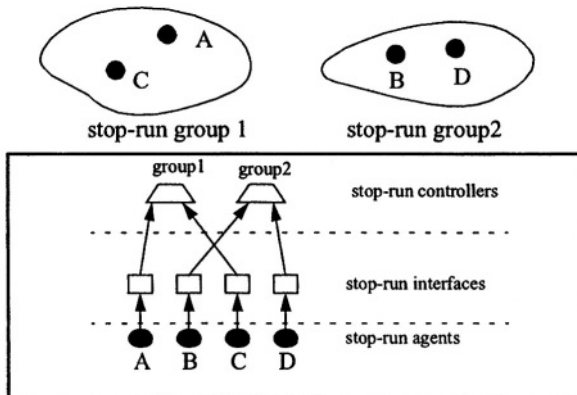


Figure 16.2 stop-run controllers and interfaces

```

1 : <
2 : import si_util/e/si_util_top.e;
3 : unit srunc_demo {
4 :     group1_srunc: si_util_stop_run_controller;
5 :     keep group1_srunc.name == "SRUNC GROUP1";
6 :     group2_srunc: si_util_stop_run_controller;
7 :     keep group2_srunc.name == "SRUNC GROUP2";
8 :
9 :     agent_A_srunc_if: si_util_stop_run_controller_if;
10 :    keep agent_A_srunc_if.name == "SRUNC AGENT A";
11 :    keep agent_A_srunc_if.srunc == group1_srunc;
12 :    agent_B_srunc_if: si_util_stop_run_controller_if;
13 :    keep agent_B_srunc_if.name == "SRUNC AGENT B";
14 :    keep agent_B_srunc_if.srunc == group2_srunc;
15 :    agent_C_srunc_if: si_util_stop_run_controller_if;
16 :    keep agent_C_srunc_if.name == "SRUNC AGENT C";
17 :    keep agent_C_srunc_if.srunc == group1_srunc;
18 :    agent_D_srunc_if: si_util_stop_run_controller_if;
19 :    keep agent_D_srunc_if.name == "SRUNC AGENT D";
20 :    keep agent_D_srunc_if.srunc == group2_srunc;
21 :
22 :    generator() @sys.any is {
23 :        -- generator body here
24 :        wait [3]*cycle;
25 :        agent_A_srunc_if.stop_run();
26 :    };
27 :    collector() @sys.any is {
28 :        -- collector body here
29 :        wait [2]*cycle;
30 :        agent_B_srunc_if.stop_run();
31 :    };
32 :    timer() @sys.any is {
33 :        -- timer body here
34 :        wait [4]*cycle;
35 :        agent_C_srunc_if.stop_run();
36 :    };
37 :    monitor() @sys.any is {
38 :        -- monitor body here
39 :        wait [5]*cycle;
40 :        agent_D_srunc_if.stop_run();
41 :    };
42 :
43 :    run() is also {
44 :        start generator();
45 :        start collector();
46 :        start timer();
47 :        start monitor();
48 :    };
49 : };
50 :
51 : extend sys {
52 :     demo: srunc_demo is instance;
53 : };
54 :
55 : >

```

In the above program, the `si_util` package is imported on line 2. Stop run controllers are instantiated on lines 4 and 6, and their names are assigned using a generation constraint. The

stop run interfaces are instantiated on lines 9, 12, 15, and 18, their names are assigned using constraints and their stop-run group membership is defined using generation constraints where their **srunc** pointers are set to stop-run controllers for their corresponding groups. In this example, the 4 stop-run agents are TCMs corresponding to a generator, a collector, a monitor, and a TCM that end the simulation based on a timer value. For each agent, the **stop_run()** method of their corresponding instance is called. In this example, a wait statement is added to each TCM to mimic program behavior when TCMs take different times to call the **stop_run()** method.

The output generated by the stop-run controllers is shown in the following.

```
Running the test ...
+D->000000000 SRUNC GROUP1 Registered (name, id)=(SRUNC AGENT A, 0)
+D->000000000 SRUNC GROUP2 Registered (name, id)=(SRUNC AGENT B, 0)
+D->000000000 SRUNC GROUP1 Registered (name, id)=(SRUNC AGENT C, 1)
+D->000000000 SRUNC GROUP2 Registered (name, id)=(SRUNC AGENT D, 1)
+D->000000002 SRUNC GROUP2 Unregistered (name, id)=(SRUNC AGENT B, 0)
+D->000000003 SRUNC GROUP1 Unregistered (name, id)=(SRUNC AGENT A, 0)
+D->000000004 SRUNC GROUP1 Unregistered (name, id)=(SRUNC AGENT C, 1)
+D->000000004 SRUNC GROUP1 All users stopped. Now stopping .....
Last specman tick - stop_run() was called
Normal stop - stop_run() is completed
```

As shown above, at time 0, all agents are registered with their corresponding stop-run groups. At time 2, stop-run agent B calls **stop_run()**. At time 4, stop-run agent C calls the **stop_run()** method. At this time, since both agents A and C have removed their requirements that simulation continue, simulation end condition is satisfied for group 1 and run phase of program execution terminates.

16.1.4 Multiple Stop-Run Groups across the Hierarchy

The implementation of the stop-run strategy for the example in figure 16.1 is shown below:

```
1 : <'
2 : import si_util/e/si_util_top.e;
3 : extend sys {
4 :     m1: module1 is instance;
5 : };
6 : unit module1 {
7 :     group2_srunc: si_util_stop_run_controller;
8 :     keep group2_srunc.name == "SRUNC GROUP2";
9 :     group3_srunc: si_util_stop_run_controller;
10 :    keep group3_srunc.name == "SRUNC GROUP3";
11 :
12 :    agent_F_srunc_if: si_util_stop_run_controller_if;
13 :    keep agent_F_srunc_if.name == "SRUNC AGENT F";
14 :    keep agent_F_srunc_if.srunc == group3_srunc;
15 :
16 :    m2: module2 is instance;
17 :    keep m2.group2_srunc == group2_srunc;
18 :    keep m2.group3_srunc == group3_srunc;
19 :    m3: module3 is instance;
20 :    keep m3.group2_srunc == group2_srunc;
21 : };
22 :
```

```

23 : unit module2 {
24 :     group2_srunc: si_util_stop_run_controller;
25 :     group3_srunc: si_util_stop_run_controller;
26 :
27 :     m4: module4 is instance;
28 :     keep m4.group2_srunc == group2_srunc;
29 :
30 :     agent_C_srunc_if: si_util_stop_run_controller_if;
31 :     keep agent_C_srunc_if.name == "SRUNC AGENT C";
32 :     keep agent_C_srunc_if.srunc == group3_srunc;
33 : };
34 : unit module3 {
35 :     group1_srunc: si_util_stop_run_controller;
36 :     keep group1_srunc.name == "SRUNC GROUP1";
37 :     group2_srunc: si_util_stop_run_controller;
38 :
39 :     agent_D_srunc_if: si_util_stop_run_controller_if;
40 :     keep agent_D_srunc_if.name == "SRUNC AGENT D";
41 :     keep agent_D_srunc_if.srunc == group2_srunc;
42 :     agent_E_srunc_if: si_util_stop_run_controller_if;
43 :     keep agent_E_srunc_if.name == "SRUNC AGENT E";
44 :     keep agent_E_srunc_if.srunc == group1_srunc;
45 : };
46 : unit module4 {
47 :     group2_srunc: si_util_stop_run_controller;
48 :
49 :     agent_A_srunc_if: si_util_stop_run_controller_if;
50 :     keep agent_A_srunc_if.name == "SRUNC AGENT A";
51 :     keep agent_A_srunc_if.srunc == group2_srunc;
52 :     agent_B_srunc_if: si_util_stop_run_controller_if;
53 :     keep agent_B_srunc_if.name == "SRUNC AGENT B";
54 :     keep agent_B_srunc_if.srunc == group2_srunc;
55 : };
56 : >

```

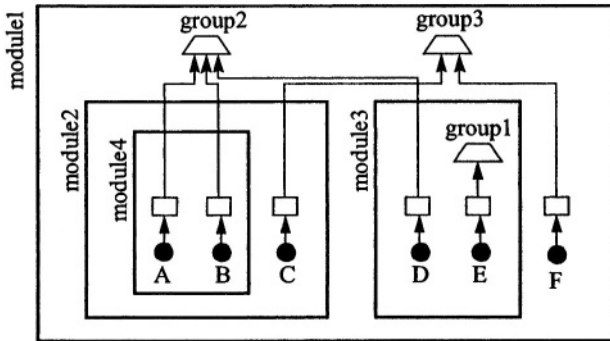
The following observation can be made about this implementation:

- A stop-run interface is instantiated in each unit containing a stop-run agent.
- The stop-run controller for each stop-run group is instantiated in the unit that is the youngest parent of units containing the stop-run agents of that group.
- The pointer for each stop-run group instance is passed to lower levels of the hierarchy by using generation constraints until it is used to constrain the **srunc** field of a stop-run interface.

A visual representation of the above implementation is shown in figure 16.3.

16.1.5 Modular Stop-Run Control

The stop-run controller utility provided by the **si_util** package gives an effective mechanism for merging the stop-run requirements of multiple modules. Consider the following two independently developed modules:



Unit hierarchy and stop-run agents, interfaces, and controllers

Figure 16.3 Hierarchical stop-run agents and groups

```

1  : <'
2  : unit module1 {
3  :   srunc: si_util_stop_run_controoler;
4  :   srunc_if: si_util_stop_run_controller_if;
5  :   keep srunc_if.srunc == srunc;
6  :
7  :   stop_this_module() is {
8  :     srunc_if.stop_run();
9  :   };
10 : };
11 : unit module2 {
12 :   srunc: si_util_stop_run_controoler;
13 :   srunc_if: si_util_stop_run_controller_if;
14 :   keep srunc_if.srunc == srunc;
15 :
16 :   stop_this_module() is {
17 :     srunc_if.stop_run();
18 :   };
19 : };
20 :

```

Now consider an environment built by combining these two modules:

```

1  : <'
2  : extend sys {
3  :   m1: module1 is instance;
4  :   m2: module2 is instance;
5  : };
6  : v'

```

The run execution phase of the above environment will end when either of the two modules call their corresponding **stop_this_module()** method. Each module has its own instance of a stop-run controller which corresponds to different stop-run groups. Since these two stop-run

controllers are not related, program execution ends when either of the two modules exist. To merge the stop-run mechanism for these two modules, the above code can be enhanced as follows:

```
1  : <'
2  : extend sys {
3  :     global srunc:si_util_stop_run_controller;
4  :     keep m1.srunc == global_srunc;
5  :     keep m2.srunc == global_srunc;
6  : };
7  : >
```

With the above enhancement, the stop-run groups of the two modules are merged into one group corresponding to **global_srunc**. The run execution phase completes only when both modules have satisfied their exit conditions.

16.2 Memory Package

Memory management in *e* consists of two packages:

- Memory Manager
- Sparse Memory Core

The memory manager provides the necessary utilities to manage an *e* based or HDL based memory block. This utility provides a mechanism for allocating and de-allocating memory segments of user-defined size. The sparse memory core models a memory block that can be used either in combination with the memory manager or independently. These utilities are described in the next sections.

16.2.1 si_util_mem_mgr Memory Manager

A memory manager is required in any scenario where multiple independent agents use the same memory block for their specific requirements. The memory manager is also needed in situations where memory segments of different sizes must be allocated and de-allocated throughout the simulation runtime. Generally, it is possible to manage a memory block without using a memory manager by manually partitioning a memory block according to the verification or system requirements. The difficulty with manual partitioning, however, is that any future changes in the memory profile would require a complete re-write of the memory partitions. Using a memory manager eliminates the need for manual partitioning, as well as the need to redefine partitions every time the system changes. A memory manager can be configured to randomly place the allocated segments in the memory block, and this leads to better coverage of system memory access mechanisms.

The following services are provided by a memory manager:

- Allocates memory segments with a user defined size
- Defines the granularity of memory allocation (byte, half-word, word, etc.)
- De-allocates (i.e. frees) a previously allocated memory segment to return it to the pool of available memory locations
- Defines reserved memory regions in the memory block so that memory in that region is not used during allocation
- Controls random placement of newly allocated memory segments within the space of available pool of memory

In general, a memory manager is not directly connected to a memory block. It only provides a database of reserved and available memory regions, and the mechanisms to reserve (allocate) and cancel a reserved memory segment (free). The addresses provided by the memory manager are then used to access either an e based or HDL based memory core.

A memory manager is implemented by instantiating the `si_util_mem_mgr` struct. Details of this struct are shown in tables 16.4 and 16.5.

Table 16.4: si_util_mem_mgr struct member

Struct Member	Type	Valid Range	Description
size	uint(bits:64)	powers of 2	Defines the size of memory core being managed by this memory manager Default: 32
alignment	uint(bits:64)	powers of 2, less than size	Gives the alignment for allocated memory. This setting implies the minimum allocated memory segment size (i.e. the minimum allocated size for alignment of 2 is 4 bytes). Default: 1
placement	si_util_mem_placement_style	BINARY_BUCKETS ANYSIZE_BUCKETS	The names correspond to memory management styles. See section 16.2.1.1 for a description. Default: BINARY_BUCKETS

Use of memory manager is shown in the example below:

```

1 : <'
2 : unit memory_module {
3 :     mgr: si_util_mem_mgr;
4 :     keep mgr.size == 2048;
5 :     keep mgr.alignment == 4;
6 :     keep mgr.placement == BINARY_BUCKETS;
7 :     write_to_memory(addr:uint(bits:64), data:byte) is {
8 :         -- write to HDL or e based memory
9 :     };
10 :     read_from_memory(addr:uint(bits:64)): byte is {

```


Table 16.5: `si_util_mem_mgr` methods

Method Name	Parameters	Return Type	Description
<code>alloc()</code>	size: uint(bits:64)	addr: uint(bits:64)	Allocates memory of size passed in as parameter. Returns the address for the allocated memory segment.
<code>alloc_fixed()</code>	addr: uint(bits:64) size: uint(bits:64)	bool	Tries to allocate memory of the given size starting at location pointer at by addr. Returns TRUE if successful, FALSE, otherwise.
<code>free()</code>	addr: uint(bits:64)		Returns the allocated memory segment corresponding to addr to the available memory pool.

```

11 :           -- read data from HDL or e based memory;
12 :       };
13 :       memory_access() is {
14 :           var addr: uint(bits:64) = mem.alloc(48);
15 :           var data: byte = 10;
16 :           write_to_memory(addr, data);
17 :           data = read_from_memory(addr);
18 :           mem.free(addr);
19 :       };
20 : };
21 : >

```

16.2.1.1 Memory Segment Placement Style

Memory managers operate by maintaining a list of free and allocated memory ranges. In the most generic form, a memory manager maintains buckets corresponding to memory ranges of different sizes. A new memory segment can only be allocated from memory ranges in buckets whose size is equal or larger than the requested memory size. When an available memory range is used to allocate a new memory segment, it is removed from its bucket, and the remaining pieces created by taking away the allocated portion are placed in buckets corresponding to the size of these new pieces. De-allocating a memory segment corresponds to returning the allocated memory segment to the pool of free memory ranges and merging it with its neighboring free memory ranges.

Different approaches can be implemented to maintain free buckets and selecting which free memory range a new memory segment is allocated from. Two memory segment placement approaches are provided by the `si_util_mem_mgr` construct:

- **BINARY_BUCKETS**
- **ANYSIZE_BUCKETS**

BINARY_BUCKETS placement style refers to a mechanism where the memory manager maintains only powers-of-2 bucket sizes (i.e. 1,2,4,8, etc.), and breaks free memory ranges into

sizes corresponding to these buckets. For example, if allocating a memory segment of size 8 from an available memory range of 64, it breaks the free memory range into one memory range of size 32, one of size 16, and 2 of size 8. It then uses a memory range of size 8 to allocate the requested memory and places the newly created ranges into their corresponding buckets. Also, for allocating a new memory segment, this placement approach uses only the smallest bucket that can accommodate the requested memory size. This placement strategy has the following properties:

- It leads to very small memory fragmentation and therefore provides very good memory utilization.
- It is efficient both for allocating and de-allocating memory segments.
- Randomization for memory segment placement is limited for the following reasons:
 - The start of memory segments are aligned with powers-of-2 addresses
 - Similar size memory segments tend to be grouped in the same memory regions

ANYSIZE_BUCKETS placement style maintains buckets of any size, and selects a free memory range of any size (i.e. not just the smallest size that can accommodate the requested size) for allocating a new memory segment. The properties of this placement style are:

- It leads quickly to high memory fragmentation so that large memory segments cannot be allocated because the placement of smaller memory segments are so spread-out.
- Memory segment allocation and de-allocation not very efficient because of the method for selecting a free range during allocation, and merging free ranges during de-allocation.
- Randomization is high as a new memory segment can start at any location in the memory space as long as the requested size can be accommodated at that starting address.

In general, it is best to use the **BINARY_BUCKETS** placement style when verifying the memory segment is not the main focus and memory is used as a utility to support the verification activity. The **ANYSIZE_BUCKETS** placement style should be used to focus on verifying the memory operation or a DMA engine that interacts with a memory core.

16.2.2 si_util_mem Sparse e Memory Core

The **si_util** package provides a sparse memory core. The **si_util_mem** construct is used to instantiate an e based memory. Details of this construct are shown in tables 16.6 and 16.7.

The memory core provided in this package is a sparse memory model. This means that not all memory locations are physically present in the memory core. Sparse models are useful in situations where memory cores are very large and physically allocating space for all the memory space is impractical or inefficient. A sparse memory model introduces overheads in space and runtime complexity of the memory model. This overhead is directly proportional to the granularity of this sparsity. For example, granularity of 1 byte indicates that space for each byte is physically present if that byte is used. Granularity of 16 bytes would indicate that a 16 byte page of memory is present if any of its bytes are used.

Table 16.6: `si_util_mem` struct members

Struct Member	Type	Valid Range	Description
size	uint(bits:64)	powers of 2	Defines the size of memory core Default is 32
granularity	uint(bits:64)	powers of 2, less than size	Specifies the size of sparse-blocks that store the memory content. Setting to 1 creates a fully sparse memory core. Setting to size creates a fully instanti- ated memory core. Default: 1
qualify_reads	bool		If set to TRUE, creates a runtime error when reading a memory location that has not been written to. Default: TRUE
default_value	byte		Default value returned when reading locations that has not been written to. Default: 0

Table 16.7: `si_util_mem_mgr` methods

Method Name	Parameters	Return Type	Description
write()	addr: uint(bits:64) data: byte		Writes data to the memory location pointed at by addr.
write_list()	addr: uint(bits:64) data_list: list of byte		Writes the list of data bytes to the memory locations starting at addr.
read()	addr: uint(bits:64)	byte	Returns the byte stored at location addr.
read_list()	addr: uint(bits:64) size: uint	list of byte	Returns the list of bytes stored at locations addr to addr+size-1.

The granularity of `si_util_mem` construct can be configured by constraining the **granularity** struct member of its instance. As indicated in table 16.6, constraining this parameter to 1 leads to a fully sparse memory model, and constraining this parameter to the memory size, leads to a fully instantiated memory where memory is modeled as one chunk of space. The setting of this parameter is application dependent. As a guideline, this parameter can be set to the average size of the memory expected to be allocated in this memory.

The implementation of this memory core is totally independent of the memory manager. Any byte in this memory can be read or written. The feature allows this memory model to be used in environments where the memory access addresses are generated in the DUV.

The following example shows the use of this memory model along with the memory manager example shown in section 16.2.1.

```

1 : <
2 : extend memory_module {
3 :     mem: si_util_mem;
4 :     keep mem.size == 2048;
5 :     keep mem.granularity ==64;
6 :
7 :     write_to_memory(addr:uint(bits:64),data:byte) is only {
8 :         mem.write(addr, data);
9 :     };
10 :    read_from_memory(addr:uint(bits:64)):byte is only {
11 :        result = mem.read(addr);
12 :    };
13 : };
14 : >

```

16.3 Native *e* Time Manager

The goal of any simulation environment is to predict a system's behavior over time. Therefore, time management is the fundamental requirement of any simulation environment. An *e* program is usually connected to an HDL simulator, therefore time management is by default relegated to the simulation kernel.

Time handling in *e* when connected to an HDL simulator is summarized as follows:

- At any simulation tick, **sys.time** reflects the time in the HDL simulator.
- Predefined **delay()** method call is used to support a notion of absolute time intervals (usable only in temporal expressions).
- Activity is synchronized to events derived from HDL signals, therefore inheriting the timing of these HDL signals.

In the absence of an HDL simulator, where every program runtime tick is assumed to represent a constant passage of time. Program behavior in the absence of an HDL simulator is summarized as follows:

- At any program runtime tick, **sys.time** reflects the number of ticks since the beginning of program execution.
- A constant time interval is assumed for a sampling period of **sys.any** event (i.e. time between occurrences of two **sys.any** events is assumed to be a constant value).
- Clocks of different periods are constructed by counting the number of **sys.any** events.
- The predefined **delay()** method cannot be used.

Handling time in the absence of an HDL simulator is very inefficient for anything but the most trivial scenarios. At the root of this inefficiency is the fact that for accurate time representation, the constant delay assumed for the sampling period of **sys.any** should be the largest

common denominator for all time values represented during program runtime. For example, consider generating two clocks with periods of 997 ns and 1009 ns respectively. These time values are prime numbers and the largest common denominator for these periods is 1 ns. This means that in the absence of an HDL simulator, accurate time representation while generating these two clocks requires that the sampling period for each **sys.any** event to represent the passage of 1 ns. This requirement basically leads to a program that can potentially run up to 1000 times slower than a program that handles time values more efficiently.

In addition to the inefficiency, code development and reuse become next to impossible with the strategy described above. During code development, knowing all time values in advance will be required for computing the largest common denominator of these time values. Code reuse also becomes almost impossible because with the addition of any new clock or time calculation requirement, all time passage assumptions have to be recalculated! One approach to solving these problems is to assign the smallest possible time passage to a **sys.any** event. Even though this approach simplifies time handling, it is impractical as it significantly slows down the program.

An *e* program may be used to develop complex environments in the absence of an HDL simulator. Such situations include the following:

- The *e* language is a powerful modeling language that may be used to model a complete device or environment. In such cases, native handling of time is a requirement for effective use of the language.
- Often, verification code is developed without attaching to the DUV or an HDL simulator. This approach holds especially true when developing an eVC. An HDL simulator is attached in the later phases of the development to complete the implementation.

The **si_util** package provides a time management utility that introduces the notion of time in the absence of an attached HDL simulator. This package provides the following features:

- A notion of time passage when no HDL simulator is attached.
- Automatic transition of time handling to the HDL simulator when attached.
- Efficient handling of time management through implementation of a time wheel

The time manager provided by the **si_util** package is automatically instantiated in the global name space as **si_util_time_mgr** and is used mostly to support the implementation of clocks and other time based data structures as described in section 16.4. This utility also maintains a parameter called **si_util_time** in the global name space that holds the current time value as computed by the **si_util** time manager. The method **update_sys_time()** is used to indicate to the environment whether or not the value of **sys.time** should be updated with the value of **si_util_time**. Under a default setting, **sys.time** is updated with the value of **si_util_time**. But since **sys.time** maintains the number of runtime ticks in the absence of an attached HDL simulator, and some legacy codes may depend on this value, it may be necessary to maintain the old behavior for **sys.time**. Methods and TCMs provided by the time manager are detailed in table 16.8.

The following program shows an example of using these methods:

Table 16.8: global.si_util_time_mgr method and TCMs

Method Name	Parameters	Sampling Event	Description
update_sys_time()	flag: bool		Indicates if sys.time should mirror the time value for si_util_time. Default: YES
wait_for()	time_interval: time	sys.any	A TCM that suspends the calling thread for time_interval amount of time
wait_until()	time_value: time	sys.any	A TCM that suspends the calling method until time_value time.

```

1 : <
2 : extend sys {
3 :   run() is also {
4 :     si_util_time_mgr.update_sys_time(TRUE);-- TRUE is the default setting
5 :     start time_consuming_example();
6 :   };
7 :
8 :   time_consuming_example()@sys.any is {
9 :     si_util_time_mgr.wait_until(10 ps);
10 :     print si_util_time: -- prints "si_util_time = 10000 "
11 :     si_util_time_mgr.wait_for(10ns);
12 :     print si_util_time; -- prints "si_util_time = 10010000"
13 :     stop_run();
14 :   };
15 : };
16 : >

```

Although this program terminates at time 10.01 ns, it only consumes two runtime ticks.

16.4 Signal Generator

Periodic and aperiodic signal generation is a common requirement in a verification environment. The **si_util** package provides predefined constructs that allow signals to be created in a declarative fashion. The implementation of these signal generation constructs are based on the time management utility described in section 16.3.

The signal generation facility can generate signals of the following types:

- Periodic: is used to implement clocks
- Pulse: is used to implement reset and other types of pulses
- Edge: is used to implement timers.

In addition, a signal management facility is also provided to provide for stopping and restarting of groups of signals. Examples of such signal groups include clocks that may need to be stopped during a reset pulse, or timers that need to be restarted after a reset is deactivated.

The signal generation utility provided by the `si_util` package is based on the following constructs:

- `si_util_signal_mgr`
- `si_util_signal`

Details of these constructs are shown in tables 16.9, 16.11, and 16.12. Table 16.13 shows the struct members for different signal types.

Table 16.9: `si_util_signal_mgr` methods

Method Name	Parameters	Return Type	Description
<code>stop()</code>			Stops all activity for signals whose <code>mgr</code> field points at this signal manager
<code>restart()</code>			Restarts all activity for signals whose <code>mgr</code> field points at this signal manager.
<code>mask()</code>	<code>flag: bool</code>		If <code>flag</code> is <code>TRUE</code> , masks all events generated by all signals whose <code>mgr</code> field points at this signal manager. The signals continue to operate. If <code>flag</code> is <code>FALSE</code> , then any masking is removed.

Table 16.10: `si_util_signal` struct members

Struct Member	Type	Valid Range	Description
<code>type</code>	<code>si_util_signal_type</code>	CLOCK JITTERED_CLOCK EDGE PULSE	Is used to indicate the signal types. Default: <code>CLOCK</code> .
<code>mgr</code>	<code>si_util_signal_mgr</code>	powers of 2, less than size	specifies the signal manager that manages this signal.

The program below shows an example using signal generators and signal managers to selectively control groups of generated signals.

```

1  : <
2  : extend sys {
3  :     clk_mgr1: si_util_signal_mgr;
4  :     clk_mgr2: si_util_signal_mgr;
5  :

```

Table 16.11: si_util_signal events

Event Name	Description
rise	emitted when signal rises
fall	emitted when signal falls
change	emitted when signal changes

Table 16.12: si_util_signal methods

Method Name	Parameters	Return Type	Description
stop()			Stops all activity for this signal.
restart()			Restarts all activity for this signal.
mask()	flag: bool		If flag is TRUE, masks all events generated by this signal. The signal continues to operate. If flag is FALSE, then any masking is removed.

```

6 :   clk1: CLOCK si_util_signal;
7 :       keepclk1.init_value == 1'b0;
8 :       keep clk1.lead_time == 6;
9 :       keep clk1.high_time == 4ns ;
10 :      keep clk1.low_time == 4ns ;
11 :      keep clk1.mgr == clk_mgr1;
12 :
13 :   clk2: CLOCK si_util_signal;
14 :       keep clk2.init_value = 1'b0;
15 :       keep clk2.lead_time = 12;
16 :       keep clk2.high_time == 2ns ;
17 :       keep clk2.low_time == 2ns ;
18 :       keep clk2.mgr == clk_mgr2;
19 :
20 :   edge1: EDGE si_util_signal;
21 :       keep edge1.init_value == 1'b0;
22 :       keep edge1.low_time == 12ns ; -- only low_time is specified since init_val = 0
23 :       keep edge1.mgr == clk_mgr1;
24 :
25 :   edge2: EDGE si_util_signal;
26 :       keep edge2.init_value == 1b1 ;
27 :       keep edge2. high_time == 28ns ; -- only high_time is specified since init_val = 1
28 :       keep edge2.mgr == clk_mgr2;
29 :
30 :   reset_pulse: PULSE si_util_signal;
31 :       keep reset_pulse.init_value == 1'b0;
32 :       keep reset_pulse.low_time == 23ns ;
33 :       keep reset_pulse.high_time == 17ns ;
34 :
35 :   event reset_rise is @reset_pulse.rise;
36 :   event reset_fall is @reset_pulse.fall;

```


Table 16.13: si_util_signal subtype struct members

Signal Type	Members	Member Type	Description
EDGE	init_value	bit	Signal value at time 0
	high_time	time	Time for signal at value 1. Only used if init_value is 1
	low_time	time	Time for signal at value 0. Only used if init_value is 0.
PULSE	init_value	bit	Signal value at time 0.
	high_time	time	Time for signal at value 1 during pulse.
	low_time	time	Time for signal at value 0 during pulse.
CLOCK	init_value	bit	Signal value at time 0.
	high_time	time	Time for signal at value 1.
	low_time	time	Time for signal at value 0.
	lead_time	time	Time during which signal stays at init_value before toggling starts.
JITTER_CLOCK	init_value	bit	Signal value at time 0.
	high_time	time	Time for signal at value 1.
	low_time	time	Time for signal at value 0.
	lead_time	time	Time during which signal stays at init_value before toggling starts.
	jitter	uint [0..99]	Jittered value defined as a percentage of high_time and low_time. During each calculation of the next transition time, high_time and low_time are randomly increased or decreased by the specified jitter percentage. Jitter does not apply to lead_time.

```

37 :
38 :     on reset_rise {
39 :         clk_mgr1.stop();
40 :         clk_mgr2.mask(TRUE)
41 :     };
42 :
43 :     on reset_fall {
44 :         clk_mgr1.restart();
45 :         clk_mgr2.mask(FALSE);
46 :     };
47 : };
48 : >

```

Figure 16.4 shows the behavior for the above program. Here, two signal managers are defined where **clk1** and **edge1** signals are controlled by **clk_mgr1**, and **clk2** and **edge2** are managed by **clk_mgr2**. Reset signal **reset_pulse** is generated using a **PULSE** signal. On the rise of reset pulse, all signals controlled by **clk_mgr1** are stopped (**clk1**, **edge1**) and their values are returned to their initial value. Also at that time, all signals controlled by **clk_mgr2** are masked (**clk2**, **edge2**) where even though signals changes occur, no events are generated based on these transitions. On the falling edge of the reset pulse, all signals controlled by **clk_mgr1** are restarted and the mask for signals controlled by **clk_mgr2** is removed so that events are generated as these signals continue to transition.

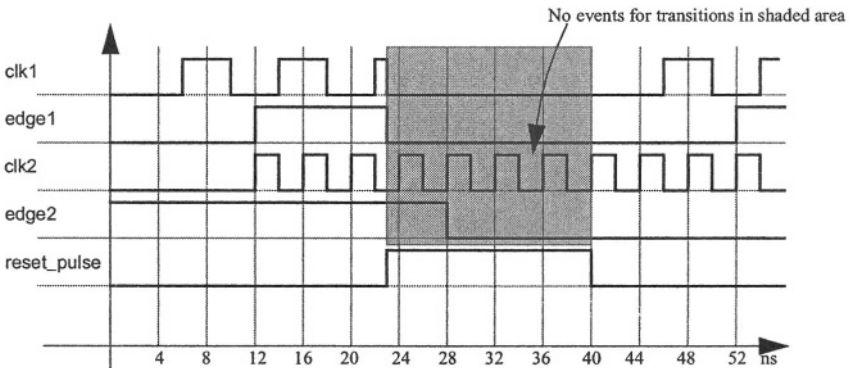


Figure 16.4 Signals and Signal Manager Operation

16.5 Native *e* Float Arithmetic Package

The float arithmetic utilities provided in the **si_util** package are designed to perform floating point arithmetic operations not inherently supported by *e* language. This implementation is compliant with the IEEE 854 standard for floating point arithmetic. The decimal base is used for performing these floating point operations.

The format for the internal representation of float numbers is given by:

$$(-1)^{\text{sign}} \times \text{coefficient} \times 10^{\text{exponent}}$$

In this representation, **sign** is either 0 or 1 representing a positive or a negative number respectively. **Coefficient** is an integer which is either zero or a positive number. **Exponent** is a signed integer. The **si_util_floatholder** struct is used to represent float numbers.

Arithmetic operations supported by the float package are described in table 16.14. These methods are a member of global name space and can be called anywhere in an *e* program.

This package also allows the user to configure the context of the floating point operations. The context is defined by the precision and the rounding mechanisms. These parameters are explained in detail in the package documentation.

Table 16.14: Float Arithmetic Operations

Method Name	Parameters	Return Type	Description
si_util_string_to_float()	A: string	si_util_floatholder	Creates float number from string.
si_util_float_to_string()	A: si_util_floatholder	string	Creates a string from a float
si_util_fdiv()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	result = A/B
si_util_fmuilt()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	result = A*B
si_util_fsub()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	result = A - B
si_util_fadd()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	result = A + B
si_util_fpow()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	result = A to the power of B
si_util_fmax()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	result = MAX(A, B)
si_util_fmin()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	result = MIN(A, B)
si_util_fdiv_int()	A: si_util_floatholder B: si_util_floatholder	int	return integer part of A/B
si_util_fremainder()	A: si_util_floatholder B: si_util_floatholder	si_util_floatholder	return the remainder of A/B
si_util_fabs()	A: si_util_floatholder	si_util_floatholder	result = absolute value of A
si_util_minus()	A: si_util_floatholder	si_util_floatholder	result = -A
si_util_round_int()	A: si_util_floatholder	int	round A to the nearest integer

The following program shows an example of using floating point arithmetic:

```

1  :  <
2  :  extend struct {
3  :      float_operations() is {
4  :          var f1: si_util_floatholder;
5  :          var f2: si_util_floatholder;
6  :          var f3: si_util_floatholder;

```

```
7 :  
8 :         f1 = si_util_string_to_float("0.6666666666");  
9 :         f2 = si_util_string_to_float("1.2E+6");  
10 :         f3 = si_util_fdiv(f2,f1);  
11 :         print si_util_float_to_string(f3);  
12 :     };  
13 : };  
14 : '>
```

16.6 Summary

This chapter presented the `si_util` package. This package provides utilities that are meant to enhance the power and usefulness of an *e* program. The utilities provided in this package at the time of this writing include mechanisms to coordinate end of runtime across multiple modules, a memory management and sparse memory core package, a time management and signal generation utility, and a native *e* float arithmetic package.

The utilities provided in `si_util` are packaged as an eRM compliant reusable component. This package can be loaded into any environment requiring the utilities provided by this package without generating any conflicts with other packages in the environment.

This page intentionally left blank

PART 7

Appendices

This page intentionally left blank

APPENDIX A

e BNF Grammar

Up to date as of Fri Mar 19 2004.

```
module :=
    statement_list
;

statement_list :=
    statements
;

statements :=
    statement
    | statements ';' statement
;

statement :=
    | package_statement
    | struct_statement
    | extend_struct_statement
    | type_statement
    | extend_type_statement
    | routine_statement
    | simulator_statement
    | unit_statement
    | sequence_statement
    | method_type_statement
    | c_export_statement
;

package_statement :=
    package id
;
```

```

struct_statement :=
  package_or_null struct id like_opt '{' struct_member_list '}'
;

like_opt :=
  | like id
;

extend_struct_statement :=
  extend struct_type '{' struct_member_list '}'
;

type_statement :=
  package_or_null type id ':' scalar_type
;

extend_type_statement :=
  extend id ':' '[' enum_item_list ']'
;

routine_statement :=
  package_or_null routine id '(' parameter_list ')' type_opt routine_name_opt
;

routine_name_opt :=
  | is c routine id
;

last_semi_opt :=
  | ';'
;

c_export_statement :=
  c export id c_export_opt
;

c_export_opt :=
  | '.' id '(' ')'
;

package_or_null :=
  | package
;

encap :=
  | package
  | private
  | protected
;

sequence_statement :=
  package_or_null sequence id sequence_opt
;

sequence_opt :=
  | using seq_name_pair_list
;

```

```

seq_name_pair_list:=
  seq_name_pair
  | seq_name_pair_list ',' seq_name_pair
;

seq_name_pair :=
  id '=' struct_type
;

method_type_statement :=
  package_or_null method type id '(' parameter_list ')' opt_return opt_event
;

opt_return :=
  | ':' type
;

opt_event :=
  | '@' event
;

struct_member_list :=
  struct_members
;

struct_members :=
  struct_member
  | more_struct_members struct_member
;

struct_member :=
  | field_declaration
  | method_declaration
  | subtype_declaration
  | constraint_declaration
  | coverage_declaration
  | temporal_declaration
  | simulator_member
  | attribute
  | cvl_declaration
;

field_declaration :=
  encap id field_type_specifier opt_instance
  | encap field_property id field_type_specifier opt_instance
;

field_property:=
  '!'
  | '%'
  | '! %'
  | '% '!'
;

field_type_specifier :=
  | '[' expr ']' ':' list_type
  | ':' type
;

```

```

method_declaration :=
  encap method_name '(' parameter_list ')' type_opt method_specifier action_block
| encap method_name '(' parameter_list ')' type_opt is empty
| encap method_name '(' parameter_list ')' type_opt is undefined
| encap method_name '(' parameter_list ')' type_opt is c routine id
| encap method_name '(' parameter_list ')' type_opt '@' event method_specifier action_block
| encap method_name '(' parameter_list ')' type_opt '@' event is empty
| encap method_name '(' parameter_list ')' type_opt '@' event is undefined
| encap method_name '(' parameter_list ')' type_opt method_specifier
    foreign_opt dynamic c routine
    libname_opt
;

method_name :=
  method_name
;

method_name :=
  method_id
;

parameter_list :=
  parameters
;

parameters :=
  parameter
| parameters ',' parameter
;

parameter :=
  id
| id ':' type
| id ':' '*' type
;

type_opt :=
  ':' type
;

method_specifier :=
  member_specifier
| is inline
| is inline only
;

foreign_opt :=
  foreign
;

member_specifier :=
  is
| is also
| is first
| is only
;

```

```

libname_opt:=
  | id ':'
  | id
  | id ':' id
;

subtype_declaration :=
  encaps when struct_subtype '{' struct_member_list '}'
;

constraint_declaration :=
  keep constraint_spec
;

list_of_constraint_spec_or_null:=
  | list_of_constraint_spec last_semi_opt
;

list_of_constraint_spec:=
  constraint_spec
  | list_of_constraint_spec ',' constraint_spec
;

constraint_spec:=
  constraint_expr
  gen before subtypes '(' field_list ')'
  reset gen before subtypes '(' ')'
;

field_list :=
  id
  field_list ',' id
;

attribute :=
  attribute id id '=' attribute_expr
;

attribute_expr :=
  id
;

unit_statement :=
  package_or_null unit id like_unit_opt '{' struct_member_list '}'
;

like_unit_opt :=
  | like id
;

opt_instance :=
  | is instance
;

cvi_declaration :=
  cvi_method
  | cvi_call
  | cvi_callback
;

```

```

cvl_method :=
  cvl method opt_async method_name '(' parameter_list ')' opt_event cvl_routine
;

cvl_call :=
  cvl call opt_async method_name '(' parameter_list ')' opt_event cvl_routine
;

cvl_callback :=
  cvl callback opt_async method_name '(' parameter_list ')' opt_event cvl_routine
;

opt_async :=
  | async
;

cvl_routine :=
  | is c routine target_struct
;

target_struct :=
  id
  | id '.' id
;

hdl_path :=
  "" hdl_pathname ""
;

simulator_statement :=
  simulator_member
  | simulator_restricted_member
;

simulator_member :=
  verilog simulator id
  | vhdl simulator id
  | verilog task hdl_path '(' vtask_parameter_list ')'
  | verilog function hdl_path '(' vfunc_parameters ')' v_size_opt
  | verilog variable hdl_path options_opt
  | verilog code expr
  | vhdl code '{ verilog_command_list last_semi_opt }'
  | vhdl procedure hdl_path options_opt
  | vhdl function "" id "" options_opt
  | vhdl driver hdl_path options_opt
  | vhdl object hdl_path
;

simulator_restricted_member :=
  verilog time verilog_timescale
  | vhdl time vhdl_timescale
;

verilog_command_list :=
  verilog_command
  | verilog_command_list ',' verilog_command
;

```

```

verilog_command :=
    STRING_LITERAL
;

vtask_parameter_list :=
    | vtask_parameters
;

vtask_parameters :=
    vtask_parameter
    | vtask_parameters ',' vtask_parameter
;

vtask_parameter :=
    id ':' expr vtask_parameter_options_opt
;

vtask_parameter_options_opt :=
    | ':' vtask_io
;

vtask_io :=
    in
    | id
    | inout
;

vfunc_parameters :=
    | vfunc_parameter_list
;

vfunc_parameter_list :=
    vfunc_parameter
    | vfunc_parameter_list ',' vfunc_parameter
;

vfunc_parameter :=
    id v_size_opt
;

v_size_opt :=
    | ':' expr
;

verilog_action :=
    force hdl_path '=' force_rhs
    | release hdl_path
;

force_rhs :=
    expr
    | verilog_literal
;

verilog_timescale :=
    NUMERIC_LITERAL id '/' NUMERIC_LITERAL id
;

```

```

vhdl_timescale :=
    NUMERIC_LITERAL id
;

action_block :=
    '{' action_list '}'
;

action_list :=
    actions
;

actions :=
    action
    | actions ';' action
;

action :=
    e_action
;

e_action :=
    | var_action
    | assign_action
    | conditional_action
    | iterative_action
    | method_call_action
    | start_tcm_action
    | compute_action
    | return_action
    | try_action
    | check_action
    | gen_action
    | emit_action
    | time_consuming_action
    | print_action
    | verilog_action
    | debug_action
    | dut_error_action
    | do_seq_action
    | action_block
;

var_action :=
    var id type_opt init_opt
    | var id "=" expr
;

init_opt :=
    | '=' expr
;

conditional_action :=
    break
    | continue
    | if_action
    | case_action
;

```

```

if_action :=
    if expr then_opt action_block else_part_opt
;

then_opt :=
    | then
;

else_part_opt :=
    | else action_block
    | else if_action
;

case_action :=
    case '{ case_list }'
    | case binary_expr '{ case_list }'
;

case_list :=
    cases last_semi_opt
;

cases :=
    case
    | cases ';' case
;

case :=
    expr colon_opt action_block
    | default
;

colon_opt :=
    | ':'
;

default :=
    default
    colon_opt action_block
;

iterative_action :=
    repeat do_opt action_block until expr
    | while expr do_opt action_block
    | for id from expr up_down binary_expr step_opt do_opt action_block
    | for '{ action ';' expr ';' action}' do_opt action_block
    | for each iterated_type_opt itemname_opt indexname_opt
        in expr do_opt action_block
    | for each iterated_type_opt itemname_opt indexname_opt
        in reverse expr do_opt action_block
    | for each file itemname_opt matching expr do_opt action_block
    | for each line itemname_opt in file expr do_opt action_block
;

up_down :=
    to
    | down to
;

```

```

iterated_type_opt :=
  | struct_type
  | enumerated_type
;

itemname_opt :=
  | (' id ')
;

indexname_opt :=
  | using index (' id ')
;

do_opt :=
  | do
;

step_opt :=
  | step expr
;

try_action :=
  try
  action_block else_try_opt
;

else_try_opt :=
  | else action_block
;

check_action :=
  check name_opt that_opt expr opt_block dut_error_opt
  | assert expr else_error_opt
;

name_opt :=
  | '<' id '>'
;

that_opt :=
  | that
;

dut_error_opt :=
  | else dut_error (' argument_list ') opt_block
;

dut_error :=
  dut error
  | dut errorf
;

else_error_opt :=
  | else error (' exprs ')
;

method_call_action :=
  method_invocation
  | method_port_invocation
;

```

```

action_opt:=
    action_block
    | with_opt
;

expr_or_default:=
    expr
    | default
;

opt_config_param:=
    | ',' exprs
;

compute_action :=
    compute expr
;

return_action :=
    return expr_opt
;

assign_action :=
    lval_expr assign_operator expr
;

assign_operator :=
    | "="
    | "+="
    | "-="
    | "*="
    | "/="
    | "%="
    | "<="
    | ">="
    | "&="
    | "&="
    | "|="
    | "and="
    | "or="
    | "<="
    | "&&="
    | "||="
;

gen_action :=
    gen reduced_gen_action_item itemname_opt keeping_opt
    | gen qualified_id itemname_opt keeping_opt
;

keeping_opt :=
    | keeping '{' constraint_list '}'
;

print_action :=
    print exprs options_opt
;

```

```

do_seq_action:=
  do when_qualified_id itemname_opt keeping_opt
;

debug_action:=
  message '(' argument_list ')' opt_block
| messagef '(' argument_list ')' opt_block
;

dut_error_action:=
  dut_error '(' argument_list ')' opt_block
| dut_errorf '(' argument_list ')' opt_block
;

opt_block :=
  | action_block
;

when_qualified_id :=
  id
  | struct_qualifier when_qualified_id
;

qualified_id:=
  path_id
  | struct_qualifier qualified_id
;

path_id :=
  id
  | id '[' expr ']'
  | me
  | path_id ':' id
  | path_id ':' id '[' expr ']'
  | path_id ':' as_a '(' type ')'
;

reduced_gen_action_item :=
  ':' id
  | ':' id '[' expr ']'
  | reduced_gen_action_item ':' id
  | reduced_gen_action_item ':' id '[' expr ']'
  | reduced_gen_action_item ':' as_a '(' type ')'
;

coverage_declaration :=
  cover id opt_cov_field coverage_group_option
;

coverage_group_option:=
  options_opt member_specifier '{' cover_item_list '}'
  | is empty
  | using also options opt_cover_item_list
;

cover_item_list :=
  | cover_items last_semi_opt
;

```

```

opt_cover_item_list :=
  | is also '{ cover_item_list }'
;

cover_items :=
  cover_item
  | cover_items ';' cover_item
;

cover_item :=
  item id item_options_opt
  | item id ':' type '=' expr options_opt
  | transition id item_options_opt
  | cross item_name_list item_options_opt
;

Item_name_list :=
  id
  | item_name_list ',' id
;

opt_cov_field:=
  | (' expr ')
;

item_options_opt:=
  | using options
  | using also options
;

temporal_declaration :=
  encap event id event_option
  | on id opt_defer do_opt action_block
  | encap expect_declaration
;

opt_defer :=
  | '$'
;

event_option:=
  | is temporal_expr
  | is only temporal_expr
;

expect_declaration:=
  expect id
  | expect temporal_expr dut_error_opt
  | expect id expect_specifier temporal_expr dut_error_opt
  | assume id
  | assume temporal_expr dut_error_opt
  | assume id expect_specifier temporal_expr dut_error_opt
;

expect_specifier:=
  is
  | is only
;

```

```

emit_action :=
    emit event
;

start_tcm_action :=
    | start method_invocation
    | start method_port_invocation
;

time_consuming_action :=
    all of action_block
    | first of action_block
    | wait
    | wait until_opt temporal_expr
    | sync
    | sync temporal_expr
    | state machine expr until_state_opt '{ transition_list }'
;

until_opt :=
    | until
;

until_state_opt :=
    | until id
;

transition_list :=
    last_semi_opt
    | transitions last_semi_opt
;

transitions :=
    transition
    | transitions ';' transition
;

transition :=
    id "<=>" id action_block
    | "*" "<=>" id action_block
    | id action_block
;

event :=
    id
    | field_access
    | primitive_expr "$"
;

temporal_expr :=
    temporal_inclusive_expression
    | temporal_expr '@' event
;

temporal_inclusive_expression :=
    temporal_or_expression
    | temporal_inclusive_expression "<=>" temporal_or_expression
;

```

```

temporal_or_expression :=
    temporal_and_expression
  | temporal_or_expression or temporal_and_expression
;

temporal_and_expression :=
    temporal_exec_expression
  | temporal_and_expression and temporal_exec_expression
;

temporal_exec_expression :=
    temporal_sampling_expression
  | temporal_sampling_expression exec action_block
;

temporal_sampling_expression :=
    temporal_eventual_expression
  | temporal_eventual_expression '@' event
;

temporal_eventual_expression :=
    temporal_repeat_expr
  | eventual temporal_repeat_expr
  | not temporal_eventual_expression
;

temporal_repeat_expr :=
    temporal_unaryexpr
  | '[' range ']' temporal_repeat_opt
  | '[' expr ']' temporal_repeat_opt
  | '~' '[' range ']' temporal_repeat_opt
;

temporal_primitive :=
    cycle
  | detach '(' temporal_expr ')'
  | true '(' expr ')'
  | rise '(' expr ')'
  | fall '(' expr ')'
  | change '(' expr ')'
  | delay '(' expr ')'
  | '(' temporal_expr ')'
  | '{ temporal_sequence last_semi_opt }'
  | consume '(' '@' event ')'
;

temporal_sequence :=
    temporal_expr
  | temporal_sequence ';' temporal_expr
;

temporal_repeat_opt :=
    | '*' temporal_exec_expression
;

temporal_unaryexpr :=
    temporal_primitive
  | '@' event
  | fail temporal_unaryexpr
;

```

```

type :=
  non_port_type
  | port_type
;

non_port_type :=
  regular_type
  | list_type
;

regular_type :=
  scalar_type
  | struct_subtype
;

scalar_type :=
  id
  | enumerated_type
  | scalar_type scalar_modifier
;

enumerated_type :=
  '{ enum_item_list }'
;

scalar_modifier :=
  '[' ranges ]'
  | '{ scalar_unit ':' expr }'
  | '{ scalar_unit ':' "" }'
;

enum_item_list :=
  | enum_items
;

enum_items :=
  enum_item
  | enum_items ',' enum_item
;

enum_item :=
  id enum_num_opt
;

enum_num_opt :=
  | '=' expr
;

scalar_unit :=
  bits
  | bytes
;

struct_type :=
  id
  | struct_subtype
;

```

```

struct_subtype :=
  struct_qualifier struct_type
;

struct_qualifier :=
  id
  | id "" id
  | FALSE "" id
  | TRUE "" id
;

list_type :=
  list of type
  | list '(' id ':' id ')' of type
;

port_type :=
  io_type simple port of non_port_type
  | io_type buffer port of non_port_type
  | io_type event port
  | serve_client call port of non_port_type
  | io_type method port of id
;

io_type :=
  id
  | in
  | inout
;

serve_client :=
  id
;

constraint_expr :=
  binary_expr
;

select :=
  select '{' selection_list last_semi_opt '}'
;

selection_list :=
  selection
  | selection_list ',' selection
;

selection :=
  expr ':' expr
;

port_binding :=
  bind '(' expr ',' port_bind_target port_constraint ')'
  ;

port_bind_target :=
  expr
  | UNDEFINED
;

```

```

port_constraint :=
  | '{ constraint_list }'
;

lval_expr :=
  id
  | field_access
  | primitive_expr '[' range_element ']'
  | hdl_path
  | bit_extract
  | bit_concat
  | primitive_expr '$'
;

primitive_expr :=
  lval_expr
  | me
  | literal
  | (' binary_expr ')
  | new
  | method_invocation
  | method_port_invocation
  | '[' ranges ']'
  | cast
  | select
  | port_binding
;

new :=
  new
  | new struct_type itemname_opt with_opt
;

with_opt :=
  | with action_block
;

field_access :=
  primitive_expr ':' when_field_access
  | ':' when_field_access
  | when_field_access_pair
;

when_field_access :=
  id
  | when_field_access_pair
;

when_field_access_pair :=
  FALSE "" id
  | TRUE "" id
  | when_field_access "" id
;

bit_extract :=
  primitive_expr '[' expr_opt ':' expr_opt slice_opt ']'
;

```

```

slice_opt:=
  | '?' scalar_type
;

bit_concat:=
  '%' '{ bit_elements }'
;

bit_elements :=
  expr
  | bit_elements ',' expr
;

method_port_invocation :=
  primitive_expr '$' '(' argument_list ')'
;

method_invocation :=
  primitive_expr ':' called_method_name '(' argument_list ')'
  | ':' called_method_name '(' argument_list ')'
  | id_or_special_method '(' argument_list ')'
  | hdl_path '(' argument_list ')'
  | all_values '(' scalar_type ')'
  | get all units '(' struct_type ')'
  | primitive_expr ':' get enclosing unit '(' struct_type ')'
  | get enclosing unit '(' struct_type ')'
  | primitive_expr ':' try enclosing unit '(' struct_type ')'
  | try enclosing unit '(' struct_type ')'
  | primitive_expr ':' seq_method '(' type ')' itemname_opt
  | ':' seq_method '(' type ')' itemname_opt
;

called_method_name :=
  method_name
;

id_or_special_method :=
  method_id
;

seq_method:=
  in sequence
  | in unit
;

argument_list :=
  | exprs
;

cast :=
  primitive_expr ':' as_a '(' type ')'
  | ':' as_a '(' type ')'
;

ranges :=
  range_element
  | ranges ',' range_element
;

```

```

range_element :=
  expr
  | range
;

range :=
  expr_opt ".." expr_opt
;

list_elements_or_null:=
  |list_elements last_semi_opt
;

list_elements :=
  expr
  | list_elements ';' expr
;

unary_expr :=
  primitive_expr
  | now '@' event
  | '{' list_elements_or_null '}'
  | '{' list_elements_or_null '[' range_element ']'
  | '{' list_elements_or_null '}' ':' id '(' argument_list ')'
  | unary_operator unary_expr
  | primitive_expr unary_post_operator
  | lval_expr time_unit
  | literal time_unit
  | constraint_for_each_expr
  | text_expansion_exp
  | "<<<" STRING_LITERAL
;

unary_operator :=
  not
  | '!'
  | '&'
  | '^'
  | nor
  | nand
  | nxor
  | '+'
  | '-'
  | '~'
  | '!'
;

unary_post_operator :=
  is empty
  | is not empty
;

binary_expr :=
  boolean_imp_expression
  | boolean_imp_expression '?' expr ':' expr
;

```

```

boolean_imp_expression :=
    logical_OR_expression
    | boolean_imp_expression "=>" logical_OR_expression
;

logical_OR_expression :=
    logical_AND_expression
    | logical_OR_expression boolor_operator logical_AND_expression
;

boolor_operator :=
    "||"
    | or
;

logical_AND_expression :=
    inclusive_OR_expression
    | logical_AND_expression booland_operator inclusive_OR_expression
;

booland_operator :=
    and
    | "&&"
;

inclusive_OR_expression :=
    exclusive_OR_expression
    | inclusive_OR_expression '!' exclusive_OR_expression
;

exclusive_OR_expression :=
    AND_expression
    | exclusive_OR_expression exclusive_operator AND_expression
;

exclusive_operator :=
    ^
    | nxor
;

AND_expression :=
    in_expression
    | AND_expression '&' in_expression
;

in_expression :=
    match_expression
    | in_expression IN_operator match_expression
;

IN_operator :=
    in
    | in range
    | not in
;

match_expression :=
    relational_expression
    | match_expression match_operator relational_expression
;

```

```

match_operator :=
    '=='
    | "!="
;

relational_expression :=
    member_expression
    | relational_expression neq_operator relational_rhs
    | verilog_literal neq_operator member_expression
;

relational_rhs:=
    member_expression
    | verilog_literal
;

neq_operator :=
    "=="
    | "!="
    | verilog_operator
;

verilog_operator :=
    "=="
    | "!="
;

member_expression:=
    equality_expression
    | member_expression is a struct_type
    | member_expression is a struct_type '(' id ')'
    | member_expression is not a struct_type
;

equality_expression :=
    soft_expression
    | equality_expression eq_operator soft_expression
;

soft_expression :=
    shift_expression
    | soft shift_expression
;

eq_operator :=
    "<="
    | ">="
    | '<'
    | '>'
;

shift_expression :=
    additive_expression
    | shift_expression shift_operator additive_expression
    | gen '(' gen_item_list ')' before '(' gen_item_list ')'
;

```

```

shift_operator :=
  "<<"
  | ">>"
;

additive_expression :=
  multiplicative_expression
  | additive_expression additive_operator multiplicative_expression
;

additive_operator :=
  '_'
  | '+'
;

multiplicative_expression :=
  unary_expr
  | multiplicative_expression multiplicative_operator unary_expr
;

multiplicative_operator :=
  '*'
  | '/'
  | '%'
  | within
;

exprs :=
  expr
  | exprs ',' expr
;

expr :=
  binary_expr
;

expr_opt :=
  | expr
;

opt_index :=
  | Index '(' id ')'
;

opt_prev :=
  | prev '(' id ')'
;

constraint_for_each_expr :=
  for each itemname_opt in gen_item do_opt
    '{' constraint_list '}'
  | for each itemname_opt using opt_index opt_prev in gen_item
    do_opt '{' constrain_list '}'
;

gen_item_list :=
  gen_item
  | gen_item_list ',' gen_item
;

```

```

gen_item :=
    primitive_expr
;

constraint_list :=
    | constraints last_semi_opt
;

constraints :=
    constraint_expr
    | constraints ';' constraint_expr
;

verilog_literal :=
    BASED_LITERAL
;

time_unit :=
    hr
    | min
    | sec
    | ms
    | us
    | ns
    | ps
    | fs
;

text_expansion_exp :=
    text begin text_list text end
;

text_list :=
    '(' expr ')'
    | STRING_LITERAL
    | text_list '(' expr ')'
    | text_list STRING_LITERAL
;

options_opt :=
    | using options
;

options :=
    option
    | options ';' option
;

option :=
    id
    | id '=' expr
    | when '=' expr
    | range_option
;

range_option :=
    ranges '=' '{ cover_ranges last_semi_opt }'
;

cover_ranges :=

```

```
    cover_range
  | cover_ranges ';' cover_range
;

cover_range :=
  range '(' '[' ranges ']' optional_range_param ')'
  | range '(' id optional_range_param ')'
;

optional_range_param :=
  !',' exprs
;

literal :=
  STRING_LITERAL
  | NUMERIC_LITERAL
  | char literal
  | TRUE
  | FALSE
  | NULL
  | UNDEF
  | MAX_INT
  | MIN_INT
;

id :=
  id
;
```


This page intentionally left blank

APPENDIX B*e Reserved Keywords*

all of	All_values	and	asa	as_a
assert	assume	Async	attribute	before
bit	Bits	bool	break	byte
bytes	c export	case	change	check that
compute	computed	consume	continue	cover
cross	cvl call	Cvl callback	cvl method	cycle
default	define	delay	detach	do
down to	Dut_error	each	edges	else
emit	event	exec	expect	extend
fail	Fall	File	first of	for
force	from	gen	global	hdl pathname
if	#ifdef	#ifndef	in	index
int	is	Is a	is also	is c routine
is empty	is first	Is inline	is instance	is not a

is not empty	is only	Is undefined	item	keep
keeping	Key	like	line	list of
matching	Me	nand	new	nor
not	Not in	now	on	only
or	others	pass	prev	print
range	ranges	release	repeat	return
reverse	Rise	routine	select	session
soft	start	state machine	step	struct
string	sync	sys	that	then
time	To	transition	true	try
type	Uint	unit	until	using
var	verilog code	verilog function	verilog import	verilog simulator
verilog task	verilog time	verilog timescale	verilog trace	verilog variable
vhdl code	vhdl driver	vhdl function	vhdl procedure	vhdl driver
vhdl simulator	vhdl time	when	while	with
within				

eRM Compliance checks are defined for different categories. These categories are:

- Packaging and Name Space Compliance Checks
- Architecture Compliance Checks
- Reset Compliance Checks
- Checking Compliance Checks
- Coverage Compliance Checks
- Sequences Compliance Checks
- Messaging Compliance Checks
- Monitor Compliance Checks
- Documentation Compliance Checks
- General Deliverables Compliance Checks
- Visualization Compliance Checks
- End of Test Compliance Checks

These checks are listed in tables in the following sections.

For each check, RQ=Required, RC=Recommended, and ST=Statistical.

C.1 Packaging and Name Space Compliance Checks

Table C.1: eRM Packaging and Name Space Compliance Checks

Check	Description	Type
Legal package name	Does the package have a legal package name including company prefix and a unique intra-company package name?	RQ
Legal directory name	Is the package directory named consistently with the package name?	RQ
Valid README file	Is there a legal PACKAGE_README.txt file at the top level of the package?	RQ
Valid distribution format	Is the package distributed as package_version_version.tar.gz?	RQ
Valid directory structure	Is the package.s directory structured correctly	RQ
Demo script available	Does the package have a demo.sh file to demonstrate the package, located at the top level of the package?	RQ
Demo script running	Does it run	RQ
Loading with golden eVC	Does the eVC run properly with the golden eVC loaded?	RQ
Simulator support	Are the simulators supported by the demo documented in the PACKAGE_README.txt file?	RQ
Simulator support documented	Are the simulators supported by the package documented in the manual?	RQ
Valid file names	Do all source code filenames in the package start with the package name?	RQ
Valid type names	Do all type names have a legal package prefix?	RQ
Valid 'define' name	Do all .define. names have a legal package prefix?	RQ
Type extensions	Do all extensions to types defined outside the package have fields that start with legal package prefix?	RQ
Top file	Is the top-level file of the eVC placed in the <i>e</i> subdirectory and called package_top.e?	RQ
Examples documented	Is there a EXAMPLES_README.txt file in the examples directory detailing the contents?	RQ
Examples documented - details	Are all examples documented?	RQ
Version number	Does the version number in PACKAGE_README.txt file match the version number in the top file?	RQ

Table C.1: eRM Packaging and Name Space Compliance Checks

Check	Description	Type
Tar file	Is the tar file named and organized according to eRM standards?	RQ
Global extends	Are there no extends to sys or global?	RC
Enum extensions	If an enum from some other package is extended, do the new values start with the package prefix?	RQ
Global extends naming	If sys or global is extended, are the extensions prefixed with the package prefix?	RQ
Configuration template	Are there config templates in the examples directory?	RQ
Package encapsulation	Does all the code of the eRM package belong to one or more <i>e</i> packages (other than main)?	RQ
Package name	Does the top file belong to an <i>e</i> package whose name is identical to the eRM package?	RQ
Multiple packages	If there is more than one <i>e</i> package, do they all have the same prefix?	RQ
Protection	How many public and non-public named types and members are there?	ST

C.2 Architecture Compliance Checks

Table C.2: eRM Architecture Compliance Checks

Check	Description	Type
Signal definition	Are there signals defined (i.e. <code>.sig_</code> strings)? Are they all within agents and <code>.env</code> units?	RC
HW access	Is all HW access done via <code>.sig_</code> strings?	RQ
Instantiation	Are all agents instantiated in envs?	RQ
Monitors	Are all monitors in envs or agents?	RQ
Active agents	Do all ACTIVE agents have a BFM in them?	RQ
BFM	Are all BFMs instantiated in ACTIVE agents?	RQ
eVC name	Does the env have a <code>.name</code> field?	RQ
Agent names	Do all agents have a <code>.name</code> field?	RQ
BFM driving signals	Are all DUT signals driven only by BFMs?	RQ
ACTIVE agents with BFM sequence drivers	Do all ACTIVE agents have a BFM sequence driver?	RQ
BFM Sequence drivers in ACTIVE agents	Are all BFM sequence drivers in ACTIVE agents?	RQ
Scoreboarding	Are hooks provided for scoreboarding?	RC

C.3 Reset Compliance Checks

Table C.3: eRM Reset Compliance Checks

Check	Description	Type
Reset Support	Does the package correctly respond to resets (of any length) generated within the DUT at the start of the test Resets..	RQ
Reset checks	Are there sufficient checks to ensure that the DUT behaves correctly after reset is de-asserted	RQ
Multiple resets	Does the package manage multiple resets during the test?	RQ

Table C.3: eRM Reset Compliance Checks

Check	Description	Type
Multiple reset checks	Are there sufficient specific checks relating to the DUT's response to assertion/de-assertion of reset?	RQ
Programmable resets	Does the package provide a mechanism for generating programmable reset(s) and can this feature be disabled?	RC
Clock generator	Does the package provide a (sufficiently programmable) clock generator?	RQ
Working with DUT supplied clock	Does the package work with DUT supplied clock?	RQ
Use of clocks	Which parts are running on unqualified clock?	ST

C.4 Checking Compliance Checks

Table C.4: eRM Checking Compliance Checks

Check	Description	Type
Checks and expects	How many checks and expects are there?	ST
Protocol checking	Does the package provide sufficient DUT checking (e.g. protocol checkers) to cover all possible DUT errors?	RQ
DUTerrormessages	Do all expects and checks have non-default dut_error message?	RQ
Error messages	Do all error messages provide sufficient detail for the user to identify the area and instance of the package/DUT that produced the error?	ST
DUT errors	How many dut_errors were defined?	ST
Error message documentation	Are all checks (both for DUT errors and user errors) sufficiently documented?	RQ
Scoreboard	Does the package provide (where appropriate) toolkits to enable the user to code complex data flow checking tasks e.g. does the package provide a scoreboard either already integrated, or as a generic tool?	RC

C.5 Coverage Compliance Checks

Table C.5: eRM Coverage Compliance Checks

Check	Description	Type
Coverage groups	How many coverage groups are defined?	ST
Coverage items	How many coverage items are defined?	ST
Coverage crosses	How many coverage crosses are defined?	ST
Coverage results	Is the coverage report produced after testing this eVC included in the package?	RQ

C.6 Sequences Compliance Checks

Table C.6: eRM Sequences Compliance Checks

Check	Description	Type
Virtual sequence driver (SD) pointers	Do virtual sequence drivers have pointers to one or more other sequence drivers?	RQ
Subdrivers	Do all virtual sequence drivers have subdrivers?	RC
SD statistics	How many sequence drivers exist? How many of them are BFM/virtual?	ST
Read/Write methods	How many sequences have their read()/write() methods extended?	RC ST
Read/Write wrappers	How many sequence drivers implement read()/write() methods?	ST
Predefined sequence types	How many predefined sequence types are provided?	ST
Error injection	Do the sequence items provide sufficient ability to inject errors into the generated data stream(s)?	RQ
Error injection . virtual fields	Are virtual fields appropriately employed?	RC

C.7 Messaging Compliance Checks

Table C.7: eRM Messaging Compliance Checks

Check	Description	Type
Loggers	Is there a screen logger in the env and in each agent?	RQ
File loggers	Are there file loggers defined in the package?	RC
Logger instantiation	Are all message loggers instantiated in env unite, agents monitors or in sys?	RQ
Message actions	How many message actions are there? At what verbosity levels are these actions?	ST
Message tags	How many message TAGs were defined?	ST
Short name and short name style	Do short_name() and short_name_style() return non-empty strings?	RC
Logger constraints	Are all loggers constrained using only soft constraints?	RC

C.8 Monitor Compliance Checks

Table C.8: eRM Monitor Compliance Checks

Check	Description	Type
Error extraction	For sequence items that collect output from the DUT, is there a sufficient number of virtual fields provided to indicate formatting errors detected in the data structure?	ST

C.9 Documentation Compliance Checks

Table C.9: eRM Documentation Compliance Checks

Check	Description	Type
Release notes	Are release notes provided in the docs directory?	RQ
Documentation	Which documentation files exist in the /docs directory (.doc, .pdf)?	RQ
Features and controls	Does the documentation cover all features and controls?	RQ
Constrainable fields	Does the documentation describe all user-constrainable fields and indicate when they are generated and what default constraints are applied to them?	RQ
Usable fields	Does the documentation describe all non-user-constrainable fields that users may use to control their constraints?	RQ
Installation and demo	Does the documentation describe the installation and demo processes?	RQ
Package architecture	Does the documentation describe the architecture of the package and give an overview of its intended use?	RQ
Examples	Does the documentation give sufficient examples to cover the most likely user scenarios?	RQ
Reset	Does the documentation explain whether the package manages multiple resets during the test?	RQ
eVC structure	Are diagrams provided to explain the structure of the eVC? Typical environments and configurations; how to use scoreboarding; the class diagram of the main units and structs.	RQ
Recommended practice	Does the documentation clearly differentiate between what is good and bad practice when using the package (e.g. which structs should and should not be extended)?	RQ

Table C.9: eRM Documentation Compliance Checks

Check	Description	Type
Support policies	Does the documentation clearly define the support policies for the package and indicate contact information for obtaining support?	RQ
Documentation format	If the documentation is to be distributed electronically, does it clearly print both on color and B&W printers and on both European A4 and US Letter paper sizes?	RC
Proper documentation	Are concepts introduced before being referred to?	RC
SD documentation	Are all sequence-driver test interfaces sufficiently documented?	RQ
BFM documented	Are all BFMs documented (their API and behavior)	RQ

C.10 General Deliverables Compliance Checks

Table C.10: eRM General Deliverables Compliance Checks

Check	Description	Type
eVC name		RQ
Protocol name and version	Name and version number of protocol or architecture the eVC models.	RQ
New functionality	New functionality added since previous release.	RQ
Customer feedback	Number of customer engagements this eVC has been involved in. Provide customer quotes or feedback indicating satisfaction level, likes and dislikes if available.	RQ
Support model	Description of the post sales support model for this eVC. (e.g. on-site support for 7 days, then telephone support).	RQ
Test plan	Test plan for the eVC and/or a description of how this eVC was tested (e.g. 5 tests were written and feedback gathered from 3 beta sites).	RQ
Code lines	How many lines of code are there?	ST

C.11 End of Test Compliance Checks

Table C.11: eRM End of Test Compliance Checks

Check	Description	Type
End of test Future	see section 16.1	

Index

Symbols

! 83
\$ port reference 93
% 95, 122
@ unary event temporal operator 173
@sim 185
@x 89
@z 89

A

abstract data types 24
abstract ranges 134
accept_message() 195
active agents 276, 277, 336
active VBFM 39
all of 75, 76, 77, 154, 174, 331
and temporal operator 173
any_env 270, 274, 275, 276
any_sequence 143, 144, 151, 152, 158
any_sequence_driver 143
any_sequence_item 143, 151
aspect 55
aspect-oriented programming 54
assume 73, 170, 331
at_least 230, 231, 236, 237, 239, 259
atomic temporal operators 172, 173

B

base temporal expressions 73, 165
base temporal operators 173
basic coverage item 100, 229
begin-code 63
BFM 38, 40, 45, 141, 145, 147, 150, 155
BFM features 39
bidirectional constraints 87
black-box verificatio n13
body() 146, 150
boolean equivalence constraints 108
boolean non-equivalence constraints 108
bus functional model 38

C

change temporal operator 173
check 138, 179, 183, 204
check that 331
check() 60
code segments 63
composite coverage items 236
composite data type 65
composite temporal operators 173
compound constraints 108, 111, 112
compound data generation 106
concurrency 29, 71, 82
concurrency actions 75

concurrency and processes 71
concurrency and resources 71
concurrency rules in e 77
concurrency support 71
Conditional Field 128
configuration 41, 268, 340
configuration interface 40
constrained random test based 25
constrained random test based verification 25
constraint evaluation step 109, 111, 112
constraint reduction step 109, 111, 112
constraint satisfaction problem 54
constraint set-scalar step 112
coordinated ranges 135
core concern 55
cover construct 227, 229
coverage 82
coverage analysis 227, 237, 245, 261
coverage bucket 226, 232
coverage bucket hit 226
coverage bucket range 232
coverage data collection 245
coverage data source 251
coverage design 246
coverage development 246
coverage driven verification methodology 27
coverage goal 226
coverage grading 226, 236, 245, 256, 258, 259, 261
coverage group 226
coverage group extension 240
coverage group sampling event 100
coverage groups 100, 225, 227, 229, 239, 249, 338
coverage hole 226
coverage illegal bucket 232
coverage item 225, 226, 227, 228, 229, 230, 231, 233, 234, 235, 236, 237, 238, 242, 246, 249, 258, 338
coverage item extension 240
coverage item range construct 230
coverage model 227, 234, 242, 243, 245, 247, 248, 250, 251, 256, 257, 261
coverage plan 246
coverage sampling event 225, 231
CPU verification 37
cross construct 237
cross coverage 237, 238, 239, 241

cross coverage items 100, 237
cross item 226
cycle accurate data checking 47

D

data abstraction 42, 45, 46, 82, 95
data abstraction translation 130
data checking 36, 46, 48, 82
data collection 45, 205
data collector 45
data generation 41, 42, 118
data generation constraints 134
data generation inconsistencies 125
data modeling conditional field 126
data modeling determinant Fields 123
data modeling using struct 122
data references 58
declarative programming 54
default buckets 235
default ranges 135
delay() 293
detach temporal operator 171, 173, 18
device under verification 18
directed test based verification methodology 22
do 130, 146, 147, 150
do_generate() 106
do_item() 150, 151, 152
do_pack() 130
do_sequence() 150, 151
do_test() 61
do_unpack() 96, 98, 132, 133
down() 80
drop_objection() 280
dut_error() 189, 203, 204
DUV 18
DUV invalid behavior 34
DUV invalid outputs 34
DUV signal coverage 252
DUV valid stimulus 35

E

e programming paradigms 54
e programming steps 61
e Reuse Methodology 267
e Verification Component (eVC) 268
emit 72
end-code 63

end-to-end scoreboarding 220
enumerated types 67
environment packages 270
e-port configuration 94
e-ports 93, 94, 275
eRM 267
erm_active_passive_t 276
eVC 270, 276, 277
eVC agents 276
eVC architectural requirements 268
eVC environment 276
event_port port type 94
events 72, 170
eventually temporal operator 177
exec 172, 179
expect 73, 170, 177
expected transactions 47
extension statements 64

F

fail temporal operator 169, 173
finalization phase 60
finalize() 61, 256
first of 75, 76, 77
first-match-variable-repeat temporal operator 175
fixed-repetition-temporal-operator 73
flat verification sequence 138, 142, 146
float arithmetic package 279, 299
for each 124
force action 89
format_message() 196
functional verification 12
function-based verification view 34

G

garbage collection 116
gen 118, 119, 120
generate() 106
generated instance of a verification item 138
generation constraints 26, 27, 28, 83, 84, 261
generation order using when block s113
generation order using gen befo re113
generation order using value ()113
generation order using when block s114
generation set-scalar step 109
generation static analysis 115
get_enclosing_unit() 91

get_next_item() 152, 153, 154
get_unit() 190
global 58, 59, 228, 229, 255
goal states 16, 32
gray-box verificatio n13

H

HDL path 91
HDL signal 88, 92
HDL signal attributes 43
HDL signal coverage 252
HDL signal driving 89
HDL signal monitored 90
HDL signal x value 89
HDL signal z value 89
hdl_path() 91
heterogeneous verification sequence 139
Heterogeneous verification sequences 155
heterogeneous verification sequences 139, 141, 142
hierarchical coverage model 247
hierarchical sequence 147
hierarchical verification sequences 137, 138, 139, 140, 143, 146, 147
homogeneous verification sequences 139

I

ignore 230, 238, 239, 240, 242, 261
ignored coverage Items 261
illegal 230, 235, 238, 239, 240, 255
illegal Bucket 235
illegal coverage items 261
imperative programming 53
implicit execution order 60
imply 108, 114, 125, 126
imply constraints 108, 113
import statement 270
import statements 64, 270
in port direction 93
init() 60, 117
initialization runtime phase 60
inout port direction 93
internal_body() 149, 150
is also 70, 240
is empty 70
is first 70
is instance 91
is only 70, 71

is undefined 70
item_done event 152, 154

K

keep 85
keep soft 86

L

layered sequences 137
Lexical Conventions 63
LIBRARY_README.txt 273
like inheritance 143
linear coverage grading 256
lock() 78
locker 78
logical interface 18
logical temporal operators 73
logical view 18
long message output format 193

M

MAIN sequence kind 145, 148
max_int_bucket 259
memory manager 288
memory package 279
message action 191
message_logger 194, 197
message_tag 192
messagef action 191
methods 66
monitor architecture 199
monitor event extraction 209
monitors 43
multi-dimensional coverage model 247
multistep scoreboarding 220
multi-valued logic 89
mutex problem 78
mutual exclusion 78

N

name 230, 237, 238, 239
no_collect 228, 229, 242
no_trace 229
none message output format 193
normative statements 34
not temporal operator 176

O

objection mechanism 280
object-oriented programming 54
on 72
on-the-fly generation 87, 115, 116
or constraints 108
or temporal operator 168, 174
others 235
out port direction 93
out() 189, 204
outf() 70, 189

P

pack action 97
pack() 96, 130
pack_options 95, 98, 130
package configuration settings 277
package directory structure 271
package e-port interface 277
package features 274
package library 270, 271, 272
package monitor 277
package name 273
package naming conventions 270
package sequence driver 277
package sequence generator 277
package shadowing 273
package title 273
package version 273
PACKAGE_README.txt 273
packages 269
packing 97
packing.high 95, 97, 132
packing.low 95, 130, 132
packing.network 95
passive agents 276
per_instance 229, 231
physical fields 95, 122
physical interface 18
physical leve 118
port-based verification view 33
post_body() 150
post_generate() 60, 106, 107
post-run result checking phase 60
pre_body() 150
pre_generate() 60, 106
pre-run generation 87
pre-run generation phase 60, 87, 197

prev 240, 241
proactive agents 276
programming concern 55
programming paradigm 53
protocol analyzers 43
protocol checker activation 204
protocol checkers 43
protocol checking 202
protocol checks 202

Q

qualified threads 75
quit() 177

R

radix 228, 230
raise_objection() 280
random generation 24, 25, 26, 27, 28, 29
RANDOM sequence kind 145, 148
random test based verification
 methodology 24
range 147
ranges 230, 242
rdv_semaphore 79
reactive agents 276
reactive sequences 137
reference model 13, 14, 36, 43, 46, 47, 48
release action 89
release() 78
relocatable verification module 91
rendezvous semaphores 79
reset_soft() 136
result 67
result checking 82
return 67
rise temporal operator 173
root-mean-square coverage grading 257
run() 60, 76, 117, 149, 152, 153, 171
runtime cycles 75

S

sampling event 73, 74, 75, 76, 77, 90, 165,
 168, 228
sampling event missing 186
sampling period 73, 74, 165
scalar range 68
scalar subtype 68
scalar subtypes 67, 68, 83

scenario generation 28, 41, 42
scoreboard configuration 215
scoreboard data checking 220
scoreboard end of simulation check 47
scoreboard ordered matching 47
scoreboard time-out feature 47
scoreboarding memory transactions 48
self-contained verification environment 37
semaphore code region 78
semaphore construct 78
semaphores 78
send_to_bfm() 145, 147, 153
separation of concerns 55
sequence 145, 157
sequence do action 140
sequence driver 276
sequence driver PULL_MODE 152
sequence driver PUSH_MODE 145, 152
sequence generator properties 137
sequence statement 148
sequence temporal operator 73, 166
setup() 61
short message output format 193
si_util_floatholder 299
si_util_mem 291
si_util_mem_mgr 288
si_util_signal 296
si_util_signal_mgr 296
si_util_stop_run_controller 281
si_util_stop_run_controller_if 281
si_util_time_mgr 294
signal generator 279
signal generator, Edge 295
signal generator, Periodic 295
signal generator, Pulse 295
sim event 90
simple constraints 106, 108, 111
SIMPLE sequence kind 145, 148
simple_port port type 94
simulation abstraction 82
simulation goa 133
simulation result reusability 21
simulation results 21
simulation run phase 60
sparse memory core 288
SPECMAN_PATH 268, 272
start 75, 77
start_sequence() 149, 151, 152

state machine 25, 26
state machine coverage 99, 253
state machine coverage groups 100
state machine verification 16
state machine verification space 17, 32
stimulus generation 82
stimulus injection 82, 94
stop_run() 78, 280
stop-run agent 280
stop-run controller 279
stop-run group 280
struct 56, 57, 59, 60, 65, 143
struct data members 66
struct declaration statement 65
struct instance hierarchy 56, 57, 59
struct subtype 67, 68, 82, 98
struct/unit declarations 64
subtype determinant struct members 84
sync 73, 75, 170
synchronization actions 75
sys 58, 59, 82, 87
sys extension 61
sys generation 115
sys message logger 190
sys predefined methods 61
sys pre-generation 115
sys.any 73, 91, 186
sys.logger 194, 196
sys.new_time 91
sys.time 90, 293

T

task driven verification methodology 23
TCMs 120
temporal evaluation termination
 condition 168
temporal evaluation success equation 167
temporal expression evaluation 167
temporal expression evaluation model 167
temporal expression reductions 178
temporal expressions 72
temporal operator 73
temporal operators 73, 166, 167
temporal struct members 203
text 228, 229, 230, 237, 239
thread creation 71
thread resume 71
thread scheduler 72

thread suspension 71
thread synchronization 71, 165
thread termination 71
threads 71, 76, 77
tick and thread scheduling 72
tick notation 129
time consuming methods 72, 74
time manager 279
transaction collection 205
transaction level data checking 46
transition coverage 242
transition coverage items 100, 226, 237, 238, 255
transition temporal operators 73, 254
true temporal operator 173
true-match-variable-repeat temporal operator 176
try_next_item() 152, 153
type and subtype declaration statements 67
type declarations 64

U

ungradeable coverage items 259
unidirectional constraints 87, 109
unit 91, 143
unpack() 98, 131, 132, 133
up() 80
user data struct hierarchy 61
using also 240
utility packages 270

V

value() 114, 115
VBFM 36, 37, 38, 39, 40, 41, 42
VBFM configuration interface 40
VBFM configuration parameters 41
VBFM configuration settings 38
VBFM error injection 38
VBFM status interface 40
VE 18
verification bench h18
verification bus functional models 36, 38
verification code reuse 267, 278
verification completeness 15, 20
verification effectiveness 20
verification efficiency 15
verification environment 18
verification environment architecture 35, 36

verification environment configuration 41
verification environment initialization 41
verification environment reusability 20
verification granularity 19
verification item 18, 138
verification item driver 139
verification plan 18, 32
verification productivity 15, 19
verification quality 18
verification reusability 15
verification scenario 18, 32, 138
verification suite 18
verification views 33
verilog variable 89
vhdl driver 89
virtual BFM 155
virtual driver implementation 156
virtual fields 95, 122
virtual verification sequence 139
VS 18

W

wait 73, 75, 166, 170, 185, 187
weight 228, 229, 230, 238, 239, 258
when 114, 123, 228, 229, 230, 237, 238, 239,
240, 242
white-box verification 13

Y

yield temporal operator 177